Fraunhofer Institut Experimentelles Software Engineering

Dissertation Eric Ras - Annex 2:

Material from the Empirical Studies

Authors:
Eric Ras

IESE-Report No. 002.09/E
Version 1.0
January 20, 2009

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

Abstract

This report contains all the material used during the controlled experiment (Section 1):

- Slides that have been used to introduce the students to the experiment (Section 1.1)
- Briefing questionnaire for assessing the disturbing factors related to experience and learning style (Section 1.2)
- Pre- and post-test questionnaires for assessing the knowledge acquisition difference (Section 1.3)
- Template of an experience package and experience packages that were used in the experiment (Section 1.4)
- Learning elements for generating learning spaces used during the experiment (Section 1.5)
- Assignments for assessing reading time and application time (Section 1.6)
- Exercises for assessing efficiency, completeness, and accuracy (Section 1.7 and Section 1.8)
- Debriefing questionnaire for assessing the other disturbing factors (Section 1.9)

In addition, the material of the "Use and Acceptance" case study is included (Section 2).

Keywords: experience management, experience factory, learning space

Table of Contents

1 Material of the Controlled Experiment

This section contains all the material used during the experiment:

- Slides that have been used to introduce the students to the experiment (Section 1.1)
- Briefing questionnaire for assessing the disturbing factors related to experience and learning style (Section 1.2)
- Pre- and post-test questionnaires for assessing the knowledge acquisition difference (Section 1.3)
- Template of an experience package and experience packages that were used in the experiment (Section 1.4)
- Learning elements for generating learning spaces used during the experiment (Section 1.5)
- Assignments for assessing reading time and application time (Section 1.6)
- Exercises for assessing efficiency, completeness, and accuracy (Section 1.7 and Section 1.8)
- Debriefing questionnaire for assessing the other disturbing factors (Section 1.9)

1.1 Slides

Experiment zum Thema Refactoring und Code Smells

Ablauf

Jörg Rech, Eric Ras

rech@iese.fraunhofer.de

Tel.: 0631-6800 2210

ras@iese.fraunhofer.de

Tel.: 0631-6800 2141

Fraunhofer Institut Experimentelles Software Engineering

Experiment zum Thema Refactoring und Code Smells

Refactoring: Übersicht

- Änderung der Software zur Verbesserung nicht-funktionaler Qualitätsaspekte
  - ohne Änderung der Funktionalität
- Basiert auf aggregierten Erfahrungen über
  - **Qualitätsdefekte:** Code Smells, Anti-patterns, Design Flaws, etc.
  - **Refaktorierungen:** Elementare und eindeutige Arbeitsschritte zur Beseitigung der Defekte

Seite 2/X

Refaktorierung (1/2)

Was? (Lernziele)

- Verstehen was Refactoring im Allgemeinen ist
- Identifizieren und Unterscheiden von verschiedenen Code Smells
- Auswahl und Durchführung von Refaktorierungen für bestimmte Code Smells

Wie?

- Bereitstellen von umfangreichen Informationen und Erfahrungspaketen
- Lernen anhand von Beispielen
- Lösen von konkreten Aufgaben (zu Eurem Kode)

Refaktorierung (2/2)

Warum?

- Damit Eurer Kode noch besser wird
- Sensibilisierung für Code Smells
- Damit ihr die Grundlagen von Refaktorierung kennen lernt – es fehlt immer noch in den Vorlesungen

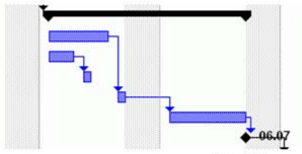
Aufwand?

- Teil des Praktikums
- Aufwand pro Student inkl. Einführung und Debriefing (insg. 10 Stunden)

Projektplan

- **Experiment** (Montag bis Donnerstag)
  - Jeder bekommt eigene Kopie eurer Systeme (Komponente)
- **Refaktorierungs-Workshops** (Freitag mit jeder Gruppe)
  - Ändern des Produktivsystems (ggf. auch noch in erster Juli-Woche)

Integrationstest	**10 dys**	**Mon 25.06.07**	**Fri 06.07.07**
Refaktorierungsexperiment	4 dys	Mon 25.06.07	Thu 28.06.07
Entwicklung Integrationsstrategie	2 dys	Mon 25.06.07	Tue 26.06.07
Präsentation Integrationsstrategie (Abgabe)	1 dy	Wed 27.06.07	Wed 27.06.07
Refaktorierungs-Workshops	1 dy	Fri 29.06.07	Fri 29.06.07
Integrationstest	5 dys	Mon 02.07.07	Fri 06.07.07
Integriertes System	0 dys	Fri 06.07.07	Fri 06.07.07



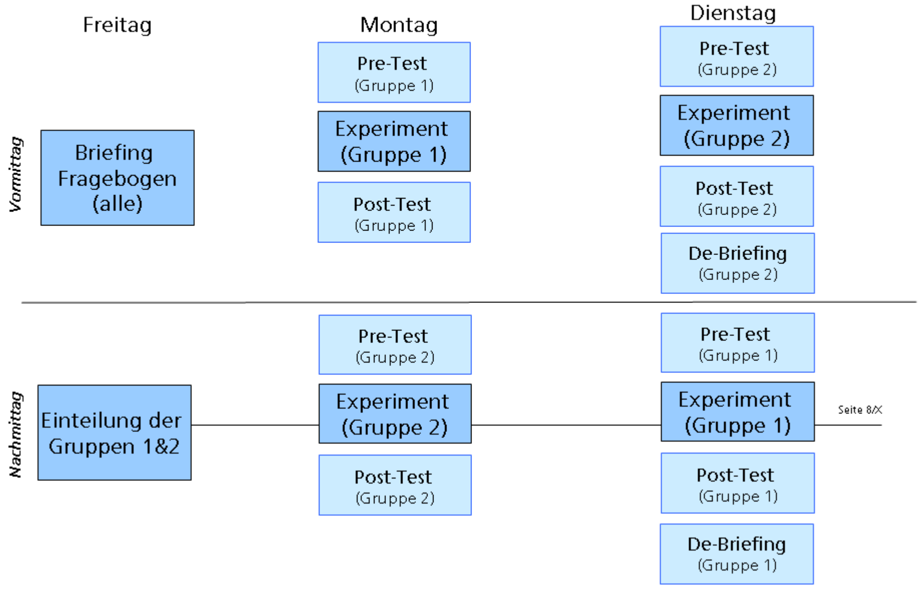
Ablauf – Genereller Ablauf der zwei Experimente

Freitag der 22.06.07	• Einführung und Fragebogen ausfüllen (vormittags)
Montags der 25.06.07 (jeweils 2 Stunden)	• Learning Space Experiment (Gruppe 1, vormittags) • Learning Space Experiment (Gruppe 2, nachmittags)
Dienstag der 26.06.07 (jeweils 2 Stunden)	• Learning Space Experiment (Gruppe 2, vormittags) • Learning Space Experiment (Gruppe 1, nachmittags)
Mittwoch der 27.06.07 (jeweils 2 Stunden)	• Doctor Q Experiment (Gruppe 1, vormittags) • Doctor Q Experiment (Gruppe 2, nachmittags)
Donnerstag der 28.06.07 (jeweils 2 Stunden)	• Doctor Q Experiment (Gruppe 2, vormittags) • Doctor Q Experiment (Gruppe 1, nachmittags)

Learning Space Experiment

Experiment zum Thema Refactoring und Code Smells

Übersicht - Learning Spaces Experiment



Ablauf – Einführung und Briefing-fragebogen

Freitag der 22.06.07
10 Uhr, ca. 1 Stunde

- Einführung in das Experiment
  - Thema Refactoring
  - Tools, Entwicklungsumgebung
  - Erläuterung der Materialien
- Ausfüllen von einem Briefing Fragebogen
- Am Freitag Nachmittag/Abend Zuordnung der Studenten zu den Gruppen für das Experiement am MO/DI.

Ablauf – Learning Spaces Experiment -

Montag der 25.06.07
9:00-11.00 Uhr
Gruppe 1

- Ausfüllen von Fragebogen 30 Minuten
- Durchführung Experiment 60 Minuten
- Ausfüllen von Fragebogen 30 Minuten

Montag der 25.06.07
14.00-16.00 Uhr
Gruppe 2

- Ausfüllen von Fragebogen 30 Minuten
- Durchführung Experiment 60 Minuten
- Ausfüllen von Fragebogen 30 Minuten

Experiment zum Thema Refactoring und Code Smells

Ablauf – Learning Spaces Experiment

Dienstag der 26.06.07
9:00-11.00 Uhr
Gruppe 1

- Ausfüllen von Fragebogen 30 Minuten
- Durchführung Experiment 90 Minuten
- Ausfüllen von Fragebogen 30 Minuten
- Ausfüllen von Debriefing-Fragebogen 5 Minuten

Dienstag der 26.06.07
14.00-16.00 Uhr
Gruppe 2

- Ausfüllen von Fragebogen 30 Minuten
- Durchführung Experiment 90 Minuten
- Ausfüllen von Fragebogen 30 Minuten
- Ausfüllen von Debriefing-Fragebogen 5 Minuten

Seite 11/X

1.2 Briefing Questionnaire

Please answer the following questions. This will take you about 5 minutes. During the analysis of the data, the data will be anonymized – your name and Matr.-Nr. (enrollment no.) will be removed.

Subject-ID	<the ID will be inserted by the evaluators>
Name:	
Matr-Nr:	

Questions on University Education

<B1>	Education	
<B1.1>	Name of study (e.g., "Angewandte Informatik")	
<B1.2>	Major Subject (i.e., "Hauptfach/Vertiefung"):	
<B1.3>	Minor Subject (i.e., "Nebenfach/Wahlfach"): *(if more than one, please mention all)*	
<B1.4>	Which lectures regarding "Software Engineering" (e.g., "SE 1-3", "GSE", ...) have you completed?	
<B1.5>	Number of terms (Fachsemester) completed (including the current one):	
<B1.6>	In how many practical courses (i.e., SE-oriented "Praktika") have you participated?	

Questions on Practical Software Engineering Experience

<B2>	Practical Software Engineering Experience	Yes	No
<B2.1>	Have you ever a written software system with more than 5 classes or 1000 lines of code?		
<B2.2>	Have you ever written software outside of university programs (e.g., private, commercial, OSS)?		
<B2.3>	Have you developed software in a large team (>4 persons) with distributed roles?		
<B2.4>	Have you developed software in a project with long duration (>6 months)?		

Questions on Experience with Programming & Java

<B3>	**Questions on Experience with Programming & Java**	
<B3.1>	How many years of computer programming experience do you have, if any?	
<B3.2>	How many different applications have you programmed?	
<B3.3>	How many different applications have you programmed in Java?	
<B3.4>	How many years were you involved in maintaining & improving a software system?	

<B4>	**What is your experience with …**	**High Experience**				**No Experience**		
<B4.1>	Java APIs (java.util, java.io, java.net, etc.)	○	○	○	○	○	○	○
<B4.2>	Java GUIs (AWT, Swing, SWT, etc.)	○	○	○	○	○	○	○
<B4.3>	Creating Java programs from scratch	○	○	○	○	○	○	○
<B4.4>	Debugging large Java programs	○	○	○	○	○	○	○
<B4.5>	The eclipse IDE (as a user, not plugin-developer)	○	○	○	○	○	○	○
<B4.6>	Other IDE such as Netbeans, Visual Studio, jBuilder, etc. (as a user, not plugin-developer)	○	○	○	○	○	○	○

Questions on Experience with Refactoring & Code Smells

<B5>	**General Questions**	
<B5.0>	Have you heard of refactoring before?	
<B5.1>	How many years of experience do you have with refactoring?	
<B5.2>	How many different applications have you refactored? (all programming languages)	
<B5.3>	How many different applications have you refactored in Java?	

<B6>	**What is your practical experience with …**	**High Experience**				**No Experience**		
<B6.1>	Identifying code smells, anti-patterns, pitfalls, design flaws, etc.	○	○	○	○	○	○	○
<B6.2>	Applying Refactorings manually	○	○	○	○	○	○	○
<B6.3>	Applying Refactorings such as "Extract Method" built into an IDE (except the "rename" refactoring)	○	○	○	○	○	○	○
<B6.4>	Working with design patterns, design heuristics, design principles, etc.	○	○	○	○	○	○	○

Questions on Experience with Software Quality Assurance & Maintenance

<B7>	What is your practical experience with …	High Experience						No Experi- ence
<B7.1>	Quality models (such as ISO 9126, FURPS, Dromey, Boehm, …)	○	○	○	○	○	○	○
<B7.2>	Testing a software system?	○	○	○	○	○	○	○
<B7.3>	Inspecting a software system regarding quality issues?	○	○	○	○	○	○	○
<B7.4>	Software measurement (Metrics)?	○	○	○	○	○	○	○
<B7.5>	Code checking tools such as PMD, checkstyle, etc.?	○	○	○	○	○	○	○

<B8>	What is your practical experience with …	High Experience						No Experi- ence
<B8.1>	Maintaining a software system? (e.g., managing defects, applying changes, etc.)	○	○	○	○	○	○	○
<B8.2>	Porting a software system to another platform? (e.g., Java 1.2 to 5.0, Java to C#, etc.)	○	○	○	○	○	○	○
<B8.3>	Improving a software system regarding efficiency (time behavior, resource behavior)?	○	○	○	○	○	○	○
<B8.4>	Improving a software system regarding reliability? (i.e., "Zuverlässig-keit")	○	○	○	○	○	○	○
<B8.5>	Improving a software system regarding usability?	○	○	○	○	○	○	○
<B8.6>	Improving a software system regarding functionality (suitability, interoperability, security)	○	○	○	○	○	○	○

Questions on Learning Style

<B9>	What is your most preferred learning style? (select one option)	
<B9.1>	Reading textbooks (with exercises)	○
<B9.2>	Classroom lectures (with exercises)	○
<B9.3>	Group work (interaction with peers and teacher / including exercises)	○
<B9.4>	Web-based training modules (with computer interaction / including examples and exercises)	○
<B9.5>	Trial and error approach (e.g., program, debug, repeat)	○

Thanks for filling out the questionnaire!

1.3 Pre- & Post-Questionnaires

This questionnaire serves to assess your competencies in the domain of refactoring and code smells. Please fill out the questionnaire as accurately as you can.

When you don't know the answer, please put your checkmark in the field "?" (Germ. Damit ist gemeint, dass Ihr nicht raten solltet – das würde die Ergebnisse verfälschen)

Before the data is processed, the data will be anonymized.

The results of the questionnaire have no impact on your grade (Germ. Note) of this practicum!

Subject-ID	**<this will be filled out by the evaluators>**
Name:	
Matr-Nr:	

General Understanding of Refactoring

<P1>	**What is refactoring about?**	**Yes**	**No**	**?**
<P1.1>	Refactoring transforms software in a way that it remains functionally identical	x		
<P1.2>	Refactoring is the art of safely removing the bad design decisions of existing code	x		
<P1.3>	Refactoring is rewriting code from scratch		x	
<P1.4>	Refactoring is dependent on eXtreme Programming (XP) methods		x	
<P1.5>	Refactoring is about a safe design-to-source transformation		x	
<P1.6>	Refactoring is about a safe source-to-source transformation	x		

<P2>	**What should be affected by refactoring?**	**Yes**	**No**	**?**
<P2.1>	The software's complexity	x		
<P2.2>	The software's flexibility	x		
<P2.3>	The software's understandability	x		
<P2.4>	The software's functionality		x	
<P2.5>	The behavior of the methods, classes, and components	x		
<P2.6>	The observable behavior of the software from the perspective of the user		x	
<P2.7>	The program's syntax	x		
<P2.8>	The software's performance	x		
<P2.9>	The program's semantics (meaning of methods, classes, etc.)	x		
<P2.10>	The program's size	x		

<P3>	**When and how should a refactoring be considered?**	**Yes**	**No**	**?**
<P3.1>	When a design choice is not explicitly addressed in one place in a system	x		
<P3.2>	When a code smell has been detected	x		
<P3.3>	When a system failure has been detected (e.g., by testing)		x	
<P3.4>	When the system design has a weakness	x		
<P3.5>	Refactoring is done on a periodical basis		x	
<P3.6>	Before implementing a new feature and if the design does not fit this change	x		
<P3.7>	Refactorings are always performed in small steps with compilation and test in-between	x		
<P3.8>	Refactorings are implemented completely. Afterwards, compilations and test are done because only completed refactorings result in a running system		x	
<P3.9>	Refactoring can be applied when the unit and acceptance tests haved failed; refactoring can help to solve the detected failures.		x	
<P3.10>	Refactoring should only be applied when the required automated unit or acceptance tests have been conducted successfully.	x		

<P4>	**What are code smells?**	**Yes**	**No**	**?**
<P4.1>	Code Smells are weaknesses in the requirements		x	
<P4.2>	Code Smells are failures observed by the user		x	
<P4.3>	Code Smells are defects observed by the tester		x	
<P4.4>	Code Smells are defects observed by the developer	x		
<P4.5>	Code Smells are weaknesses in the design	x		
<P4.6>	All Code Smells can be easily determined by using appropriate measures		x	
<P4.7>	Determining what is and is not a Code Smell is often a subjective judgment	x		

Assigment of Refactoring Methods to Code Smells

<P5>	**What refactorings are used to remove the following code smells?** **<place checkmarks in the columns for each code smell>** **<for those refactorings where you don't know which code smells they are suitable for, choose "?" >**								
		?	Comment	Long Method	Type Embedded in Name	Uncommunicative Name	Long Parameter List	Lazy Class	Data Class
<P5.1>	AddParameter						x		
<P5.2>	DecomposeConditional			x					
<P5.3>	EncapsulateCollection								x
<P5.4>	EncapsulateField								x
<P5.5>	ExtractMethod		x	x					
<P5.6>	HideMethod								
<P5.7>	IntroduceAssertion		x						
<P5.8>	IntroduceParameterObject			x			x		
<P5.9>	MoveMethod								x

<P5.10>	PreserveWholeObject			x			x		
<P5.11>	RemoveParameter								
<P5.12>	RemoveSettingMethod								
<P5.13>	RenameMethod		x		x	x			
<P5.14>	ReplaceMethodwithMethodObject			x					
<P5.15>	ReplaceParameterwithMethod						x		
<P5.16>	ReplaceTempwithQuery			x					

Questions related to the code smell *Long Method*

<C2>	**Questions related to the code smell *Long Method***	
<C2.1>	Explain in your own words what a *Long Method* code smell is? What are the problems it brings to the code?	**?**
	<Your answer:>	
<C2.2>	Mark the blocks in the following method that you would extract in order to make the method shorter (with your text marker)	

```
//example from Wakes p. 23
import java.util.*;
import java.io.*;

public class Report {
        public static void report(Writer out, List machines, Robot
robot)
        throws IOException {
                out.write("FACTORY REPORT\n");
                out.write("This list includes information on
"+machines.size()+ " machines")
                Iterator line = machines.iterator();
                while (line.hasNext() {
                        Machine machine = (Machine) line.next();
                        out.write("Machine " + machine.name());
                        if (machine.status() != null)
                                out.write(" status=" + ma-
chine.status());
                        out.write("\n");
                }
                out.write("\n");

                out.write("Robot ");
                if (robot.location() != null)
                        out.write("location=" + ro-
bot.location().name()):
                if(robot.status() != null)
                        out.write("status=" + robot.status());

                out.write("\n");
                out.write("=========\n")
        }
}
```

C2.3>	Rewrite the report(…) method, as you have done the extract method for each block. (don't describe the new methods – only the new report() with the call of the extracted methods	**?**

	<Your answer:>
	`public static void report (`
	`Printstream out, L i s t machines. Robot robot) {`
	`        reportHeader(out);`
	`        reportMachines(out, machines);`
	`        reportRobot(out, robot);`
	`        reportFooter(out);`
	`}`	

<C2.4>	What refactorings are suitable for the code smell *Long Method* in general? Name them all.	**?**
	<Your answer:> ExtractMethod IntroduceParameterObject PreserveWholeObject ReplaceTempWithQuery ReplaceMethodWithMethodObject	
<C2.5>	In what order should the previously listed refactorings be applied? <put "no sequence" if the sequence is not important>	**?**
	<Your answer:> 1. ExtractMethod 2. IntroduceParameterObject, 3. PreserveWholeObject, 4. ReplaceTempWithQuery 5. ReplaceMethodWithMethodObject	

<C2.6>	What refactoring would you apply for this *Long Method* code smell example first? Please mark the code smell and explain why you apply this refactoring.	?

```
class Customer ...
        public String statement() (
                double totalAmount = 0;
                int frequentRenterPoints = 0;
                Enumeration rentals = _rentals.elements();
                String result = "Rental Record for " + getName() +
"\n";
                while (rentals.hasMoreElements()) {
                        Rental each = (Rental) rentals.nextElement();

                        //add frequent renter points
                        frequentRenterPoints ++;

                        //add bonus for a two day new release rental
                        if ((each.getMovie().getPriceCode() ==
Movie.NEW_RELEASE)
                                        && each.getDaysRented() > 1) {

                                frequentRenterPoints ++;
                        }
                        //show figures far this rental
                        result += "\t" + each.getMovie().getTitle()+
"\t" +
                        String.valueOf(each.getCharge()) + "\n";
                        totalAmount += each.getCharge();
                }

                //add footer lines
                result += "Amount owed is " +
Strinq,valueOf(totalAmount) + "\n";
                result += "You earned " +
String.valueOf(frequentRenterPoint) + "frequent renter points";
                return result;
}
```

<Your answer:>

Use Extract Method

Questions related to the code smell *Type Embedded in Name*

<C3>	**Questions related to the code smell *Type Embedded in Name***			
<C3.1>	Explain in your own words what a *Type Embedded in Name* code smell is? What are the problems it brings to the code?	?		

<Your answer:>

The following problems are related to the code smell Type Embedded in Name.

- Method names are compound words, consisting of a word plus the type of the argument(s).For example, a method addCourse(Course c).
- Names are in Hungarian notation, where the type of an object is encoded into the name; e.g., icount as an integer member variable.
- Variables names reflect their type rather than their purpose or role.

		Yes	**No**	**?**
<C3.2>	Which of the following examples included is a *Type Embedded in Name* code smell? <please mark the smell with your pen>			
	`public Class getColumnClass(final int columnIndex) {`	x		

	return String.class; }			

	public class Texts {
	private static final String BUNDLE_NAME =
	"de.frewert.dndinfo.gui.dndinfo"; //$NON-NLS-1$
	
	private static final ResourceBundle RESOURCE_BUNDLE =
	ResourceBundle.getBundle(BUNDLE_NAME);
	
	private Texts()
	
	public static String getString(String key) {
	try {
	return RESOURCE_BUNDLE.getString(key);
	} catch (MissingResourceException e) {
	return '!' + key + '!';
	}
	}
	}
		x		

	public void paintComponent(Graphics g) {
	super.paintComponent(g);
	
	Graphics2D g2 = (Graphics2D) g;
	g2.setFont(bgFont);
	g2.setColor(fontColor);
	
	int dividerPos = getDividerLocation();
	drawCentered(g2, info[0], 0, dividerPos);
	drawCentered(g2, info[1], divider-
	Pos + getDividerSize(),
	getHeight());
	}
			x	

	private Observer dndObserver = new Observer() {
	public void update(Observable o, Object arg) {
	if (arg instanceof DataFlavor[]) {
	
	gui.displayFlavors((DataFlavor[]) arg);
	
	} else if (arg instanceof String) {
	
	gui.appendData((String) arg);
	
	} else if (arg instanceof int[]) {
	
	int [] action = (int[]) arg;
	gui.setSourceActions(action[0]);
	gui.setUserAction(action[1]);
	}
	}
	}
		x		

	private ActionListener quitListener = new ActionLis-
	tener() {
	public void actionPerformed(ActionEvent e) {
	Main.this.quit();
	}
			x	

<C3.3>	Give another simple example of a code smell *Type Embedded in Name*		**?**	
	<Your answer:>			

<C3.4> Please name the refactoring applied to the following *Type Embedded in Name* code smell: **?**

<your answer:> RenameMethod

```java
public void storeTask (Task t) {
        t.setTaskId(numberTasks+1);
        currentTaskList.addTask(t);
        numberTasks++;
                                    }
```

is transformed to:

```java
public void store (Task t) {
        t.setTaskId(numberTasks+1);
        currentTaskList.addTask(t);
        numberTasks++;
                                    }
```

<C3.5> List the refactorings that are suitable for the code smell *Type Embedded in Name* in general **?**

<Your answer:>

RenameMethod

<C3.6> In what order should the previously listed refactorings be applied? <put “no sequence” if the sequence is not important> **?**

<Your answer:>

RenameMethod

<C3.7> What refactoring would you apply for this *Type Embedded in Name* code smell example? Please mark each code smell with your text marker and explain why you apply this refactoring. **?**

```java
public void addDropTargetListener(DropTargetListener
dtl) {
 /*
 * Using the GlassPane as only DropTarget would be more
 * elegant, but Drag&Drop doesn't work with a
 * GlassPane in Java <= 1.4.0. (See Java bug #4435403)
 */

// Use the following block if JRE 1.3 compatibility
// isn't neccessary any longer.

// Component c = SwingUtilities.getRoot(this);
// if ((c != null) && (c instanceof JFrame)) {
// JFrame f = (JFrame) c;
// Component glassPane = f.getGlassPane();
// glassPane.setVisible(true);
// DropTarget dropTarget = new DropTarget(glassPane,
dtl);
// }

 new DropTarget(flavorArea, dtl);
 new DropTarget(dataArea, dtl);
                            }
```

	<Your answer:> addDropTargetListener(DropTargetListener is a type embedded in name code smell. The variable type is embedded in the method name. When the type changes, the method also needs to be renamed.	

Questions related to the code smell *Comments*

<C4>	**Questions related to the code smell *Comments***			
<C4.1>	Explain in your own words what a *Comments* code smell is? What are the problems if brings to the code?			**?**
	<Your answer:> Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program and should be added when the author realizes that something isn't as clear as it could be and adds a comment. In addition, the frequency of comments sometimes reflects poor quality of code. A lot of comments can be reflected just as well in the code itself.			
<C4.2>	Which of the following examples includes at least one *Comments* code smell? <please mark the smell(s) with your text marker>	**Yes**	**No**	**?**

	`private JScrollPane getFlavorScrollPane(final Map map,`
	` String header1,`
	` String header2) {`
	` JTable table = new JTable(new FlavorTableModel(map,`
	` header1,`
	` header2));`
	` final int viewportHeight = 12 * table.getRowHeight();`
	` table.setPreferredScrollableViewportSize(new Dimension(450,`
	`viewportHeight));`
	
	` // table.getColumn(header1).setPreferredWidth(header1.);`
	
	` JScrollPane scrollPane = new JScrollPane(table);`
	` scroll-`
	`Pane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLB`
	`AR_ALWAYS);`
	
	` return scrollPane;`
	`}`
		x		

	`public class AboutDialog extends JDialog {`
	` private static final long serialVersionUID =`
	`3257853194578048567L;`
	
	` /**`
	` * Create a new AboutDialog.`
	` * @param parent the parent frame.`
	` * @param title the title of the dialog`
	` * @param version the version of the application`
	` */`
	` public AboutDialog(Frame parent, String title, String ver-`
	`sion) {`
	` super(parent,`
	` Texts.getString("AboutDialog.title.prefix") + title);`
		x		

```
//$NON-NLS-1$
 createGui(title, version);
 pack();
 setResizable(false);
                              }
```

```
/**
 * Liest die Refaktorierungen eines Diagnose-Plug-Ins aus
der Extension-
 * Beschreibung aus.
 *
 * @param extensionID
 * ID der Extension, die den Extension-Point <co-
de>Diagnosis</code>
 * implementiert
 * @return Refaktorierungen als Komma-separierte Liste in
einem String
 */
        public String getRefactorings(String extensionID) {
                return getAttributeValue(extensionID,
EP_DIAGNOSIS,
                                "refactorings", ELE-
MENT_FRONTEND);
        }
```

x

```
private ActionListener buttonListener = new ActionListener()
{
 public void actionPerformed(ActionEvent e) {
 // Don't dispose, dialog is reused in Main class
 FlavorDialog.this.hide();
 }
                              };
```

x

```
//      Constructor where the Id is set
        public TaskList(int taskListId){
                this.taskListId =taskListId;
                state=false;
                tasks=new HashSet<Task>();
        }
```

x

<C4.3>	Give another simple example of a code smell *Comment*	**?**
	<Your answer:>	
<C4.4>	Please name the refactoring applied to the following *Comments* code smell:	**?**
	<your answer:> IntroduceAssertion	

```
/**
 * @param clipLimit has to be larger than zero
 * @param delta has to have a positive value
 */
public boolean match(int[] expected, int[] actual, int
clipLimit, int delta)
        {
                // Clip " too- large" values
                for (int i = 0; i < actual.length; i++)
                        if (actual [i] > clipLimit)
                                actual [i] = clipLimit;
                // Check for length differences
                if (actual.length != expected.length)
                        return false;
```

	<pre> // Check that each entry within expected +/-
	delta
	                for (int i = 0; i < actual.length; i++)
	                        if (Math.abs(expected[i] - actual[i]
	> delta)
	                                        return false;
	
	                        return true;
	                                   }</pre>
	is transformed to:
	<pre>public boolean match(int[] expected, int[] actual, int
	clipLimit, int delta)
	        {
	                assert expected != null;
	                assert actual != null;
	                assert clipLimit >= 0;
	                assert delta >= 0;
	
	                // Clip " too- large" values
	                for (int i = 0; i < actual.length; i++)
	                        if (actual [i] > clipLimit)
	                                actual [i] = clipLimit;
	                // Check for length differences
	                if (actual.length != expected.length)
	                        return false;
	                // Check that each entry within expected +/-
	delta
	                for (int i = 0; i < actual.length; i++)
	                        if (Math.abs(expected[i] - actual[i]
	> delta)
	                                        return false;
	
	                        return true;
	        }</pre>	

<C4.5>	List the refactorings that are suitable for the code smell *Comments* in general	**?**
	<Your answer:> ExtractMethod IntroduceAssertion RenameMethod	
<C4.6>	In what order should the previously listed refactorings be applied? <put "no sequence" if the sequence is not important>	**?**
	<Your answer:> doesn't matter, depends on the type of the *comments code smell*.	
<C4.7>	What refactoring(s) would you apply for this(these) *Comments* code smell example(s)? Mark each code smell with your text marker and explain why you apply this refactoring.	**?**

	<pre>/** Simulation of a Tic-Tac-Toe game (does not do strategy).
	*/
	public class TicTacToe {
	 protected static final int X = 1, O = -1; // players
	 protected static final int EMPTY = 0; //
	empty cell
	 protected int board[][] = new int[3][3]; // game board
	 protected int player; // current
	player
	 /** Constructor */</pre>	

```
public TicTacToe() { clearBoard(); }
/** Clears the board */
public void clearBoard() {
for (int i = 0; i < 3; i++)
for (int j = 0; j < 3; j++)
board[i][j] = EMPTY;         // every cell should be empty
player = X;                          // the first player is 'X'
}
/** Puts an X or O mark at position i,j */
public void putMark(int i, int j) throws IllegalArgumentException {
if ((i < 0) { (i > 2) { (j < 0) { (j > 2))
throw new IllegalArgumentException("Invalid board position");
if (board[i][j] != EMPTY)
throw new IllegalArgumentException("Board position occupied");
board[i][j] = player;        // place the mark for the current player
player = - player;           // switch players (uses fact that O = - X)
}
/** Checks whether the board configuration is a win for the given player */
public boolean isWin(int mark) {
return ((board[0][0] + board[0][1] + board[0][2] == mark*3) // row 0
        { (board[1][0] + board[1][1] + board[1][2] == mark*3) // row 1
        { (board[2][0] + board[2][1] + board[2][2] == mark*3) // row 2
        { (board[0][0] + board[1][0] + board[2][0] == mark*3) // column 0
        { (board[0][1] + board[1][1] + board[2][1] == mark*3) // column 1
        { (board[0][2] + board[1][2] + board[2][2] == mark*3) // column 2
        { (board[0][0] + board[1][1] + board[2][2] == mark*3) // diagonal
        { (board[2][0] + board[1][1] + board[0][2] == mark*3)); // diagonal
}
/** Returns the winning player or 0 to indicate a tie */
public int winner() {
if (isWin(X))
return(X);
else if (isWin(O))
return(O);
else
return(0);
}
```

<Your answer:>

It is clear that the constructor is the constructor!

The name of the method clearBoard tells the reader what the method does. The comment is redundant. The same is true for the methods putMark and isWin.

Questions related to the code smell *Uncommunicative Name*

<C4>	**Questions related to the code smell *Uncommunicative Name***			
<C4.1>	Explain in your own words what a *Uncommunicative Name* code smell is? What are the problems it brings to the code?		**?**	
	<Your answer:> A name doesn't communicate its intent of a method, variable, classes, etc. well enough - One- or two-character names - Names with vowels omitted - Numbered variables (e.g., panel, pane2, and so on) - Odd abbreviations - Misleading names			
<C4.2>	Which of the following examples includes at least one *Uncommunicative Name* code smell? <please mark the smell(s) with your text marker>	**Yes**	**No**	**?**

	`public Class getColumnClass(final int columnIndex) {`
	` return String.class;`
	`}`
			x	

	`public void addDropTargetListener(DropTargetListener dtl)`
	`{`
	` new DropTarget(flavorArea, dtl);`
	` new DropTarget(dataArea, dtl);`
	`}`
		x		

	`public String getColumnName(final int column) {`
	` String name = (column >= columnHeader.length)`
	` ? ""`
	` : columnHeader[column];`
	` return (name == null) ? "" : name;`
	` }`
			x	

	`public void insertUpdate(DocumentEvent e) {`
	` /* using invokeLater seems neccessary */`
	` SwingUtilities.invokeLater(new Runnable() {`
	` public void run() {`
	` scrollbar.setValue(scrollbar.getMaximum());`
	` }`
	` });`
	`}`
		x		

	`public void paintComponent(Graphics g) {`
	` super.paintComponent(g);`
	
	` Graphics2D g2 = (Graphics2D) g;`
	` g2.setFont(bgFont);`
	` g2.setColor(fontColor);`
	
	` int dividerPos = getDividerLocation();`
	` drawCentered(g2, info[0], 0, dividerPos);`
	` drawCentered(g2, info[1],`
	` dividerPos + getDividerSize(), getHeight());`
	`}`
		x		

	`public Object getValueAt(final int arg0, final int arg1)`
	`{`
	` return data[arg0][arg1];`
	` }`
		x		

<C4.3> Give another simple example of a *Uncommunicative Name* code smell **?**

<Your answer:>

<C4.4> Please name the refactoring applied to the following *Uncommunicative Name* code smell: **?**

<your answer:> RenameMethod (or RenameVariable)

```
//data contains the colortable

public Object getValueAt(final int arg0, final int arg1)
{
 return data[arg0][arg1];
                            }
```

is transformed to:

```
public Object getValueAt(final int x, final int y) {
 return data[x][y];
}
```

<C4.5> List the refactorings that are suitable for the code smell *Uncommunicative Name* in general **?**

<Your answer:>

RenameMethod (or RenameVariable)

<C4.6> In what order should the previously listed refactorings be applied? <put "no sequence" if the sequence is not important> **?**

<Your answer:>

RenameMethod (or RenameVariable)

<C4.7> What refactoring would you apply for this(these) *Uncommunicative Name* code smell example(s)? Mark the code smell with your text marker and explain why you apply this refactoring. **?**

```
private Observer dndObserver = new Observer() {
 public void update(Observable o, Object arg) {
 if (arg instanceof DataFlavor[]) {

 gui.displayFlavors((DataFlavor[]) arg);

 } else if (arg instanceof String) {

 gui.appendData((String) arg);

 } else if (arg instanceof int[]) {

 int [] action = (int[]) arg;
 gui.setSourceActions(action[0]);
 gui.setUserAction(action[1]);
 }
 }
                            };
```

<Your answer:>

o is a one-character variable

	arg is a variable name with no meaning	

1.4 Experience Packages for Experimentation

This section illustrates first the template for experience packages and afterwards the experience packages used in the controlled experiment.

1.4.1 Experience Package Template

Titel of EP			**Type**	Experience
Action (A)				
	Abstract:			
	Problem:			
	Solution:			
Benefit (B)				
	Effect:			
Context (C)				
	Product:			
	Process:			
	Project:			
	Knowledge:			
	Organization:			
	People:			
	Group:			
Description (D)				
	Explanation:			
	Example:			
Evidence (E)	Analysis Technique:		Hypothesis:	
Administrative				
	Author:		Date:	
	Version:		Relation EPs:	
	Status:			
Remark				

1.4.2 Experience Package: Code Smell *Long Method*

Titel of EP	Code Smell Long Method		**Type**	Experience
Action (A)				
	Abstract:	Large methods consist of a large number of lines. You should be suspicious when a method has more than 5 to 10 lines. The refactorings ExtractMethod, ReplaceTempwithQuery, ReplaceMethodwithMethodObject, DecomposeConditional can be used to reduce this kind of code smell. They will improve the class structure and abstraction levels.		
	Problem:	A method starts down a path and, rather than break the flow or identify the helper classes, the author adds more and more. Code is often easier to write than it is to read, so there's a temptation to write blocks that are too big, which means that they get difficult to maintain, understand, etc.		
	Solution:	The refactoring *ExtractMethod* could be used to break up the method into smaller parts. Look for comments or white space delineating interesting blocks. You want to extract methods that are semantically meaningful, not just introduce a function call every seven lines. In addition, the following three methods can be used, too: - *ReplaceTempwithQuery*: Temporary variables are used to hold the result of an expression. This expression should be replaced with a method. Extract the expression into a method. - *ReplaceMethodwithMethodObject*: The difficulty in decomposing a method lies in local variables. If they are rampant (Germ. üppig), decomposition can be difficult. Applying it turns all the local variables into fields on the method object and ExtractMethod can be applied on this new object afterwards. - *DecomposeConditional*: Methods named after the intention of that block of code replace the parts of the conditional part and each of the alternatives. This way you highlight the condition and make it clear what you are branching on.		
Benefit (B)				
	Effect:	Improves communication. May expose duplication. Often helps to get new classes and abstractions		
Context (C)				
	Product:	Java Code		
	Process:	ExtractMethod, ReplaceTempwithQuery, ReplaceMethodwithMethodObject, DecomposeConditional		
	Project:	OO projects		
	Knowledge:	Code Smell Long Method		
	Organization:	Fraunhofer IESE		
	Individual:	Eric Ras		
	Group:	SOP-Dev		
Evidence (E)	Analysis Technique:	-	Hypothesis	-
Administrative				
	Author:	Martin Fowler	Date:	1999
	Version:		Relation EPs:	
	Status:			

remark	Exercise	W. Exercise 4 23ff		

1.4.3 Experience Package: Code Smell *Type Embedded in Name*

Titel of EP	Code Smell Type Embedded in Name	**Type**		Experience
Action (A)				
	Abstract:	When types are embedded in names, it's not only redundant, but it forces you to change the name if the type changes. This often results. Therefore, the refactoring RenameMethod is applied to avoid this kind of code smell, which is called Type Embedded in Name. Avoid placing types in method names!		
	Problem:	The embedded name can create unnecessary troubles because later changes of the parameter (i.e., type) will lead to a renaming of the method and the related calls.		
	Solution:	The refactoring *RenameMethod* (the same is done for fields or constants) should be applied, which leads to a new name that communicates the intent of the method without being so much tied to a type.		
Benefit (B)				
	Effect:	Improves communication. May make it easier to spot duplication.		
Context (C)				
	Product:	Java code		
	Process:	RenameMethod		
	Project:	OO projects		
	Knowledge:	Code Smell Type Embedded in Name		
	Organization:	Fraunhofer IESE		
	Individual:	Eric Ras		
	Group:	Sop-Dev		
Evidence (E)	Analysis Technique:		Hypothesis	
Administrative				
	Author:	Wakes	Date:	
	Version:	1.0	Relation EPs:	
	Status:	stable		
remark	exercise	no one in W.		

1.4.4 Experience Package: Code Smell *Comments*

Titel of EP	Code Smell Comments	**Type**	Experience
Action (A)			
	Abstract:	Comments serve for a better communication and explanation of code. It's surprising how often the code is badly commented and that the comments are there because the code is bad. Hence, comments can be substituted by refactoring methods.	

	Problem:	Comments are often used to explain bad code. Programmers must add a lot of comment to explain their classes and methods because their naming does not give a hint what they itend to do.		
	Solution:	The first action in refactoring is to remove the bad code smells. When this is done many comments get superfluous. In fact, the goal of a routine can often be communicated as well through the routine's name as it can through a comment. The following refactorings should be used to reduce the comments and to improve the code: - When a comment explains a block of code, you can often use the refactoring *ExtractMethod* to pull the block out into a separate method. The comment will often suggest a name for the new method. - When a comment explains what a method does (better than the methods name), use the refactoring *RenameMethod* using the comment as the basis of the new name. - When a comment explains preconditions, consider using the refactoring *IntroduceAssertion* to replace the comment with code.		
Benefit (B)				
	Effect:	Improves communication. May expose duplication		
Context (C)				
	Product:	Java Code		
	Process:	ExtractMethod, IntroduceAssertion, RenameMethod		
	Project:	OO projects		
	Knowledge:	Code Smell Comment		
	Organization:	Fraunhofer IESE		
	People:	Eric Ras		
	Group:	SOP-Dev		
Evidence (E)	Analyse Technique:	NA	Hypothesis	NA
Administrative				
	Author:	Martin Fowler (p 87)	Date:	1999
	Version:	1.0	Relation EPs:	
	Status:	stable		
Remark	Exercise	Wakes s19-20		

1.4.5 Experience Package: Code Smell *Uncommunicative Name*

Titel of EP	Code Smell Uncommunicative Name	**Type**	Experience
Action (A)			
	Abstract:	The name does not explain the intent of a method. This makes understanding a time consuming activity. The refactoring *RenameMethod* should be applied to remove uncommunicative names.	
	Problem:	A name doesn't communicate its intent of a method well enough.	
	Solution:	Use *RenameMethod* (or held, constant, etc.) to give it a better name.	

Benefit (B)				
	Effect:	Improves communication		
	Product:	Java Code		
	Process:	RenameMethod		
	Project:	OO projects		
	Knowledge:	Code Smell Uncommunicative Name		
	Organization:	Fraunhofer IESE		
	People:	Eric Ras		
	Group:	Sop-Dev		
Description (D)				
	Explanation:			
	Example:			
	Exceptions			
Evidence (E)	Analyse Technique:		Hypothesis	
Administrative				
	Author:		Date:	
	Version:		Relation EPs:	
	Status:			
remark				

1.4.6 Experience Package: Code Smell *Long Parameter List*

Titel of EP	Code Smell Long Parameter List	**Type**	Experience
Action (A)			
	Abstract:	A method that has more than 2 parameters. Long parameter list are difficult to understand. Apply the refactorings ReplaceParameterwithMethod, IntroduceParameterObject, PreserveWholeObject to remove this problem.	
	Problem:	Long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because you are forever changing them as you need more data. Reasons for long parameter list are often routines that provide a general algorithm, which need to have a lot of parameters in order to cover all the needed variations.	
	Solution:	Most problems are removed by passing objects instead of using a lot of parameters because you are much more likely to make only a couple of requests to get at a new piece of data. You pass only the minimum so that the method can get everything it needs. The following refactoring can be used for reducing parameter lists: - Use *ReplaceParameterwithMethod* when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter. - If the parameter comes from a single object use *PreserveWholeObject* and replace it with the object itself.	

		- If you have several data items from different logical objects, use *IntroduceParameterObject* to group the parameters.
Benefit (B)		
	Effect:	Improves communication. May expose duplication. Often reduces size
Context (C)		
	Product:	Java Code
	Process:	ReplaceParameterwithMethod, IntroduceParameterObject, PreserveWholeObject
	Project:	OO projects
	Knowledge:	Code Smell Long Parameter List
	Organization:	Fraunhofer IESE
	People:	Eric Ras
	Group:	Sop-Dev

Evidence (E)	Analyse Technique:		Hypothesis	
Administrative				
	Author:		Date:	
	Version:		Relation EPs:	
	Status:			
remark	exercise	W. 31		

1.4.7 Experience Package: Code Smell *Lazy Class*

Titel of EP	Code Smell Lazy Class	**Type**	Experience
Action (A)			
	Abstract:	A class that isn't doing enoug to pay for itself should be removed for reasons of code size, code simplicity, and understandability. The refactorings such as *CollapseHierarchy* and *InlineClass* help in this situation.	
	Problem:	A class isn't doing much – its parents, children, or callers seem to be doing all the associated work, and there isn't enough behavior left in the class to justify its continued existence. They have a negative impact on size, simplicity, and understandability of the code.	
	Solution:	If parents or children of the class seem like the right place for the class's behavior, fold it into one of them via the refactoring *CollapseHierarchy*. Otherwise, fold its behavior into its caller via the refactoring *InlineClass*.	

Benefit (B)				
	Effect:	Reduces size. Improves communication. Improves simplicity.		
Context (C)				
	Product:	Java Code		
	Process:	CollapseHierarchy, InlineClass,		
	Project:	OO projects		
	Knowledge:	Code Smell Lazy Class		
	Organization:	Fraunhofer IESE		
	People:	Eric Ras		
	Group:	Sop-Dev		
Evidence (E)	Analyse Technique:		Hypothesis	
Administrative				
	Author:		Date:	
	Version:		Relation EPs:	
	Status:			
remark	exercise	W. 91		

1.4.8 Experience Package: Code Smell *Data Class*

Titel of EP	Code Smell Data Class	**Type**	Experience
Action (A)			
	Abstract:	These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are being manipulated too much by other classes. Bad understanding and communcication is the consequence. Appropriate refactorings such as *EncapsulateCollection, RemoveSettingMethod,* or *EncapsulateField* solve the problem of data classes.	
	Problem:	The existence of these kind of data classes is bad for understanding and communcication. The way how they happen is because it's common for classes to begin like this: You realize that some data is part of an independent object, so you extract it. But objects are about the commonality of behavior; and these objects aren't developed enough as yet to have much behavior. Bad understanding and communcication is the consequence.	
	Solution:	1. If classes have public fields. If so, you should immediately apply *EncapsulateField* to block direct access to the fields (allowing access only through getters and setters). 2. If you have collection fields, check to see whether they are properly encapsulated and apply *EncapsulateCollection* if they aren't, use *RemoveSettingMethod* on any field that should not be changed. 3. You'll find clients accessing the fields and manipulating the results when the class could do it for them. Use *ExtractMethod* on the client to pull out the class-related code, then *MoveMethod* to put it over on the data class. If you can't move a whole method, use *ExtractMethod* to create a method that can be moved.	

		4. After-doing this awhile, you may find that you have several similar methods on the class. Use refactorings such as *RenameMethod*, *ExtractMethod*, *AddParameter*, or *RemoveParameter* to harmonize signatures and remove duplication. 5. Most access to the fields shouldn't be needed anymore because the moved methods cover the real use. So use *HideMethod* to eliminate access to the getters and setters.		
Benefit (B)				
	Effect:			
Context (C)				
	Product:	Java Code		
	Process:	*EncapsulateField, EncapsulateCollection, RemoveSettingMethod, ExtractMethod, MoveMethod, RenameMethod*, *AddParameter*, *RemoveParameter, HideMethod*		
	Project:	OO Project		
	Knowledge:	Code Smell Data Class		
	Organization:	Fraunhofer IESE		
	People:	Eric Ras		
	Group:	Sop-Dev		
Evidence (E)	Analyse Technique:		Hypothesis	
Administrative				
	Author:		Date:	
	Version:		Relation EPs:	
	Status:			
remark	exercise	W. 31		

1.5 Learning Spaces for Experimentation

This section describes the learning elements of the different learning spaces. The learning elements have been created in the software organization platform by using the learning element authoring tool.

1.5.1 Learning Space: Code Smell *Comments*

Learning Space Page				
0	Experience Package - Code Smell Comment	Exerience A+B		
		Ontology	**Type of**	**Learning Content**

		Instance (Ontology Main Class)	Learning Element	
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software ystem in such a way that it does not alter the external behavior of the code yet improves its internal structure. [Fowler] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real smells that can keep you busy.

3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. – During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring - Overview	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException.

				6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experience Package - Code Smell Comment	Repeat Experience (AB)		
8	Comment - Introductio n	Comment (Knowledge)	Description	Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program and should be added when the author realizes that something isn't as clear as it could be and adds a comment. Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves. In addition, the frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer. Some comments are particularly helpful: - Those that tell why something is done a particular way (or why it wasn't) - Those that cite algorithms that are not obvious (where a simpler algorithm won't do) Other comments can be reflected just as well in the code itself! The refactorings ExtractMethod, IntroduceAssertion, RenameMethod should be used to remove this kind of code smells.
9	ExtractMet hod	ExtractMeth od (process)	Description	The refactoring ExtractMethod could be used to break up the method into smaller parts. Look for comments or white space delineating interesting blocks. You want to extract methods that are semantically meaningful, not just introduce a function call every seven lines. ExtractMethod is one of the most common refactorings. Look at a method that is too long or look at code that needs a comment to understand its purpose. Then turn that fragment of code into its own method whose name explains the purpose of the method. Short, well-named methods should be preferred for several reasons: • First, it increases the chances that other methods can use a method when the method is finely grained.

				• Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained. Small methods really work only when you have good names, so you need to pay attention to naming. What is the optimal length for a method? In fact, length is not the issue. The key is the semantic distance between the method name and the method body. If extracting improves clarity, do it, even if the name is longer than the code you have extracted.

			Process	• Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
				    o If the code you want to extract is very simple, such as a simple message or function call, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code.
				• Copy the extracted code from the source method into the new target method.
				• Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
				• See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
				• Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method as it stands. You may need to use *SplitTemporaryVariable* (128) and try again. You can eliminate temporary variables with ReplaceTempwithQuery (see the discussion in the examples).
				• Pass into the target method as parameters local-scope variables that are rad from the extracted code.
				• Compile when you have dealt with all the locally-scoped variables.
				• Replace the extracted code in the source method with a call to the target method.
				    o If you have moved any temporary variables over to the target method, look to see whether they were declared outside of the extracted code. If so, you can now remove the declaration.
				• Compile and test.

		ExtractMethod (process)	Example	void printOwing() {
				        printBanner();
				
				        //print details
				        System.out.println ("name: " + _name);
				        System.out.println ("amount: " + amount);
				}
				
				It is easy to extract the code that prints the banner. You just cut, paste, and put in a call:
				
				void printOwing(dounble amount) {
				    printBanner();
				    printDetails(amount);
				}
				
				void printDetails (double amount) {
				    System.out.println ("name:" + _name);
				    System.out.println ("amount: " + amount);
				}

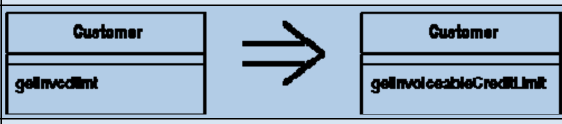
10	IntroduceA	IntroduceAs	Description	Often sections of code work only if certain conditions are true. This may be as simple

	ssertion	sertion (process)		as a square root calculation's working only on a positive input value. With an object it may be assumed that at least one of a group of fields has a value in it. Such assumptions often are not stated but can only be decoded by looking through an algorithm. Sometimes the assumptions are stated with a comment. A better technique is to make the assumptions explicit by writing an assertion. An assertion is a conditional statement that is assumed to be always true. Failure of an assertion indicates programmer error. As such, assertion failures should always result in unchecked exceptions. Assertions should never be used by other parts of the system. Indeed assertions usually are removed for production code. It is therefore important to signal something is an assertion. Assertions act as communication and debugging aids. In communcication they help the reader understand the assumption the code is making. In debugging, assertions can help catch bugs closer to their origin. It has been noticed the debugging help is less important when write self-testing code is writing, but the value of assertions is still appreciated in communciation.
		IntroduceAssertion (process)	Process	Because assertions should not affect the running of a system, adding one is always behavior preserving. • When you see that a condition is assumed to be true, add an assertion to state it. o Have an assert class that you can use for assertion behavior Beware of overusing assertions. Don't use assertions to check everything that you think is true for a section of code. Use assertions only to check things that need to be true. Overusing assertions can lead to duplicate logic that is awkward to maintain. Logic that covers an assumption is good because it forces you to rethink the section of the code. If the code works without the assertion, the assertion is confusing rather than helpful and may hinder modification in the future. Always ask whether the code still works if an assertion fails. If the code does work, remove the assertion. Beware of duplicate code in assertions. Duplicate code smells just as bad in assertion checks as it does anywhere else. Use Extract Method liberally to get rid of the duplication.

		IntroduceAssertion (process)	Example	Here's a simple tale of expense limits. Employees can be given an individual expense limit. If they are assigned a primary project, they can use the expense limit of that primary project. They don't have to have an expense limit or a primary project, but they must have one or the other. This assumption is taken for granted in the code that uses expense limits:
				
				class Employee …
				private static final double NULL-EXPENSE = -1.0;
				private double -expenseLimit = NULL-EXPENSE;
				private Project -primaryProject;
				
				double getExpenseLimit() {
				    // should have either expense limit or a primary project
				    return (_expenseLimit != NULL_EXPENSE) ?
				        _expenseLimit:
				        _primaryProject.getMemberExpenseLimit();
				  }
				This code contains an implicit assumption that the employee has either a project or a personal expense limit. Such an assertion should be clearly stated in the code:
				
				  double getExpenseLimit() {
				    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
				    return (_expenseLimit != NULL_EXPENSE) ?
				        _expenseLimit:
				        _primaryProject.getMemberExpenseLimit();
				  }
				
				This assertion does not change any aspect of the behavior of the program. Either

				way, if the condition is not true, you get a runtime exception: either a null pointer exception in withinLimit or a runtime exception inside Assert.isTrue. In some circumstances the assertion helps find the bug, because it is closer to where things went wrong. Mostly, however, the assertion helps to communicate how the code works and what it assumes.
11	RenameMethod	RenameMethod (process)	Description	If the name of a method does not reveal its purpose, you should change the name of this method. An important part of the code style Fowler is advocating is small methods to factor complex processes. Done badly, this can lead you on a merry dance to find out what all the little methods do. The key to avoiding this merry dance is naming the methods. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method. If you see a badly named method, it is imperative that you change it. Remember your code is for a human first and a computer second. Humans need good names. Take note of when you have spent ages trying to do something that would have been easier if a couple of methods had been better named. Good naming is a skill that requires practice; improving this skill is the key to being a truly skillful programmer. Remark: The same applies to other aspects of the signature. If reordering parameters clarifies matters, do it (see Add Parameter (275) and RemoveParameter [277]).

		RenameMethod (process)	Process	• Check to see whether the method signature is implemented by a superclass or subclass. If it is, perform these steps for each implementation.
				• Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
				• Compile.
				• Change the body of the old method so that it calls the new one.
				    o If you have only a few references, you can reasonable sklp thls step.
				• Compile and test.
				• Find all references to the old method name and change them to refer to the new one. Compile and test after each change.
				• Remove the old method.
				    o If the old method is part of the rnterface and you cannot remove it, leave leave it in place and mark it as deprecated.
				• Complle and test.

		RenameMethod (process)	Example	

		RenameMethod (process)	Example	You have a method to get a person's telephone number:
				
				public String getTelephoneNumber() {
				return ("(" + _0fficeAreaCode + ") " + _officeNumber);
				}
				
				You want to rename the method to get0fficeTelephoneNumber. You begin by creating the new method and copying the body over to the new method. The old method now changes to call the new one:
				
				class Person. . .
				    public String getTelephoneNumber(){
				        return getOfficeTelephoneNumber() ;
				}

				public String getOfficeTelephoneNumber() {
				return ("(" + _officeAreaCode t ") " + _officeNumber);
				}
				
				Now find the callers of the old method, and switch them to call the new one. When you have switched them all, you can remove the old method.
				The procedure is the same if you need to add or remove a parameter.
				If there aren't many callers, you change the callers to call the new method without sing the old method as a delegating method. If your tests throw a problem, you back out and make the changes the slow way.

1.5.2 Learning Space: Code Smell *Long Method*

Learning Space Page				
0	Experience Package - Code Smell Long Method	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on

		(Process)		or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the

				design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring - Overview	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp, or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experience Package - Code Smell Long Method	Repeat Experience (AB)		
8	Code Smell Long Method	Code Smell Long Method (knowledge)	Description	The object programs that live best and longest are those with short methods. Programmers new to OO development often feel that no computation ever takes place, that object programs are endless sequences of delegation. When you have

				lived with such a program for a few years, however, you learn just how valuable all those little methods are. All of the payoffs of indirection-explanation, sharing, and choosing-are supported by little methods. Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand. In fact, it is a kind the "Columbo syndrome". Columbo was the detective who always had "just one more thing." A method starts down a path and, rather than break the flow or identify the helper classes, the author adds one more thing. Code is often easier to write than it is to read, so there's a temptation to write blocks that are too big. You may find other refactorings (those that clean up straight-line code, conditionals, and variable usage) helpful before you even begin splitting up the method.
			CounterExample	It may be that a somewhat longer method is just the best way to express something. (Like almost all smells, the length is a warming sign – not a guarantee – of a problem.)
			Process	Missing (see Fowler p 77)
9	ExtractMethod	ExtractMethod (process)	Description	The refactoring ExtractMethod could be used to break up the method into smaller parts. Look for comments or white space delineating interesting blocks. You want to extract methods that are semantically meaningful, not just introduce a function call every seven lines. Extract Method is one of the most common refactorings. Look at a method that is too long or look at code that needs a comment to understand its purpose. Then turn that fragment of code into its own method whose name explains the purpose of the method. Short, well-named methods should be preferred for several reasons: o First, it increases the chances that other methods can use a method when the method is finely grained. o Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained. Small methods really work only when you have good names, so you need to pay attention to naming. What is the optimal length for a method? In fact, length is not the issue. The key is the semantic distance between the method name and the method body. If extracting improves clarity, do it, even if the name is longer than the code you have extracted.
		ExtractMethod (process)	Process	• Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it). o If the code you want to extract is very simple, such as a simple message or function call, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code. • Copy the extracted code from the source method into the new target method. • Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method. • See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables. • Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method as it stands. You may need to use *SplitTemporaryVariable* (128) and try again. You can eliminate temporary variables with ReplaceTempwithQuery (120) (see the discussion in the examples).

				• Pass into the target method as parameters local-scope variables that are rad from the extracted code. • Compile when you have dealt with all the locally-scoped variables. • Replace the extracted code in the source method with a call to the target method. o If you have moved any temporary variables over to the target method, look to see whether they were declared outside of the extracted code. If so, you can now remove the declaration. • Compile and test.

		ExtractMethod (process)	Example	void printOwing() {
				printBanner();
				    //print details
				    System.out.println ("name: " + _name);
				    System.out.println ("amount " + amount);
				}
				
				It is easy to extract the code that prints the banner. Just cut, paste, and put in a call:
				
				void printOwing() {
				    printBanner();
				    printDetails(getOutstanding());
				}
				
				void printDetails (double outstanding) {
				    System.out.println ("name: " + _name);
				    System.out.println ("amount " + outstanding);
				}

10	ReplaceTempwithQuery	ReplaceTempwithQuery (process)	Description	Temporary variable are used that to hold the result of an expression. This expression should be replace with a method. Replace all references to the temp with the expression. The new method can then be used in other methods. The problem with temps is that they are temporary and local. Because they can be seen only in the context of the method in which they are used, temps tend to encourage longer methods, because that's the only way you can reach the temp. By replacing the temp with a query method, any method In the class can get at the informatlon. That helps a lot In coming up with cleaner code for the class. ReplaceTempwithQuery often is an important step before ExtractMethod. Local variables make it difficult to extract, so replace as many variables as you can with queries. The straightforward cases of this refactoring are those in which temps are assigned only to once and those in which the expression that generates the assignment is free of side effects. Other cases are trickier but possible. You may need to use SplitTemporaryVariable or SeparateQueryfromModifier(279) first to make things easier. If the temp is used to collect a result (such assumming over a loop), you need to copy some logic into the query method.

			Process	Here is the sample case:
				
				• Look for a temporary variable that is assigned to once.
				    o If a temp is set more than once consider SplitTemporaryVariable(128).
				• Declare the temp as final.
				• Compile.
				    o This will ensure that the temp is only assigned to once.
				• Extract the right-hand side of the assignment into a method.
				    o Initially mark the method as private. You may find more use for it later; but you can easily relax the protection later.
				    o Ensure the extracted method is free of side effects, that is, it does not modify any object. If it is not free of side effects, use SeparateQueryfromModifier (279).

				• Compile and test. • Use ReplaceTempwithQuery on the temp

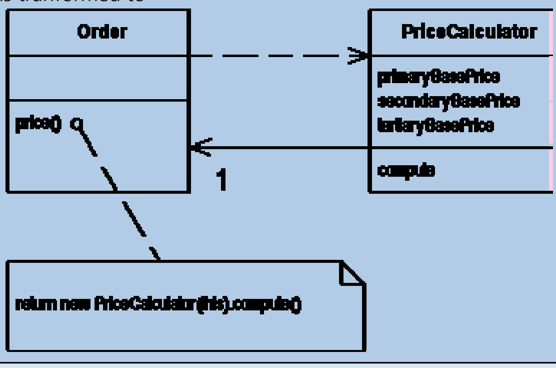
			Example	double basePrice = _quantity * _itemPrice;
				if (basePrice > 1000)
				return basePrice * 0.95;
				else
				return basePrice * 0.98;
				
				Is transformed to:
				
				if (basePrice() > 1000)
				return basePrice() * 0.95;
				else
				return basePrice() * 0.98;
				...
				double basePrice() {
				return _quantity * _itemPrice;
				}

			Example	double getPrice() {
				int basePrice = _quantity * _itemprice;
				double discountFactor;
				if (baseprice > 1000) discountFactor = 0.95;
				else discountfactor = 0.98;
				return basePrice * discountFactor;
				}
				
				Don’t replace both temps, replace one at a time. Although it's pretty clear in this case, you can test that they are assigned only to once by declaring them as final:
				
				double getPrice() {
				final int basePrice = _quantity * _itemPrice;
				final double discountFactor;
				if (baseprice > 1000)
				discountFactor = 0.95;
				else discountFactor = 0.98;
				return basePrice * discountFactor;
				}
				
				Compiling will then alert me to any problems. Do this first, because if there is a problem, you shouldn't be doing this refactoring. Replace the temps one at a time. First, extract the right-hand side of the assignment:
				
				double getPrice() {
				final int basePrice = basePrice();
				final double discountFactor;
				if (basePrice > 1000)
				discountFactor = 0.95;
				else discountFactor = 0.98;
				return basePrice * discountFactor;
				}
				
				double getPrice() {
				final int basePrice = basePrice();
				final double discountFactor;
				if (basePrice > 1000)
				discountfactor = 0.95;
				else discountFactor = 0.98;
				return basePrice * discountfactor;
				}
				
				Compile and test, then begin with ReplaceTempwithQuery. First replace the first

				reference to the temp:
				
				double getPrice() {
				final int basePrice = basePrice();
				final double discountFactor;
				if (basePrice() > 1000)
				discountfactor = 0.95;
				else discountFactor = 0.98;
				return basePrice * discountfactor;
				}
				
				Compile and test and do the next (sounds like a caller at a line dance). Because it's the last, remove the temp declaration:
				
				double getPrice() {
				final double discountFactor;
				if (basePrice() > 1000)
				discountfactor = 0.95;
				else discountFactor = 0.98;
				return basePrice() * discountfactor;
				}
				
				With this done, extract discountFactor in a similar way:
				
				double getPrice() {
				final double discountFactor = discountfactor();
				return basePrice() * discountfactor;
				
				private double discountfactor() {
				if (basePrice() > 1000) return 0.95;
				else return 0.98;
				}
				
				See how it would have been difficult to extract discountFactor if you had not replaced basePrice with a query. The getPrice method ends up as follows:
				
				double getPrice() {
				return basePrice() * discountfactor();
				}

			Counterexample	Paul Haahr pointed out that you can't do this refactoring if the code in between the the assignment to the temp and the use of the temp changes the value of the expression that calculates the temp. In these cases the code is using the temp to snapshot the value of the temp when it's assigned. The name of the temp should convey this fact (and you should change the name if it doesn't).
11	ReplaceMethodwithMethodObject	ReplaceMethodwithMethodObject (process)	Description	You have a long method that uses local variables in such a way that you cannot apply ExtractMethod. Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object. By extracting pieces out of a large method, you make things much more comprehensible. The difficulty in decomposing a method lies in local variables. If they are rampant (Germ. üppig), decomposition can be difficult. Using ReplaceTempwithQuery helps to reduce this burden, but occasionally you may find you cannot break down a method that needs breaking. In this case you reach deep into the tool bag and get out your *method object* [Beck]. Applying ReplaceMethodwithMethodObject turns all these local variables into fields on the method object. You can then use ExtractMethod on this new object to create additional methods that break down the original method.

			Process	o Create a new class, name it after the method. o Give the new class a final field for the object that hosted the original method (the source object) and a field for each temporary variable and each parameter in the method o Give the new class a constructor that takes the source object and each parameter. o Give the new class a method named "compute." o Copy the body of the original method Into compute. Use the source object field for any mvocatlons of methods on the original object. o Replace the old method with one that creates the new object and calls compute. Now comes the fun part. Because all the local variables are now fields, you can freely decompose the method without having to pass any parameters.

			Example	class Order...
				double price() {
				double primaryBasePrice;
				double secondaryBasePrice;
				double tertiaryBasePrice;
				// long computation;
				...
				}
				
				Is tranformed to
				

			Example	A proper example of this requires a long chapter, so this refactoring is showed for a method that doesn't need it. (The logic of this method is not important to understand!)
				
				Class Account
				int gamma (int inputVal, int quantity, int yearToDate) {
				int importantValue1 = (inputVal * quantity) + delta();
				int importantValue2 = (inputVal * yearToDate) + 100;
				if ((yearToDate - importantValue1) > 100)
				importantValue2 -= 20;
				int importantValue3 = importantValue2 * 7;
				// and so on.
				return importantValue3 - 2 * importantValue1;
				}
				
				To turn this into a method object, begin by declaring a new class. Provide a final field for the original object and a field for each parameter and temporary variable in the method.
				
				class Gamma. . .
				private final Account _account;
				private int inputVal ;
				private int quantity;
				private int yearToDate;
				private int importantValue1;
				private int importantValue2;
				private int importantValue3;
				
				Should should usually use the underscore prefix convention for marking fields. But to keep small steps leave the names as they are for the moment.
				Add a constructor:
				
				Gamma (Account source, int inputValArg, int quantityArg, int yearToDateArg) {
				_account = source;
				inputVal = inputValArg;
				quantity = quantityArg;
				yearToDate = yearToDateArg;

				}
				
				Now you can move the original method over you need to modify any calls of features of account to use the _account field
				int compute () {
				importantValue1 = (inputVal Quantity) + _account.delta();
				importantValue2 = (inputVal * yearToDate) + 100;
				if ((yearToDate – importantvalue1) > 100)
				importantValue2 -= 20;
				int importantValue3 = importantValue2 *7;
				// and so on.
				return importantValue3 - 2 * importantValue1;
				}
				
				You then modify the old method to delegate to the method object:
				
				int gamma (int inputVal, int quantity, int yearToDate) {
				return new Gamma(this, inputVal, quantity, yearToDate) .compute();
				}
				
				That's the essential refactoring. The benefit is that you can now easily use ExtractMethod on the compute method without ever worrying about the argument's passing:
				
				int compute () {
				importantValue1 = (inputVal "quantity) + _account .delta() ;
				importantValue2 = (inputVal * yearToDate) + 100;
				importantThing() ;
				int importantValue3 = importantValue2 * 7;
				// and so on.
				return importantValue3 - 2 * importantValue1;
				}
				
				void importantThing() {
				if ((yearToDate – importantValue1) > 100)
				importantValue2 -= 20;
				}

12	DecomposeConditional	DecomposeConditional (process)	Description	You have a complicated conditional (if-then-else) statement. Extract methods from the condition, then part, and else parts. One of the most common areas of complexity in a program lies in complex conditional logic. As you write code to test conditions and to do various things depending on various conditions, you quickly end up with a pretty long method. Length of a method is in itself a factor that makes it harder to read, but conditions increase the difficulty. The problem usually lies in the fact that the code, both in the condition checks and in the actions, tells you what happens but can easily make it difficult to understand why it happens. As with any large block of code, you can make your intention clearer by decomposing it and replacing chunks of code with a method call named after the intention of that block of code. With conditions you can receive further benefit by doing this for the conditional part and each of the alternatives. This way you highlight the condition and make it clearly what you are branching on. You also highlight the reason for the branching.

			Example p239	if (date.before (SUMMER_START) || date.after(SUMMER_END))
				charge = quantity * _winterRate + _winterServiceCharge;
				else charge = quantity * _summerRate;
				
				is transformed to:
				
				if (notSummer(date))
				charge = winterCharge(quantity);

				else charge = summerCharge (quantity);
				
				private boolean notSummer(Date date) {
				return date.before (SUMMER_START) || date.after(SUMMER_END) ;
				}
				
				private double summerCharge(int quantity) {
				return quantity * _summerRate;
				}
				
				private double winterCharge(int quantity) {
				return quantity * _winterRate + _winterServiceCharge;
				}

			Process	o Extract the condition into its own method. o Extract the then part and the else part into their own methods.

1.5.3 Learning Space: Code Smell *Type Embedded in Name*

Learning Space Page				
0	Experience Package - Code Smell Type Embedded in Name	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all

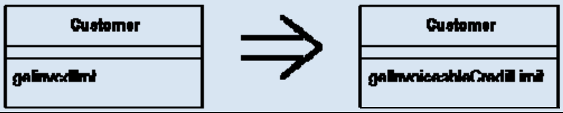
				considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introducti on	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactori ng - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this

				case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring - Overview	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experience Package - Code Smell	Repeat Experience (AB)		

	Type Embedded in Name			
8	Code Smell Type Embedded in Name	Code Smell Type Embedded in Name	Description	The following problems are related to the code smell Type Embedded in Name. o Method names are compound words, consisting of a word plus the type of the argument(s).For example, a method addCourse(Course c). o Names are in Hungarian notation, where the type of an object is encoded into the name; e.g., icount as an integer member variable. o Variable names reflect their type rather than their purpose or role. Explanation of the problems: The type may be added in the name of communication. For example, schedule.addCourse(course) might be regarded as more readable than schedule.add(course). The embedded type name represents duplication: Both the argument and the name mention the same type. The embedded name can create unnecessary troubles later on. For example, suppose we introduce a parent class for Course to cover both courses and series or courses. Now, all the places that refer to addcourse() have a name that's not quite appropriate. We either change the name at every call site or live with a poor name. Finally, by naming things for the operation alone, we make it easier to see duplication and recognize new abstractions. Hungarian notation is often introduced as part of a coding standard. In pointer-based languages (like C), it was useful to know that **ppc is in fact a character, but in object-oriented languages it overcouples a name to its type. Some programmers or teams use a convention where a prefix indicates that something is a member variable (_count or in-count) or that something is a constant (ALL-UPPER-CASE). Again, this adds friction as we change whether something is a local variable, a member, and so on. Aren't there times when we need to know which is which? Sure-and if it's not easy to tell, then it may be a sign that a class is too big. The solution for these type of code smells is to apply the refactoring *RenameMethod*.
			Counter Example	Rarely, you might have a class that wants to do the same sort of operation to two different but related types. For example, we might have a Graph class with addPoint() and addlink() methods. If the abstract behavior for the two cases is tlie same, it may be appropriate to overload the method name (add()). Sometimes you're using a coding standard that uses typographical conventions to distinguish different classes of variables. You may then value the team's readability of code above the flexibility of untyped names, and follow those conventions.
11	Rename Method	RenameMethod (process)	Description	The name of a method does not reveal its purpose. Change the name of the method. An important part of the code style Martin Fowler advocating is small methods to factor complex processes. Done badly, this can lead you on a merry dance to find out what all the little methods do. The key to avoiding this merry dance is naming the methods. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method. If you see a badly named method, it is imperative that you change it. Remember your code is for a human first and a computer second. Humans need good names. Take note of when you have spent ages trying to do something that would have been easier if a couple of methods had been better named. Good naming is a skill that requires practice; improving this skill is the key to being a truly skillful programmer. Remark: The same applies to other aspects of the signature.

				If reordering parameters clarifies matters, do it (see Add Parameter (275) and RemoveParameter [277]).

		RenameMethod (process)	Process	• Check to see whether the method signature is implemented by a superclass or subclass. If it is, perform these steps for each implementation.
				• Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
				• Compile.
				• Change the body of the old method so that it calls the new one.
				    o If you have only a few references, you can reasonable sklp thls step.
				• Compile and test.
				• Find all references to the old method name and change them to refer to the new one. Compile and test after each change.
				• Remove the old method.
				    o If the old method is part of the rnterface and you cannot remove it, leave leave it in place and mark it as deprecated.
				• Complle and test.

		RenameMethod (process)	Example	

		RenameMethod (process)	Example	You have a method to get a person's telephone number:
				
				public String getTelephoneNumber() {
				return ("(" + _0fficeAreaCode + ") " + _officeNumber);
				}
				
				You want to rename the method to get0fficeTelephoneNumber. You begin by creating the new method and copying the body over to the new method. The old method now changes to call the new one:
				
				class Person. . .
				public String getTelephoneNumber(){
				return getOfficeTelephoneNumber() ;
				}
				
				public String getOfficeTelephoneNumber() {
				return ("(" + _officeAreaCode + ") " + _officeNumber);
				}
				
				Now you find the callers of the old method, and switch them to call the new one. When you have switched them all, you can remove the old method.
				The procedure is the same if you need to add or remove a parameter.
				If there aren't many callers, you change the callers to call the new method without sing the old method as a delegating method. If your tests throw a problem, you back out and make the changes the slow way.

1.5.4 Learning Space: Code Smell *Uncommunicative Name*

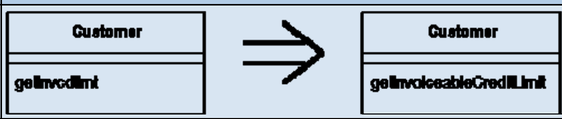
0	Experience Package - Code Smell Uncommunicative Name	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the

				term increased after it was featured in the book “Refactoring. Improving the Design of Existing Code” by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers’ incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software’s requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactori	Refactoring	Overview	Classification of Martin Fowler:

	ng - Overview	(Process)		1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactori ng - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experienc e Package - Code Smell Uncomm unicative Name	Repeat Experience (AB)		
8	Code Smell Uncomm unicative Name	Code Smell Uncommunic ative Name	Description	A name doesn't communicate its intent of a method, variable, classes, etc. well enough - One- or two-character names - Names with vowels omitted. - Numbered variables (e.g., panel, pane2, and so on) - Odd abbreviations - Misleading names When you first implement something, you have to name things somehow. You give the best name you can think of at the time and move on. Later, you may have an insight that lets you pick a better name.
			Counter Example	Some teams use i/j/k for loop indexes or c for characters; these aren't too confusing if the scope is reasonably short. Similarly, you may occassionally find that numbered variables communicates better.
9	Rename Method	RenameMeth od (process)	Description	If the name of a method does not reveal its purpose, you should change the name of this method.

				An important part of the code style Fowler is advocating is small methods to factor complex processes. Done badly, this can lead you on a merry dance to find out what all the little methods do. The key to avoiding this merry dance is naming the methods. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method. If you see a badly named method, it is imperative that you change it. Remember your code is for a human first and a computer second. Humans need good names. Take note of when you have spent ages trying to do something that would have been easier if a couple of methods had been better named. Good naming is a skill that requires practice; improving this skill is the key to being a truly skillful programmer. Remark: The same applies to other aspects of the signature, i.e., variables, class names, etc. If reordering parameters clarifies matters, do it (see Add Parameter (275) and RemoveParameter [277]).

		RenameMethod (process)	Process	• Check to see whether the method signature is implemented by a superclass or subclass. If it is, perform these steps for each implementation.
				• Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
				• Compile.
				• Change the body of the old method so that it calls the new one.
				    o If you have only a few references, you can reasonable sklp thls step.
				• Compile and test.
				• Find all references to the old method name and change them to refer to the new one. Compile and test after each change.
				• Remove the old method.
				    o If the old method is part of the rnterface and you cannot remove it, leave leave it in place and mark it as deprecated.
				• Compile and test.

		RenameMethod (process)	Example	

		RenameMethod (process)	Example	You have a method to get a person's telephone number:
				
				public String getTelephoneNumber() {
				return ("(" + _0fficeAreaCode + ") " + _officeNumber);
				}
				
				You want to rename the method to get0fficeTelephoneNumber. You begin by creating the new method and copying the body over to the new method. The old method now changes to call the new one:
				
				class Person. . .
				public String getTelephoneNumber(){
				return getOfficeTelephoneNumber() ;
				}
				
				public String getOfficeTelephoneNumber() {
				return ("(" + _officeAreaCode + ") " + _officeNumber);
				}
				
				Now you find the callers of the old method, and switch them to call the new one. When you have switched them all, you can remove the old method.

				The procedure is the same if you need to add or remove a parameter. If there aren't many callers, you change the callers to call the new method without sing the old method as a delegating method. If your tests throw a problem, you back out and make the changes the slow way.

1.5.5 Learning Space: Code Smell *Long Parameter List*

Learning Space Page				
0	Experience Package - Code Smell Uncommunicative Name	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.

		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells -	Code Smells within	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings

	Overview	Classes		
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring - Overview	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experience Package - Code Smell Long Parameter List	Repeat Experience (AB)		
8	Code Smell Long Parameter List	Code Smell Long Parameter List	Description	Long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because you are forever changing them as you need more data. The cause for this is that in the early programming days we were taught to pass in as parameters everything needed by a routine. This was understandable because the alternative was global data, and global data is evil and usually painful. Objects change this situation because if you don't have something you need, you can always ask another object to get it for you. Thus with objects you don't pass in everything the method needs; instead you pass enough so that the method can get to everything it needs. A lot of what a method needs is available on the method's host

				class. In object-oriented programs parameter lists tend to be much smaller than in traditional programs. Hence most problems can be removed by passing objects because you are much more likely to make only a couple of requests to get at a new piece of data. This smell is easy to identify. However, be aware they are not necessarily the easiest to fix.
			Counter Example	There is one important exception when the refactoring should not be applied. This is when you explicitly do not want to create a dependency from the called object to the larger object. In those cases unpacking data and sending it along as parameters is seasonable, but pay attention to the possible upcoming problems. If the parameter list is too long or changes too often, you need to rethink your dependency structure. Or, the parameters have no meaningful grouping – they don't go together.
10	Replace Parameterwith Method	ReplaceParameterwithMethod (process)	Description	Long parameter lists are difficult to understand, and we should reduce them as much as possible. For example an object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method. Remove the parameter and let the receiver invoke the method! Hence,use ReplaceParameterwithMethod when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter. So, look to see whether the receiving method can make the same calculation. If an object is calling a method on itself, and the calculation for the parameter does not reference any of the parameters of the calling method, you should be able to remove the parameter by turning the calculation into its own method. This is also true if you are calling a method on a different object that has a reference to the calling object. You can't remove the parameter if the calculation relies on a parameter of the calling method, because that parameter may change with each call (unless, of course, that parameter can be replaced with a method). You also can't remove the parameter if the receiver does not have a reference to the sender, and you don't want to give it one. In some cases the parameter may be there for a future parameterization of the method. In this case you should still remove it.
			Counter Example	You should make an exception to this rule only when the resulting change in the interface would have painful consequences around the whole program, such as a long build or changing of a lot of embedded code. If this worries you, look into how painful such a change would really be. You should also look to see whether you can reduce the dependencies that cause the change to be so painful. Stable interfaces are good, but freezing a poor interface is a problem.
			Process	- If necessary, extract the calculation of the parameter into a method. - Replace references to the parameter in method bodies with references to the method. - Compile and test after each replacement. - Use RemoveParameter on the parameter.

			Example	int basePrice = _quantity * _itemPrice;
				discountLevel = getDiscountLevel();
				double finalPrice = discountedPrice (basePrice, discountLevel);
				
				Is transformed to:
				
				int basePrice = _quantity * _itemPrice;
				double finalPrice = discountedPrice (basePrice);

			Example	public double getPrice0 {
				    int basePrice = _quantity * _itemPrice;
				    int discountlevel;
				    if (_quantity > 100) discountLevel = 2;
				    else discountLevel = 1;

```
    double finalPrice = discountedPrice (basePrice, discountLevel) ;
    return finalPrice;
}

private double discountedPrice (int basePrice, int discountLevel) {
if (discountLevel == 2) return basePrice * 0.1;
else return basePrice * 0.05;
}
```

You can begin by extracting the calculation of the discount level:

```
public double getPrice() {
    int basePrice = _quantity * _itemprice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice (basePrice, discountLevel) ;
    return finalPrice;
}
private int getDiscountLevel() {
    if (_quantity > 100) return 2;
    else return 1;
}
```

You then replace references to the parameter in discountedPrice:

```
private double discountedPrice (int basePrice, int discountLevel) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Then you can use the refactoring RemoveParameter:

```
public double getPrice() {
    int basePrice = _quantity * _itemprice;
    int discountLevel = getDiscountLevel() ;
    double finalPrice = discountedPrice (basePrice) ;
    return finalPrice;
}
private double discountedPrice (int basePrice) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

You can now remove the temp:

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double finalPrice = discountedPrice (basePrice) ;
    return finalPrice;
}
```

Then it's time to remove the other parameter and its temp.

```
public double getPrice() {
    return discountedPrice();
}

private double discountedPrice() {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}

private double getBasePrice() {
```

				return _quantity * _itemPrice;
				}
				
				so you might as well use the refactoring InlineMethod on discountedPrice:
				
				private double getPrice () {
				if (getDiscountLevel() == 2) return getBasePrice() *0.1;
				else return getBasePrice() * 0.05;
				}

11	IntroduceParameterObject	IntroduceParameterObject (process)	Description	If you have several data items with no logical object, use IntroduceParameterObject to group the parameters. So, if you have a group of parameters that naturally go together, replace them with an object. Often you see a particular group of parameters that tend to be passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced with an object that carries all of this data. It is worthwhile to turn these parameter into objects just to group the data together. This refactoring is useful because it reduces the size of the parameter lists, and long parameter lists are hard to understand. The defined accessors on the new object also make the code more consistent, which again makes it easier to understand and modify. You get a deeper benefit, however, because once you have clumped together the parameters, you soon see behavior that you can also move into the new class. Often the bodies of the methods have common manipulations of the parameter values. By moving this behavior into the new object, you can remove a lot of duplicated code.
			Process	• Create a new class to represent the group of parameters you are replacing. • Compile • Use AddParameter for the new data clump. Use a null for this parameter in all the callers. - If you have many callers, you can retain the old signature and let it call the new method. Apply the refactoring or the old method first. You can then move the callers over one by one and remove the old method when you're done. • For each parameter in the data clump, remove the parameter from the signature.Modify the callers and method body to use the parameter object for that value. • Compile and test after you remove each parameter. • When you have removed the parameters, look for behavior that you can move into the parameter object with MoveMethod. o This may be a whole method or part of a method. If it is part of a method, use ExtractMethod first and then move the new method over

			Example	The example begins with an account and entries. The entries are simple data holders.
				
				class Entry ..
				Entry (double value, Date chargeDate) {
				_value = value;
				_chargeDate = chargeDate;
				}
				Date getDate(){
				return _chargeDate;
				}
				double getvalue() {
				return -value;
				}
				private Date _chargeDate;
				private double _value;
				
				The focus is on the account, which holds a collection of entries and has a method for determining the flow of the account between two dates:

```
class Account ..
        double getFlowBetween (Date start, Date end) {
                double result = 0;
                Enumeration e = _entries.elements();
                        while (e.hasMoreElements()) {
                        Entry each = (Entry) e.nextElement();
                        i f (each.getDate() .equals(start) I I
                        each.getDate() .equals(end) I I (each.getDate().after(start)
                        && each.getDate().before(end)))
                        {
                        result+= each.getValue();
                        }
                }
        return result ;
}
private Vector _entries = new Vector();

client code ..
        double flow = anAccount.getFlowBetween(startDate, endDate);
```

You should always try to use ranges instead of pairs of values that show a range. The first step is to declare a simple data holder for the range:

```
class DateRange {
        DateRange (Date start, Date end) {
                _start = start;
                _end = end;
        }
        Date getstart() {
                return _start;
        }
        Date getEnd() {
                return _end;
        }
        private final Date _start;
        private final Date _end;
}
```

You have made the date range class immutable; that is, all the values for the date range are final and set in the constructor, hence there are no methods for modifying the values. This is a wise move to avoid aliasing bugs. Because Java has pass-by-value parameters, making the class immutable mimics the way Java's parameters work, so this is the right assumption for this refactoring.
Next you add the date range into the parameter list for the getFlowBetween method:

```
class Account ..
        double getFlowBetween (Date start, Date end, DateRange range) {
                double result = 0;
                Enumeration e = _entries.elements() ;
                while (e. hasMoreElements()) {
                        Entry each = (Entry) e.nextElement();
                        if (each.getDate().equals(start) I I
                        each.getDate().equal(end) | | (each.getDate().after(start)
                        && each.getDate().before(end)))
                        {
                        result += each.getValue();
                        }
                }
        return result;
}
client code.
        double flow = anAccount.getFlowBetween(startDate, endDate, null);
```

At this point you only need to compile, because you haven't altered any behavior yet. The next step is to remove one of the parameters and use the new object instead. To do this you delete the start parameter and modify the method and its callers to use the new object instead:

```
class Account..
        double getFlowBetween (Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
                while (e. hasMoreElements()) {
                        Entry each = (Entry) e.nextElement();
                                if (each.getDate().equals(range.getStart()) I I
                                each.getDate().equals(end) I I
                                (each.getDate().after(range.getStart()) &&
                                each.getDate().before(end)))
                        {
                        result += each.getValue();
                        }
                }
        return result;
}
client code..
        double flow = anAccount.getFlowBetween(endDate, new DateRange
        (startDate, null));
```

You then remove the end date:

```
class Account..
        double getFlowBetween (DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
                while (e. hasMoreElements()) {
                        Entry each = (Entry) e.nextElement();
                                if (each.getDate().equals(range.getStart()) I I
                                each.getDate().equals(range.getEnd()) I I
                                (each.getDate().after(range.getStart()) &&
                                each.getDate().before(range.getEnd())))
                        {
                        result += each.getValue();
                        }
                }
        return result;
}
client code..
        double flow = anAccount.getFlowBetween(new DateRange (startDate,
        endDate));
```

You have introduced the parameter object; however, you can get more value from this refactoring by moving behavior from other methods to the new object. In this case you can take the code in the condition and use ExtractMethod and Move Method to get

```
class Account..
        double getFlowBetween (DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
                while (e. hasMoreElements()) {
                        Entry each = (Entry) e.nextElement();
                        if (range.includes(each.getDate())) {
                                result += each.getValue();
                        }
                }
```

				return result;
				}
				
				class DateRange..
				boolean includes (Date arg) {
				return (arg.equals(_start) ||
				arg.equals(_end) ||
				(arg.after(_start) && arg.before(_end))) ;
				}
				
				You usually should do simple extracts and moves such as this in one step. If you run into a bug, you can back out and take the two smaller steps.

			Counter Example	Ralph Johnson pointed out to me that a common case isn't clear in the Refactoring book. This case is when you have a bunch of methods that call each other, all of which have a clump of parameters that need this refactoring. In this case you don't want to apply Introduce Parameter Object because it would lead to lots of new objects when you only want to have one object that's passed around.
12	PreserveWholeObject	PreserveWholeObject (process)	Description	This type of situation arises when an object passes several data values from a single object as parameters in a method call. The problem with this is that if the called object needs new data values later, you have to find and change all the calls to this method. You can avoid this by passing in the whole object from which the data came. The called object then can ask for whatever it wants from the whole object. In addition to making the parameter list more robust to changes, PreserveWholeObject often makes the code more readable. Long parameter lists can be hard to work with because both caller and callee have to remember which values were there. They also encourage duplicate code because the called object can't take advantage of any other methods on the whole object to calculate intermediate values. That a called method uses lots of values from another object is a signal that the called method should really be defined on the object from which the values come. When you are considering PreserveWholeObject, consider the refactoring MoveMethod as an alternative. You may not already have the whole object defined. In this case you need the refactoring IntroduceParameterObject. A common case is that a calling object passes several of its own data values as parameters. In this case you can make the call and pass in this instead of these values, if you have the appropriate getting methods and you don't mind the dependency.
			Process	- Create a new pxameter for the whole oblect from which the data comes. - Compile and test. - Determine which parameters should be obtained from the whole object. - Take one parameter and replace references to it within the method body by invoking an appropriate method on the whole object parameter. - Delete the parameter. - Compile and test. - Repeat for each parameter that can be got from the whole object. - Remove the code in the calling method that obtains the deleted parameters. o Unless, of course, the code is using these parameters somewhere else. - Compile and test.

			Example	int low = daysTempRange().getLow();
				int high = daysTempRange().getHigh();
				withinPlan = plan.withinRange(low, high);
				
				Is transformed to:
				
				withinPlan = plan.withinRange(daysTempRange());

			Counter Example	Passing objects to methods has also a down side. When you pass in values, the called object has a dependency on the values, but there isn't any dependency to the object from which the values were extracted. Passing in the required object causes a dependency between the required object and the called object. If this is going to mess

				up our dependency structure, don't use PreserueWholeObject.
			Counter Example	A reason not to use PreserveWholeObject is that when a calling object need only one value from the required object, it is better to pass in the value than to pass in the whole object. You don't subscribe to that view. One value and one object amount to the same thing when you pass them in, at least for clarity's sake (there may be a performance cost with pass by value parameters). The driving force is the dependency issue.

			Example	Consider a room object that records high and low temperatures during a day. It needs to compare its range in a predefined plan:
				
				class Room …
				    boolean withinPlan(HeatingPlan plan) {
				        i nt low = daysTempRange() .getlow();
				        i nt high = daysTempRang() .getHigh();
				        return plan.withinRange(low, high);
				    }
				
				class HeatingPlan..
				    boolean withinRange (int low, int high) {
				        return (low >= _range.getLow() && high <= _range.getHigh());
				    }
				    private TempRange _range;
				
				Rather than unpack the range informtation when pass you it, you can pass the whole range object. In this simple case you can do this in one step. When more parameters are involved, you can do it in smaller steps. First, you add the whole object to the parameter list.
				
				class HeatinqPlan..
				    boolean withinRange (TempRange roomRange, int low, int high) {
				        return (low >= -range.getLow() && high <= _range.getHigh());
				    }
				
				class Room. . .
				    boolean withinPlan(HeatingPlan plan) {
				        i nt low = daysTenlpRange() .getlow();
				        i nt high = daysTempRange().getHigh();
				        return plan.withinRange(daysTempRange() , low, high);
				    }
				
				Then you use the mehthod on the whole object instead of one of the parameters:
				
				class HeatingPlan ..
				    boolean withinRange (TempRange roomRange, int high) {
				        return (roomRange.getLow() >= _range.getLow() && high <=
				        _range.getHigh()) ;
				}
				class Room ..
				    boolean withinPlan(HeatingPlan plan) {
				        int low = daysTempRange() .getlow();
				        i nt high = daysTempRange() .getHigh() ;
				        return plan.withinRange(daysTempRange(), high);
				}
				
				You continue until you habe changes all you need:
				
				class HeatingPlan..
				    boolean withinRange (TempRange roomRange) {
				        return (roomRange.getLow() >= _range.getLow() &&
				        roomRange.getHigh() <= _range.getHigh()) ;
				}
				class Room..

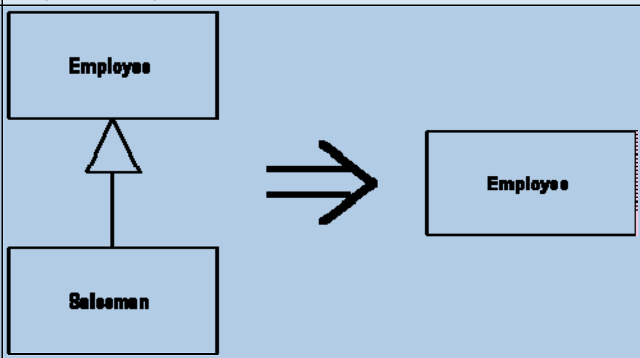
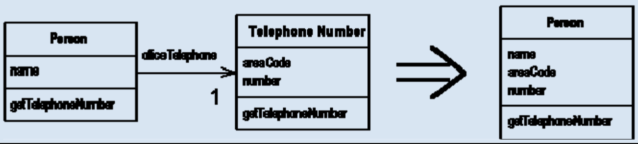
				boolean withinPlan(HeatingPlan plan) {
				int low = daysTempRange().getLow();
				int high = daysTempRange() .getHigh() ;
				return plan.withinRange(daysTempRange()) ;
				}
				
				Now you don't need the temps anymore:
				class Room..
				boolean withinPlan(HeatingPlan plan) {
				int low = daysTempRange().getLow();
				int high = daysTempRange() .getHigh() ;
				return plan.withinRange(daysTempRange()) ;
				}
				
				Using whole objects this way soon leads you to realize that you can usefully move behavior into the whole object to make it easier to work with.
				
				class HeatingPlan..
				boolean withinRange (TempRange roomRange) {
				return (_range.includes(roomRange)) ;
				}
				class TempRange . . .
				boolean includes (TempRange arg) {
				return arg.getLow() >= this.getLow() && arg.getHigh() <= this.getHigh();
				}

1.5.6 Learning Space: Code Smell *Lazy Class*

Learnin Space age				
0	Experience Package - Code Smell Lazy Class	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of

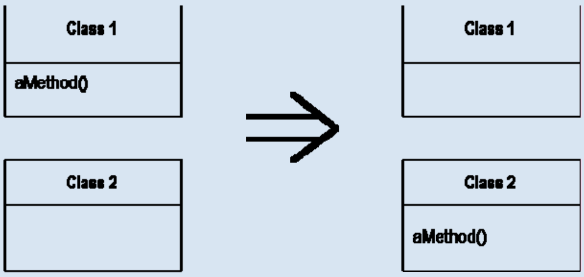
				reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis

				• Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring - Overview	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level. Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term

				• To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experienc e Package - Code Smell Lazy Class	Repeat Experience (AB)		
8	Code Smell Lazy Class	Code Smell Lazy Class	Description	Data class is a code smell between classes. Each class you create costs money to maintain and understand. A class that isn't doing enough to pay for itself should be eliminated. Often this might be a class that is used to pay its way but has been downsized with refactoring. Or it might be a class that was added because of changes that were planned but not made. Either way, you let the class die with dignity. If you have subclasses that aren't doing enough, try to use *CollapseHierarchy*. Nearly useless components should be subjected to InlineClass.
			Counter Example	Sometimes a lazy class is present to communicate intent. You may have to balance communication cersion simplicity.
9	Collapse Hierarchy	CollapseHiera rchy (process)	Description	If you have been working for a while with a class hierarchy, it can easily become too tangled for its own good. Refactoring the hierarchy often involves pushing methods and fields up and down the hierarchy. After you've done this you can well find you have a subclass that isn't adding any value, so you need to merge the classes together. So, if a superclass and subclass are not very different, the refactoring CollapseHierarchy merges them together.
			Example	
			Process	• Choose which class is going to be removed: the superclass or the subclasses. • Use the refactorings PullUpField and PullUpMethod or PushDownMethod and PushDownField to move all the behavior and data of the removed class to the class with which it is being merged. • Compile and test with each move. • Adjust references to the class that will be removed to use the merged class. This will affect variable declarations, parameter types, and constructors. • Remove the empty class. • Compile and test.
10	InlineClas s	InlineClass (process)	Description	The refactoring InlineClass move all features of a class that isn’t doing very much into another class and delete it afterwards.
			Example	
			Process	• Declare the public protocol of the source class onto the absorbing class.

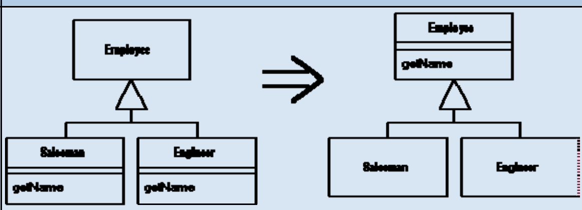
				Delegate all these methods to the source class o If a separate interface makes sense for the source class methods, use ExtractInterface before inlining • Change all references from the source class to the absorbing class. o Declare the source class private to remove out-of-package references. Also change the name of the source class so the compiler catches any dangling references to the source class. • Compile and test • Use MoveMethod and MoveField to move features from the source class to the absorbing class until there is nothing left. • Died!

			Example	We start with separate classes:
				
				class Person..
				public String getName() {
				return _name;
				}
				public String getTelephoneNumber(){
				return _officeTelephone.getTelephoneNumber() ;
				}
				TelephoneNumber getOfficeTelephone() {
				return _officeTelephone;
				}
				private String _name;
				private TelephoneNumber _0fficeTelephone = new TelephoneNumber();
				
				class TelephoneNumber..
				public String getTelephoneNumber() {
				return ("(" + _areacode + ") " + _number);
				}
				String getAreaCode() {
				return _areacode;
				}
				void setAreaCode(Stri ng arg) {
				_areacode = arg;
				}
				String getNumber() {
				return _number;
				}
				void setNumber(5tring arg) {
				_number = arg;
				}
				private string _number;
				private string _areacode;
				
				You begin by declaring all the visible methods on telephone number on person:
				
				class Person..
				String getAreaCode() {
				return _officeTelephone.getAreaCode() ;
				}
				void setAreaCode(String arg) {
				_officeTelephone.setAreaCode(arg1;
				}
				String getNumber() {
				return _officeTelephone.getNumber();
				}
				void setNumber(String arg) {
				_officeTelephone.setNumber(arg) ;
				}

				Now you find clients of telephone number and switch them to use the person's interface. So Person martin = new Person(); martin.getOfficeTelephone().setAreaCode ("781"); becomes Person martin = new Person(); martin.setAreaCode ("781"); Now I can use MoveMethod and MoveField until the telephone class is no more.
10	MoveMethod	MoveMethod (process)	Description	When a method is, or will be, using or used by more features of another class than the class on which it is defined, then you should create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether. Moving methods is the bread and butter of refactoring. You should move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By moving methods around, you can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities. You should look through the methods on a class to find a method that seems to reference another object more than the object it lives on. It's not always an easy decision to make. If I am not sure whether to move a method, you should go on to look at other methods. Moving other methods often makes the decision easier. Sometimes the decision still is hard to make.
			Example	

			Process	• Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.
				  o If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a set of methods than to move them one at a time.
				• Check the sub- and superclasses of the source class for other declarations of the method.
				  o If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target.
				• Declare the method in the target class.
				  o You may choose to use a different name, one that makes more sense in the target class.
				• Copy the code from the source method to the target. Adjust the method to make it work in its new home.
				• If the method uses its source, you need to determine how to reference the source object from the target method. If there is no mechanism in the target class, pass the source object reference to the new method as a parameter,
				• If the method includes exception handlers handlers, decide which class should logically handle the exception. If the source class should be responslble, leave the handlers behind.

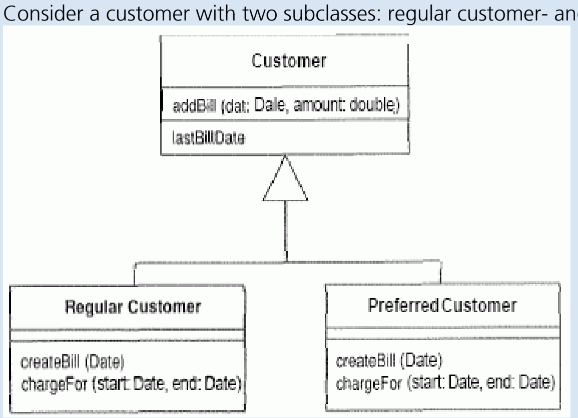
				• Compile the target class.
				• Determine how to reference the correct target object from the source.
				    o There may be an existing field or method that wdl give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also make it temporarily until you have refactored enough to remove it.
				• Turn the source method into a delegating method.
				• Compile and test.
				• Decide whether to remove the source method or retain it as a delegating method.
				• Leaving the source as a delegating method is easier if you have many references.
				• If you remove the source method, replace all the eferences with references to the target method.
				    o You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace.
				• Compile and test.

			Example	• F p.144
12	PullUpMethod (322)	PullUpMethod (process)	Description	This refactoring is applied when you have methods with identical results on subclasses. Then move them to the superclass. Eliminating duplicate behaviour is important. Although two duplicate methods work fine as they are, they are nothing more than a breeding ground for bugs in the future. Whenever there is dupllcation, you face the risk that an alteration to one will not be made to the other. Usually it is difficult to find the duplicates. The easiest case of using PullUpMethod occurs when the methods have the same body, implying there's been a copy and paste. Of course it's not always as obvious as that. You could just do the refactoring and see if the test fails, but that puts a lot of reliance on your tests.So, look for the differences; often they show up behavior that youl forgot to test for. Often PullUpMethod comes after other steps. You see two methods in different classes that can be parameterized In such a way that they end up as essentially the same method. In that case the smallest step is to parameterize each method separately and then generalize them. A special case of the need for Pull Up Method occurs when you have a subclass method that overrides a superclass method yet does the same thing. The most awkward element of PullUpMethod is that the body of the methods may refer to features that are on the subclass but not on the superclass. If the feature is a method, you can either generalize the other method or create an abstract method in the superclass. You may need to change a method's signature or create a delegating method to get this to work. (If you have two methods that are similar hut not the same, you may be able to use *FormTemplateMethod*.) This refactoring is often used in the scope of lazy class code smell in combination with the refactorings PullUpField or PushDownMethod and PushDownField to move all the behavior and data of the removed class to the class with which it is being merged.
			Example	
			Process	• Inspect the methode to ensure they are identical. o If the methods look like they do the same thing but are not

<table>
<tr><td></td><td></td><td></td><td></td><td>identical, use SubstituteAlgorithm on one of them to make them identical.

If the methods have different signatures, change the signatures to the one you want to use in the superclass.
Create a new method in the superclass, copy the body of one of the methods to it, adjust, and compile.

If you are in a strongly typed language and the method calls another method that is present on both subclasses but not on the superclass, declare an abstract methodon the superclass.
If the method uses a subclass field, use PullUpField and declare and use an abstract getting method.

Delete one subclass method.
Compile and test.
Keep deleting subclass methods and testing until only the superclass method remains.
Take a look at the callers of this method to see whether you can change a required type to the superclass.
</td></tr>
<tr><td></td><td></td><td></td><td>Example</td><td>Consider a customer with two subclasses: regular customer- and preferred customer.


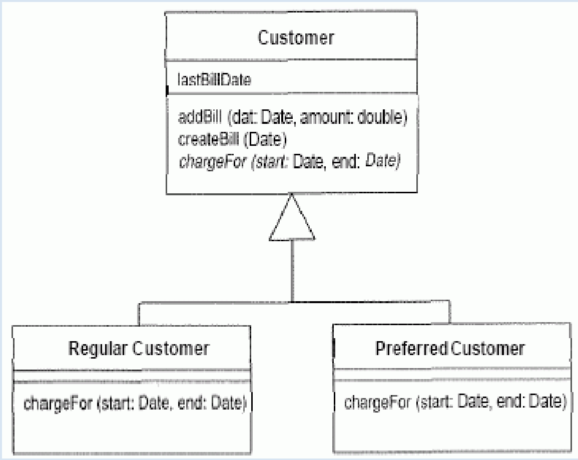
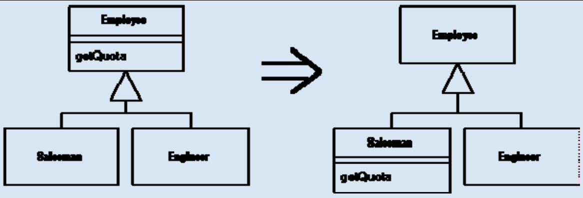
The createBill method is identical for each class:

void createBill (date Date) {
    double chargeAmount = charge (lastBillDate, date);
    addBill (date, charge);
}

You can't move the method up into the superclass, because chargeFor is different on each subclass. First you have to declare it on the superclass as abstract:

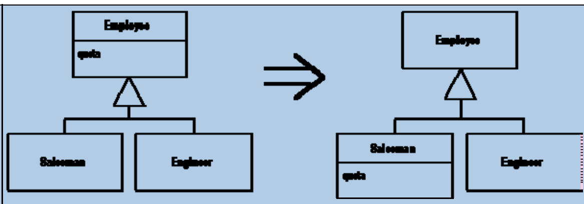
class Customer.. .
    abstract double chargeFor(date start, date end)

Then you can copy createBill from one of the subclasses. You compile with that in place and then remove the createBill method from one of the subclasses, compile, and test. I then remove it from the other, compile, and test:</td></tr>
</table>

				
13	PushDownMethod (328)	(process)	Description	If the behavior on a superclass is relevant only for some of its subclasses then you should apply this refactoring and move it to those subclasses. PullDownMethod is the opposite of PullUpMethod. You use it when you need to move behavior from a superclass to a specific subclass, usually because it makes sense only there. This refactoring is often used in the scope of lazy class code smell in combination with the refactorings PullUpField and PullUpMethod or PushDownField to move all the behavior and data of the removed class to the class with which it is being merged.
			Example	

			Process	• Declare a method in all subclasses and copy the body into each subclass.
				  o You may need to declare fields as protected for the method to access them. Usually you do this if you intend to push down the field later Otherwise use an accessor on the superclass. If the accessor is not public, you need to declare it as protected.
				• Remove method from superclass.
				  o You may have to change callers to use the subclass in variable and parameter declarations
				  o If it makes sense to access the method through a superclass variable, you don't intend to remove the method from any subclasses, and the superclass is abstract, you can declare the method as abstract, in the superclass.
				• Compile and test.
				• Remove the method from each subclass that does not need it.
				• Compile and test.

14	PushDownField	PushDownField (process)	Description	When a field is used only by some subclasses use the refactoring PushDownField and move the field to those subclasses. PushDownField is the opposite of PullUpField. Use it when you don't need a field in the superclass but only in a subclass. This refactoring is often used in the scope of lazy class code smell in combination with the refactorings PullUpField and PullUpMethod or PushDownMethod to move all the behavior and data of the removed class to the class with which it is being merged.

			Example	
			Process	• Declare the field in all subclasses. • Remove the field from the superclass. • Compile and test. • Remove the field from all subclasses that don't need it. • Compile and test.
15	SubstituteAlgorithm 139	SubstituteAlgorithm (process)	Description	This refactoring is applied when you want to replace an algorithm with one that is clearer. Then you must replace the body of the method with the new algorithm. Refactoring can break down something complex into simpler pieces, but sometimes you just reach the point at which you have to remove the whole algorithm and replace it with something simpler, This occurs as you learn more about the problem and realize that there's an easier way to do it. It also happens if you start using a library that supplies features that duplicate your code. Sometimes when you want to change the algorithm to do something slightly different, it is easier to substitute the algorithm first into something easier for the change you need to make. When you have to take this step, make sure you have decomposed the method as much as you can. Substituting a large, complex algorithm is very difficult; only by making it simple can you make the substitution tractable.

			Example	String foundPerson(String people){
				    for (int i = 0; i < people.length; i++) {
				        if (people[i].equals ("Don")){
				            return "Don";
				        }
				        if (people[i].equals ("John")){
				            return "John";
				        }
				        if (people[i].equals ("Kent")){
				            return "Kent";
				        } }
				    return "";
				}
				
				is transformed to:
				String foundPerson(String[] people){
				    List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});
				    for (int i=0; i<people.length; i++)
				        if (candidates.contains(people[i]))
				            return people[i];
				    return "";
				}

			Process	• Prepare your alternative algorithm. Get it so that it compiles. • Run the new algorithm against your tests. If the results are the same, you are finished. • If the results aren't the same, use the old algorithm for comparison in testing and debugging. o Run each test case with old and new algorithms and watch both results. That will help you see which test cases are causing trouble, and how.

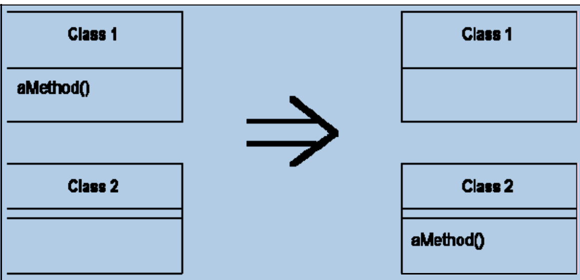
1.5.7 Learning Space: Code Smell *Data Class*

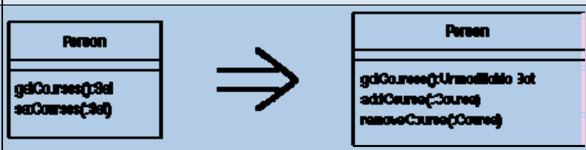
Learnin Space age				
0	Experience Package - Code Smell Data Class	Exerience A+B		
		Ontology Instance (Ontology Main Class)	**Type of Learning Element**	**Learning Content**
1	Refactoring - Introduction	Refactoring (Process)	Definition	Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Fowler [] -- [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
		Refactoring (Process)	Definition	A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck1999]
		Refactoring (Process)	Definition	A behaviour-preserving source-to-source program transformation [Roberts1998]
		Refactoring (Process)	Description	Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an equivalence - the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking. Refactoring is a powerful technique for improving existing software. Having source code that is understandable helps ensure a system is maintainable and extensible. Originally conceived in the Smalltalk community, it has now become a mainstream development technique. • Refactoring is the art of safely removing the design of existing code • Refactoring does not include just any changes in a system. Changes that represent design improvements or add new functionality are not all considered as refactoring • Refactoring is not rewriting from scratch • Refactoring is not just any restructuring intended to improve the code. Refactorings strive to be safe transformations. Even big refactorings that change large amounts of code are divided into smaller, safe refactorings
		Refactoring (Process)	Description	ExtremeProgramming is dependent on refactoring. Refactoring is *not* dependent on or from XP.
2	Code Smell - Introduction	Code Smell (Knowledge)	Definition	In the domain of programming, code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined.
		Code Smell (Knowledge)	Description	Refactorings are no end in itself, but always aim at eliminating a weakness in design. Weaknesses are present when the existing system structure hampers or even prevents modifications. Such weaknesses are also referred to as bad smelling code – so-called code smells. Bad smells often emerge when the so-called Once and Only Once Principle has been disregarded: each design choice shall be expressed exactly in one place in the system. The term appears to have been coined by Kent Beck on WardsWiki. Usage of the term

				increased after it was featured in the book "Refactoring. Improving the Design of Existing Code" by Martin Fowler. Determining what is and is not a code smell is often a subjective judgment, and will often vary by language, developer and development methodology. A code smell can either be a long and complex method in a class, a cyclical uses relation between two classes, or a parallel inheritance hierarchy. Often developers will encounter code smells during their daily work – more specifically whenever the system refuses to accept a modification. Most code smells can be cured with the appropriate refactoring. Finally, rememher that a smell is an indication of a potential problem, not a guarantee of an actual problem. You will occasionally find false-positives – things that smell to you, but are actually better than the alternatives. But most code has plenty of real srnells that can keep you busy.
3	Refactoring - Process	Refactoring (Process)	Process	Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
		Refactoring (Process)	Process	The general refactoring cycle has four steps: • Detect a problem: Choose a working program where smells remain. Is there a problem? What is the problem? • Characterise the problem: Why is it necessary to change something? What are the benefits? Are there any risks? Choose the worst smell • Design a solution: What should be the "goal state" of the code? Which code transformation(s) will move the code towards the desired state? Select a refactoring that will address the smell • Apply the refactoring: Modify the code: Steps that will carry out the code transformation(s) that leave the code functioning the same way as it did before. In addition, when should a refactoring be applied? • When you think it is necessary – Not on a periodical basis • Apply the rule of three – first time: implement solution from scratch – second time: implement something similar by duplicating code – third time: do not reimplement or duplicate, but refactor! • Consolidation before adding new functionality – Before implementing a new feature, the developers analyze the code and debate how this new feature can be realized. It is possible that the new feature will integrate badly with the existing design, or not at all. In this case, in a first step refactoring must be used to rearrange the design to fit the new feature, followed by the developers' incorporation of it in the software. - During debugging – If it is difficult to trace an error, refactor to make the code more comprehensible • After a new feature has been implemented, the developers notice that the design does no longer meet the software's requirements. Using suitable refactorings, the developers can continue to improve the software design until it meets the required functional range. • During formal code inspections (code reviews)
4	Code Smells - Overview	Code Smells within Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
		Code Smells between Classes	Overview	http://wiki.java.net/bin/view/People/SmellsToRefactorings
5	Refactoring -	Refactoring (Process)	Overview	Classification of Martin Fowler: 1. Composing Methods: These refactorings serve restructurings on the method-level.

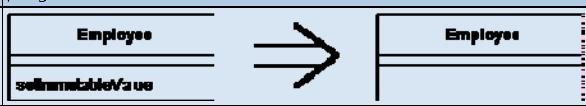
	Overview			Examples of refactorings from this group are: ExtractMethod, InlineTemp or ReplaceTempwithQuery. 2. Moving Features Between Objects: These refactorings support the moving of methods and fields between classes. Among them, refactorings like MoveMethod, ExtractClass or RemoveMiddleMan can be found. 3. Organizing Data: These refactorings restructure the data organization. Examples are: SelfEncapsulateField, ReplaceTypeCodewithClass, or ReplaceArraywithObject. 4. Simplifying Conditional Expressions: These refactorings simplify conditional expressions, such as Introduce NullObject or DecomposeConditional. 5. Making Method Calls Simpler: These refactorings simplify method calls, such as RenameMethod, AddParameter, or ReplaceErrorCodewithException. 6. Dealing with Generalization: These refactorings help to organize inheritance hierarchies, such as PullUpField, ExtractInterface, or FormTemplateMethod.
6	Refactoring - Benefits	Refactoring (Process)	Effort	Most refactorings tend to take from a minute to an hour to apply. The average is probably five to ten minutes.
		Refactoring (Process)	Benefit	Kent Beck states that refactoring adds to the value of any program that has at least one of the following shortcomings: • Programs that are hard to read are hard to modify. • Programs that have duplicate logic are hard to modify • Programs that require additional behaviour that requires you to change running code are hard to modify. • Programs with complex conditional logic are hard to modify
		Refactoring (Process)	Benefit	To improve the software design • To reduce – software decay / software aging – software complexity – software maintenance costs • To increase – software understandibility e.g., by introducing design patterns – software productivity • at long term, not at short term • To facilitate future changes • Improve the software design until it meets the required functional range.
7	Experience Package - Code Smell Data Class	Repeat Experience (AB)		
8	Code Smell Data Class	Code Smell Data Class	Description	Data class is a code smell between classes. 1. In early stages these classes may have public fields. If so, you should immediately apply *EncapsulateField* before anyone notices. Use the refactoring *EncapsulateField* to block direct access to the fields (allowing access only through getters and setters). 2. If you have collection fields, check to see whether they are properly encapsulated and apply *EncapsulateCollection* if they aren't, use *RemoveSettingMethod* on any field that should not be changed. 3. Look at each client of the object. Almost invariably, you'll find clients accessing the fields and manipulating the results when the class could do it for them. (This is often a source of duplication, because many callers will tend to do the same things with the data.) Use *ExtractMethod* on the client to pull out the class-related code, then *MoveMethod* to put it over on the data class. If you can't move a whole method, use *ExtractMethod* to create a method that can be moved. 4. After-doing this awhile, you may find that you have several similar methods on the

				class. Use refactorings such as *RenameMethod*, *ExtractMethod*, *AddParameter*, or *RemoveParameter* to harmonize signatures and remove duplication. 5. Most access to the fields shouldn't be needed anymore because the moved methods cover the real use. So use *HideMethod* to eliminate access to the getters and setters. (You may decide to keep them with private access and have all internal access go through them.)
9	EncapsulateField	EncapsulateField (process)	Description	One of the principal tenets of object orientation is encapsulation, or data hiding. This says that you should never make your data public. When you make data public, other objects can change and access data values without the owning object's knowing about it. This separates data from behavior. This is seen as a bad thing because it reduces the modularity of the program. When the data and behavior that uses it are clustered together, it is easier to change the code, because the changed code is in one place rather than scattered all over the program. If a class has a public field, it can be solved by making it private and providing accessors. *EncapsulateField* begins the process by hiding the data and adding accessors. But this is only the first step. A class with only accessors is a dumb class that doesn't really take advantage of the opportunities of objects, and an object is terrible thing to waste. Once you have done *EncapsulateField* you look for methods that are used by more features of another class than the class on which it is defined. If you find one you use the refactoring *MoveMethod* to move the method to the class.
			Example	public String _name is transformed to. private String _name; public String getName() {return _name;} public void setName(String arg) {_name = arg;}
			Process	• Create getting and setting methods for the field. • Find all clients outside the class that reference the field. If the client uses the value, replace the reference with a call to the getting method. If the client changes the value, replace the reference with a call to the setting method. o If the field is an object and the client invokes modifier on the object, that is a use. Only use the setting method to replace an assignment. • Compile and test after each change. • Once all clients are changed, declare the field as private. • Compile and test.
10	MoveMethod	MoveMethod (process)	Description	When a method is, or will be, using or used by more features of another class than the class on which it is defined, then you should create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether. Moving methods is the bread and butter of refactoring. You should move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By moving methods around, you can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities. You should look through the methods on a class to find a method that seems to reference another object more than the object it lives on. It's not always an easy decision to make. If I am not sure whether to move a method, you should go on to look at other methods. Moving other methods often makes the decision easier. Sometimes the decision still is hard to make.

			Example	
			Process	• Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved. o If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a set of methods than to move them one at a time. • Check the sub- and superclasses of the source class for other declarations of the method. o If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target. • Declare the method in the target class. o You may choose to use a different name, one that makes more sense in the target class. • Copy the code from the source method to the target. Adjust the method to make it work in its new home. • If the method uses its source, you need to determine how to reference the source object from the target method. If there is no mechanism in the target class, pass the source object reference to the new method as a parameter, • If the method includes exception handlers handlers, decide which class should logically handle the exception. If the source class should be responslble, leave the handlers behind. • Compile the target class. • Determine how to reference the correct target object from the source. o There may be an existing field or method that wdl give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also make it temporarily until you have refactored enough to remove it. • Turn the source method into a delegating method. • Compile and test. • Decide whether to remove the source method or retain it as a delegating method. • Leaving the source as a delegating method is easier if you have many references. • If you remove the source method, replace all the eferences with references to the target method. o You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace. • Compile and test.
			Example	• F p.144
11	EncapsulateCollection	EncapsulateCollection (process)	Description	Often a class contains a collection of instances. This collection might be an array, list, set, or vector. Such cases often have the usual getter and setter for the collection. However, collections should use a protocol slightly different from that for other kinds of data. The getter should not return the collection object itself, because that allows clients to manipulate the contents of the collection without the owning class's knowing

				what is going on. It also reveals too much to clients about the object's internal data structures. A getter for a multivalued attribute should return something that prevents manipulation of the collection and hides unnecessary details about its structure. How you do this varies depending on the version of Java you are using. In addition there should not be a setter for collection: rather there should be operations to add and remove elements. This gives the owning object control over adding and removing elements from the collection. With this protocol the collection is properly encapsulated, which reduces the coupling of the owning class to its clients.
			Example	

			Process	• Add an add and remove method for the collection.
				• Initialize the field to an empty collection.
				• Compile.
				• Find callers of the setting method. Either modify the setting method to use the add and remove operations or have the clients call those operations instead.
				  o Setters are used in two cases: when the collection is empty and when the setter is replacing a non-empty collection.
				  o You may wish to use *RenameMethod* to rename the setter. Change it from set to initialize or replace.
				• Compile and test.
				• Find all users of the getter that modify the collection. Change them to use the add and remove methods. Compile and test after each change.
				• When all uses of the getter that modify have been changed, modify the getter to return a read-only view of the collection.
				  o In Java 2, thrs rs the appropriate unmodifiable collection view
				  o In Java 1.1, you should return a copy of the collection
				• Compile and test.
				• Find the users of the getter. Look for code that should be on the host object. Use ExtractMethod and MoveMethod to move the code to the host object.
				• For Java 2, you are done with that. For Java 1.1, however, clients may prefer to use an enumeration. To provide the enumeration:
				• Change the name of the current getter and add a new getter to return an enumeration. Find users of the old getter and change them to use one of the new methods.
				  o If this is a to bog change , use RenameMethod on the old getter; create a new method that returns an enumeration, and change callers to use the new method.
				• Compile and test.

12	RemoveSettingMethod	RemoveSettingMethod (process)	Description	A field should be set at creation time and never altered. There, you should remove any setting method for that field. Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created, then don't provide a setting method (and make the field final). That way your intention is clear and you often remove the very possibility that the field will change. This situation often occurs when programmers blindly use direct variable access. Such programmers then use setters even in a constructor.
			Example	
			Process	o If the field isn't final, make it so. o Compile and test.

				o Check that the setting method is called only in the constructor, or in a method called by the constructor. o Modify the constructor to access the variables directly. o You cannot do this if you have a subclass setting the private fields of a superclass. In this case you should try to provide a protected superclass method (ideally a constructor) to set these values. Whatever you do, don't give the superclass method a name that will confuse it with a setting method. o Compile and test. o Remove the setting method. o Compile.

			Process	A simple example is as follows:
				class Account {
				    private String _id;
				    Account (String id) {
				    setId(id) ;
				    }
				    void setId (String arg) {
				    _id = arg;
				}
				
				which can be replaced with
				
				class Account {
				    private final String _id;
				    Account (String id) {
				    _id = id;
				}

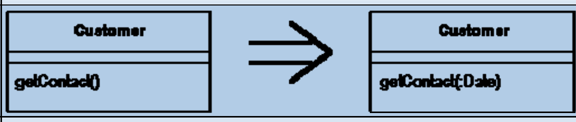
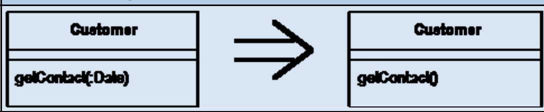
			Example	The problems come in some variations. First is the case in which you are doing computation on the argument:
				
				class Account {
				    private String _id;
				    Account (String id) {
				    setId(id) ;
				    }
				    void setId (String arg) {
				    _id = "ZZ" + arg;
				}
				If the change is simple (as here) and there is only one constructor, you can make the change in the constructor. If the change is complex or you need to call it from separate methods, you need to provide a method. In that case you need to name the method to make its intention clear:
				
				class Account {
				    private final String _id;
				    Account (String i d) {
				    initializeId (id);
				    }
				    void initializeId (String arg) {
				    _id = "ZZ" + arg;
				}

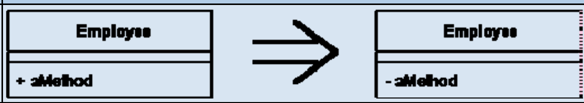
			Example	An awkward case lies with subclasses that initialize private superclass variables:
				
				class InterestAccount extends Account..
				    private double _interestRate;
				    InterestAccount (String id , double rate) {
				    setId(id);
				    _interestRate = rate;
				}

				The problem is that you cannot access id directly to set it. The best solution is to use a superclass constructor:
				
				class InterestAccount..
				InterestAccount (String id , double rate) {
				super(id) ;
				_interestRate = rate;
				}
				
				If that is not possible, a well-named method is the best thing to use:
				
				class InterestAccount..
				
				InterestAccount (String id , double rate) {
				initializeId(id);
				_interestRate = rate;
				}

			Example	Another case to consider is setting the value of a collection:
				
				class Person {
				Vector getCourses0 {
				return _courses;
				}
				void setCourses(Vector arg) {
				_courses = arg;
				}
				private Vector _courses;
				
				Here I want to replace the setter with add and remove operations. This can be done by using the refactoring *EncapsulateCollection*.

13	ExtractMethod	see other EP		
14	AddParameter	AddParameter (process)	Description	AddParameter is a very common refactoring, one that you almost certainly have already done. The motivation is simple. You have to change a method, and the change requires information that wasn't passed in before, so you add a parameter. Actually most of what i have to say is motivation against doing this refactoring. Often you have other alternatives to adding a parameter. If available, these alternatives are better because they don't lead to increasing the length of parameter lists. Long parameter lists smell bad because they are hard to remember and often involve data clumps. Look at the existing parameters. Can you ask one of those objects for the information you need? If not, would it make sense to give them a method to provide that information? What are you using the information for? Should that behavior be on another object, the one that has the information? Look at the existing parameters and think about them with the new parameter. Perhaps you should consider Introduce Parameter Object (295). I'm not saying that you should never add parameters; I do it frequently, but you need to be aware of the alternatives.
			Process	The mechanics of AddParameter are very similar to those of RenameMethod: • Check to see whether this method signature is implemented by a superclass or subclass. If it is, carry out these steps for each implementation. • Declare a new method with the added parameter. Copy the old body of • code over to the new method. o If you need to add more than one paranzeter, it is easier to add them at the same time. • Compile. • Change the body of the old method so that it calls the new one. o If you only have a few references, you can reasonably skip this step. o You can supply any value for the parameter, but usually you use

				null for object parameter and clearly odd value for built-in types. It's a good idea to use something other than zero for numbers so you can spot this case more easily. • Compile and test. • Find all references to the old method and change them to refer to the new one. • Compile and test after each change. • Remove the old method. o If the old method is part of the interface and you cannot remove it, leave it in place and mark it as deprecated. • Compile and test.
			Example	
15	RemoveParameter	RemoveParameter (procees)	Description	Programmers often add parameters but are reluctant to remove them. After all, a spurious parameter doesn't cause any problems, and you might need it again later. This is bad! A parameter indicates information that is needed; different values make a difference. Your caller has to worry about what values to pass. By not removing the parameter you are making further work for everyone who uses the method. That's not a good trade-off, especially because removing parameters is an easy refactoring. The case to be wary of here is a polymorphic method. In this case you may well find that other implementations of the method do use the parameter. In this case you shouldn't remove the parameter. You might choose to add a separate method that can be used in those cases, but you need to examine how your callers use the method to see whether it is worth doing that. If some callers already know they are dealing with a certain subclass and doing extra work to find the parameter or are using Itnowledge of the class hierarchy to know they can get away with a null, add an extra method without the parameter. If they do not need to know about which class has which method, the callers should be left in blissful ignorance.
			Process	The mechanics of RemoveParameter are very similar to those of RenameMethod and AddParameter. • Check to see whether this method signature is implemented by a superclass or subclass. Check to see whether the class or superclass uses the parameter. If it does, don't do this refactoring. • Declare a new method without the parameter. Copy the old body of code to the new method. o If you need to remove more than one parameter, it is easier to remove them together. • Compile. • Change the body of the old method so that it calls the new one. o If you only have a few references, you can reasotzably skip this step. • Compile and test. • Find all references to the old method and change them to refer to the new one. Compile and test after each change. • Remove the old method. o If the old method 1s part of the rnterface and you cannot remove it, leave it in place and mark it as deprecated. • Compile and test. Because I'm pretty comfortable with adding and removing parameters, I often do a batch in one go.
			Figure	

16	HideMethod	HideMethod (process)	Description	If a method is not used by any other class, then you should make it private. Refactoring often causes you to change decisions about the visibility of methods. It is easy to spot cases in which you need to make a method more visible: another class needs it and you thus relax the visibility. It is somewhat more difficult to tell when a method is too visible. Ideally a tool should check all methods to see whether they can be hidden. If it doesn't, you should make this check at regular intervals. A particularly common case is hiding getting and setting methods as you work up a richer interface that provides more behavior. This case is most common when you are starting with a class that is little more than an encapsulated data holder. As more behavior is built into the class, you may find that many of the getting and setting methods are no longer needed publicly, in which case they can be hidden. If you make a getting or setting method private and you are using direct variable access, you can remove the method.
			Example	
			Process	o Check regularly for opportunities to make a method more private. o Make each method as private as you can. o Compile after doing a group of hidings.

1.6 Assignments

In the following, all the assignments used during the controlled experiment are provided. Five different developer teams were involved during the experiment. The code used for the assignments was code produced by the corresponding teams themselves, i.e., the assignments contain their own code. Therefore, 20 different assignments were produced for the two periods of the experiment.

The first page of the assignment provides instructions for solving the assignment and asks the subject to enter the time when he starts to solve the assignment (see Section 1.6.1 until Section 1.6.4). After that, two exercises with Java code were given to the subjects (see Section 1.7 and Section 1.8). It was up to the students to decide whether they completely read the provided information first (i.e., information of an experience package or a learning space) or directly started to solve the exercises. The sheet for describing the solutions used by the subjects is provided in Section 1.6.5 (example) and Appendix 1.6.6 (empty sheet).

1.6.1 Assignment Information and Related Exercises (Mo-Mo-G1): (Group:__________________)

Your Name: ______________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Goal of the experiment:
The goal of the experiment is to apply the knowledge from an experience package to your own context (in this case DCGA project). Information about the experience package will be

provided in a Wiki. Further, additional information in a so-called learning space will help you to understand and apply the experience package. In order to apply the experience packages an exercise should be solved.

Selected Experience Packages
This sheet explains in which order you should work through the experience packages. Two experience packages have been assigned to you. Please access them in the following sequence as assigned in the parentheses. When you have read the information in the Wiki and when you think you are ready to solve the exercise, please put the actual time behind the corresponding experience package when you start to access the experience package in the Wiki.

- Experience Package Code Smell *Comments* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Method* (___) starting time [___ : ___]

- Experience Package Code Smell *Type Embedded in Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Uncommunicative Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Parameter List* (___) starting time [___ : ___]

- Experience Package Code Smell *Lazy Class* (___) starting time [___ : ___]

- Experience Package Code Smell *Data Class* (___) starting time [___ : ___]

Please access the Wiki by using your web browser:
http://watt.informatik.uni-kl.de/gseprojekt1/index.php/Spezial:Experiences (use gseprojekt "1" !)
Login: experiment
Passwd: geiermeier

The exercises are provided in the following.

1.6.2 Assignment Information and Related Exercises (Mo-Aft-G2): (Group:________________)

Your Name: ____________________________

Your Subject-ID: ______<your ID will be filled out by evaluators>

Goal of the experiment:
The goal of the experiment is to apply the knowledge from an experience package to your own context (in this case DCGA project). Information about the experience package will be provided in a Wiki.

Selected Experience Packages
This sheet explains in which order you should work through the experience packages. Two experience packages have been assigned to you. Please access them in the following sequence as assigned in the parentheses. When you have read the information in the Wiki and when you think you are ready to solve the exercise, please put the actual time behind the corresponding experience package when you start to access the experience package in the Wiki.

- Experience Package Code Smell *Comments* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Method* (___) starting time [___ : ___]

- Experience Package Code Smell *Type Embedded in Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Uncommunicative Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Parameter List* (___) starting time [___ : ___]

- Experience Package Code Smell *Lazy Class* (___) starting time [___ : ___]

- Experience Package Code Smell *Data Class* (___) starting time [___ : ___]

To each of the experience package an exercise should be solved when you have read the information in the Wiki and when you think you are ready to solve the exercise.
Please access the Wiki by using your web browser:
http://watt.informatik.uni-kl.de/gseprojekt3/index.php/Spezial:Experiences (use gseprojekt "3" !)
Login: experiment
Passwd: auawaua

The exercises are provided in the following.

1.6.3 Assignment Information and Related Exercises (Tu-Mo-G2): (Group:__________________)

Your Name: ____________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Goal of the experiment:
The goal of the experiment is to apply the knowledge from an experience package to your own context (in this case DCGA project). Information about the experience package will be provided in a Wiki. Further, additional information in a so-called learning space will help you to understand and apply the experience package. In order to apply the experience packages an exercise should be solved.

Selected Experience Packages
This sheet explains in which order you should work through the experience packages. Two experience packages have been assigned to you. Please access them in the following sequence as assigned in the parentheses. When you have read the information in the Wiki and when you think you are ready to solve the exercise, please put the actual time behind the corresponding experience package when you start to access the experience package in the Wiki.

- Experience Package Code Smell *Comments* (___)
starting time [___ : ___]

- Experience Package Code Smell *Long Method* (___) starting time [___ : ___]

- Experience Package Code Smell *Type Embedded in Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Uncommunicative Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Parameter List* (___) starting time [___ : ___]

- Experience Package Code Smell *Lazy Class* (___)
starting time [___ : ___]

- Experience Package Code Smell *Data Class* (___)
starting time [___ : ___]

To each of the experience package an exercise should be solved when you have read the information in the Wiki and when you think you are ready to solve the exercise.
Please access the Wiki by using your web browser:

http://watt.informatik.uni-kl.de/gseprojekt1/index.php/Spezial:Experiences (use gseprojekt "1" !)
Login: experiment
Passwd: eierweier

The exercises are provided in the following.

1.6.4 Assignment Information and Related Exercises (Tu-Aft-G1): (Group:________________)

Your Name: ______________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Goal of the experiment:
The goal of the experiment is to apply the knowledge from an experience package to your own context (in this case DCGA project). Information about the experience package will be provided in a Wiki.

Selected Experience Packages
This sheet explains in which order you should work through the experience packages. Two experience packages have been assigned to you. Please access them in the following sequence as assigned in the parentheses. When you have read the information in the Wiki and when you think you are ready to solve the exercise, please put the actual time behind the corresponding experience package when you start to access the experience package in the Wiki.

- Experience Package Code Smell *Comments* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Method* (___) starting time [___ : ___]

- Experience Package Code Smell *Type Embedded in Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Uncommunicative Name* (___) starting time [___ : ___]

- Experience Package Code Smell *Long Parameter List* (___) starting time [___ : ___]

- Experience Package Code Smell *Lazy Class* (___) starting time [___ : ___]

- Experience Package Code Smell *Data Class* (___) starting time [___ : ___]

To each of the experience package an exercise should be solved when you have read the information in the Wiki and when you think you are ready to solve the exercise.
Please access the Wiki by using your web browser:
http://watt.informatik.uni-kl.de/gseprojekt3/index.php/Spezial:Experiences
(use gseprojekt "3" !)

Login: experiment; Passwd: balabala

The exercises are provided in the following.

1.6.5 Answer Sheet for Exercises (example)

This is an example at how to mark a code smell and how to describe it in the *Answer Sheet for Exercises*.

Code example:

```
void printOwing() {
        printBanner();

        //print details
        System.out.println ("name:  " + _name);
        System.out.println ("amount" + amount);
}
```

Your explanation can be provided in different ways:

<table>
<tr><th>Number</th><th>Explanation of your descision</th></tr>
<tr><td>1</td><td>I would use the Extract Method refactoring. This is a solution:
<pre>void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
        System.out.println ("name: " + _name);
        System.out.println ("amount   " + outstanding);
}</pre></td></tr>
<tr><td></td><td>or describe it in this way</td></tr>
<tr><td>1</td><td>I would use the Extract Method refactoring.
The first step is to extract both system.out.println statements into a separate method (e.g., method printDetails(double outstanding) with the double variable oustanding). This method call to this new method will replace the println statements in the printOwning method.
That’s it.</td></tr>
<tr><td></td><td>It is not necessary to state the compile and test steps !</td></tr>
</table>

1.6.6 Answer Sheet for Assignments

The answer can also be stated in German if this is more appropriate for you.

Number	Explanation of your decision

1.7 Exercises of the Assignments (Monday)

1.7.1 Exercise to Experience Package for Amica Interaction Group: Long Method

Your Name: ______________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import org.belami.dcga.amica_interaction.Situation;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/* Data structure containing the information of one "match" element
from the XML mapping file.
 */
public class Match {
    /**
* Fact ID that has to be matched with the Situation object
     */
    private String factName = null;
    /**
     * Comparator method for the fact ID from the mapping-file
```

```
     */
    private String factNameComparator = null;

    /**
     * Start date that has to be matched with the Situation object
     */
    private Date startDate = null;
    /**
     * Comparator method for the start date from the mapping-file
     */
    private String startDateComparator = null;

    /**
     * End date that has to be matched with the Situation object
     */
    private Date endDate = null;
    /**
     * Comparator method for the end date from the mapping-file
     */

    private String endDateComparator = null;

    /**
     * Description that has to be matched with the Situation object
     */
    private String description = null;
    /**
     * Comparator method for the description from the mapping-file
     */
    private String descriptionComparator = null;

    /**
     * Source that has to be matched with the Situation object
     */
    private String source = null;
    /**
     * Comparator method for the source identifier from the mapping-
file
     */
    private String sourceComparator = null;

    /**
     * Location that has to be matched with the Situation object
     */
    private String location = null;
    /**
     * Comparator method for the location identifier from the map-
ping-file
     */
```

```java
    private String locationComparator = null;

    /**
     * NodeList used to map a matching Situation to an Information.
Might be null if not applicable.
     */
    public NodeList mapInformationNodes = null;
    /**
     * NodeList used to map a matching Situation to a Task. Might be
null if not applicable.
     */
    public NodeList mapTaskNodes = null;
    /**
     * Boolean value that specifies if a matching Situation is mapped
a TaskEvent.
     */
    public boolean mapTaskEvent = false;


    private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd");

    private static DateFormat dateTimeFormat = new SimpleDateFor-
mat("yyyy-MM-dd k:m:s");

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
```

```
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    /**
     * Returns true if the given situation is matched.
     * @param situation A Situation
     * @return True if the given situation is matched.
     */
    public boolean matches(Situation situation) {
```

```
        if(factNameComparator != null) {
            if(!compare(situation.getFactName(), factName, factName-
Comparator)) {
                return false;
            }
        }
        if(startDateComparator != null) {
            if(!compare(situation.getStartDate(), startDate, start-
DateComparator)) {
                return false;
            }
        }
        if(endDateComparator != null) {
            if(!compare(situation.getEndDate(), endDate, endDateCom-
parator)) {
                return false;
            }
        }
        if(descriptionComparator != null) {
            if(!compare(situation.getDescription(), description, de-
scriptionComparator)) {
                return false;
            }
        }
        if(sourceComparator != null) {
            if(!compare(situation.getSource(), source, sourceCompara-
tor)) {
                return false;
            }
        }
        if(locationComparator != null) {
            if(!compare(situation.getLocation()+"", location, loca-
tionComparator)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
    public boolean mapsInformation(Situation situation) {
```

```
        return mapInformationNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }

        return information;
    }

    /**
     * Map the given Situation to a Task object.
     * @param situation A Situation
```

```
     * @return Mapped Task object
     */
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Map the given Situation to a TaskEvent object.
     * @param situation A Situation
     * @return Mapped TaskEvent object
     */
    public TaskEvent mapTaskEvent(Situation situation) {
        TaskEvent taskEvent = new TaskEvent(situation.getSource(),
situation.getLocation(), situation.getFactName());
        return taskEvent;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
     * @param a Original object
```

```
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "startsWith",
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
            return false;
```

```
        } else {
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Replaces keywords in a String using data from the given Situa-
tion object
     * @param text Untreated input String
     * @param situation A Situation
     * @return Treated Text
     */
    protected static String prepareString(String text, Situation
situation) {
        text = text.replaceAll("\\{\\{priority\\}\\}", situa-
tion.getPriority()+"");
        if (situation.getDescription()!= null) {
           text = text.replaceAll("\\{\\{description\\}\\}", situa-
tion.getDescription());
        }
        if (situation.getLocation()!= null) {
           text = text.replaceAll("\\{\\{location\\}\\}", situa-
tion.getLocation()+"");
        }
        text = text.replaceAll("\\{\\{startDate\\}\\}",  dateTimeFor-
mat.format(situation.getStartDate()));
        if (situation.getEndDate()!= null) {
           text = text.replaceAll("\\{\\{endDate\\}\\}", dateTimeFor-
mat.format(situation.getEndDate()));
        }
        text = text.replaceAll("\\{\\{source\\}\\}", situa-
tion.getSource());
        text = text.replaceAll("\\{\\{factName\\}\\}", situa-
tion.getFactName());

        return text;
     }
}
```

1.7.2 Exercise to Experience Package for Amica Interaction Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type embedded in name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import org.belami.dcga.amica_interaction.Situation;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/* Data structure containing the information of one "match" element
from the XML mapping file.
 *
 */
public class Match {
	/**
* Fact ID that has to be matched with the Situation object
	 */
	private String factName = null;
	/**
```

```
     * Comparator method for the fact ID from the mapping-file
     */
    private String factNameComparator = null;

    /**
     * Start date that has to be matched with the Situation object
     */
    private Date startDate = null;
    /**
     * Comparator method for the start date from the mapping-file
     */
    private String startDateComparator = null;

    /**
     * End date that has to be matched with the Situation object
     */
    private Date endDate = null;
    /**
     * Comparator method for the end date from the mapping-file
     */

    private String endDateComparator = null;

    /**
     * Description that has to be matched with the Situation object
     */
    private String description = null;
    /**
     * Comparator method for the description from the mapping-file
     */
    private String descriptionComparator = null;

    /**
     * Source that has to be matched with the Situation object
     */
    private String source = null;
    /**
     * Comparator method for the source identifier from the mapping-
file
     */
    private String sourceComparator = null;

    /**
     * Location that has to be matched with the Situation object
     */
    private String location = null;
    /**
     * Comparator method for the location identifier from the map-
ping-file
```

```
     */
    private String locationComparator = null;

    /**
     * NodeList used to map a matching Situation to an Information.
Might be null if not applicable.
     */
    public NodeList mapInformationNodes = null;
    /**
     * NodeList used to map a matching Situation to a Task. Might be
null if not applicable.
     */
    public NodeList mapTaskNodes = null;
    /**
     * Boolean value that specifies if a matching Situation is mapped
a TaskEvent.
     */
    public boolean mapTaskEvent = false;


    private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd");

    private static DateFormat dateTimeFormat = new SimpleDateFor-
mat("yyyy-MM-dd k:m:s");

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
```

```
                    }
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    /**
     * Returns true if the given situation is matched.
     * @param situation A Situation
     * @return True if the given situation is matched.
     */
```

```
    public boolean matches(Situation situation) {
        if(factNameComparator != null) {
            if(!compare(situation.getFactName(), factName, factName-
Comparator)) {
                return false;
            }
        }
        if(startDateComparator != null) {
            if(!compare(situation.getStartDate(), startDate, start-
DateComparator)) {
                return false;
            }
        }
        if(endDateComparator != null) {
            if(!compare(situation.getEndDate(), endDate, endDateCom-
parator)) {
                return false;
            }
        }
        if(descriptionComparator != null) {
            if(!compare(situation.getDescription(), description, de-
scriptionComparator)) {
                return false;
            }
        }
        if(sourceComparator != null) {
            if(!compare(situation.getSource(), source, sourceCompara-
tor)) {
                return false;
            }
        }
        if(locationComparator != null) {
            if(!compare(situation.getLocation()+"", location, loca-
tionComparator)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
```

```
    public boolean mapsInformation(Situation situation) {
        return mapInformationNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }

        return information;
    }

    /**
     * Map the given Situation to a Task object.
```

```
     * @param situation A Situation
     * @return Mapped Task object
     */
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Map the given Situation to a TaskEvent object.
     * @param situation A Situation
     * @return Mapped TaskEvent object
     */
    public TaskEvent mapTaskEvent(Situation situation) {
        TaskEvent taskEvent = new TaskEvent(situation.getSource(),
situation.getLocation(), situation.getFactName());
        return taskEvent;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
```

```
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "startsWith",
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
```

```
            return false;
        } else {
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Replaces keywords in a String using data from the given Situa-
tion object
     * @param text Untreated input String
     * @param situation A Situation
     * @return Treated Text
     */
    protected static String prepareString(String text, Situation
situation) {
        text = text.replaceAll("\\{\\{priority\\}\\}", situa-
tion.getPriority()+"");
        if (situation.getDescription()!= null) {
           text = text.replaceAll("\\{\\{description\\}\\}", situa-
tion.getDescription());
        }
        if (situation.getLocation()!= null) {
           text = text.replaceAll("\\{\\{location\\}\\}", situa-
tion.getLocation()+"");
        }
        text = text.replaceAll("\\{\\{startDate\\}\\}",  dateTimeFor-
mat.format(situation.getStartDate()));
        if (situation.getEndDate()!= null) {
           text = text.replaceAll("\\{\\{endDate\\}\\}", dateTimeFor-
mat.format(situation.getEndDate()));
        }
        text = text.replaceAll("\\{\\{source\\}\\}", situa-
tion.getSource());
        text = text.replaceAll("\\{\\{factName\\}\\}", situa-
tion.getFactName());

        return text;
   }
}
```

1.7.3 Exercise to Experience Package for Computation: Long Method

Your Name: ________________________
Your Subject-ID: _____ <your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java→ in your code code smells of long method couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;

… code removed …

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
```

```
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    … code removed …
```

```
    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
    public boolean mapsInformation(Situation situation) {
        return mapInformationNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
```

```
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }

        return information;
    }

    /**
     * Map the given Situation to a Task object.
     * @param situation A Situation
     * @return Mapped Task object
     */
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
```

```
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "startsWith", 
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by 
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
```

```
            return false;
        } else {
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    ... code removed …
}
```

1.7.4 Exercise to Experience Package for Computation Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type embedded in name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Computation: taskmanager

```
package org.belami.dcga.computation.taskmanager;

import java.util.Observer;
import java.util.Vector;

import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;

/**
 * This component is responsible for all task related computations.
 * It stores the task list for the current elderly person and per-
forms
 * the following computations (from initial problem description)so 
far:
 * sort task list when new room is entered, auto check tasks for com-
pletion
 * if possible, change task status
 *
 * @author j_koehle
 *
 */
public interface TaskManager {

      public static TaskManager INSTANCE = TaskManager-
Impl.getInstance();

      /**
```

```
         * Loads task list for current patient from persistence
         */
        public void initialize();

        /**
         * Checks if care giver was in the rooms demanded for
         * the specified task yet and changes task status to
         * "done by care giver"
         * @param taskID
         * ID of task to change
         * @param override
         * if true, no exception is thrown. Reason: Care Giver can mark
         * task as completed, although isn't marked as visited (in case
         * of defective rfid-system)
         * @throws RoomNotVisitedException
         */
        public void markTaskAsCompletedManually(int taskID, boolean
override)
                    throws RoomNotVisitedException;

        /**
         * Changes the state of the task event specified by TaskEvendID
at
         * all tasks waiting for this event as "done" and check whether
         * the task is finished or not. A task is finished when all
         * taskEvents are done.
         * @param event
         * Incoming task event tracked by amiCA
         */
        public void setTaskEventDone(TaskEvent event);

        /**
         * Adds a task from the caller to the current TaskList in Task-
Manager
         * and persistence
         * @param unplannedTask
         * Incoming unplanned Task
         */
        public void addUnplannedTask(Task unplannedTask);

        /**
         * Returns TaskList for the current patient to the caller
         * @return
         * Task list for current elderly person
         */
        public Vector<Task> getTaskList();

        //WER DAS INTERFACE ÄNDERT OHNE MICH ZU FRAGEN WIRD GEKÖPFT :D
        /**
```

```
     * Registeres an observer in the observable task list. It's no-
tified
     * every time the list changes.
     * @param taskListObserver
     * Observer to add
     */
    public void addTaskListObserver(Observer taskListObserver);

    /**
     * Deletes an observer from the observable task list.
     * @param taskListObserver
     * Observer to delete
     */
    public void deleteTaskListObserver(Observer taskListObserver);

    /**
     * Registeres an observer that will be notified if a task is
     * in state "undone" when the appartment is left.
     * @param warningObserver
     * Observer to add
     */
    public void addOpenTaskWarningObserver(Observer warningOb-
server);

    /**
     * Deletes an observer for open task warnings
     * @param warningObserver
     * Observer to delete
     */
    public void deleteOpenTaskWarningObserver(Observer warningOb-
server);

    /**
     * If the apartment is left, this function checks, if there are
tasks
     * left undone. If this is the case, oben task warning observers
will
     * be notified.
     */
    public void onApartmentLeft();

    /**
     * sorts the task list according task priority and room the care
giver is
     * currently in
     */
    public void sort();

}
```

1.7.5 Exercise to Experience Package for Location Manager Group: Long Method

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method
2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"
3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java → in your code code smells of long method couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;

… code removed …

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
```

```
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    … code removed …
```

```
    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
    public boolean mapsInformation(Situation situation) {
        return mapInformationNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
```

```
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }

        return information;
    }

    /**
     * Map the given Situation to a Task object.
     * @param situation A Situation
     * @return Mapped Task object
     */
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
```

```
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "startsWith",
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
```

```
            return false;
        } else {
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    ... code removed …
}
```

1.7.6 Exercise to Experience Package for Location Manager Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type embedded in name
2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"
3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Locationmanager: locationmanagerImpl.java

```
package org.belami.dcga.location_manager;

import org.belami.dcga.common_datastructures.PositionData;
import org.belami.dcga.computation.Computation;

class LocationManagerImpl implements LocationManager {
      private RFIDConnector rfidConnector = new RFIDConnector();

      private RoomMapping roomMapping = new RoomMapping();

      private VisitedRoomList visitedRoomList = new VisitedRoomList();

      private int currentRoomId = noRoom;

      static final int noRoom = -1;

      /**
       * initilize the connection between System and RFID, Persis-
tence, clear
       * Visited roomlist
       *
       * @return True if connections are initilized, false otherwise.
       */
      public boolean initialize(int apartmentId) {
            if (!roomMapping.loadRoomMapping(apartmentId)
                        || !rfidConnector.initialize())
```

```
                return false;

            visitedRoomList.clear();
            currentRoomId = noRoom;

            return true;
        }

        /**
         * when new RFID coordination entered 1. Coordination will be in
RoomId
         * translated 2. is this RoomId a NEW roomID? 3. if the roomId
is new, call
         * Computation.INSTANCE.onNewRoomEntered(newRoom)
         *
         */
        public void onRefresh(PositionData newPos) {
            int newRoom = roomMapping.convertToRoom(newPos);
            if (currentRoomId != newRoom && newRoom != noRoom) {
                currentRoomId = newRoom;
                visitedRoomList.add(newRoom);
                Computation.INSTANCE.onNewRoomEntered(newRoom);
            }
        }

        /**
         * stop the connection to RFID.
         */
        public void stop() {
            rfidConnector.stop();
        }

        /**
         * call the method visitedRoomList.wasRoomEntered(roomId);.
         */
        public boolean wasRoomEntered(int roomId) {
            return visitedRoomList.wasRoomEntered(roomId);
        }
}
```

1.7.7 Exercise to Experience Package for Persistence Group: Long Method

Your Name: ______________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Persistence:persistenceImpl.java

```
package org.belami.dcga.persistence;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.*;

import org.belami.dcga.common_datastructures.*;

class PersistenceImpl implements Persistence{
/**Overview
 *Project: DCGA, Summer semester 2007, GSE-Project, Technische Uni-
versität Kaiserslautern
 *Subsystem: Persistence
 *Desing version: Persistence.doc (Date: --/06/2007)
 *Last modification: 21.06.2007
 **/

/**   Atributes:
 *    int n: Number of elderly persons that are stored (number of pa-
tients).
 *    ElderlyPerson curentPatient: Temporary copy of the elderly per-
son selected by the care giver.
 *    TaskList currentTaskList: First uncompleted Task List associated
the selected elderly person.
 *    int numberTasks: number of tasks of the current Task List.
```

```
 *   int numberInformations: number of informations associated to the
selected elderly person.
 *   int numberComments: number of comments associated to the se-
lected elderly person.
 *   int lastCommentId: Id associated to the last stored comment.
 **/

ElderlyPerson currentPatient = new ElderlyPerson();
int n=0;
TaskList currentTaskList =new TaskList();
int numberTasks=0;
int numberInformations=0;
int numberComments=0;
int lastCommentId=0;

/** Methods:**/

/** Name: getPatientList()
 * Komponent: Persistence
 * Function: getPatientList()
 * Input : -
 * Output:
 *   name: epData
 *   description: : List of current patients
 *   type: Set<ElderlyPersonShortInfo>
 *
 * Description:
 *   1. Reads ElderlyPersonsShortInformation list (epData) from Eld-
erlyPersonMap.text file
 *   2. Sets the number of patients (n)
 *   3. Return List of current patients (epData)
 *
 *  Variables:
 *         name : eData
 *         description: Summarize of the elderly patient information:
id, name, address,
 *         type: ElderlyPersonsShortInformation
 *
 * Last modificaction: 21.07.2007
 * Test cases:
 * **/
public Collection getPatientList(){

      Set<ElderlyPersonShortInfo> epData = new HashSet();
      try {
       FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       epData = (Set<ElderlyPersonShortInfo>)ois.readObject();
```

```
        ois.close();
        }catch(Exception e){
           e.printStackTrace();
      }

      n=0;
      ElderlyPersonShortInfo myPatient=new ElderlyPersonShortInfo();

      Iterator myIterator = epData.iterator();
      while (myIterator.hasNext()) {
            myPatient= (ElderlyPersonShortInfo) myIterator.next();
            n++; }

      return epData;
}

/**
* Name: setPatientId()
* Komponent: Persistence
* Function: setPatientId()
* Input:
*     name:pId
*     description: id of the selected elderly person
*     type: int
* Output: -
* Description: Sets the data of the currentPatient
*     1. Reads the data of the selected elderly person from the file
pID.txt
*     2. Updates currentPatient
*     3. Sets numberInformations, numberComments
*     5. Searchs and sets lastCommentId
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void setPatientId(int pId){
      currentPatient=null;
      try{
      FileInputStream fis = new FileInputStream(pId+".txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
    currentPatient = (ElderlyPerson )ois.readObject();
    ois.close();

    numberInformations= currentPatient.getInformations().size();
      numberComments= currentPatient.getComments().size();

      //checks for the highest commentId (if some comment was deleted,
its Id, won´t be used any more)
      lastCommentId= numberComments;
```

```
        Comment myComment=new Comment();
        Iterator myIterator = currentPatient.getComments().iterator();
        while (myIterator.hasNext()) {
              myComment = (Comment) myIterator.next();
              if (myComment.getCommentId()>lastCommentId){
                   lastCommentId=myComment.getCommentId();
              }
         }
        }catch(Exception e){
             e.printStackTrace();
       }

}

/**
* Name: getPatient()
* Komponent: Persistence
* Function: getPatient()
* Input: -
* Output: currentPatient
* Description: Returns the current patient
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//precondition: setPatient(id) has been called
public ElderlyPerson getPatient(){
     return currentPatient;
}

/**
* Name: getTaskList()
* Komponent: Persistence
* Function: getTaskList()
* Input: -
* Output: currentTaskList
* Description: Sets and returns currentTaskList for the selected Eld-
erlyPerson,
* Assumes the undone task list with lower id as the next task list to
be done.
*     1. Initializes currentTaskList=null;
*     2. For each task list (myTaskList)
*          if (first task list) then initializes id
*          else if (task list id <id) and (task list state = undone)
then
*                     sets id=task list id, currentTask-
List=myTaskList; numberTasks
*     3. Return currentTaskList
* Variables:
```

```
*     name: myTaskList
*     description: task list
*     type:TaskList
*
*     name: erste
*     description: first task list
*     type:boolean
*
* Last modificaction: 21.07.2007
* Test cases:
* **/
//precondition: setPatient(id) has been called
public TaskList getTaskList(){
      boolean taskListSelected=false;
      currentTaskList=null;

      TaskList myTaskList=new TaskList();
      boolean erste= true;
      int id=0;
      Iterator myIterator = currentPatient.getTaskLists().iterator();
      while (myIterator.hasNext()) {
             myTaskList= (TaskList) myIterator.next();
             //finds the first undone TaskList in the Collection
             if (erste && (myTaskList.getState()==false)){
                   id=myTaskList.getTaskListId();
                   currentTaskList =myTaskList;
                   erste=false;
                   taskListSelected=true;
             }
             //looks for undone TaskList with smaller ID
             else{
                   if ((myTaskList.getTaskListId()<id) && (myTask-
List.getState()==false)) {
                          id=myTaskList.getTaskListId();
                          currentTaskList =myTaskList;
                   }
             }
     }
      numberTasks= currentTaskList.getTasks().size();

      if(!taskListSelected)
            throw new NoSuchElementException("There are no unodne
TaskLists for the currentPatient");

      return currentTaskList;

}

/**
```

```
* Name: storeTask()
* Komponent: Persistence
* Function: storeTask()
* Input: -
* Output: -
* Description: Sets task id and adds it to the current task list
*     1. Sets task id
*     2. Adds task to currentTaskList
*     3. Updates numberTasks
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//preconditions: setPatient(id) and getTaskList() has been called
public void storeTask(Task t){
      t.setTaskId(numberTasks+1);
      currentTaskList.addTask(t);
      numberTasks++;
}

/**
* Name: updateTask()
* Komponent: Persistence
* Function: updateTask()
* Input:
*     name:tId
*     description: id of the selected task
*     type: int
*
*     name:newstate
*     description: task new state
*     type: int

* Output: -
* Description: Updates the selected task state as newstate
*     1. Search selected task (myTask)in the currect tas list (cur-
rentTaskList)
*     2. Updates task state as newstate
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//preconditions: setPatient(id) and getTaskList() has been called

public void updateTask(int tId, int newstate){
      boolean updated=false;
      Task myTask = new Task();
      Iterator myIterator = currentTaskList.getTasks().iterator();
      while (myIterator.hasNext()) {
```

```
            myTask= (Task) myIterator.next();
            if (myTask.getTaskId()==tId){
                  myTask.setState(newstate);
                  updated=true;
            }
      }
      if(!updated)
            throw new NoSuchElementException("Task with taskId=
"+tId+" doesn't exist");

}

/**
* Name: getCommentList()
* Komponent: Persistence
* Function: getCommentList()
* Input: -
* Output:
*     name: comm
*     description: list of comments
*     type: CommentShorInfo
*
* Description: Returns comment list (summary)
*     1. Copy the comments (complete information)in an array (com-
ments)
*     2. For each comment (c) in the array, adds  the comment to the
summary list of short
*     comment (comm)
*     3. Returns comment list (comm)
*
* Variables: -
*     name: comments
*     description: copy of the current comment list
*     type: Comment[]
*
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection<CommentShortInfo> getCommentList(){
      Set<CommentShortInfo> comm = new HashSet();
      Comment[] comments =new Comment[numberComments];
      System.arraycopy((currentPatient.getComments()).toArray(), 0,
comments, 0, numberComments);

      for(int i=0; i<numberComments;i++){
            CommentShortInfo c= new Com-
mentShortInfo(comments[i].getCommentId(),
String.valueOf(comments[i].getLocation())
,comments[i].getDescription(), comments[i].getCommentDate());
```

```
            comm.add(c);
        }
        //in case that there are no comments for the currentPatient
stored: returnes empty Collection
        return comm;
}

/**
* Name: getComment()
* Komponent: Persistence
* Function: getComment()
* Input:
*      name: comId
*      description: selected comment id
*      type: int
*
* Output:
*      name: myComment
*      description: selected comment
*      type: Comment
*
* Description: Searchs and returns the selected comment
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Comment getComment(int comId){

        Comment  myComment= new Comment();
        Comment  c= new Comment();
        Iterator myIterator = currentPatient.getComments().iterator();
        boolean found=false;
        while (myIterator.hasNext()) {
               c= (Comment) myIterator.next();
               if (c.getCommentId()==comId){
                     myComment=c;
                     found=true;
               }
         }
        if (found){
             return myComment;
             }else {
                   throw new NoSuchElementException("Comment with com-
mentId=  "+comId+" doesn't exist");
                   //return null;
                   }
}
```

```
/**
* Name: storeComment()
* Komponent: Persistence
* Function: storeComment()
* Input:
*     name: com
*     description: new comment
*     type: Comment
*
* Output: -
* Description: Adds a comment to current list comments.
*     1. Adds a comment (comm)
*     2. Updates numberComments
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeComment(Comment com){
      com.setCommentId(lastCommentId+1);
      currentPatient.addComment(com);
      numberComments++;
      lastCommentId++;
}

/** Name: deleteComment()
* Komponent: Persistence
* Function: deleteComment()
* Input:
*     name: comId
*     description: selected comment id
*     type: int
*
* Output: -
* Description: Remove the selected comment
*     1. Removes selected comment
*     2. Updates numberComments
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void deleteComment(int comId){
      boolean deleted=false;
      Comment  c= new Comment();
      Iterator myIterator = currentPatient.getComments().iterator();
      while (myIterator.hasNext()) {
             c= (Comment) myIterator.next();
             if (c.getCommentId()==comId){
                  myIterator.remove();
                  numberComments--;
```

```
                  deleted=true;
            }
      }
      if(!deleted)
            throw new NoSuchElementException("Comment with commentId=
"+comId+" doesn't exist");
}

/**
* Name: getInformationList()
* Komponent: Persistence
* Function: getInformationList()
* Input: -
* Output: Information List
* Description: Returns Information list
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection getInformationList(){
      return currentPatient.getInformations();
}

/**Name: storeInformation()
* Komponent: Persistence
* Function: storeInformation()
* Input:
*     name: info
*     description: new information
*     type: Information
*
* Output: -
* Description: Adds an information to current list comments.
*     1. Adds an Information
*     2. Updates numberInformations
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeInformation(Information info){
      info.setInformationId(numberInformations+1);
      currentPatient.addInformation(info);
      numberInformations++;
}

/**Name: storeData
* Komponent: Persistence
* Function: storeData
* Input:
```

```
*     name: elderlyPersons
*     description: list of elderly person, including care tasks, com-
ments and informations
*     type: Information
*
* Output: -
* Description: Store the list of elderly person for the current day
*     1. Sets n (number of elderly persons)
*     2. Sets the List ElderlyPersonShortInfo (epData)
*     3. Stores the summary file of elderly persons (ElderlyPer-
sonsMap.txt) based on epData
*     4. Stores each elderly person into patientID.txt;
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeData(Collection elderlyPersons){

      n = elderlyPersons.size();

      //Copies an array from the specified source collection
      ElderlyPerson[] ePersons = new ElderlyPerson[n];
    System.arraycopy(elderlyPersons.toArray(), 0, ePersons, 0, n );

    //creates and writes the ElderlyPersonShortInfo - needed for the
getPatientList()
      Set<ElderlyPersonShortInfo> epData = new HashSet();
      ElderlyPersonShortInfo eData=new ElderlyPersonShortInfo();
      for(int i=0;i<n;i++){
            eData= new ElderlyPerson-
ShortInfo(ePersons[i].getPatientId(),ePersons[i].getName(), ePer-
sons[i].getAddress());
      epData.add(eData);
      }

      try{
            FileOutputStream fos = new FileOutput-
Stream("ElderlyPersonsMap.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(epData);
         oos.flush();
         oos.close();
    } catch(Exception e) {
      e.printStackTrace();
    }

    //writes every ElderlyPerson in a separate txt-file
    for(int i=0;i<n;i++){
      try{
```

```
            int id= ePersons[i].getPatientId();
                  FileOutputStream fos = new FileOutputStream(id
+".txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ePersons[i]);
            oos.flush();
            oos.close();

      }catch(Exception e){
             e.printStackTrace();
         }

    }

}

/**Name: loadData
* Komponent: Persistence
* Function: loadData
* Input:
* Output:
*     name: elderlyPersons
*     description: list of elderly person, including care tasks, com-
ments and informations
*     type: Collection
*
* Description: Returns the current list of elderly person
*     1. Reads ElderlyPersonsMap.txt and sets n (number of elderly
persons)
*     2. For each elderly person
*           Reads elderly person (patient)from the patientId.txt file
*           Adds elderly person to elderly persons set (elderlyPer-
sons)
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection loadData(){


      Set<ElderlyPersonShortInfo> epData = new HashSet();
      try {
       FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       epData = (Set<ElderlyPersonShortInfo>)ois.readObject();
       ois.close();
       }catch(Exception e){
          e.printStackTrace();
```

```
    }
     //counts the ElderlyPerson entrys
    n=0;;
    ElderlyPersonShortInfo myPatient=new ElderlyPersonShortInfo();

    Iterator myIterator = epData.iterator();
    while (myIterator.hasNext()) {
          myPatient= (ElderlyPersonShortInfo) myIterator.next();
          n++; }

    //reads all ElderlyPerson
    Set elderlyPersons = new HashSet();
    ElderlyPerson patient =new ElderlyPerson();
    elderlyPersons=null;
    try{
          for(int i=0;i<n;i++){
                FileInputStream fis = new FileInput-
Stream((i+1)+".txt");
                ObjectInputStream ois = new ObjectInputStream(fis);

                patient = (ElderlyPerson )ois.readObject();
                elderlyPersons.add(patient);
                ois.close();
               }
         }catch(Exception e){
                e.printStackTrace();
        }

         return elderlyPersons;

    }



/**Name: markTaskListDone()
* Komponent: Persistence
* Function: markTaskListDone()
* Input: -
* Output:
* Description: Stores the elderly person information that has been
added/changed during  last visit
*     1. Updates task list state as done
*     2. Updates elderly person file (patientId.txt) based on current-
Patient data
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void markTaskListDone(){
```

```
            currentTaskList.setState(true);
            try{
            FileOutputStream fos = new FileOutputStream( currentPa-
tient.getPatientId() +".txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(currentPatient);
          oos.flush();
          oos.close();
      }catch(Exception e){
             e.printStackTrace();
        }

      }

/**Name: setLastVisit()
* Komponent: Persistence
* Function: (No reference)
* Input: -
* Output:  date
* Description: Set the date of the last visit to the current elderly
person
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void setLastVisit(Date date){
            currentPatient.setLastVisit(date);
      }

/** Name: loadRoomMapping()
* Komponent: Persistence
* Function: loadRoomMapping()
* Input: -
* Output:
* Description:
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//the method parameters were sujested by the location manager...
public boolean loadRoomMapping(int apartmentId, Array-
List<MappingItem> data){
            System.arraycopy(currentPatient.getAppMap(), 0, data, 0,
currentPatient.getAppMap().size());
            return true;
      }


}
```

1.7.8 Exercise to Experience Package for Persistence Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type embedded in name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Persistence:persistenceImpl.java

```
package org.belami.dcga.persistence;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.*;

import org.belami.dcga.common_datastructures.*;

class PersistenceImpl implements Persistence{
/**Overview
 *Project: DCGA, Summer semester 2007, GSE-Project, Technische Universität Kaiserslautern
 *Subsystem: Persistence
 *Desing version: Persistence.doc (Date: --/06/2007)
 *Last modification: 21.06.2007
 **/

/**  Atributes:
 *   int n: Number of elderly persons that are stored (number of patients).
 *   ElderlyPerson curentPatient: Temporary copy of the elderly person selected by the care giver.
 *   TaskList currentTaskList: First uncompleted Task List associated the selected elderly person.
 *   int numberTasks: number of tasks of the current Task List.
```

```
 *    int numberInformations: number of informations associated to the
selected elderly person.
 *    int numberComments: number of comments associated to the se-
lected elderly person.
 *    int lastCommentId: Id associated to the last stored comment.
 **/

ElderlyPerson currentPatient = new ElderlyPerson();
int n=0;
TaskList currentTaskList =new TaskList();
int numberTasks=0;
int numberInformations=0;
int numberComments=0;
int lastCommentId=0;

/** Methods:**/

/** Name: getPatientList()
 * Komponent: Persistence
 * Function: getPatientList()
 * Input : -
 * Output:
 *    name: epData
 *    description: : List of current patients
 *    type: Set<ElderlyPersonShortInfo>
 *
 * Description:
 *    1. Reads ElderlyPersonsShortInformation list (epData) from Eld-
erlyPersonMap.text file
 *    2. Sets the number of patients (n)
 *    3. Return List of current patients (epData)
 *
 *  Variables:
 *          name : eData
 *          description: Summarize of the elderly patient information:
id, name, address,
 *          type: ElderlyPersonsShortInformation
 *
 * Last modificaction: 21.07.2007
 * Test cases:
 * **/
public Collection getPatientList(){

      Set<ElderlyPersonShortInfo> epData = new HashSet();
      try {
       FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       epData = (Set<ElderlyPersonShortInfo>)ois.readObject();
```

```
         ois.close();
         }catch(Exception e){
            e.printStackTrace();
        }

        n=0;
        ElderlyPersonShortInfo myPatient=new ElderlyPersonShortInfo();

        Iterator myIterator = epData.iterator();
        while (myIterator.hasNext()) {
              myPatient= (ElderlyPersonShortInfo) myIterator.next();
              n++; }

        return epData;
}

/**
* Name: setPatientId()
* Komponent: Persistence
* Function: setPatientId()
* Input:
*     name:pId
*     description: id of the selected elderly person
*     type: int
* Output: -
* Description: Sets the data of the currentPatient
*     1. Reads the data of the selected elderly person from the file
pID.txt
*     2. Updates currentPatient
*     3. Sets numberInformations, numberComments
*     5. Searchs and sets lastCommentId
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void setPatientId(int pId){
        currentPatient=null;
        try{
        FileInputStream fis = new FileInputStream(pId+".txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
      currentPatient = (ElderlyPerson )ois.readObject();
      ois.close();

      numberInformations= currentPatient.getInformations().size();
        numberComments= currentPatient.getComments().size();

        //checks for the highest commentId (if some comment was deleted,
its Id, won´t be used any more)
        lastCommentId= numberComments;
```

```
      Comment myComment=new Comment();
      Iterator myIterator = currentPatient.getComments().iterator();
      while (myIterator.hasNext()) {
            myComment = (Comment) myIterator.next();
            if (myComment.getCommentId()>lastCommentId){
                  lastCommentId=myComment.getCommentId();
            }
       }
      }catch(Exception e){
            e.printStackTrace();
      }

}

/**
* Name: getPatient()
* Komponent: Persistence
* Function: getPatient()
* Input: -
* Output: currentPatient
* Description: Returns the current patient
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//precondition: setPatient(id) has been called
public ElderlyPerson getPatient(){
      return currentPatient;
}

/**
* Name: getTaskList()
* Komponent: Persistence
* Function: getTaskList()
* Input: -
* Output: currentTaskList
* Description: Sets and returns currentTaskList for the selected Eld-
erlyPerson,
* Assumes the undone task list with lower id as the next task list to
be done.
*     1. Initializes currentTaskList=null;
*     2. For each task list (myTaskList)
*           if (first task list) then initializes id
*           else if (task list id <id) and (task list state = undone)
then
*                       sets id=task list id, currentTask-
List=myTaskList; numberTasks
*     3. Return currentTaskList
* Variables:
```

```
*     name: myTaskList
*     description: task list
*     type:TaskList
*
*     name: erste
*     description: first task list
*     type:boolean
*
* Last modificaction: 21.07.2007
* Test cases:
* **/
//precondition: setPatient(id) has been called
public TaskList getTaskList(){
      boolean taskListSelected=false;
      currentTaskList=null;

      TaskList myTaskList=new TaskList();
      boolean erste= true;
      int id=0;
      Iterator myIterator = currentPatient.getTaskLists().iterator();
      while (myIterator.hasNext()) {
             myTaskList= (TaskList) myIterator.next();
             //finds the first undone TaskList in the Collection
             if (erste && (myTaskList.getState()==false)){
                   id=myTaskList.getTaskListId();
                   currentTaskList =myTaskList;
                   erste=false;
                   taskListSelected=true;
             }
             //looks for undone TaskList with smaller ID
             else{
                   if ((myTaskList.getTaskListId()<id) && (myTask-
List.getState()==false)) {
                         id=myTaskList.getTaskListId();
                         currentTaskList =myTaskList;
                   }
             }
    }
      numberTasks= currentTaskList.getTasks().size();

      if(!taskListSelected)
            throw new NoSuchElementException("There are no unodne
TaskLists for the currentPatient");

      return currentTaskList;

}

/**
```

```
* Name: storeTask()
* Komponent: Persistence
* Function: storeTask()
* Input: -
* Output: -
* Description: Sets task id and adds it to the current task list
*     1. Sets task id
*     2. Adds task to currentTaskList
*     3. Updates numberTasks
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//preconditions: setPatient(id) and getTaskList() has been called
public void storeTask(Task t){
      t.setTaskId(numberTasks+1);
      currentTaskList.addTask(t);
      numberTasks++;
}

/**
* Name: updateTask()
* Komponent: Persistence
* Function: updateTask()
* Input:
*     name:tId
*     description: id of the selected task
*     type: int
*
*     name:newstate
*     description: task new state
*     type: int

* Output: -
* Description: Updates the selected task state as newstate
*     1. Search selected task (myTask)in the currect tas list (cur-
rentTaskList)
*     2. Updates task state as newstate
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//preconditions: setPatient(id) and getTaskList() has been called

public void updateTask(int tId, int newstate){
      boolean updated=false;
      Task myTask = new Task();
      Iterator myIterator = currentTaskList.getTasks().iterator();
      while (myIterator.hasNext()) {
```

```
            myTask= (Task) myIterator.next();
            if (myTask.getTaskId()==tId){
                 myTask.setState(newstate);
                 updated=true;
            }
       }
      if(!updated)
           throw new NoSuchElementException("Task with taskId=
"+tId+" doesn't exist");

}

/**
* Name: getCommentList()
* Komponent: Persistence
* Function: getCommentList()
* Input: -
* Output:
*     name: comm
*     description: list of comments
*     type: CommentShorInfo
*
* Description: Returns comment list (summary)
*     1. Copy the comments (complete information)in an array (com-
ments)
*     2. For each comment (c) in the array, adds  the comment to the
summary list of short
*     comment (comm)
*     3. Returns comment list (comm)
*
* Variables: -
*     name: comments
*     description: copy of the current comment list
*     type: Comment[]
*
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection<CommentShortInfo> getCommentList(){
      Set<CommentShortInfo> comm = new HashSet();
      Comment[] comments =new Comment[numberComments];
      System.arraycopy((currentPatient.getComments()).toArray(), 0,
comments, 0, numberComments);

      for(int i=0; i<numberComments;i++){
           CommentShortInfo c= new Com-
mentShortInfo(comments[i].getCommentId(),
String.valueOf(comments[i].getLocation())
,comments[i].getDescription(), comments[i].getCommentDate());
```

```
            comm.add(c);
        }
        //in case that there are no comments for the currentPatient
stored: returnes empty Collection
        return comm;
}

/**
* Name: getComment()
* Komponent: Persistence
* Function: getComment()
* Input:
*       name: comId
*       description: selected comment id
*       type: int
*
* Output:
*       name: myComment
*       description: selected comment
*       type: Comment
*
* Description: Searchs and returns the selected comment
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Comment getComment(int comId){

        Comment  myComment= new Comment();
        Comment  c= new Comment();
        Iterator myIterator = currentPatient.getComments().iterator();
        boolean found=false;
        while (myIterator.hasNext()) {
                c= (Comment) myIterator.next();
                if (c.getCommentId()==comId){
                        myComment=c;
                        found=true;
                }
         }
        if (found){
            return myComment;
            }else {
                throw new NoSuchElementException("Comment with com-
mentId=  "+comId+" doesn't exist");
                //return null;
                }
}
```

```
/**
* Name: storeComment()
* Komponent: Persistence
* Function: storeComment()
* Input:
*     name: com
*     description: new comment
*     type: Comment
*
* Output: -
* Description: Adds a comment to current list comments.
*     1. Adds a comment (comm)
*     2. Updates numberComments
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeComment(Comment com){
      com.setCommentId(lastCommentId+1);
      currentPatient.addComment(com);
      numberComments++;
      lastCommentId++;
}

/** Name: deleteComment()
* Komponent: Persistence
* Function: deleteComment()
* Input:
*     name: comId
*     description: selected comment id
*     type: int
*
* Output: -
* Description: Remove the selected comment
*     1. Removes selected comment
*     2. Updates numberComments
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void deleteComment(int comId){
      boolean deleted=false;
      Comment  c= new Comment();
      Iterator myIterator = currentPatient.getComments().iterator();
      while (myIterator.hasNext()) {
             c= (Comment) myIterator.next();
             if (c.getCommentId()==comId){
                  myIterator.remove();
                  numberComments--;
```

```
                    deleted=true;
                }
            }
        if(!deleted)
            throw new NoSuchElementException("Comment with commentId=
"+comId+" doesn't exist");
}

/**
* Name: getInformationList()
* Komponent: Persistence
* Function: getInformationList()
* Input: -
* Output: Information List
* Description: Returns Information list
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection getInformationList(){
      return currentPatient.getInformations();
}

/**Name: storeInformation()
* Komponent: Persistence
* Function: storeInformation()
* Input:
*     name: info
*     description: new information
*     type: Information
*
* Output: -
* Description: Adds an information to current list comments.
*     1. Adds an Information
*     2. Updates numberInformations
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeInformation(Information info){
      info.setInformationId(numberInformations+1);
      currentPatient.addInformation(info);
      numberInformations++;
}

/**Name: storeData
* Komponent: Persistence
* Function: storeData
* Input:
```

```
*     name: elderlyPersons
*     description: list of elderly person, including care tasks, com-
ments and informations
*     type: Information
*
* Output: -
* Description: Store the list of elderly person for the current day
*     1. Sets n (number of elderly persons)
*     2. Sets the List ElderlyPersonShortInfo (epData)
*     3. Stores the summary file of elderly persons (ElderlyPer-
sonsMap.txt) based on epData
*     4. Stores each elderly person into patientID.txt;
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void storeData(Collection elderlyPersons){

      n = elderlyPersons.size();

      //Copies an array from the specified source collection
      ElderlyPerson[] ePersons = new ElderlyPerson[n];
    System.arraycopy(elderlyPersons.toArray(), 0, ePersons, 0, n );

    //creates and writes the ElderlyPersonShortInfo - needed for the
getPatientList()
      Set<ElderlyPersonShortInfo> epData = new HashSet();
      ElderlyPersonShortInfo eData=new ElderlyPersonShortInfo();
      for(int i=0;i<n;i++){
            eData= new ElderlyPerson-
ShortInfo(ePersons[i].getPatientId(),ePersons[i].getName(), ePer-
sons[i].getAddress());
      epData.add(eData);
      }

      try{
            FileOutputStream fos = new FileOutput-
Stream("ElderlyPersonsMap.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(epData);
         oos.flush();
         oos.close();
    } catch(Exception e) {
      e.printStackTrace();
    }

    //writes every ElderlyPerson in a separate txt-file
    for(int i=0;i<n;i++){
      try{
```

```
            int id= ePersons[i].getPatientId();
                  FileOutputStream fos = new FileOutputStream(id
+".txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ePersons[i]);
            oos.flush();
            oos.close();

      }catch(Exception e){
             e.printStackTrace();
         }

    }

}

/**Name: loadData
* Komponent: Persistence
* Function: loadData
* Input:
* Output:
*     name: elderlyPersons
*     description: list of elderly person, including care tasks, com-
ments and informations
*     type: Collection
*
* Description: Returns the current list of elderly person
*     1. Reads ElderlyPersonsMap.txt and sets n (number of elderly
persons)
*     2. For each elderly person
*           Reads elderly person (patient)from the patientId.txt file
*           Adds elderly person to elderly persons set (elderlyPer-
sons)
* Variables:
* Last modificaction: 21.07.2007
* Test cases:
* **/
public Collection loadData(){


      Set<ElderlyPersonShortInfo> epData = new HashSet();
      try {
       FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       epData = (Set<ElderlyPersonShortInfo>)ois.readObject();
       ois.close();
       }catch(Exception e){
          e.printStackTrace();
```

```
      }
       //counts the ElderlyPerson entrys
      n=0;;
      ElderlyPersonShortInfo myPatient=new ElderlyPersonShortInfo();

      Iterator myIterator = epData.iterator();
      while (myIterator.hasNext()) {
            myPatient= (ElderlyPersonShortInfo) myIterator.next();
            n++; }

      //reads all ElderlyPerson
      Set elderlyPersons = new HashSet();
      ElderlyPerson patient =new ElderlyPerson();
      elderlyPersons=null;
      try{
            for(int i=0;i<n;i++){
                  FileInputStream fis = new FileInput-
Stream((i+1)+".txt");
                  ObjectInputStream ois = new ObjectInputStream(fis);

                  patient = (ElderlyPerson )ois.readObject();
                  elderlyPersons.add(patient);
                  ois.close();
                 }
           }catch(Exception e){
                  e.printStackTrace();
          }

           return elderlyPersons;

      }


/**Name: markTaskListDone()
* Komponent: Persistence
* Function: markTaskListDone()
* Input: -
* Output:
* Description: Stores the elderly person information that has been
added/changed during  last visit
*     1. Updates task list state as done
*     2. Updates elderly person file (patientId.txt) based on current-
Patient data
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void markTaskListDone(){
```

```
            currentTaskList.setState(true);
            try{
            FileOutputStream fos = new FileOutputStream( currentPa-
tient.getPatientId() +".txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(currentPatient);
          oos.flush();
          oos.close();
      }catch(Exception e){
             e.printStackTrace();
         }

      }

/**Name: setLastVisit()
* Komponent: Persistence
* Function: (No reference)
* Input: -
* Output:  date
* Description: Set the date of the last visit to the current elderly
person
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
public void setLastVisit(Date date){
            currentPatient.setLastVisit(date);
      }

/** Name: loadRoomMapping()
* Komponent: Persistence
* Function: loadRoomMapping()
* Input: -
* Output:
* Description:
* Variables: -
* Last modificaction: 21.07.2007
* Test cases:
* **/
//the method parameters were sujested by the location manager...
public boolean loadRoomMapping(int apartmentId, Array-
List<MappingItem> data){
            System.arraycopy(currentPatient.getAppMap(), 0, data, 0,
currentPatient.getAppMap().size());
            return true;
      }


}
```

1.7.9 Exercise to Experience Package for Synchronization Group: Long Method

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Persistence:persistenceImpl.java → in your code code smells of long method couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.persistence;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.HashSet;
import java.util.Set;
import java.util.*;
import org.belami.dcga.common_datastructures.*;

class PersistenceImpl implements Persistence{


//delcare variables needed to handle the currentPatient


//creates a temporary copy of the ElderlyPerson
//data will be added when setPatient() is called
ElderlyPerson curentPatient = new ElderlyPerson();

//number of ElderlyPersons stored
int n=0;
```

```
//currentTaskList Object
TaskList currentTaskList =new TaskList();

//number of tasks in the currentTaskList, number of Comments and In-
formations for the currentPatient
int numberTasks=0;
int numberInformations=0;

//comments can be deleted (the ID of deleted comment will not be used
for that ElderlyPerson for that day
int numberComments=0;
int lastCommentId=0;

public Collection getPatientList(){
     //returns List of ElderlyPerson`s patientId,name
     //information is stored when storeData() called, number of EP-
Data Objects = n

     //the List that will be returned
     Set<ElderlyPersonShortInfo> epData = new HashSet();

     //reads the number of EPs
     Integer cant = new Integer(0);
     try {
      FileInputStream fis = new FileInput-
Stream("NumberOfElderlyPersons.txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
      cant= (Integer)ois.readObject();
      n = cant.intValue();
      ois.close();
      }catch(Exception e){
         e.printStackTrace();
     }


     //reads all ElderlyPersons from the txt-files


           try{
                FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
                ObjectInputStream ois = new ObjectInputStream(fis);
                ElderlyPersonShortInfo eData =new ElderlyPerson-
ShortInfo();
                for(int i=0;i<n;i++){
                     eData = (ElderlyPerson-
ShortInfo)ois.readObject();
                     epData.add(eData);
```

```
                }
                ois.close();
            }catch(Exception e){
             e.printStackTrace();
         }

      return epData;
}
public void setPatientId(int pId){
      //reads the currentPatient from the (pId).txt


      try{
      FileInputStream fis = new FileInputStream(pId+".txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
    curentPatient = (ElderlyPerson )ois.readObject();
    ois.close();
            }catch(Exception e){
            e.printStackTrace();
        }
       numberInformations= curentPatient.getInformations().size();
       numberComments= curentPatient.getComments().size();

       //retrieves the last commentId (example for commentList with
IDs: 1,2,5,6 (3,4 were deleted)
       lastCommentId= numberComments;
       Comment[] comments =new Comment[numberComments];
       System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, curentPatient.getComments().size());

       for (int i=0;i<numberComments;i++){
            if (comments[i].getCommentId()>lastCommentId)
                 lastCommentId=comments[i].getCommentId();
          }


}
public ElderlyPerson getPatient(){
      //return currently ElderlyPerson
      return curentPatient;
}

public TaskList getTaskList(){
      //retrieve current TaskList and returns it


      if (curentPatient.getTaskLists().size()>0) {
      TaskList[] tLists = new Task-
List[curentPatient.getTaskLists().size()];
```

```
      System.arraycopy(curentPatient.getTaskLists().toArray(), 0,
tLists, 0, curentPatient.getTaskLists().size());

      //selects the current TaskList from the array tLists[] and makes
a reference to currentTaskList
      //looks for the TaskList with the smallest TaskListId that is
still unfinished

            int Id = tLists[0].getTaskListId();
            int pos= 0;
            for(int i=1;i<curentPatient.getTaskLists().size();i++){
                  if ((tLists[i].getTaskListId()< Id)&&
(tLists[i].getState() == false)){
                        Id = tLists[i].getTaskListId();
                        pos=i;
                        }
            }
            currentTaskList = tLists[pos];
            numberTasks = tLists[pos].getTasks().size();
      //counts the Tasks in the currentTaskList

    }else {
      currentTaskList= null;
      numberTasks =0;
    }
      System.out.println(numberTasks);
    return currentTaskList;

}

public void storeTask(Task t){
      // retrieve and set taskId, set current TaskListId, create a
Task and stores it
      t.setTaskId(numberTasks+1);
      currentTaskList.addTask(t);
      numberTasks++;
}
public void updateTask(int tId, int newstate){
      // update Task with taskId==tId state=newstate

      Task[] tasks =new Task[numberTasks];
      System.arraycopy((currentTaskList.getTasks()).toArray(), 0,
tasks, 0,numberTasks-1);
      for(int i=0;i<numberTasks;i++){
            if(tasks[i].getTaskId()==tId){
                  tasks[i].setState(newstate);
                  break;
            }
```

```
      }

}

public Collection<CommentShortInfo> getCommentList(){
      //returns a Collection of CommentShortInfo for the cureent Eld-
erlyPerson
      Set<CommentShortInfo> comm = new HashSet();

      Comment[] comments =new Comment[numberComments];
      System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, numberComments);
      for(int i=0; i<numberComments;i++){
            String mylocation = ""+ comments[i].getLocation();
            CommentShortInfo c= new Com-
mentShortInfo(comments[i].getCommentId(), mylocation
,comments[i].getDescription(), comments[i].getCommentDate());
            comm.add(c);
      }
      return comm;
}

public Comment getComment(int comId){
      //returns Comment with commentId=comId
      Comment[] comments =new Comment[numberComments];
      System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, curentPatient.getComments().size());

    for (int i=0;i<numberComments;i++){
      if (comments[i].getCommentId()==comId)
            return comments[i];
    }
    throw new NoSuchElementException("Doen't exist");
}

public void storeComment(Comment com){
      //retrieves commonId, create a Comment with description, and
stores it
      com.setCommentId(lastCommentId+1);
      curentPatient.addComment(com);
      numberComments++;
      lastCommentId++;
}
//when a Comment is deleted, there will be no Comment will comId for
that person any more
//the free comId won`t be set to another Comment

public void deleteComment(int comId){
      //deletes the Comment with commentId==comId from the database
```

```
        //finds the comment to be deleted
        Comment[] comments =new Comment[numberComments];
        Comment myComment =new Comment(-1);
        System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, curentPatient.getComments().size());
        for (int i=0;i<curentPatient.getComments().size();i++){
        if (comments[i].getCommentId()==comId){
                myComment=comments[i];
                numberComments--;
                break;
        }
      }
        //deletes the Comment from the Collection
        if (myComment.getCommentId()>0) {
                curentPatient.getComments().remove(myComment);
        }

    // if the comment with comId doesn`t exist:  trows new NoSuchEle-
ment();
}

public Collection getInformationList(){
        //returns a Collection of Information about the current Elderly-
Person
        return curentPatient.getInformations();
}

public void storeInformation(Information info){
        //retrieves an informationId, creates an Information with de-
scription=descr and stores it
        info.setInformationId(numberInformations+1);
        curentPatient.addInformation(info);
        numberInformations++;
}
public void storeData(Collection elderlyPersons){
        //stores the Collection elderlyPersons

        //    transform the Collection of ElderlyPersons to an array
        n = elderlyPersons.size();
        ElderlyPerson[] ePersons = new ElderlyPerson[n];
      System.arraycopy(elderlyPersons.toArray(), 0, ePersons, 0, n );


      //write a ElderlyPersonsMap.txt containig for all ElderlyPerson:
Id, name, address
      try{
        FileOutputStream fos = new FileOutput-
Stream("ElderlyPersonsMap.txt");
```

```
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      ElderlyPersonShortInfo[] epData =new ElderlyPersonShortInfo[n];
         for(int i=0;i<n;i++){
            epData[i]= new ElderlyPerson-
ShortInfo(ePersons[i].getPatientId(),ePersons[i].getName(), ePer-
sons[i].getAddress());
            oos.writeObject(epData[i]);
           oos.flush();
         };
         oos.close();

         }catch(Exception e){
             e.printStackTrace();
         }

      //writes the number of EPs

      Integer num = new Integer(elderlyPersons.size());
     try{
      FileOutputStream fos = new FileOutput-
Stream("NumberOfElderlyPersons.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(num);
            oos.flush();
            oos.close();

         }catch(Exception e){
             e.printStackTrace();
         }


   //writtes for every ElderlyPerson seperate file: (patientId).txt
     for(int i=0;i<n;i++){
      try{
            int id= ePersons[i].getPatientId();

            FileOutputStream fos = new FileOutputStream(id +".txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ePersons[i]);
            oos.flush();
            oos.close();

      }catch(Exception e){
             e.printStackTrace();
         }

     }
 }
public Collection loadData(){
```

```
	//returns a Collection of elderlyPersons

	//number of stored ElderlyPerson: - n (the number is set during
the storeData())

	//the Collection that will be returned
	Set elderlyPersons = new HashSet();
	// set number of elderlyPersons

	Integer cant = new Integer(0);
	try {
	 FileInputStream fis = new FileInput-
Stream("NumberOfElderlyPersons.txt");
	 ObjectInputStream ois = new ObjectInputStream(fis);
	 cant= (Integer)ois.readObject();
	 n = cant.intValue();
	 ois.close();
	 }catch(Exception e){
	    e.printStackTrace();
	}
	//reads all ElderlyPersons from the txt-files
		try{


	 for(int i=0;i<n;i++){
		FileInputStream fis = new FileInputStream((i+1)+".txt");
		ObjectInputStream ois = new ObjectInputStream(fis);

		curentPatient = (ElderlyPerson )ois.readObject();
		elderlyPersons.add(curentPatient);
		ois.close();
	 }

		}catch(Exception e){
		 e.printStackTrace();
	  }

	return elderlyPersons;

}
	//marks currentTaskList as done
	//writtes the changed currentlyPerson down into its txt-file
	public void markTaskListDone(){
		currentTaskList.setState(true);
		try{
		FileOutputStream fos = new FileOutputStream( curentPa-
tient.getPatientId() +".txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(curentPatient);
```

```
            oos.flush();
            oos.close();
        }catch(Exception e){
                e.printStackTrace();
          }

        }

        public void setLastVisit(Date date){
                curentPatient.setLastVisit(date);
        }

        public boolean loadRoomMapping(int apartmentId, Array-
List<MappingItem> data){
                System.arraycopy(curentPatient.getAppMap(), 0, data, 0,
curentPatient.getAppMap().size()-1);
                return true;

        }
}
```

1.7.10 Exercise to Experience Package for Synchronization Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type Embedded in Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Synchronization:SynConnectorImpl.java
```
package org.belami.dcga.synchronization.syncconnector;

import java.io.IOException;

import org.belami.dcga.synchronization.MappedData;

/**
 * Implementation of the SyncConnector Interface.
 * @see SyncConnector
 */
class SyncConnectorImpl implements SyncConnector {
	private ConnectionManager connectionManager = null;

	/**
	 * Associates the only instance of the ConnectionManager to the
SyncConnector.
	 */
	public void initSync() {
		connectionManager = ConnectionManager.getInstance();
	}

	/**
	 * This method writes a MappedData object (download type) into
the OutputStream and
	 * receives the filled MappedData object through the Input-
Stream.
```

```
     *
     * @return data: MappedData filled with data from the Operator-
System
     */
    public MappedData downloadData() {
        MappedData data = new MappedData(true, null);
        try {
            connectionMan-
ager.getOutputStream().writeObject(data);
            data = (MappedData) connectionMan-
ager.getInputStream().readObject();
            System.out.println("Download Ok");
        } catch (IOException e) {
            System.out.println("Download failed: "+
e.getMessage());
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
         return data;
    }

    /**
     * This method writes a MappedData object (upload type) into the
OutputStream.
     */
    public void sendMappedData(MappedData data) {
        try {
            connectionMan-
ager.getOutputStream().writeObject(data);
            String result = (String) connectionMan-
ager.getInputStream().readObject();
            System.out.println(result);

        } catch (IOException e) {
            System.out.println("Upload failed: "+
e.getMessage());
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

    }

    /**
     * Get-method for the ConnectionManager
     *
     * @return connectionManager
     */
```

```
	public ConnectionManager getConnectionManager() {
		return connectionManager;
	}
}
```

1.7.11 Exercise to Experience Package for UI Group: Long Method

Your Name: ________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Long Method

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Persistence:persistenceImpl.java → in your code code smells of long method couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.persistence;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.HashSet;
import java.util.Set;
import java.util.*;
import org.belami.dcga.common_datastructures.*;

class PersistenceImpl implements Persistence{


//delcare variables needed to handle the currentPatient


//creates a temporary copy of the ElderlyPerson
//data will be added when setPatient() is called
ElderlyPerson curentPatient = new ElderlyPerson();

//number of ElderlyPersons stored
int n=0;
```

```
//currentTaskList Object
TaskList currentTaskList =new TaskList();

//number of tasks in the currentTaskList, number of Comments and In-
formations for the currentPatient
int numberTasks=0;
int numberInformations=0;

//comments can be deleted (the ID of deleted comment will not be used
for that ElderlyPerson for that day
int numberComments=0;
int lastCommentId=0;

public Collection getPatientList(){
      //returns List of ElderlyPerson`s patientId,name
      //information is stored when storeData() called, number of EP-
Data Objects = n

      //the List that will be returned
      Set<ElderlyPersonShortInfo> epData = new HashSet();

      //reads the number of EPs
      Integer cant = new Integer(0);
      try {
       FileInputStream fis = new FileInput-
Stream("NumberOfElderlyPersons.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       cant= (Integer)ois.readObject();
       n = cant.intValue();
       ois.close();
       }catch(Exception e){
          e.printStackTrace();
     }


      //reads all ElderlyPersons from the txt-files


            try{
                  FileInputStream fis = new FileInput-
Stream("ElderlyPersonsMap.txt");
                  ObjectInputStream ois = new ObjectInputStream(fis);
                  ElderlyPersonShortInfo eData =new ElderlyPerson-
ShortInfo();
                  for(int i=0;i<n;i++){
                        eData = (ElderlyPerson-
ShortInfo)ois.readObject();
                        epData.add(eData);
```

```
                }
                ois.close();
            }catch(Exception e){
             e.printStackTrace();
         }

      return epData;
}
public void setPatientId(int pId){
      //reads the currentPatient from the (pId).txt


      try{
      FileInputStream fis = new FileInputStream(pId+".txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
    curentPatient = (ElderlyPerson )ois.readObject();
    ois.close();
            }catch(Exception e){
            e.printStackTrace();
        }
       numberInformations= curentPatient.getInformations().size();
       numberComments= curentPatient.getComments().size();

       //retrieves the last commentId (example for commentList with
IDs: 1,2,5,6 (3,4 were deleted)
       lastCommentId= numberComments;
       Comment[] comments =new Comment[numberComments];
       System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, curentPatient.getComments().size());

       for (int i=0;i<numberComments;i++){
            if (comments[i].getCommentId()>lastCommentId)
                  lastCommentId=comments[i].getCommentId();
          }


}
public ElderlyPerson getPatient(){
      //return currently ElderlyPerson
      return curentPatient;
}

public TaskList getTaskList(){
      //retrieve current TaskList and returns it


      if (curentPatient.getTaskLists().size()>0) {
      TaskList[] tLists = new Task-
List[curentPatient.getTaskLists().size()];
```

```
      System.arraycopy(curentPatient.getTaskLists().toArray(), 0,
tLists, 0, curentPatient.getTaskLists().size());

      //selects the current TaskList from the array tLists[] and makes
a reference to currentTaskList
      //looks for the TaskList with the smallest TaskListId that is
still unfinished

            int Id = tLists[0].getTaskListId();
            int pos= 0;
            for(int i=1;i<curentPatient.getTaskLists().size();i++){
                  if ((tLists[i].getTaskListId()< Id)&&
(tLists[i].getState() == false)){
                        Id = tLists[i].getTaskListId();
                        pos=i;
                        }
            }
            currentTaskList = tLists[pos];
            numberTasks = tLists[pos].getTasks().size();
      //counts the Tasks in the currentTaskList

    }else {
      currentTaskList= null;
      numberTasks =0;
    }
      System.out.println(numberTasks);
    return currentTaskList;

}

public void storeTask(Task t){
      // retrieve and set taskId, set current TaskListId, create a
Task and stores it
      t.setTaskId(numberTasks+1);
      currentTaskList.addTask(t);
      numberTasks++;
}
public void updateTask(int tId, int newstate){
      // update Task with taskId==tId state=newstate

      Task[] tasks =new Task[numberTasks];
      System.arraycopy((currentTaskList.getTasks()).toArray(), 0,
tasks, 0,numberTasks-1);
      for(int i=0;i<numberTasks;i++){
            if(tasks[i].getTaskId()==tId){
                  tasks[i].setState(newstate);
                  break;
            }
```

```
        }

}

public Collection<CommentShortInfo> getCommentList(){
        //returns a Collection of CommentShortInfo for the cureent Eld-
erlyPerson
        Set<CommentShortInfo> comm = new HashSet();

        Comment[] comments =new Comment[numberComments];
        System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, numberComments);
        for(int i=0; i<numberComments;i++){
                String mylocation = ""+ comments[i].getLocation();
                CommentShortInfo c= new Com-
mentShortInfo(comments[i].getCommentId(), mylocation
,comments[i].getDescription(), comments[i].getCommentDate());
                comm.add(c);
        }
        return comm;
}

public Comment getComment(int comId){
        //returns Comment with commentId=comId
        Comment[] comments =new Comment[numberComments];
        System.arraycopy((curentPatient.getComments()).toArray(), 0,
comments, 0, curentPatient.getComments().size());

      for (int i=0;i<numberComments;i++){
        if (comments[i].getCommentId()==comId)
                return comments[i];
      }
      throw new NoSuchElementException("Doen't exist");
}

public void storeComment(Comment com){
        //retrieves commonId, create a Comment with description, and
stores it
        com.setCommentId(lastCommentId+1);
        curentPatient.addComment(com);
        numberComments++;
        lastCommentId++;
}
//when a Comment is deleted, there will be no Comment will comId for
that person any more
//the free comId won`t be set to another Comment

public void deleteComment(int comId){
        //deletes the Comment with commentId==comId from the database
```

```
      //finds the comment to be deleted
      Comment[] comments =new Comment[numberComments];
      Comment myComment =new Comment(-1);
      System.arraycopy((curentPatient.getComments()).toArray(), 0, 
comments, 0, curentPatient.getComments().size());
      for (int i=0;i<curentPatient.getComments().size();i++){
      if (comments[i].getCommentId()==comId){
            myComment=comments[i];
            numberComments--;
            break;
      }
    }
      //deletes the Comment from the Collection
      if (myComment.getCommentId()>0) {
            curentPatient.getComments().remove(myComment);
      }

   // if the comment with comId doesn`t exist:  trows new NoSuchEle-
ment();
}

public Collection getInformationList(){
      //returns a Collection of Information about the current Elderly-
Person
      return curentPatient.getInformations();
}

public void storeInformation(Information info){
      //retrieves an informationId, creates an Information with de-
scription=descr and stores it
      info.setInformationId(numberInformations+1);
      curentPatient.addInformation(info);
      numberInformations++;
}
public void storeData(Collection elderlyPersons){
      //stores the Collection elderlyPersons

      //    transform the Collection of ElderlyPersons to an array
      n = elderlyPersons.size();
      ElderlyPerson[] ePersons = new ElderlyPerson[n];
    System.arraycopy(elderlyPersons.toArray(), 0, ePersons, 0, n );


    //write a ElderlyPersonsMap.txt containig for all ElderlyPerson:
Id, name, address
    try{
      FileOutputStream fos = new FileOutput-
Stream("ElderlyPersonsMap.txt");
```

```
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      ElderlyPersonShortInfo[] epData =new ElderlyPersonShortInfo[n];
        for(int i=0;i<n;i++){
           epData[i]= new ElderlyPerson-
ShortInfo(ePersons[i].getPatientId(),ePersons[i].getName(), ePer-
sons[i].getAddress());
           oos.writeObject(epData[i]);
          oos.flush();
        };
        oos.close();

        }catch(Exception e){
            e.printStackTrace();
        }

      //writes the number of EPs

      Integer num = new Integer(elderlyPersons.size());
     try{
      FileOutputStream fos = new FileOutput-
Stream("NumberOfElderlyPersons.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
           oos.writeObject(num);
           oos.flush();
           oos.close();

        }catch(Exception e){
            e.printStackTrace();
        }


    //writtes for every ElderlyPerson seperate file: (patientId).txt
     for(int i=0;i<n;i++){
      try{
           int id= ePersons[i].getPatientId();

           FileOutputStream fos = new FileOutputStream(id +".txt");
           ObjectOutputStream oos = new ObjectOutputStream(fos);
           oos.writeObject(ePersons[i]);
           oos.flush();
           oos.close();

      }catch(Exception e){
            e.printStackTrace();
        }

     }
 }
public Collection loadData(){
```

```
	//returns a Collection of elderlyPersons

	//number of stored ElderlyPerson: - n (the number is set during the storeData())

	//the Collection that will be returned
	Set elderlyPersons = new HashSet();
	// set number of elderlyPersons

	Integer cant = new Integer(0);
	try {
	 FileInputStream fis = new FileInput-
Stream("NumberOfElderlyPersons.txt");
	 ObjectInputStream ois = new ObjectInputStream(fis);
	 cant= (Integer)ois.readObject();
	 n = cant.intValue();
	 ois.close();
	 }catch(Exception e){
	   e.printStackTrace();
	}
	//reads all ElderlyPersons from the txt-files
		try{


	 for(int i=0;i<n;i++){
		 FileInputStream fis = new FileInputStream((i+1)+".txt");
		 ObjectInputStream ois = new ObjectInputStream(fis);

		 curentPatient = (ElderlyPerson )ois.readObject();
		 elderlyPersons.add(curentPatient);
		 ois.close();
	 }

		}catch(Exception e){
		 e.printStackTrace();
	  }

	return elderlyPersons;

}
	//marks currentTaskList as done
	//writtes the changed currentlyPerson down into its txt-file
	public void markTaskListDone(){
		currentTaskList.setState(true);
		try{
		FileOutputStream fos = new FileOutputStream( curentPa-
tient.getPatientId() +".txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(curentPatient);
```

```
            oos.flush();
            oos.close();
        }catch(Exception e){
                e.printStackTrace();
           }

        }

        public void setLastVisit(Date date){
                curentPatient.setLastVisit(date);
        }

        public boolean loadRoomMapping(int apartmentId, Array-
List<MappingItem> data){
                System.arraycopy(curentPatient.getAppMap(), 0, data, 0,
curentPatient.getAppMap().size()-1);
                return true;

        }
}
```

1.7.12 Exercise to Experience Package for UI Group: Type Embedded in Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Type embedded in name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

uiSystem :visualizationUnit.java

```
package org.belami.dcga.ui.ui_system.visualization_unit;

import java.lang.reflect.InvocationTargetException;
import java.util.Collection;
import java.util.GregorianCalendar;
import java.util.Vector;

import javax.swing.JOptionPane;

import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.ui.ui_system.interaction_unit.InteractionUnit;

/**
 * The visualisatin unit creates the display of the DCGA.
 *
 * @author A-Team
 * @version 1.0
 */
public class VisualizationUnit {

	/**
	 * The controller of the gui.
```

```
         */
        InteractionUnit interactionUnit;

        /**
         * The main frame of the gui.
         */
        private MainFrame mainFrame;

        /**
         * The dialog to choose a patient manually
         */
        PatientsDialog patientsDialog;

        /**
         * The dialog to show the patient informations
         */
        PatientInfoDialog patientInfoDialog;

        /**
         * The synchronization dialog to show while uploading /
downloading data
         */
        SynchronizationDialog syncDialog;

        /**
         * The frame to enter text comment
         */
        CommentInputFrame commentInputFrame;

        /**
         * Creates an instance of the visualisation unit.
         * The main frame is automatically created by the creation.
         * <code>VisualisationUnit</code> needs an interaction unit as
controller,
         * which must be set using the <code>setInteractionUnit</code>
Method.
         */
        public VisualizationUnit() {

              /*
               * Schedules a job for the event-dispatching thread
               * to create and show the main frame.
               */
              try {
                    javax.swing.SwingUtilities.invokeAndWait(new Run-
nable() {
                          public void run() {
                                createAndShowMainFrame();
                          }
```

```
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }

      }

      /**
       * Sets the controller for the display.
       * @param interactionUnit  The controller
       */
      public void setInteractionUnit(InteractionUnit interactionUnit)
{
            this.interactionUnit = interactionUnit;
      }

      /**
      * Creates the main frame and shows it.  For thread safety,
      * this method should be invoked from the event-dispatching
thread.
      */
      private void createAndShowMainFrame() {
            mainFrame = MainFrame.createMainFrame(this);
      }

      /**
       * Used to set the title of the main window. It contains the
Customer-No,
       * the Patientname and furthermore the actual date and time.
       * (e.g. "Customer: 0815, Ms. Schmidt | Monday, 10/10/2010 |
10:15 PM")
       * @param elderlyPerson
       */
      public void updateTitleBar(ElderlyPerson elderlyPerson) {
            GregorianCalendar today = new GregorianCalendar();
            String minute = ("0"+today.get(GregorianCalendar.MINUTE));
            minute = minute.substring(minute.length()-2, min-
ute.length());
            String hour = ("0"+today.get(GregorianCalendar.HOUR));
            hour = hour.substring(hour.length()-2, hour.length());
                  mainFrame.setTitle("Customer: " + elderlyPer-
son.getName()+" | " //Name of elderly Person
                        +(today.get(GregorianCalendar.MONTH)+1)  /*+1,
because of format 0-11*/+ "/"
                        +today.get(GregorianCalendar.DAY_OF_MONTH)+"/"
            //american format
```

```
                    +today.get(GregorianCalendar.YEAR)+" | "
          //month/day/year
                    +hour + ":"
                    +minute);
     }

     /**
      * Updates the "Current Comments" and the "Old Comments"
      * @param commentList
      */
     public void updateComments(Collection<CommentShortInfo> current-
CommentsList, Collection<CommentShortInfo> oldCommentsList) {

          CommentTabbedPane commentTabbedPane =
                    main-
Frame.infoAndCommentPane.commentPanel.commentTabbedPane;

          commentTabbed-
Pane.setNewCurrentCommentsList(currentCommentsList);
          commentTabbedPane.setNewOldCommentsList(oldCommentsList);

     }

     /**
      * Updates the "Done Tasks", "Open Tasks" and the ProgressBar.
      * @param doneTasks, openTasks
      */
     public void updateTasks(Vector<Task> openTaskList, Vector<Task>
doneTaskList) {
          main-
Frame.taskPanel.taskTabbedPane.openTasksListTable.setNewTaskList(open
TaskList);
          main-
Frame.taskPanel.taskTabbedPane.doneTasksListTable.setNewTaskList(done
TaskList);

          main-
Frame.taskPanel.progressPanel.progressBar.setMaximum(openTaskList.siz
e() + doneTaskList.size());
          main-
Frame.taskPanel.progressPanel.progressBar.setValue(openTaskList.size(
));
          main-
Frame.taskPanel.progressPanel.progressBar.setString(String.valueOf(do
neTaskList.size()) + "/"
                    + String.valueOf(openTaskList.size() + do-
neTaskList.size()) + " Tasks completed.");
```

```
		main-
Frame.taskPanel.progressPanel.markAsDoneButton.setEnabled(false);
	}

	/**
	 * Updates the visualization of amiCA information box
	 * @param infoList
	 */
	public void updateInformation(Collection<Information> infoList)
{

	this.mainFrame.infoAndCommentPane.infoPanel.setNewInformationLis
t(infoList);
	}

	/**
	 * opens a new window where the user is able to select the cur-
rent patient.
	 * @param patientList
	 */
	public void showPatientList(Collection<ElderlyPersonShortInfo>
patientList) {
		patientsDialog = new PatientsDialog(this, patientList);
	}

	/**
	 * opens a new window where the user can see further information
about the actual patient.
	 * @param dumdidum
	 */
	public void showPatientInformation(ElderlyPerson ep) {
		patientInfoDialog = new PatientInfoDialog(this, ep);
	}

	/**
	 * Changes the image of record button to “start button”,
	 * activates the comment buttons
	 */
	public void showNormalButtonState() {

		// Get the panel with the buttons
		ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

		// Set the state of the buttons
		buttonPanel.recordButton.setEnabled(true);
		buttonPanel.recordButton.setIcon(new
javax.swing.ImageIcon(getClass().getResource(
				buttonPanel.getrecordButtonRes())));
```

```
            buttonPanel.playButton.setEnabled(false);
            buttonPanel.stopButton.setEnabled(false);
            buttonPanel.deleteButton.setEnabled(false);

      }

      /**
       * Changes the image of record button to “stop button”,
       * deactivates the other comment buttons.
       */
      public void showRecordingState() {

            // Get the panel with the buttons
            ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

            // Set the state of the buttons
            buttonPanel.recordButton.setEnabled(true);
            buttonPanel.recordButton.setIcon(new
javax.swing.ImageIcon(getClass().getResource(
                        buttonPanel.getrecordStopButtonRes())));
            buttonPanel.playButton.setEnabled(false);
            buttonPanel.stopButton.setEnabled(false);
            buttonPanel.deleteButton.setEnabled(false);

      }

      /**
       * Deactivates all comment buttons, except the “stop button”
       * @param dumdidum
       */
      public void showPlayingState() {

            // Get the panel with the buttons
            ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

            // Set the state of the buttons
            buttonPanel.recordButton.setEnabled(false);
            buttonPanel.playButton.setEnabled(false);
            buttonPanel.stopButton.setEnabled(true);
            buttonPanel.deleteButton.setEnabled(false);

      }

      /**
       * Shows a confirmation dialog with a custom text mes-sage, ok
and cancel buttons.
       */
```

```
	public boolean showConfirmationDialog() {

		int dialogReturn = JOptionPane.showConfirmDialog(
				this.mainFrame,
				"The task could not have been completed. Would
you like to complete it anyway?",
				"Confirmation",
				JOptionPane.YES_NO_OPTION);

		if (dialogReturn == JOptionPane.YES_OPTION) {
			return true;
		} else {
			return false;
		}

	}

	/**
	 * Shows a dialog with a comment text message, ok button
	 */
	public void showCommentDialog(String comment) {
		JOptionPane.showMessageDialog(this.mainFrame,
								comment,
								"Text Comment",
								JOption-
Pane.PLAIN_MESSAGE);
	}

	/**
	 * Shows a dialog with a custom text message, ok button
	 * @param dumdidum
	 */
	public void showDialog(String message) {
		JOptionPane.showMessageDialog(this.mainFrame, message);
	}

	/**
	 * Shows a dialog with a text input field, ok and cancel buttons
	 * @param dumdidum
	 */
	public void showTextCommentInputDialog() {
		commentInputFrame = new CommentInputFrame(this);
	}

	/**
	 * Shows a modal window with a custom text message during the
synchronization process.
	 * @param message
	 * @param dumdidum
```

```
	 */
	public void showSynchronizationWindow(String message) {
		syncDialog = new SynchronizationDialog(this, message);
	}

	/**
	 * Closes the modal window after the synchronization process.
	 * @param dumdidum
	 */
	public void closeSynchronizationWindow() {
		syncDialog.dispose();
	}

	/**
	 * Obtains the controller of the gui.
	 * @return a reference to the <code>InteractionUnit</code>
	 */
	public InteractionUnit getInteractionUnit() {
		return interactionUnit;
	}

	/**
	 * Obtains the main frame of the <code>VisualizationUnit</code>.
	 * @return the main frame of the <code>VisualizationUnit</code>.
	 */
	public MainFrame getMainFrame() {
		return mainFrame;
	}

	/***
	 * Obtains the patients dialog of the
<code>VisualizationUnit</code>.
	 * @return the patients dialog of the
<code>VisualizationUnit</code>.
	 */
	public PatientsDialog getPatientsDialog() {
		return patientsDialog;
	}

}
```

1.8 Exercises of the Assignments (Tuesday)

1.8.1 Exercise to Experience Package for Amica Interaction Group: Comments

Your Name: ______________________________

Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import org.belami.dcga.amica_interaction.Situation;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * Data structure containing the information of one "match" element
from the XML mapping file.
 *
 * @author Marc Giombetti
 * @author Philip Preissing
 * @author Michel Weimerskirch
 */
public class Match {
    /**
     * Fact ID that has to be matched with the Situation object
     */
    private String factName = null;
    /**
     * Comparator method for the fact ID from the mapping-file
     */
    private String factNameComparator = null;
```

```
    /**
     * Start date that has to be matched with the Situation object
     */
    private Date startDate = null;
    /**
     * Comparator method for the start date from the mapping-file
     */
    private String startDateComparator = null;

    /**
     * End date that has to be matched with the Situation object
     */
    private Date endDate = null;
    /**
     * Comparator method for the end date from the mapping-file
     */
    private String endDateComparator = null;

    /**
     * Description that has to be matched with the Situation object
     */
    private String description = null;
    /**
     * Comparator method for the description from the mapping-file
     */
    private String descriptionComparator = null;

    /**
     * Source that has to be matched with the Situation object
     */
    private String source = null;
    /**
     * Comparator method for the source identifier from the mapping-
file
     */
    private String sourceComparator = null;

    /**
     * Location that has to be matched with the Situation object
     */
    private String location = null;
    /**
     * Comparator method for the location identifier from the map-
ping-file
     */
    private String locationComparator = null;

    /**
```

```
     * NodeList used to map a matching Situation to an Information.
Might be null if not applicable.
     */
    public NodeList mapInformationNodes = null;
    /**
     * NodeList used to map a matching Situation to a Task. Might be
null if not applicable.
     */
    public NodeList mapTaskNodes = null;
    /**
     * Boolean value that specifies if a matching Situation is mapped
a TaskEvent.
     */
    public boolean mapTaskEvent = false;


    private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd");

    private static DateFormat dateTimeFormat = new SimpleDateFor-
mat("yyyy-MM-dd k:m:s");

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
```

```
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    /**
     * Returns true if the given situation is matched.
     * @param situation A Situation
     * @return True if the given situation is matched.
     */
    public boolean matches(Situation situation) {
        if(factNameComparator != null) {
            if(!compare(situation.getFactName(), factName, factName-
Comparator)) {
```

```
                return false;
            }
        }
        if(startDateComparator != null) {
            if(!compare(situation.getStartDate(), startDate, start-
DateComparator)) {
                return false;
            }
        }
        if(endDateComparator != null) {
            if(!compare(situation.getEndDate(), endDate, endDateCom-
parator)) {
                return false;
            }
        }
        if(descriptionComparator != null) {
            if(!compare(situation.getDescription(), description, de-
scriptionComparator)) {
                return false;
            }
        }
        if(sourceComparator != null) {
            if(!compare(situation.getSource(), source, sourceCompara-
tor)) {
                return false;
            }
        }
        if(locationComparator != null) {
            if(!compare(situation.getLocation()+"", location, loca-
tionComparator)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
    public boolean mapsInformation(Situation situation) {
        return mapInformationNodes != null;
    }
```

```
    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }

        return information;
    }

    /**
     * Map the given Situation to a Task object.
     * @param situation A Situation
     * @return Mapped Task object
     */
```

```
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Map the given Situation to a TaskEvent object.
     * @param situation A Situation
     * @return Mapped TaskEvent object
     */
    public TaskEvent mapTaskEvent(Situation situation) {
        TaskEvent taskEvent = new TaskEvent(situation.getSource(),
situation.getLocation(), situation.getFactName());
        return taskEvent;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
     * @param a Original object
     * @param b Compared object
```

```
     * @param comparator One of "isNull", "notNull", "startsWith",
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
            return false;
        } else {
```

```
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Replaces keywords in a String using data from the given Situa-
tion object
     * @param text Untreated input String
     * @param situation A Situation
     * @return Treated Text
     */
    protected static String prepareString(String text, Situation
situation) {
        text = text.replaceAll("\\{\\{priority\\}\\}", situa-
tion.getPriority()+"");
        if (situation.getDescription()!= null) {
           text = text.replaceAll("\\{\\{description\\}\\}", situa-
tion.getDescription());
        }
        if (situation.getLocation()!= null) {
           text = text.replaceAll("\\{\\{location\\}\\}", situa-
tion.getLocation()+"");
        }
        text = text.replaceAll("\\{\\{startDate\\}\\}",  dateTimeFor-
mat.format(situation.getStartDate()));
        if (situation.getEndDate()!= null) {
           text = text.replaceAll("\\{\\{endDate\\}\\}", dateTimeFor-
mat.format(situation.getEndDate()));
        }
        text = text.replaceAll("\\{\\{source\\}\\}", situa-
tion.getSource());
        text = text.replaceAll("\\{\\{factName\\}\\}", situa-
tion.getFactName());

        return text;
    }
}
```

1.8.2 Exercise to Experience Package for Amica Interaction Group: Uncommunicative Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Uncommunicative Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Amica_Interaction:match.java

```
package org.belami.dcga.amica_interaction.mapping;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import org.belami.dcga.amica_interaction.Situation;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.w3c.dom.DOMException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * Data structure containing the information of one "match" element
from the XML mapping file.
 *
 * @author Marc Giombetti
 * @author Philip Preissing
 * @author Michel Weimerskirch
 */
public class Match {
    /**
```

```
     * Fact ID that has to be matched with the Situation object
     */
    private String factName = null;
    /**
     * Comparator method for the fact ID from the mapping-file
     */
    private String factNameComparator = null;

    /**
     * Start date that has to be matched with the Situation object
     */
    private Date startDate = null;
    /**
     * Comparator method for the start date from the mapping-file
     */
    private String startDateComparator = null;

    /**
     * End date that has to be matched with the Situation object
     */
    private Date endDate = null;
    /**
     * Comparator method for the end date from the mapping-file
     */
    private String endDateComparator = null;

    /**
     * Description that has to be matched with the Situation object
     */
    private String description = null;
    /**
     * Comparator method for the description from the mapping-file
     */
    private String descriptionComparator = null;

    /**
     * Source that has to be matched with the Situation object
     */
    private String source = null;
    /**
     * Comparator method for the source identifier from the mapping-
file
     */
    private String sourceComparator = null;

    /**
     * Location that has to be matched with the Situation object
     */
    private String location = null;
```

```
    /**
     * Comparator method for the location identifier from the map-
ping-file
     */
    private String locationComparator = null;

    /**
     * NodeList used to map a matching Situation to an Information.
Might be null if not applicable.
     */
    public NodeList mapInformationNodes = null;
    /**
     * NodeList used to map a matching Situation to a Task. Might be
null if not applicable.
     */
    public NodeList mapTaskNodes = null;
    /**
     * Boolean value that specifies if a matching Situation is mapped
a TaskEvent.
     */
    public boolean mapTaskEvent = false;


    private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd");

    private static DateFormat dateTimeFormat = new SimpleDateFor-
mat("yyyy-MM-dd k:m:s");

    /**
     * Creates a new instance of Match
     * @param matchNode DOM Node from the XML mapping document
     */
    public Match(Node matchNode) {
        NodeList childNodes = matchNode.getChildNodes();
        for(int i=0, l=childNodes.getLength(); i<l; i++) {
            Node currentNode = childNodes.item(i);
            String nodeName  = currentNode.getNodeName();

            if(nodeName.equals("factName")) {
                Node comparator = currentNode.getFirstChild();
                factNameComparator = comparator.getNodeName();
                factName = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("startDate")) {
                Node comparator = currentNode.getFirstChild();
                startDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
```

```
                        startDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("endDate")) {
                Node comparator = currentNode.getFirstChild();
                endDateComparator = comparator.getNodeName();
                if(comparator.getFirstChild() != null) {
                    try {
                        endDate = dateFor-
mat.parse(comparator.getFirstChild().getNodeValue());
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            } else if(nodeName.equals("description")) {
                Node comparator = currentNode.getFirstChild();
                descriptionComparator = comparator.getNodeName();
                description = compara-
tor.getFirstChild().getNodeValue();
            } else if(nodeName.equals("source")) {
                Node comparator = currentNode.getFirstChild();
                sourceComparator = comparator.getNodeName();
                source = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("location")) {
                Node comparator = currentNode.getFirstChild();
                locationComparator = comparator.getNodeName();
                location = comparator.getFirstChild().getNodeValue();
            } else if(nodeName.equals("map")) {
                NodeList mapNodes = currentNode.getChildNodes();
                for (int j=0, k=mapNodes.getLength(); j<k; j++) {
                    Node node = mapNodes.item(j);
                    if(node.getNodeName().equals("task")) {
                        mapTaskNodes = node.getChildNodes();
                    } else if(node.getNodeName().equals("taskEvent"))
{
                        mapTaskEvent = true;
                    } else
if(node.getNodeName().equals("information")) {
                        mapInformationNodes = node.getChildNodes();
                    }
                }
            }
        }
    }

    /**
```

```
     * Returns true if the given situation is matched.
     * @param situation A Situation
     * @return True if the given situation is matched.
     */
    public boolean matches(Situation situation) {
        if(factNameComparator != null) {
            if(!compare(situation.getFactName(), factName, factName-
Comparator)) {
                return false;
            }
        }
        if(startDateComparator != null) {
            if(!compare(situation.getStartDate(), startDate, start-
DateComparator)) {
                return false;
            }
        }
        if(endDateComparator != null) {
            if(!compare(situation.getEndDate(), endDate, endDateCom-
parator)) {
                return false;
            }
        }
        if(descriptionComparator != null) {
            if(!compare(situation.getDescription(), description, de-
scriptionComparator)) {
                return false;
            }
        }
        if(sourceComparator != null) {
            if(!compare(situation.getSource(), source, sourceCompara-
tor)) {
                return false;
            }
        }
        if(locationComparator != null) {
            if(!compare(situation.getLocation()+"", location, loca-
tionComparator)) {
                return false;
            }
        }

        return true;
    }

    /**
     * Returns true if the given situation can be mapped to an Infor-
mation.
```

```
     * @param situation A Situation
     * @return True if the given situation can be mapped to an Infor-
mation.
     */
    public boolean mapsInformation(Situation situation) {
        return mapInformationNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a Task.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a Task.
     */
    public boolean mapsTask(Situation situation) {
        return mapTaskNodes != null;
    }

    /**
     * Returns true if the given situation can be mapped to a
TaskEvent.
     * @param situation A Situation
     * @return True if the given situation can be mapped to a
TaskEvent.
     */
    public boolean mapsTaskEvent(Situation situation) {
        return mapTaskEvent;
    }

    /**
     * Map the given Situation to an Information object.
     * @param situation A Situation
     * @return Mapped Information object
     */
    public Information mapInformation(Situation situation) {
        Information  information = new Information();
        for(int i=0, l=mapInformationNodes.getLength(); i<l; i++) {
            Node node = mapInformationNodes.item(i);
            if (node.getNodeName().equals("location")) {
                informa-
tion.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {
                informa-
tion.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            }
        }
```

```
        return information;
    }

    /**
     * Map the given Situation to a Task object.
     * @param situation A Situation
     * @return Mapped Task object
     */
    public Task mapTask(Situation situation) {
        Task task = new Task();
        for(int i=0, l=mapTaskNodes.getLength(); i<l; i++) {
            Node node = mapTaskNodes.item(i);
            if(node.getNodeName().equals("priority")) {

task.setPriority(Integer.parseInt(prepareString(node.getFirstChild().
getNodeValue(), situation)));
            } else if (node.getNodeName().equals("location")) {

task.setLocation(prepareString(node.getFirstChild().getNodeValue(),
situation));
            } else if (node.getNodeName().equals("description")) {

task.setDescription(prepareString(node.getFirstChild().getNodeValue()
, situation));
            } else if (node.getNodeName().equals("autoMarkable")) {
                TaskEvent taskEvent = new
TaskEvent(situation.getSource(), situation.getLocation(), situa-
tion.getFactName());
                ArrayList<TaskEvent> taskEventCollection = new Array-
List<TaskEvent>();
                taskEventCollection.add(taskEvent);

                task.setAutoMarkable(true);
                task.addTaskEvents(taskEventCollection);
            }
        }

        return task;
    }

    /**
     * Map the given Situation to a TaskEvent object.
     * @param situation A Situation
     * @return Mapped TaskEvent object
     */
    public TaskEvent mapTaskEvent(Situation situation) {
        TaskEvent taskEvent = new TaskEvent(situation.getSource(),
situation.getLocation(), situation.getFactName());
```

```
        return taskEvent;
    }

    /**
     * Compare two String objects using the comparison method given
by the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "startsWith",
"endsWith", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(String a, String b, String com-
parator) {
        if(comparator.equals("notNull")) {
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if (b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("startsWith")) {
                if(a.startsWith(b)) return true;
                else return false;
            } else if(comparator.equals("endsWith")) {
                if(a.endsWith(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Compare two Date objects using the comparison method given by
the "comparator" String.
     * @param a Original object
     * @param b Compared object
     * @param comparator One of "isNull", "notNull", "before", "af-
ter", "equals"
     * @return True if the comparison is successful.
     */
    protected static boolean compare(Date a, Date b, String compara-
tor) {
        if(comparator.equals("notNull")) {
```

```
            if(a != null) return true;
            else return false;
        } else if(comparator.equals("isNull")) {
            if(a == null) return true;
            else return false;
        } else if(b == null || a == null) {
            return false;
        } else {
            if(comparator.equals("before")) {
                if(a.before(b)) return true;
                else return false;
            } else if(comparator.equals("after")) {
                if(a.after(b)) return true;
                else return false;
            } else { //default: equals
                if(a.equals(b)) return true;
                else return false;
            }
        }
    }

    /**
     * Replaces keywords in a String using data from the given Situa-
tion object
     * @param text Untreated input String
     * @param situation A Situation
     * @return Treated Text
     */
    protected static String prepareString(String text, Situation
situation) {
        text = text.replaceAll("\\{\\{priority\\}\\}", situa-
tion.getPriority()+"");
        if (situation.getDescription()!= null) {
           text = text.replaceAll("\\{\\{description\\}\\}", situa-
tion.getDescription());
        }
        if (situation.getLocation()!= null) {
           text = text.replaceAll("\\{\\{location\\}\\}", situa-
tion.getLocation()+"");
        }
        text = text.replaceAll("\\{\\{startDate\\}\\}",  dateTimeFor-
mat.format(situation.getStartDate()));
        if (situation.getEndDate()!= null) {
           text = text.replaceAll("\\{\\{endDate\\}\\}", dateTimeFor-
mat.format(situation.getEndDate()));
        }
        text = text.replaceAll("\\{\\{source\\}\\}", situa-
tion.getSource());
```

```
        text = text.replaceAll("\\{\\{factName\\}\\}", situa-
tion.getFactName());

        return text;
    }
}
```

AmicaInteraction: UnplannedTaskHandler.java

```
package org.belami.dcga.amica_interaction;

import org.belami.dcga.amica_interaction.mapping.Match;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.computation.Computation;

/**
 * The Handler for the unplanned tasks.
 *
 * @author Marc Giombetti
 * @author Philip Preissing
 * @author Michel Weimerskirch
 *
 */
class UnplannedTaskHandler {
    Computation computation;

    /** Creates a new instance of UnplannedTaskHandler
     * @param computation2 */
    public UnplannedTaskHandler(Computation computation2) {
        computation = computation2;
    }

    /**
     * Creation of a new unplanned task for a given situation
     *
     * @param s A situation
     */
    public void handleUnplannedTask(Situation s, Match match) {
        Task task = match.mapTask(s);

        computation.addUnplannedTask(task);
    }
}
```

1.8.3 Exercise to Experience Package for Computation Group: Comments

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Computation:ComputationImpl.java

```
package org.belami.dcga.computation;

import java.util.Collection;
import java.util.Date;
import java.util.Observer;
import java.util.Vector;

import org.belami.dcga.common_datastructures.Comment;
import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.belami.dcga.computation.commentmanager.CommentManager;
import org.belami.dcga.computation.commentmanager.CommentManagerImpl;
import
org.belami.dcga.computation.informationmanager.InformationManager;
import
org.belami.dcga.computation.informationmanager.InformationManagerImpl
;
import org.belami.dcga.computation.patientmanager.PatientManager;
import org.belami.dcga.computation.patientmanager.PatientManagerImpl;
import
org.belami.dcga.computation.taskmanager.RoomNotVisitedException;
import org.belami.dcga.computation.taskmanager.TaskManager;
import org.belami.dcga.computation.taskmanager.TaskManagerImpl;
```

```java
import org.belami.dcga.location_manager.LocationManager;
import org.belami.dcga.synchronization.Synchronization;
import org.belami.dcga.ui.UI;

/**
 * This Class is an implementation of the Computation Interface where
the
 * communication is controlled. For a detailed description have a
look at the
 * interface {@link Computation}
 *
 * @see Computation
 * @author Daniel Schneider
 * @version 1.0
 *
 */
class ComputationImpl implements Computation {



      /**
       * Main method for the program. The computation controller is
instantiated
       * which begins to execute a startup sequence
       *
       * @param args
       *            command line arguments (not specified yet)
       */
      public static void main(String[] args) {
            Computation.INSTANCE.startUp();
      }

      /**
       * Provides a singleton instance for the computation component
       */
      private static ComputationImpl INSTANCE = null;

      /**
       * Store the current room. The value -1 means that the room was
not yet set.
       */
      private int currentRoom = -1;

      /**
       * A singleton instance of the TaskManager. We need this in-
stance to work on
       * it and this is also needed for the testcases
       *
       * @see TaskManager
```

```
         * @see TaskManagerImpl
         */
        private TaskManager taskManager = TaskManager.INSTANCE;

        /**
         * An instance of the CommentManager. We need this instance to
work on it
         * and this is also needed for the testcases
         *
         * @see CommentManager
         * @see CommentManagerImpl
         */
        private CommentManager commentManager = new CommentManager-
Impl();

        /**
         * An instance of the InformationManager. We need this instance
to work on
         * it and this is also needed for the testcases
         *
         * @see InformationManager
         * @see InformationManagerImpl
         */
        private InformationManager informationManager = new Information-
ManagerImpl();

        /**
         * An instance of the PatientManager. We need this instance to
work on it
         * and this is also needed for the testcases
         *
         * @see PatientManager
         * @see PatientManagerImpl
         */
        private PatientManager patientManager = new PatientManager-
Impl();

        /**
         * An instance of the LocationManager. We need this instance to
work on it
         * and this is also needed for the testcases
         *
         * @see LocationManager
         */
        private LocationManager locationManager = LocationMan-
ager.INSTANCE;

        /**
```

```
	 * An instance of the Synchronization. We need this instance to 
work on it
	 * and this is also needed for the testcases
	 * 
	 * @see Synchronization
	 */
	private Synchronization synchronization = Synchroniza-
tion.INSTANCE;

	/**
	 * An instance of the UI. We need this instance to work on it
	 * and this is also needed for the testcases
	 * 
	 * @see UI
	 */
	private UI ui = UI.INSTANCE;

	/**
	 * By using the singleton pattern we have to make a private con-
structor. By
	 * this we assure that there can only be one instance at any 
time.
	 * 
	 */
	private ComputationImpl() {
	}

	/**
	 * Its the startUp sequence for DCGA. We have to create all the 
required
	 * components and initialize them if needed.
	 * 
	 * @see org.belami.dcga.computation.Computation#startUp()
	 */
	public void startUp() {
		ui.initialize();

	}

	/**
	 * This is part of the singleton pattern. We provide the only 
existing
	 * interface with this method
	 * 
	 * @return instance of the computation
	 */
	protected static Computation getInstance() {
		if (INSTANCE == null) {
			INSTANCE = new ComputationImpl();
```

```
        }
        return INSTANCE;
    }

    /**
     * The controller is told to be initialized. This initialization
means to
     * tell the subcomponents also to initialize themselves. This
method is
     * called when a new patientId is set.
     *
     */
    private void initialize() {
        // PatientManager does not have to be initialized because
the
        // patientManager itself performs this function call
        taskManager.initialize();
        // DO we still need this?

    }

    /**
     * Set the current room variable in this component to a new
value. Its a
     * setter methods for the private variable {@link #currentRoom}
     *
     * @param id
     *           of the current room
     *
     */
    private void setCurrentRoom(int roomId) {
        this.currentRoom = roomId;
    }

    /**
     * @see
org.belami.dcga.computation.Computation#addInformation(org.belami.dcg
a.common_datastructures.Information)
     */
    public void addInformation(Information information) {
        informationManager.addInformation(information);
    }

    /**
     * @see
org.belami.dcga.computation.Computation#deleteComment(int)
     */
    public void deleteComment(int commentId) {
        commentManager.deleteComment(commentId);
```

```
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getComment(int)
	 */
	public Comment getComment(int commentId) {
		return commentManager.getComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getCurrentRoom()
	 */
	public int getCurrentRoom() {
		return currentRoom;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#getInformationList()
	 */
	public Collection<Information> getInformationList() {
		return informationManager.getInformationList();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientInfo()
	 */
	public ElderlyPerson getPatientInfo() {
		return patientManager.getPatientInfo();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getTaskList()
	 */
	public Vector<Task> getTaskList() {
		return taskManager.getTaskList();
	}

	/**
	 * @throws RoomNotVisitedException
	 * @see
org.belami.dcga.computation.Computation#markTaskAsCompleted(int)
	 */
	public void markTaskAsCompleted(int taskId, boolean override)
			throws RoomNotVisitedException {
		taskManager.markTaskAsCompletedManually(taskId, override);
	}
```

```java
	/**
	 * @see
org.belami.dcga.computation.Computation#setTaskEventDone(TaskEvent)
	 */
	public void setTaskEventDone(TaskEvent taskEvent) {
		taskManager.setTaskEventDone(taskEvent);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addUnplannedTask(Task)
	 */
	public void addUnplannedTask(Task unplannedTask) {
		taskManager.addUnplannedTask(unplannedTask);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onAmiCaConnected(int)
	 */
	public ElderlyPerson onAmiCaConnected(int patientId) {
		setPatientId(patientId);
		return patientManager.getPatientInfo();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onNewRoomEntered(int)
	 */
	public void onNewRoomEntered(int roomId) {
		setCurrentRoom(roomId);
		taskManager.sort();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setPatientId(int)
	 */
	public void setPatientId(int patientId) {
		patientManager.setPatientId(patientId);
		initialize();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startDownload()
	 */
	public void startDownload() {
		synchronization.initDownload();
	}
```

```
	/**
	 * @see org.belami.dcga.computation.Computation#startUpload()
	 */
	public void startUpload() {
		synchronization.initUpload();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#storeComment(org.belami.dcga.
common_datastructures.Comment)
	 */
	public void storeComment(Comment comment) {
		commentManager.storeComment(comment);
	}

	/**
	 * Register the Observer at the subcomponents.
	 *
	 * @see
org.belami.dcga.computation.Computation#unregisterObserver(java.util.
Observable)
	 */
	public void unregisterObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
			case OpenTaskWarning:
				taskMan-
ager.deleteOpenTaskWarningObserver(observer);
				break;
			case TaskList:
				taskManager.deleteTaskListObserver(observer);
				break;
			case CommentList:
				commentManager.deleteObserver(observer);
				break;
			case Patient:
				patientManager.deleteObserver(observer);
				break;
			case InformationList:
				informationManager.deleteObserver(observer);
				break;
		}
	}

	/**
```

```
     * @see
org.belami.dcga.computation.Computation#registerObserver(java.util.Ob
servable)
     */
    public void registerObserver(Observer observer, ControllerOb-
servables observables) {
        switch (observables) {
            case OpenTaskWarning:
                taskMan-
ager.addOpenTaskWarningObserver(observer);
                break;
            case TaskList:
                taskManager.addTaskListObserver(observer);
                break;
            case CommentList:
                commentManager.addObserver(observer);
                break;
            case Patient:
                patientManager.addObserver(observer);
                break;
            case InformationList:
                informationManager.addObserver(observer);
                break;
        }

    }

    /**
     * @see
org.belami.dcga.computation.Computation#onApartmentLeft()
     */
    public void onApartmentLeft(Date date) {
        patientManager.setLastVisit(date);
        taskManager.onApartmentLeft();
    }

    /**
     * @see
org.belami.dcga.computation.Computation#wasRoomVisited(int)
     */
    public boolean wasRoomVisited(int roomId) {
        /**
         * This method was formerly called wasRoomVisited but Lo-
cationManager
         * implemented it with another name. Perhaps it was a non-
consistent
         * specification
         */
        return locationManager.wasRoomEntered(roomId);
```

```
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientList()
	 */
	public Collection<ElderlyPersonShortInfo> getPatientList() {
		return patientManager.getPatientList();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getCommentList()
	 */
	public Collection<CommentShortInfo> getCommentList() {
		return commentManager.getCommentList();
	}
}
```

1.8.4 Exercise to Experience Package for Computation Group: Uncommunicative Name

Your Name: ________________________
Your Subject-ID: _____<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the follwing type: Uncommunicative Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

CommonDataStructures:Task.java → in your code code smells of uncommunicative name couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.common_datastructures;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;

import org.belami.dcga.computation.Computation;

public class Task implements Serializable, Comparable<Task> {

     /**
      * Task is not yet done.
      */
     public static final int UNDONE = 0;

     public static final int DONE_SYSTEM = 1;

     public static final int DONE_CG = 2;

     private int taskId;

     private int priority;

     private String description;

     // room-Id
```

```
	private String location;

	// room-ID as Integer, needed by taskmanager!
	private int roomID;

	/**
	 * @see org.belami.common_datastructures.Task
	 */
	private int state;

	/**
	 * indicates, whether the task can be automatically marked as
completed or
	 * not. If the task can be auto-marked it is still possible to
mark it
	 * manually.
	 */
	private boolean autoMarkable;

	private boolean unplannedTask;

	// Stores TaskEvents needed for auto-completion, set it with
addTaskEvents()
	private HashSet<TaskEvent> taskEvents = new Hash-
Set<TaskEvent>();

	// required: a no-args constructor
	public Task() {
		taskId = -1;
		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;

	}

	// Constructor where the Id is set
	public Task(int tId) {
		taskId = tId;

		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;
	}
```

```
// implements the getter methods of the class
public int getTaskId() {
      return taskId;
}

public int getPriority() {
      return priority;
}

public String getDescription() {
      return description;
}

public String getLocation() {
      return location;
}

public int getState() {
      return state;
}

public boolean isAutoMarkable() {
      return autoMarkable;
}

public boolean getUnplannedTask() {
      return unplannedTask;
}

// implements the setter methods of the class
public void setTaskId(int Id) {
      taskId = Id;
}

public void setPriority(int prio) {
      priority = prio;
}

public void setDescription(String desc) {
      description = desc;
}

public void setLocation(String loc) {
      location = loc;
}

public void setState(int st) {
```

```
		state = st;
	}

	public void setAutoMarkable(boolean mM) {
		autoMarkable = mM;
	}

	public void setUnplannedTask(boolean uT) {
		unplannedTask = uT;
	}

	public void addTaskEvents(Collection<TaskEvent> events) {
		taskEvents = new HashSet<TaskEvent>(events);
	}

	/**
	 * Removes TaskEvent "event" from the taskEvents Set.
	 *
	 * @param event
	 *            event to delete (equivalent to mark as done)
	 */
	public void setTaskEventDone(TaskEvent event) {
		taskEvents.remove(event);
	}

	/**
	 * checks, if the Task is ready. If the task has to be marked as
done
	 * manually (check for completeness not possible), false is re-
turned.
	 * Otherwise it returns true, if all taskevents are done (in
this case
	 * taskEvents hashset is empty
	 *
	 * @return
	 */
	public boolean isReady() {
		if (autoMarkable == true && taskEvents.isEmpty())
			return true;
		else
			return false;
	}

	public int getRoomID() {
		return roomID;
	}

	public int compareTo(Task task) {
		int curRoom = Computation.INSTANCE.getCurrentRoom();
```

```
		if (task.getRoomID() != curRoom) {
			if (getRoomID() != curRoom)
				return getPriority() - task.getPriority();
			else
				return 1;
		} else {
			if (getRoomID() != curRoom)
				return -1;
			else
				return getPriority() - task.getPriority();
		}
	}

	public String toString() {
		return description;
	}

	public void setRoomID(int roomID) {
		this.roomID = roomID;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		final Task other = (Task) obj;
		if (autoMarkable != other.autoMarkable)
			return false;
		if (description == null) {
			if (other.description != null)
				return false;
		} else if (!description.equals(other.description))
			return false;
		if (location == null) {
			if (other.location != null)
				return false;
		} else if (!location.equals(other.location))
			return false;
		if (priority != other.priority)
			return false;
		if (roomID != other.roomID)
			return false;
		if (state != other.state)
```

```
            return false;
        if (taskId != other.taskId)
            return false;
        if (unplannedTask != other.unplannedTask)
            return false;
        return true;
    }

}
```

1.8.5 Exercise to Experience Package for Location Manager Group: Comments

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Computation:ComputationImpl.java → in your code code smells of comments couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.computation;

import java.util.Collection;
import java.util.Date;
import java.util.Observer;
import java.util.Vector;

import org.belami.dcga.common_datastructures.Comment;
import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.belami.dcga.computation.commentmanager.CommentManager;
import org.belami.dcga.computation.commentmanager.CommentManagerImpl;
import
org.belami.dcga.computation.informationmanager.InformationManager;
import
org.belami.dcga.computation.informationmanager.InformationManagerImpl
;
import org.belami.dcga.computation.patientmanager.PatientManager;
import org.belami.dcga.computation.patientmanager.PatientManagerImpl;
import
org.belami.dcga.computation.taskmanager.RoomNotVisitedException;
import org.belami.dcga.computation.taskmanager.TaskManager;
```

```
import org.belami.dcga.computation.taskmanager.TaskManagerImpl;
import org.belami.dcga.location_manager.LocationManager;
import org.belami.dcga.synchronization.Synchronization;
import org.belami.dcga.ui.UI;

/**
 * This Class is an implementation of the Computation Interface where
the
 * communication is controlled. For a detailed description have a
look at the
 * interface {@link Computation}
 */
class ComputationImpl implements Computation {


      /**
       * Main method for the program. The computation controller is
instantiated
       * which begins to execute a startup sequence
       *
       * @param args
       *            command line arguments (not specified yet)
       */
      public static void main(String[] args) {
            Computation.INSTANCE.startUp();
      }

      /**
       * Provides a singleton instance for the computation component
       */
      private static ComputationImpl INSTANCE = null;

      /**
       * Store the current room. The value -1 means that the room was
not yet set.
       */
      private int currentRoom = -1;

      /**
       * A singleton instance of the TaskManager. We need this in-
stance to work on
       * it and this is also needed for the testcases
       *
       * @see TaskManager
       * @see TaskManagerImpl
       */
      private TaskManager taskManager = TaskManager.INSTANCE;
```

```
	/**
	 * An instance of the CommentManager. We need this instance to
work on it
	 * and this is also needed for the testcases
	 *
	 * @see CommentManager
	 * @see CommentManagerImpl
	 */
	private CommentManager commentManager = new CommentManager-
Impl();

	/**
	 * An instance of the InformationManager. We need this instance
to work on
	 * it and this is also needed for the testcases
	 *
	 * @see InformationManager
	 * @see InformationManagerImpl
	 */
	private InformationManager informationManager = new Information-
ManagerImpl();

	/**
	 * An instance of the PatientManager. We need this instance to
work on it
	 * and this is also needed for the testcases
	 *
	 * @see PatientManager
	 * @see PatientManagerImpl
	 */
	private PatientManager patientManager = new PatientManager-
Impl();

	/**
	 * An instance of the LocationManager. We need this instance to
work on it
	 * and this is also needed for the testcases
	 *
	 * @see LocationManager
	 */
	private LocationManager locationManager = LocationMan-
ager.INSTANCE;

	/**
	 * An instance of the Synchronization. We need this instance to
work on it
	 * and this is also needed for the testcases
	 *
	 * @see Synchronization
```

```
	 */
	private Synchronization synchronization = Synchroniza-
tion.INSTANCE;

	/**
	 * An instance of the UI. We need this instance to work on it
	 * and this is also needed for the testcases
	 *
	 * @see UI
	 */
	private UI ui = UI.INSTANCE;

	/**
	 * By using the singleton pattern we have to make a private con-
structor. By
	 * this we assure that there can only be one instance at any
time.
	 *
	 */
	private ComputationImpl() {
	}

	/**
	 * Its the startUp sequence for DCGA. We have to create all the
required
	 * components and initialize them if needed.
	 *
	 * @see org.belami.dcga.computation.Computation#startUp()
	 */
	public void startUp() {
		ui.initialize();

	}

	/**
	 * This is part of the singleton pattern. We provide the only
existing
	 * interface with this method
	 *
	 * @return instance of the computation
	 */
	protected static Computation getInstance() {
		if (INSTANCE == null) {
			INSTANCE = new ComputationImpl();
		}
		return INSTANCE;
	}

	/**
```

```
	 * The controller is told to be initialized. This initialization 
means to
	 * tell the subcomponents also to initialize themselves. This 
method is
	 * called when a new patientId is set.
	 * 
	 */
	private void initialize() {
		// PatientManager does not have to be initialized because 
the
		// patientManager itself performs this function call
		taskManager.initialize();
		// DO we still need this?

	}

	/**
	 * Set the current room variable in this component to a new 
value. Its a
	 * setter methods for the private variable {@link #currentRoom}
	 * 
	 * @param id
	 *            of the current room
	 * 
	 */
	private void setCurrentRoom(int roomId) {
		this.currentRoom = roomId;
	}

	/**
	 * @see 
org.belami.dcga.computation.Computation#addInformation(org.belami.dcg
a.common_datastructures.Information)
	 */
	public void addInformation(Information information) {
		informationManager.addInformation(information);
	}

	/**
	 * @see 
org.belami.dcga.computation.Computation#deleteComment(int)
	 */
	public void deleteComment(int commentId) {
		commentManager.deleteComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getComment(int)
	 */
```

```
	public Comment getComment(int commentId) {
		return commentManager.getComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getCurrentRoom()
	 */
	public int getCurrentRoom() {
		return currentRoom;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#getInformationList()
	 */
	public Collection<Information> getInformationList() {
		return informationManager.getInformationList();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientInfo()
	 */
	public ElderlyPerson getPatientInfo() {
		return patientManager.getPatientInfo();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getTaskList()
	 */
	public Vector<Task> getTaskList() {
		return taskManager.getTaskList();
	}

	/**
	 * @throws RoomNotVisitedException
	 * @see
org.belami.dcga.computation.Computation#markTaskAsCompleted(int)
	 */
	public void markTaskAsCompleted(int taskId, boolean override)
			throws RoomNotVisitedException {
		taskManager.markTaskAsCompletedManually(taskId, override);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setTaskEventDone(TaskEvent)
	 */
	public void setTaskEventDone(TaskEvent taskEvent) {
```

```
		taskManager.setTaskEventDone(taskEvent);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addUnplannedTask(Task)
	 */
	public void addUnplannedTask(Task unplannedTask) {
		taskManager.addUnplannedTask(unplannedTask);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onAmiCaConnected(int)
	 */
	public ElderlyPerson onAmiCaConnected(int patientId) {
		setPatientId(patientId);
		return patientManager.getPatientInfo();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onNewRoomEntered(int)
	 */
	public void onNewRoomEntered(int roomId) {
		setCurrentRoom(roomId);
		taskManager.sort();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setPatientId(int)
	 */
	public void setPatientId(int patientId) {
		patientManager.setPatientId(patientId);
		initialize();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startDownload()
	 */
	public void startDownload() {
		synchronization.initDownload();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startUpload()
	 */
	public void startUpload() {
```

```
		synchronization.initUpload();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#storeComment(org.belami.dcga.
common_datastructures.Comment)
	 */
	public void storeComment(Comment comment) {
		commentManager.storeComment(comment);
	}

	/**
	 * Register the Observer at the subcomponents.
	 *
	 * @see
org.belami.dcga.computation.Computation#unregisterObserver(java.util.
Observable)
	 */
	public void unregisterObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
			case OpenTaskWarning:
				taskMan-
ager.deleteOpenTaskWarningObserver(observer);
				break;
			case TaskList:
				taskManager.deleteTaskListObserver(observer);
				break;
			case CommentList:
				commentManager.deleteObserver(observer);
				break;
			case Patient:
				patientManager.deleteObserver(observer);
				break;
			case InformationList:
				informationManager.deleteObserver(observer);
				break;
		}
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#registerObserver(java.util.Ob
servable)
	 */
	public void registerObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
```

```
                case OpenTaskWarning:
                     taskMan-
ager.addOpenTaskWarningObserver(observer);
                     break;
                case TaskList:
                     taskManager.addTaskListObserver(observer);
                     break;
                case CommentList:
                     commentManager.addObserver(observer);
                     break;
                case Patient:
                     patientManager.addObserver(observer);
                     break;
                case InformationList:
                     informationManager.addObserver(observer);
                     break;
           }

     }

     /**
      * @see
org.belami.dcga.computation.Computation#onApartmentLeft()
      */
     public void onApartmentLeft(Date date) {
          patientManager.setLastVisit(date);
          taskManager.onApartmentLeft();
     }

     /**
      * @see
org.belami.dcga.computation.Computation#wasRoomVisited(int)
      */
     public boolean wasRoomVisited(int roomId) {
          /**
           * This method was formerly called wasRoomVisited but Lo-
cationManager
           * implemented it with another name. Perhaps it was a non-
consistent
           * specification
           */
          return locationManager.wasRoomEntered(roomId);
     }

     /**
      * @see org.belami.dcga.computation.Computation#getPatientList()
      */
     public Collection<ElderlyPersonShortInfo> getPatientList() {
          return patientManager.getPatientList();
```

```
    }

    /**
     * @see org.belami.dcga.computation.Computation#getCommentList()
     */
    public Collection<CommentShortInfo> getCommentList() {
        return commentManager.getCommentList();
    }
}
```

1.8.6 Exercise to Experience Package for Location Manager Group: Uncommunicative Name

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Uncommunicative Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

CommonDataStructures:: Task.java → in your code code smells of uncommunicative name couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.common_datastructures;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;

import org.belami.dcga.computation.Computation;

public class Task implements Serializable, Comparable<Task> {

     /**
      * Task is not yet done.
      */
     public static final int UNDONE = 0;

     public static final int DONE_SYSTEM = 1;

     public static final int DONE_CG = 2;

     private int taskId;

     private int priority;

     private String description;
```

```java
	// room-Id
	private String location;

	// room-ID as Integer, needed by taskmanager!
	private int roomID;

	/**
	 * @see org.belami.common_datastructures.Task
	 */
	private int state;

	/**
	 * indicates, whether the task can be automatically marked as
completed or
	 * not. If the task can be auto-marked it is still possible to
mark it
	 * manually.
	 */
	private boolean autoMarkable;

	private boolean unplannedTask;

	// Stores TaskEvents needed for auto-completion, set it with
addTaskEvents()
	private HashSet<TaskEvent> taskEvents = new Hash-
Set<TaskEvent>();

	// required: a no-args constructor
	public Task() {
		taskId = -1;
		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;

	}

	// Constructor where the Id is set
	public Task(int tId) {
		taskId = tId;

		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
```

```java
        unplannedTask = false;
    }

    // implements the getter methods of the class
    public int getTaskId() {
        return taskId;
    }

    public int getPriority() {
        return priority;
    }

    public String getDescription() {
        return description;
    }

    public String getLocation() {
        return location;
    }

    public int getState() {
        return state;
    }

    public boolean isAutoMarkable() {
        return autoMarkable;
    }

    public boolean getUnplannedTask() {
        return unplannedTask;
    }

    // implements the setter methods of the class
    public void setTaskId(int Id) {
        taskId = Id;
    }

    public void setPriority(int prio) {
        priority = prio;
    }

    public void setDescription(String desc) {
        description = desc;
    }

    public void setLocation(String loc) {
        location = loc;
    }
```

```
	public void setState(int st) {
		state = st;
	}

	public void setAutoMarkable(boolean mM) {
		autoMarkable = mM;
	}

	public void setUnplannedTask(boolean uT) {
		unplannedTask = uT;
	}

	public void addTaskEvents(Collection<TaskEvent> events) {
		taskEvents = new HashSet<TaskEvent>(events);
	}

	/**
	 * Removes TaskEvent "event" from the taskEvents Set.
	 *
	 * @param event
	 *            event to delete (equivalent to mark as done)
	 */
	public void setTaskEventDone(TaskEvent event) {
		taskEvents.remove(event);
	}

	/**
	 * checks, if the Task is ready. If the task has to be marked as
done
	 * manually (check for completeness not possible), false is re-
turned.
	 * Otherwise it returns true, if all taskevents are done (in
this case
	 * taskEvents hashset is empty
	 *
	 * @return
	 */
	public boolean isReady() {
		if (autoMarkable == true && taskEvents.isEmpty())
			return true;
		else
			return false;
	}

	public int getRoomID() {
		return roomID;
	}
```

```java
public int compareTo(Task task) {
	int curRoom = Computation.INSTANCE.getCurrentRoom();
	if (task.getRoomID() != curRoom) {
		if (getRoomID() != curRoom)
			return getPriority() - task.getPriority();
		else
			return 1;
	} else {
		if (getRoomID() != curRoom)
			return -1;
		else
			return getPriority() - task.getPriority();
	}
}

public String toString() {
	return description;
}

public void setRoomID(int roomID) {
	this.roomID = roomID;
}

@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	final Task other = (Task) obj;
	if (autoMarkable != other.autoMarkable)
		return false;
	if (description == null) {
		if (other.description != null)
			return false;
	} else if (!description.equals(other.description))
		return false;
	if (location == null) {
		if (other.location != null)
			return false;
	} else if (!location.equals(other.location))
		return false;
	if (priority != other.priority)
		return false;
	if (roomID != other.roomID)
```

```java
                return false;
            if (state != other.state)
                return false;
            if (taskId != other.taskId)
                return false;
            if (unplannedTask != other.unplannedTask)
                return false;
            return true;
        }

}
```

1.8.7 Exercise to Experience Package for Persistence Group: Comments

Your Name: ______________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Computation:ComputationImpl.java → in your code code smells of comments couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.computation;

import java.util.Collection;
import java.util.Date;
import java.util.Observer;
import java.util.Vector;

import org.belami.dcga.common_datastructures.Comment;
import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.belami.dcga.computation.commentmanager.CommentManager;
import org.belami.dcga.computation.commentmanager.CommentManagerImpl;
import
org.belami.dcga.computation.informationmanager.InformationManager;
import
org.belami.dcga.computation.informationmanager.InformationManagerImpl
;
import org.belami.dcga.computation.patientmanager.PatientManager;
import org.belami.dcga.computation.patientmanager.PatientManagerImpl;
import
org.belami.dcga.computation.taskmanager.RoomNotVisitedException;
import org.belami.dcga.computation.taskmanager.TaskManager;
```

```java
import org.belami.dcga.computation.taskmanager.TaskManagerImpl;
import org.belami.dcga.location_manager.LocationManager;
import org.belami.dcga.synchronization.Synchronization;
import org.belami.dcga.ui.UI;

/**
 * This Class is an implementation of the Computation Interface where
the
 * communication is controlled. For a detailed description have a
look at the
 * interface {@link Computation}
 */
class ComputationImpl implements Computation {


	/**
	 * Main method for the program. The computation controller is
instantiated
	 * which begins to execute a startup sequence
	 *
	 * @param args
	 *            command line arguments (not specified yet)
	 */
	public static void main(String[] args) {
		Computation.INSTANCE.startUp();
	}

	/**
	 * Provides a singleton instance for the computation component
	 */
	private static ComputationImpl INSTANCE = null;

	/**
	 * Store the current room. The value -1 means that the room was
not yet set.
	 */
	private int currentRoom = -1;

	/**
	 * A singleton instance of the TaskManager. We need this in-
stance to work on
	 * it and this is also needed for the testcases
	 *
	 * @see TaskManager
	 * @see TaskManagerImpl
	 */
	private TaskManager taskManager = TaskManager.INSTANCE;
```

```
    /**
     * An instance of the CommentManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see CommentManager
     * @see CommentManagerImpl
     */
    private CommentManager commentManager = new CommentManager-
Impl();

    /**
     * An instance of the InformationManager. We need this instance
to work on
     * it and this is also needed for the testcases
     *
     * @see InformationManager
     * @see InformationManagerImpl
     */
    private InformationManager informationManager = new Information-
ManagerImpl();

    /**
     * An instance of the PatientManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see PatientManager
     * @see PatientManagerImpl
     */
    private PatientManager patientManager = new PatientManager-
Impl();

    /**
     * An instance of the LocationManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see LocationManager
     */
    private LocationManager locationManager = LocationMan-
ager.INSTANCE;

    /**
     * An instance of the Synchronization. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see Synchronization
```

```
	 */
	private Synchronization synchronization = Synchroniza-
tion.INSTANCE;

	/**
	 * An instance of the UI. We need this instance to work on it
	 * and this is also needed for the testcases
	 *
	 * @see UI
	 */
	private UI ui = UI.INSTANCE;

	/**
	 * By using the singleton pattern we have to make a private con-
structor. By
	 * this we assure that there can only be one instance at any
time.
	 *
	 */
	private ComputationImpl() {
	}

	/**
	 * Its the startUp sequence for DCGA. We have to create all the
required
	 * components and initialize them if needed.
	 *
	 * @see org.belami.dcga.computation.Computation#startUp()
	 */
	public void startUp() {
		ui.initialize();

	}

	/**
	 * This is part of the singleton pattern. We provide the only
existing
	 * interface with this method
	 *
	 * @return instance of the computation
	 */
	protected static Computation getInstance() {
		if (INSTANCE == null) {
			INSTANCE = new ComputationImpl();
		}
		return INSTANCE;
	}

	/**
```

```
	 * The controller is told to be initialized. This initialization
means to
	 * tell the subcomponents also to initialize themselves. This
method is
	 * called when a new patientId is set.
	 * 
	 */
	private void initialize() {
		// PatientManager does not have to be initialized because
the
		// patientManager itself performs this function call
		taskManager.initialize();
		// DO we still need this?

	}

	/**
	 * Set the current room variable in this component to a new
value. Its a
	 * setter methods for the private variable {@link #currentRoom}
	 * 
	 * @param id
	 *            of the current room
	 * 
	 */
	private void setCurrentRoom(int roomId) {
		this.currentRoom = roomId;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addInformation(org.belami.dcg
a.common_datastructures.Information)
	 */
	public void addInformation(Information information) {
		informationManager.addInformation(information);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#deleteComment(int)
	 */
	public void deleteComment(int commentId) {
		commentManager.deleteComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getComment(int)
	 */
```

```java
	public Comment getComment(int commentId) {
		return commentManager.getComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getCurrentRoom()
	 */
	public int getCurrentRoom() {
		return currentRoom;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#getInformationList()
	 */
	public Collection<Information> getInformationList() {
		return informationManager.getInformationList();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientInfo()
	 */
	public ElderlyPerson getPatientInfo() {
		return patientManager.getPatientInfo();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getTaskList()
	 */
	public Vector<Task> getTaskList() {
		return taskManager.getTaskList();
	}

	/**
	 * @throws RoomNotVisitedException
	 * @see
org.belami.dcga.computation.Computation#markTaskAsCompleted(int)
	 */
	public void markTaskAsCompleted(int taskId, boolean override)
			throws RoomNotVisitedException {
		taskManager.markTaskAsCompletedManually(taskId, override);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setTaskEventDone(TaskEvent)
	 */
	public void setTaskEventDone(TaskEvent taskEvent) {
```

```
		taskManager.setTaskEventDone(taskEvent);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addUnplannedTask(Task)
	 */
	public void addUnplannedTask(Task unplannedTask) {
		taskManager.addUnplannedTask(unplannedTask);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onAmiCaConnected(int)
	 */
	public ElderlyPerson onAmiCaConnected(int patientId) {
		setPatientId(patientId);
		return patientManager.getPatientInfo();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onNewRoomEntered(int)
	 */
	public void onNewRoomEntered(int roomId) {
		setCurrentRoom(roomId);
		taskManager.sort();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setPatientId(int)
	 */
	public void setPatientId(int patientId) {
		patientManager.setPatientId(patientId);
		initialize();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startDownload()
	 */
	public void startDownload() {
		synchronization.initDownload();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startUpload()
	 */
	public void startUpload() {
```

```
		synchronization.initUpload();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#storeComment(org.belami.dcga.
common_datastructures.Comment)
	 */
	public void storeComment(Comment comment) {
		commentManager.storeComment(comment);
	}

	/**
	 * Register the Observer at the subcomponents.
	 *
	 * @see
org.belami.dcga.computation.Computation#unregisterObserver(java.util.
Observable)
	 */
	public void unregisterObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
			case OpenTaskWarning:
				taskMan-
ager.deleteOpenTaskWarningObserver(observer);
				break;
			case TaskList:
				taskManager.deleteTaskListObserver(observer);
				break;
			case CommentList:
				commentManager.deleteObserver(observer);
				break;
			case Patient:
				patientManager.deleteObserver(observer);
				break;
			case InformationList:
				informationManager.deleteObserver(observer);
				break;
		}
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#registerObserver(java.util.Ob
servable)
	 */
	public void registerObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
```

```
				case OpenTaskWarning:
					taskMan-
ager.addOpenTaskWarningObserver(observer);
					break;
				case TaskList:
					taskManager.addTaskListObserver(observer);
					break;
				case CommentList:
					commentManager.addObserver(observer);
					break;
				case Patient:
					patientManager.addObserver(observer);
					break;
				case InformationList:
					informationManager.addObserver(observer);
					break;
			}

	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onApartmentLeft()
	 */
	public void onApartmentLeft(Date date) {
		patientManager.setLastVisit(date);
		taskManager.onApartmentLeft();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#wasRoomVisited(int)
	 */
	public boolean wasRoomVisited(int roomId) {
		/**
		 * This method was formerly called wasRoomVisited but Lo-
cationManager
		 * implemented it with another name. Perhaps it was a non-
consistent
		 * specification
		 */
		return locationManager.wasRoomEntered(roomId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientList()
	 */
	public Collection<ElderlyPersonShortInfo> getPatientList() {
		return patientManager.getPatientList();
```

```java
}

/**
 * @see org.belami.dcga.computation.Computation#getCommentList()
 */
public Collection<CommentShortInfo> getCommentList() {
	return commentManager.getCommentList();
}
```

1.8.8 Exercise to Experience Package for Persistence Group: Uncommunicative Name

Your Name: ________________________

Your Subject-ID: _____<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Uncommunicative Name
2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"
3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

CommonDataStructures:: Task.java → in your code code smells of uncommunicative name couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.common_datastructures;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;

import org.belami.dcga.computation.Computation;

public class Task implements Serializable, Comparable<Task> {

	/**
	 * Task is not yet done.
	 */
	public static final int UNDONE = 0;

	public static final int DONE_SYSTEM = 1;

	public static final int DONE_CG = 2;

	private int taskId;

	private int priority;

	private String description;

	// room-Id
```

```java
	private String location;

	// room-ID as Integer, needed by taskmanager!
	private int roomID;

	/**
	 * @see org.belami.common_datastructures.Task
	 */
	private int state;

	/**
	 * indicates, whether the task can be automatically marked as
completed or
	 * not. If the task can be auto-marked it is still possible to
mark it
	 * manually.
	 */
	private boolean autoMarkable;

	private boolean unplannedTask;

	// Stores TaskEvents needed for auto-completion, set it with
addTaskEvents()
	private HashSet<TaskEvent> taskEvents = new Hash-
Set<TaskEvent>();

	// required: a no-args constructor
	public Task() {
		taskId = -1;
		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;

	}

	// Constructor where the Id is set
	public Task(int tId) {
		taskId = tId;

		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;
	}
```

```java
// implements the getter methods of the class
public int getTaskId() {
	return taskId;
}

public int getPriority() {
	return priority;
}

public String getDescription() {
	return description;
}

public String getLocation() {
	return location;
}

public int getState() {
	return state;
}

public boolean isAutoMarkable() {
	return autoMarkable;
}

public boolean getUnplannedTask() {
	return unplannedTask;
}

// implements the setter methods of the class
public void setTaskId(int Id) {
	taskId = Id;
}

public void setPriority(int prio) {
	priority = prio;
}

public void setDescription(String desc) {
	description = desc;
}

public void setLocation(String loc) {
	location = loc;
}

public void setState(int st) {
```

```
		state = st;
	}

	public void setAutoMarkable(boolean mM) {
		autoMarkable = mM;
	}

	public void setUnplannedTask(boolean uT) {
		unplannedTask = uT;
	}

	public void addTaskEvents(Collection<TaskEvent> events) {
		taskEvents = new HashSet<TaskEvent>(events);
	}

	/**
	 * Removes TaskEvent "event" from the taskEvents Set.
	 *
	 * @param event
	 *            event to delete (equivalent to mark as done)
	 */
	public void setTaskEventDone(TaskEvent event) {
		taskEvents.remove(event);
	}

	/**
	 * checks, if the Task is ready. If the task has to be marked as
done
	 * manually (check for completeness not possible), false is re-
turned.
	 * Otherwise it returns true, if all taskevents are done (in
this case
	 * taskEvents hashset is empty
	 *
	 * @return
	 */
	public boolean isReady() {
		if (autoMarkable == true && taskEvents.isEmpty())
			return true;
		else
			return false;
	}

	public int getRoomID() {
		return roomID;
	}

	public int compareTo(Task task) {
		int curRoom = Computation.INSTANCE.getCurrentRoom();
```

```java
		if (task.getRoomID() != curRoom) {
			if (getRoomID() != curRoom)
				return getPriority() - task.getPriority();
			else
				return 1;
		} else {
			if (getRoomID() != curRoom)
				return -1;
			else
				return getPriority() - task.getPriority();
		}
	}

	public String toString() {
		return description;
	}

	public void setRoomID(int roomID) {
		this.roomID = roomID;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		final Task other = (Task) obj;
		if (autoMarkable != other.autoMarkable)
			return false;
		if (description == null) {
			if (other.description != null)
				return false;
		} else if (!description.equals(other.description))
			return false;
		if (location == null) {
			if (other.location != null)
				return false;
		} else if (!location.equals(other.location))
			return false;
		if (priority != other.priority)
			return false;
		if (roomID != other.roomID)
			return false;
		if (state != other.state)
```

```
            return false;
        if (taskId != other.taskId)
            return false;
        if (unplannedTask != other.unplannedTask)
            return false;
        return true;
    }

}
```

1.8.9 Exercise to Experience Package for Synchronization Group: Comments

Your Name: ________________________
Your Subject-ID: _____<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

Computation:ComputationImpl.java → in your code code smells of comments couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.computation;

import java.util.Collection;
import java.util.Date;
import java.util.Observer;
import java.util.Vector;

import org.belami.dcga.common_datastructures.Comment;
import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.common_datastructures.TaskEvent;
import org.belami.dcga.computation.commentmanager.CommentManager;
import org.belami.dcga.computation.commentmanager.CommentManagerImpl;
import
org.belami.dcga.computation.informationmanager.InformationManager;
import
org.belami.dcga.computation.informationmanager.InformationManagerImpl
;
import org.belami.dcga.computation.patientmanager.PatientManager;
import org.belami.dcga.computation.patientmanager.PatientManagerImpl;
import
org.belami.dcga.computation.taskmanager.RoomNotVisitedException;
import org.belami.dcga.computation.taskmanager.TaskManager;
```

```
import org.belami.dcga.computation.taskmanager.TaskManagerImpl;
import org.belami.dcga.location_manager.LocationManager;
import org.belami.dcga.synchronization.Synchronization;
import org.belami.dcga.ui.UI;

/**
 * This Class is an implementation of the Computation Interface where
the
 * communication is controlled. For a detailed description have a
look at the
 * interface {@link Computation}
 */
class ComputationImpl implements Computation {



     /**
      * Main method for the program. The computation controller is
instantiated
      * which begins to execute a startup sequence
      *
      * @param args
      *            command line arguments (not specified yet)
      */
     public static void main(String[] args) {
          Computation.INSTANCE.startUp();
     }

     /**
      * Provides a singleton instance for the computation component
      */
     private static ComputationImpl INSTANCE = null;

     /**
      * Store the current room. The value -1 means that the room was
not yet set.
      */
     private int currentRoom = -1;

     /**
      * A singleton instance of the TaskManager. We need this in-
stance to work on
      * it and this is also needed for the testcases
      *
      * @see TaskManager
      * @see TaskManagerImpl
      */
     private TaskManager taskManager = TaskManager.INSTANCE;
```

```
    /**
     * An instance of the CommentManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see CommentManager
     * @see CommentManagerImpl
     */
    private CommentManager commentManager = new CommentManager-
Impl();

    /**
     * An instance of the InformationManager. We need this instance
to work on
     * it and this is also needed for the testcases
     *
     * @see InformationManager
     * @see InformationManagerImpl
     */
    private InformationManager informationManager = new Information-
ManagerImpl();

    /**
     * An instance of the PatientManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see PatientManager
     * @see PatientManagerImpl
     */
    private PatientManager patientManager = new PatientManager-
Impl();

    /**
     * An instance of the LocationManager. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see LocationManager
     */
    private LocationManager locationManager = LocationMan-
ager.INSTANCE;

    /**
     * An instance of the Synchronization. We need this instance to
work on it
     * and this is also needed for the testcases
     *
     * @see Synchronization
```

```
     */
    private Synchronization synchronization = Synchroniza-
tion.INSTANCE;

    /**
     * An instance of the UI. We need this instance to work on it
     * and this is also needed for the testcases
     *
     * @see UI
     */
    private UI ui = UI.INSTANCE;

    /**
     * By using the singleton pattern we have to make a private con-
structor. By
     * this we assure that there can only be one instance at any
time.
     *
     */
    private ComputationImpl() {
    }

    /**
     * Its the startUp sequence for DCGA. We have to create all the
required
     * components and initialize them if needed.
     *
     * @see org.belami.dcga.computation.Computation#startUp()
     */
    public void startUp() {
        ui.initialize();

    }

    /**
     * This is part of the singleton pattern. We provide the only
existing
     * interface with this method
     *
     * @return instance of the computation
     */
    protected static Computation getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new ComputationImpl();
        }
        return INSTANCE;
    }

    /**
```

```
	 * The controller is told to be initialized. This initialization
means to
	 * tell the subcomponents also to initialize themselves. This
method is
	 * called when a new patientId is set.
	 *
	 */
	private void initialize() {
		// PatientManager does not have to be initialized because
the
		// patientManager itself performs this function call
		taskManager.initialize();
		// DO we still need this?

	}

	/**
	 * Set the current room variable in this component to a new
value. Its a
	 * setter methods for the private variable {@link #currentRoom}
	 *
	 * @param id
	 *            of the current room
	 *
	 */
	private void setCurrentRoom(int roomId) {
		this.currentRoom = roomId;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addInformation(org.belami.dcg
a.common_datastructures.Information)
	 */
	public void addInformation(Information information) {
		informationManager.addInformation(information);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#deleteComment(int)
	 */
	public void deleteComment(int commentId) {
		commentManager.deleteComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getComment(int)
	 */
```

```
	public Comment getComment(int commentId) {
		return commentManager.getComment(commentId);
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getCurrentRoom()
	 */
	public int getCurrentRoom() {
		return currentRoom;
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#getInformationList()
	 */
	public Collection<Information> getInformationList() {
		return informationManager.getInformationList();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getPatientInfo()
	 */
	public ElderlyPerson getPatientInfo() {
		return patientManager.getPatientInfo();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#getTaskList()
	 */
	public Vector<Task> getTaskList() {
		return taskManager.getTaskList();
	}

	/**
	 * @throws RoomNotVisitedException
	 * @see
org.belami.dcga.computation.Computation#markTaskAsCompleted(int)
	 */
	public void markTaskAsCompleted(int taskId, boolean override)
			throws RoomNotVisitedException {
		taskManager.markTaskAsCompletedManually(taskId, override);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setTaskEventDone(TaskEvent)
	 */
	public void setTaskEventDone(TaskEvent taskEvent) {
```

```
		taskManager.setTaskEventDone(taskEvent);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#addUnplannedTask(Task)
	 */
	public void addUnplannedTask(Task unplannedTask) {
		taskManager.addUnplannedTask(unplannedTask);
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onAmiCaConnected(int)
	 */
	public ElderlyPerson onAmiCaConnected(int patientId) {
		setPatientId(patientId);
		return patientManager.getPatientInfo();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#onNewRoomEntered(int)
	 */
	public void onNewRoomEntered(int roomId) {
		setCurrentRoom(roomId);
		taskManager.sort();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#setPatientId(int)
	 */
	public void setPatientId(int patientId) {
		patientManager.setPatientId(patientId);
		initialize();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startDownload()
	 */
	public void startDownload() {
		synchronization.initDownload();
	}

	/**
	 * @see org.belami.dcga.computation.Computation#startUpload()
	 */
	public void startUpload() {
```

```
		synchronization.initUpload();
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#storeComment(org.belami.dcga.
common_datastructures.Comment)
	 */
	public void storeComment(Comment comment) {
		commentManager.storeComment(comment);
	}

	/**
	 * Register the Observer at the subcomponents.
	 *
	 * @see
org.belami.dcga.computation.Computation#unregisterObserver(java.util.
Observable)
	 */
	public void unregisterObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
			case OpenTaskWarning:
				taskMan-
ager.deleteOpenTaskWarningObserver(observer);
				break;
			case TaskList:
				taskManager.deleteTaskListObserver(observer);
				break;
			case CommentList:
				commentManager.deleteObserver(observer);
				break;
			case Patient:
				patientManager.deleteObserver(observer);
				break;
			case InformationList:
				informationManager.deleteObserver(observer);
				break;
		}
	}

	/**
	 * @see
org.belami.dcga.computation.Computation#registerObserver(java.util.Ob
servable)
	 */
	public void registerObserver(Observer observer, ControllerOb-
servables observables) {
		switch (observables) {
```

```
                case OpenTaskWarning:
                     taskMan-
ager.addOpenTaskWarningObserver(observer);
                     break;
                case TaskList:
                     taskManager.addTaskListObserver(observer);
                     break;
                case CommentList:
                     commentManager.addObserver(observer);
                     break;
                case Patient:
                     patientManager.addObserver(observer);
                     break;
                case InformationList:
                     informationManager.addObserver(observer);
                     break;
           }

      }

      /**
       * @see
org.belami.dcga.computation.Computation#onApartmentLeft()
       */
      public void onApartmentLeft(Date date) {
           patientManager.setLastVisit(date);
           taskManager.onApartmentLeft();
      }

      /**
       * @see
org.belami.dcga.computation.Computation#wasRoomVisited(int)
       */
      public boolean wasRoomVisited(int roomId) {
           /**
            * This method was formerly called wasRoomVisited but Lo-
cationManager
            * implemented it with another name. Perhaps it was a non-
consistent
            * specification
            */
           return locationManager.wasRoomEntered(roomId);
      }

      /**
       * @see org.belami.dcga.computation.Computation#getPatientList()
       */
      public Collection<ElderlyPersonShortInfo> getPatientList() {
           return patientManager.getPatientList();
```

```java
}

/**
 * @see org.belami.dcga.computation.Computation#getCommentList()
 */
public Collection<CommentShortInfo> getCommentList() {
     return commentManager.getCommentList();
}
```

1.8.10 Exercise to Experience Package for Synchronization Group: Uncommunicative Name

Your Name: ________________________

Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Uncommunicative Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

CommonDataStructures:: Task.java → in your code code smells of uncommunicative name couldn't be found. Therefore, another DCGA file is used.

```
package org.belami.dcga.common_datastructures;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashSet;

import org.belami.dcga.computation.Computation;

public class Task implements Serializable, Comparable<Task> {

     /**
      * Task is not yet done.
      */
     public static final int UNDONE = 0;

     public static final int DONE_SYSTEM = 1;

     public static final int DONE_CG = 2;

     private int taskId;

     private int priority;

     private String description;
```

```
	// room-Id
	private String location;

	// room-ID as Integer, needed by taskmanager!
	private int roomID;

	/**
	 * @see org.belami.common_datastructures.Task
	 */
	private int state;

	/**
	 * indicates, whether the task can be automatically marked as
completed or
	 * not. If the task can be auto-marked it is still possible to
mark it
	 * manually.
	 */
	private boolean autoMarkable;

	private boolean unplannedTask;

	// Stores TaskEvents needed for auto-completion, set it with
addTaskEvents()
	private HashSet<TaskEvent> taskEvents = new Hash-
Set<TaskEvent>();

	// required: a no-args constructor
	public Task() {
		taskId = -1;
		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
		unplannedTask = false;

	}

	// Constructor where the Id is set
	public Task(int tId) {
		taskId = tId;

		priority = 0;
		description = "INITIAL";
		location = "";
		state = UNDONE;
		autoMarkable = false;
```

```
        unplannedTask = false;
    }

    // implements the getter methods of the class
    public int getTaskId() {
        return taskId;
    }

    public int getPriority() {
        return priority;
    }

    public String getDescription() {
        return description;
    }

    public String getLocation() {
        return location;
    }

    public int getState() {
        return state;
    }

    public boolean isAutoMarkable() {
        return autoMarkable;
    }

    public boolean getUnplannedTask() {
        return unplannedTask;
    }

    // implements the setter methods of the class
    public void setTaskId(int Id) {
        taskId = Id;
    }

    public void setPriority(int prio) {
        priority = prio;
    }

    public void setDescription(String desc) {
        description = desc;
    }

    public void setLocation(String loc) {
        location = loc;
    }
```

```
	public void setState(int st) {
		state = st;
	}

	public void setAutoMarkable(boolean mM) {
		autoMarkable = mM;
	}

	public void setUnplannedTask(boolean uT) {
		unplannedTask = uT;
	}

	public void addTaskEvents(Collection<TaskEvent> events) {
		taskEvents = new HashSet<TaskEvent>(events);
	}

	/**
	 * Removes TaskEvent "event" from the taskEvents Set.
	 *
	 * @param event
	 *            event to delete (equivalent to mark as done)
	 */
	public void setTaskEventDone(TaskEvent event) {
		taskEvents.remove(event);
	}

	/**
	 * checks, if the Task is ready. If the task has to be marked as
done
	 * manually (check for completeness not possible), false is re-
turned.
	 * Otherwise it returns true, if all taskevents are done (in
this case
	 * taskEvents hashset is empty
	 *
	 * @return
	 */
	public boolean isReady() {
		if (autoMarkable == true && taskEvents.isEmpty())
			return true;
		else
			return false;
	}

	public int getRoomID() {
		return roomID;
	}
```

```java
public int compareTo(Task task) {
	int curRoom = Computation.INSTANCE.getCurrentRoom();
	if (task.getRoomID() != curRoom) {
		if (getRoomID() != curRoom)
			return getPriority() - task.getPriority();
		else
			return 1;
	} else {
		if (getRoomID() != curRoom)
			return -1;
		else
			return getPriority() - task.getPriority();
	}
}

public String toString() {
	return description;
}

public void setRoomID(int roomID) {
	this.roomID = roomID;
}

@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	final Task other = (Task) obj;
	if (autoMarkable != other.autoMarkable)
		return false;
	if (description == null) {
		if (other.description != null)
			return false;
	} else if (!description.equals(other.description))
		return false;
	if (location == null) {
		if (other.location != null)
			return false;
	} else if (!location.equals(other.location))
		return false;
	if (priority != other.priority)
		return false;
	if (roomID != other.roomID)
```

```
            return false;
        if (state != other.state)
            return false;
        if (taskId != other.taskId)
            return false;
        if (unplannedTask != other.unplannedTask)
            return false;
        return true;
    }

}
```

1.8.11 Exercise to Experience Package for UI Group: Comments

Your Name: ___________________________
Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Comments

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

uiSystem :visualizationUnit.java

```
package org.belami.dcga.ui.ui_system.visualization_unit;

import java.lang.reflect.InvocationTargetException;
import java.util.Collection;
import java.util.GregorianCalendar;
import java.util.Vector;

import javax.swing.JOptionPane;

import org.belami.dcga.common_datastructures.CommentShortInfo;
import org.belami.dcga.common_datastructures.ElderlyPerson;
import org.belami.dcga.common_datastructures.ElderlyPersonShortInfo;
import org.belami.dcga.common_datastructures.Information;
import org.belami.dcga.common_datastructures.Task;
import org.belami.dcga.ui.ui_system.interaction_unit.InteractionUnit;

/**
 * The visualisatin unit creates the display of the DCGA.
 *
 * @author A-Team
 * @version 1.0
 */
public class VisualizationUnit {

	/**
	 * The controller of the gui.
	 */
	InteractionUnit interactionUnit;
```

```
	/**
	 * The main frame of the gui.
	 */
	private MainFrame mainFrame;

	/**
	 * The dialog to choose a patient manually
	 */
	PatientsDialog patientsDialog;

	/**
	 * The dialog to show the patient informations
	 */
	PatientInfoDialog patientInfoDialog;

	/**
	 * The synchronization dialog to show while uploading / 
downloading data
	 */
	SynchronizationDialog syncDialog;

	/**
	 * The frame to enter text comment
	 */
	CommentInputFrame commentInputFrame;

	/**
	 * Creates an instance of the visualisation unit.
	 * The main frame is automatically created by the creation.
	 * <code>VisualisationUnit</code> needs an interaction unit as 
controller,
	 * which must be set using the <code>setInteractionUnit</code> 
Method.
	 */
	public VisualizationUnit() {

		/*
		 * Schedules a job for the event-dispatching thread
		 * to create and show the main frame.
		 */
		try {
			javax.swing.SwingUtilities.invokeAndWait(new Run-
nable() {
				public void run() {
					createAndShowMainFrame();
				}
			});
		} catch (InterruptedException e) {
```

```
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }

        }

        /**
         * Sets the controller for the display.
         * @param interactionUnit  The controller
         */
        public void setInteractionUnit(InteractionUnit interactionUnit)
{
            this.interactionUnit = interactionUnit;
        }

        /**
        * Creates the main frame and shows it.  For thread safety,
        * this method should be invoked from the event-dispatching
thread.
        */
        private void createAndShowMainFrame() {
            mainFrame = MainFrame.createMainFrame(this);
        }

        /**
         * Used to set the title of the main window. It contains the
Customer-No,
         * the Patientname and furthermore the actual date and time.
         * (e.g. "Customer: 0815, Ms. Schmidt | Monday, 10/10/2010 |
10:15 PM")
         * @param elderlyPerson
         */
        public void updateTitleBar(ElderlyPerson elderlyPerson) {
            GregorianCalendar today = new GregorianCalendar();
            String minute = ("0"+today.get(GregorianCalendar.MINUTE));
            minute = minute.substring(minute.length()-2, min-
ute.length());
            String hour = ("0"+today.get(GregorianCalendar.HOUR));
            hour = hour.substring(hour.length()-2, hour.length());
                mainFrame.setTitle("Customer: " + elderlyPer-
son.getName()+" | " //Name of elderly Person
                    +(today.get(GregorianCalendar.MONTH)+1)  /*+1,
because of format 0-11*/+ "/"
                    +today.get(GregorianCalendar.DAY_OF_MONTH)+"/"
            //american format
                    +today.get(GregorianCalendar.YEAR)+" | "
            //month/day/year
                    +hour + ":"
```

```
                    +minute);
    }

    /**
     * Updates the "Current Comments" and the "Old Comments"
     * @param commentList
     */
    public void updateComments(Collection<CommentShortInfo> current-
CommentsList, Collection<CommentShortInfo> oldCommentsList) {

          CommentTabbedPane commentTabbedPane =
                    main-
Frame.infoAndCommentPane.commentPanel.commentTabbedPane;

          commentTabbed-
Pane.setNewCurrentCommentsList(currentCommentsList);
          commentTabbedPane.setNewOldCommentsList(oldCommentsList);

    }

    /**
     * Updates the "Done Tasks", "Open Tasks" and the ProgressBar.
     * @param doneTasks, openTasks
     */
    public void updateTasks(Vector<Task> openTaskList, Vector<Task>
doneTaskList) {
          main-
Frame.taskPanel.taskTabbedPane.openTasksListTable.setNewTaskList(open
TaskList);
          main-
Frame.taskPanel.taskTabbedPane.doneTasksListTable.setNewTaskList(done
TaskList);

          main-
Frame.taskPanel.progressPanel.progressBar.setMaximum(openTaskList.siz
e() + doneTaskList.size());
          main-
Frame.taskPanel.progressPanel.progressBar.setValue(openTaskList.size(
));
          main-
Frame.taskPanel.progressPanel.progressBar.setString(String.valueOf(do
neTaskList.size()) + "/"
                    + String.valueOf(openTaskList.size() + do-
neTaskList.size()) + " Tasks completed.");

          main-
Frame.taskPanel.progressPanel.markAsDoneButton.setEnabled(false);
    }
```

```
	/**
	 * Updates the visualization of amiCA information box
	 * @param infoList
	 */
	public void updateInformation(Collection<Information> infoList)
{
	
	this.mainFrame.infoAndCommentPane.infoPanel.setNewInformationLis
t(infoList);
	}

	/**
	 * opens a new window where the user is able to select the cur-
rent patient.
	 * @param patientList
	 */
	public void showPatientList(Collection<ElderlyPersonShortInfo>
patientList) {
		patientsDialog = new PatientsDialog(this, patientList);
	}

	/**
	 * opens a new window where the user can see further information
about the actual patient.
	 * @param dumdidum
	 */
	public void showPatientInformation(ElderlyPerson ep) {
		patientInfoDialog = new PatientInfoDialog(this, ep);
	}

	/**
	 * Changes the image of record button to "start button",
	 * activates the comment buttons
	 */
	public void showNormalButtonState() {
		boolean isCommentSelected;

		// Get the panel with the buttons
		ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

		// Set the state of the buttons
		buttonPanel.recordButton.setSelected(false);
		buttonPanel.recordButton.setEnabled(true);
		buttonPanel.recordButton.setIcon(new
javax.swing.ImageIcon(getClass().getResource(
				buttonPanel.getrecordButtonRes())));
		buttonPanel.playButton.setEnabled(false);
		buttonPanel.stopButton.setEnabled(false);
```

```
		buttonPanel.deleteButton.setEnabled(false);

		isCommentSelected = !main-
Frame.infoAndCommentPane.commentPanel.
			commentTabbed-
Pane.curCommentList.isSelectionEmpty();

		if (isCommentSelected) {
			getMainFrame().infoAndCommentPane.commentPanel.
					buttonPanel.playButton.setEnabled(true);
			getMainFrame().infoAndCommentPane.commentPanel.
					buttonPanel.deleteButton.setEnabled(true);
		}
	}

	/**
	 * Changes the image of record button to “stop button”,
	 * deactivates the other comment buttons.
	 */
	public void showRecordingState() {

		// Get the panel with the buttons
		ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

		// Set the state of the buttons
		buttonPanel.recordButton.setEnabled(true);
		buttonPanel.recordButton.setIcon(new
javax.swing.ImageIcon(getClass().getResource(
			buttonPanel.getrecordStopButtonRes())));
		buttonPanel.playButton.setEnabled(false);
		buttonPanel.stopButton.setEnabled(false);
		buttonPanel.deleteButton.setEnabled(false);

	}

	/**
	 * Deactivates all comment buttons, except the “stop button”
	 * @param dumdidum
	 */
	public void showPlayingState() {

		// Get the panel with the buttons
		ButtonPanel buttonPanel = main-
Frame.infoAndCommentPane.commentPanel.buttonPanel;

		// Set the state of the buttons
		buttonPanel.recordButton.setEnabled(false);
		buttonPanel.playButton.setEnabled(false);
```

```
			buttonPanel.stopButton.setEnabled(true);
			buttonPanel.deleteButton.setEnabled(false);

		}

		/**
		 * Shows a confirmation dialog with a custom text mes-sage, ok
and cancel buttons.
		 */
		public boolean showConfirmationDialog() {

			int dialogReturn = JOptionPane.showConfirmDialog(
					this.mainFrame,
					"The task could not have been completed. Would
you like to complete it anyway?",
					"Confirmation",
					JOptionPane.YES_NO_OPTION);

			if (dialogReturn == JOptionPane.YES_OPTION) {
				return true;
			} else {
				return false;
			}

		}

		/**
		 * Shows a dialog with a comment text message, ok button
		 */
		public void showCommentDialog(String comment) {
			JOptionPane.showMessageDialog(this.mainFrame,
							comment,
							"Text Comment",
							JOption-
Pane.PLAIN_MESSAGE);
		}

		/**
		 * Shows a dialog with a custom text message, ok button
		 * @param dumdidum
		 */
		public void showDialog(String message) {
			JOptionPane.showMessageDialog(this.mainFrame, message);
		}

		/**
		 * Shows a dialog with a text input field, ok and cancel buttons
		 * @param dumdidum
		 */
```

```
    public void showTextCommentInputDialog() {
        commentInputFrame = new CommentInputFrame(this);
    }

    /**
     * Shows a modal window with a custom text message during the
synchronization process.
     * @param message
     * @param dumdidum
     */
    public void showSynchronizationWindow(String message) {
        syncDialog = new SynchronizationDialog(this, message);
    }

    /**
     * Closes the modal window after the synchronization process.
     * @param dumdidum
     */
    public void closeSynchronizationWindow() {
        syncDialog.dispose();
    }

    /**
     * Obtains the controller of the gui.
     * @return a reference to the <code>InteractionUnit</code>
     */
    public InteractionUnit getInteractionUnit() {
        return interactionUnit;
    }

    /**
     * Obtains the main frame of the <code>VisualizationUnit</code>.
     * @return the main frame of the <code>VisualizationUnit</code>.
     */
    public MainFrame getMainFrame() {
        return mainFrame;
    }

    /***
     * Obtains the patients dialog of the
<code>VisualizationUnit</code>.
     * @return the patients dialog of the
<code>VisualizationUnit</code>.
     */
    public PatientsDialog getPatientsDialog() {
        return patientsDialog;
    }
```

1.8.12 Exercise to Experience Package for UI Group: Uncommunicative Name

Your Name: ____________________________

Your Subject-ID: ______<your ID will be filled out by evaluators>

Please put the starting time in here [___ : ___]
Please put the ending time in here [___ : ___]
Exercise:

1. **Identify and mark** with a text marker code smells of the following type: Uncommunicative Name

2. For each identified code smell **state the refactoring** you would apply into the code and **give a subsequent number** - start with "1"

3. **Use the *Answer Sheet for Exercises*. Put the related number in the first column** in order to relate your answer to the identified code smell. Then **explain your decision** (i.e., your stepwise solution in your own words or why you wouldn't remove the code smell).

uiSystem: tastList.java

```
package org.belami.dcga.ui.ui_system.interaction_unit;

import java.util.Observable;
import java.util.Observer;

public class TaskList implements Observer {

	private InteractionUnit interactionUnit;

	TaskList(InteractionUnit interactionUnit) {
		this.interactionUnit = interactionUnit;
	}

	public void update(Observable arg0, Object arg1) {
		interactionUnit.updateTasks();
	}
}
```

uiSystem:CommentTabbedPane.java

```
package org.belami.dcga.ui.ui_system.visualization_unit;

import java.util.Collection;

import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JTabbedPane;
```

```
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import org.belami.dcga.common_datastructures.CommentShortInfo;

/**
 * Appears on the right side of the Mainframe: includes "current" and
"old" comments.
 *
 * @version 1.0
 */
public class CommentTabbedPane extends JTabbedPane {

      /**
       * CommentTabbedPane implements Serializable
       * and should have a <code>serialVersionUID</code>.
       */
      private static final long serialVersionUID = 1L;

      /**
       * The title of the current comments tab.
       */
      private final String currentCommentsTabTitle = "Current Com-
ments";

      /**
       * The title of the old comments tab.
       */
      private final String oldCommentsTabTitle = "Old Comments";

      CurrentCommentsList curCommentList;

      OldCommentsList oldCommentList;



      /**
       * Creates an instance of the <code>CommentTabbedPane</code>.
       * Adds the subcomponents.
       */
      CommentTabbedPane(final VisualizationUnit visUnit) {

            curCommentList = new CurrentCommentsList();
            curCom-
mentList.getSelectionModel().addListSelectionListener(new ListSelec-
tionListener() {
                  public void valueChanged(ListSelectionEvent e) {
                        curCommentListListener(e, visUnit);
```

```
                    }
                });
            oldCommentList = new OldCommentsList();
            oldCom-
mentList.getSelectionModel().addListSelectionListener(new ListSelec-
tionListener() {
                public void valueChanged(ListSelectionEvent e) {
                    oldCommentListListener(e, visUnit);
                    }
                });


            this.addTab(currentCommentsTabTitle, new JScroll-
Pane(curCommentList));
            this.addTab(oldCommentsTabTitle, new JScroll-
Pane(oldCommentList));

      }

      public void setNewCurrentComment-
sList(Collection<CommentShortInfo> commentsList) {
      curCommentList.setListData(commentsList.toArray(new Com-
mentShortInfo[0]));
     }

      public void setNewOldCommentsList(Collection<CommentShortInfo>
commentsList) {
      oldCommentList.setListData(commentsList.toArray(new Com-
mentShortInfo[0]));
     }

      /**
       * curCommentListListener (ActionListener for "Current Comments"
JList)
       * @param e
       * @param visUnit
       */
      private void curCommentListListener(ListSelectionEvent e, Visu-
alizationUnit visUnit) {
            ListSelectionModel curCommentListSelectionModel = (ListSe-
lectionModel)e.getSource();

            if (!curCommentListSelectionModel.isSelectionEmpty()) {
                enableButtons(true, visUnit);
            } else {
                enableButtons(false, visUnit);
            }
      }
      /**
```

```
	 * oldCommentListListener (ActionListener for "Old Comments" 
JList)
	 * @param e
	 * @param visUnit
	 */
	private void oldCommentListListener(ListSelectionEvent e, Visu-
alizationUnit visUnit) {
		ListSelectionModel curCommentListSelectionModel = (ListSe-
lectionModel)e.getSource();

		if (!curCommentListSelectionModel.isSelectionEmpty()) {
			enableButtons(true, visUnit);
		} else {
			enableButtons(false, visUnit);
		}
	}

	/**
	 * 
	 * @param state
	 * @param visUnit
	 */
	private void enableButtons(boolean state, VisualizationUnit vis-
Unit) {
		vis-
Unit.getMainFrame().infoAndCommentPane.commentPanel.buttonPanel.playB
utton.setEnabled(state);
		vis-
Unit.getMainFrame().infoAndCommentPane.commentPanel.buttonPanel.delet
eButton.setEnabled(state);
	}
}


class CurrentCommentsList extends JList {

	/**
	 * CommentCommentsList implements Serializable
	 * and should have a <code>serialVersionUID</code>.
	 */
	private static final long serialVersionUID = 1L;

	CurrentCommentsList() {

	this.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
	}

}
```

```java
class OldCommentsList extends JList {

	/**
	 * OldCommentsList implements Serializable
	 * and should have a <code>serialVersionUID</code>.
	 */
	private static final long serialVersionUID = 1L;

	OldCommentsList() {

	this.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
	}
}
```

Experience Package using Common Datastructures

Common_datastructure: comments.java

```
package org.belami.dcga.common_datastructures;

import java.io.Serializable;
import java.util.GregorianCalendar;
import javax.sound.sampled.AudioInputStream;
public class Comment implements Serializable {
	private int commentId;
	private int location;
	private String description;
	private boolean isSpeech;
	private GregorianCalendar commentDate;
		//the speech file
	private AudioInputStream ais;
	public Comment(){
	}
	public Comment(int cId){
		commentId=cId;
		location = 0;
		description=null;
	}
	public Comment(int commentId, int location, String description, 
boolean isSpeech, GregorianCalendar commentDate, AudioInputStream 
ais) {
		super();
		this.commentId = commentId;
		this.location = location;
		this.description = description;
		this.isSpeech = isSpeech;
		this.commentDate = commentDate;
		this.ais = ais;
	}

	/**
	 * This contructor is needed by the UI.
	 * The commentId is created by Persistance.
	 * @param location          The id of the room, where the comment 
was added
	 * @param isSpeech          <code>true</code>, if this is a 
speech comment
	 * @param description The text comment. <code>null</code>, if 
this is
	 *                               a speech comment.
	 * @param ais               The audio data. <code>null</code>, if 
this is
```

```
         *                                a text comment.
         */
        public Comment(int location, boolean isSpeech,
                          String description, AudioInputStream ais) {

             this.commentDate = new GregorianCalendar(); // The date of
today
             this.location = location;
             this.isSpeech = isSpeech;
             this.description = description;
             this.ais = ais;

        }

        //   implements the getter methods
        public int getCommentId(){
             return commentId;
        }

        public int getLocation(){
             return location;
        }
        public String getDescription(){
             return description;
        }
//implements the setter methods
        public void setCommentId(int id){
             commentId=id;
        }
        public void setLocation(int newLocation){
             location = newLocation;
        }
        public void setDescription(String newDescription){
             description=newDescription;
        }
        public boolean isSpeech() {
             return isSpeech;
        }
        public void setSpeech(boolean isSpeech) {
             this.isSpeech = isSpeech;
        }
        public GregorianCalendar getCommentDate() {
             return commentDate;
        }
        public void setCommentDate(GregorianCalendar comDate) {
             this.commentDate = comDate;
        }
        public AudioInputStream getAudioInputStream() {
```

```java
		return ais;
	}

	public void setAudioStream(AudioInputStream ais) {
		this.ais = ais;
	}
}
```

1.9 Debriefing Questionnaire

Questions on Complexity of the Tasks

<D1>		**Agree**				**Disagree**		
<D1.1>	The complexity of the experience packages used in both runs (Monday and Tuesday) were comparable	○	○	○	○	○	○	○
<D1.2>	The complexity of the code in the exercises used in both runs (Monday and Tuesday) were comparable	○	○	○	○	○	○	○
<D1.3>	I knew most of the code in the exercises during both runs	○	○	○	○	○	○	○

Questions on Time Needed

<D2>	**I had enough time to**	**Yes**	**No**
<D2.1>	read the information provided by the learning spaces in run 1 (Monday)		
<D2.2>	read the information provided by the learning spaces in run 2 (Tuesday)		
<D2.3>	solve the exercises in run 1 (Monday)		
<D2.4>	solve the exercises in run 2 (Tuesday)		
<D2.5>	familiarize myself with the Wiki and the learning space		

Questions on Learning Spaces

These questions are related to the run where you had access to the Learning Space.

<D3>	**How did you use the Learning Space (LS)? <choose one option>**	
<D3.1>	I first read the LS completely and started to solve the exercises without accessing the LS again	○
<D3.2>	I first read the LS completely and started to solve the exercises by accessing the LS again	○
<D3.3>	I first read the LS partially and started to solve the exercises without accessing the LS again	○
<D3.4>	I first read the LS partially and started to solve the exercises by accessing the LS again	○
<D3.5>	I didn't read the LS and started with the exercise without accessing the LS at all	○
<D3.6>	I didn't read the LS and started with the exercise by accessing the LS later	○

<D4>	**What kind of information did you find useful in the Learning Space with regard to solving the exercise?**	**Agree**				**Disagree**		
<D4.1>	Descriptions of items → labeled as *Description*	○	○	○	○	○	○	○
<D4.2>	Definitions of items → labeled as *Definition*	○	○	○	○	○	○	○
<D4.3>	Example descriptions of items → labeled as *Example*	○	○	○	○	○	○	○
<D4.4>	Counterexample descriptions of items → labeled as *Counterexample*	○	○	○	○	○	○	○
<D4.5>	Process descriptions of items → labeled as *Process*	○	○	○	○	○	○	○

Questions on Stand-Alone Experience Package vs. Learning Spaces

Below you will find a number of opposing adjectives on both sides of each line. You can react to the statements by checking the appropriate point on the line, as in this example:

Useful				Useless		
●	○	○	○	○	○	○

when you think that it was very useful.

<D5>	useful				useless		
I consider the explanations / information provided in a Learning Space in addition to an experience package descrption in general	○	○	○	○	○	○	○
	boring				absorbing		
	○	○	○	○	○	○	○
	easy				difficult		
	○	○	○	○	○	○	○
	clear				confusing		
	○	○	○	○	○	○	○
	complete				incomplete		
	○	○	○	○	○	○	○

<D6>	useful				useless		
I consider the explanations / information provided in a stand-alone experience package description (without Learning Space) in general	○	○	○	○	○	○	○
	boring				absorbing		
	○	○	○	○	○	○	○
	easy				difficult		
	○	○	○	○	○	○	○
	clear				confusing		
	○	○	○	○	○	○	○
	complete				incomplete		
	○	○	○	○	○	○	○

<D7> I would like to make the following comment(s) / improvement suggestion(s) (can be in German)

<D8> I had a problem with … <please explain (can be in German)>:

Questions on Evaluating of the Use and Acceptance of Learning Spaces

<D9> Performance expectancy		**Agree**				**Disagree**		
<D9.1>	I would find the system useful in my job.	○	○	○	○	○	○	○
<D9.2>	Using the Learning Space enables me to accomplish tasks more quickly.	○	○	○	○	○	○	○
<D9.3>	Using the Learning Space increases my productivity.	○	○	○	○	○	○	○
<D9.4>	If I use the Learning Space, I will increase my chances of getting a pay raise.	○	○	○	○	○	○	○

<D10> Effort expectancy		**Agree**				**Disagree**		
<D10.1>	My interaction with the Learning Space would be clear and understandable.	○	○	○	○	○	○	○
<D10.2>	It would be easy for me to become skillful at using the Learning Space.	○	○	○	○	○	○	○
<D10.3>	I would find the Learning Space easy to use.	○	○	○	○	○	○	○

<D11> Attitude toward using technology		**Agree**				**Disagree**		
<D11.1>	Using the Learning Space is a good idea.	○	○	○	○	○	○	○
<D11.2>	The Learning Space makes work more interesting.	○	○	○	○	○	○	○
<D11.3>	Working with the Learning Space is fun.	○	○	○	○	○	○	○
<D11.4>	I like working with the Learning Space.	○	○	○	○	○	○	○

Thanks for filling out the questionnaire!

2 Material of the "Use and Acceptance" Case Study

Evaluierung des Learning Space Tools

Name: ________________________

(Der Name wird nur für Nachfragen bei Unklarheiten der Antworten benötigt. Der Fragebogen wird natürlich anonym ausgewertet.)

Zugangsdaten zum Server:

http://ls.sop-world.org/

Login: lernen

Passwort: lsdevserver

Dann bitte als Benutzer „test" mit dem Passwort „erfahrung" rechts oben im Wiki Fenster einloggen

I. ISONORM Fragebogen zur Software Ergonomie

Füllen Sie bitte den nachfolgenden Fragebogen aus. Die Fragen, die Ihrer Meinung nach nicht für dieses System zutreffen, lassen Sie bitte unbeantwortet.

Der Fragebogen entspricht dem ISONORM 9142/10 Fragebogen.

Die folgenden Fragen beziehen sich ausschließlich auf die Arbeitsaufgabe der Wiederverwendung von Erfahrung und der Verwendung von Lernräumen und nicht auf die anderen Wiki-Funktionalitäten.

Aufgabenangemessenheit

<E1>	**Unterstützt die Software die Erledigung Ihrer Arbeitsaufgaben (Wiederverwendung von Erfahrung), ohne Sie als Benutzer unnötig zu belasten?**								
	Die Software...	---	--	-	-/+	+	++	+++	
<E1.1>	ist kompliziert zu bedienen.	○	○	○	○	○	○	○	ist unkompliziert zu bedienen.
<E1.2>	bietet nicht alle Funktionen, um die anfallenden Aufgaben effizient zu bewältigen.	○	○	○	○	○	○	○	bietet alle Funktionen, die anfallenden Aufgaben effizient zu bewältigen.

<E1.4>	erfordert überflüssige Eingaben.	○	○	○	○	○	○	○	erfordert keine überflüssigen Eingaben.
<E1.5>	ist schlecht auf die Anforderungen der Arbeit zugeschnitten.	○	○	○	○	○	○	○	ist gut auf die Anforderungen der Arbeit zugeschnitten.

Selbstbeschreibungsfähigkeit

<E2>	**Gibt Ihnen die Software genügend Erläuterungen und ist sie in ausreichendem Maße verständlich?**								
	Die Software...	---	--	-	-/+	+	++	+++	
<E2.1>	bietet einen schlechten Überblick über ihr Funktionsangebot.	○	○	○	○	○	○	○	bietet einen guten Überblick über ihr Funktionsangebot.
<E2.2>	verwendet schlecht verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.	○	○	○	○	○	○	○	verwendet gut verständliche Begriffe, Bezeichnungen, Abkürzungen oder Symbole in Masken und Menüs.
<E2.3>	liefert in unzureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.	○	○	○	○	○	○	○	liefert in zureichendem Maße Informationen darüber, welche Eingaben zulässig oder nötig sind.

Steuerbarkeit

<E3>	**Können Sie als Benutzer die Art und Weise, wie Sie mit der Software arbeiten, beeinflussen?**								
	Die Software...	---	--	-	-/+	+	++	+++	
<E3.1>	bietet keine Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.	○	○	○	○	○	○	○	bietet die Möglichkeit, die Arbeit an jedem Punkt zu unterbrechen und dort später ohne Verluste wieder weiterzumachen.
<E3.2>	erzwingt eine unnötig starre Einhaltung von Bearbeitungsschritten.	○	○	○	○	○	○	○	erzwingt keine unnötig starre Einhaltung von Bearbeitungsschritten.
<E3.3>	ermöglicht keinen leichten Wechsel zwischen einzelnen Menüs oder Masken.	○	○	○	○	○	○	○	ermöglicht einen leichten Wechsel zwischen einzelnen Menüs oder Masken.
<E3.4>	ist so gestaltet, dass der Benutzer nicht beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.	○	○	○	○	○	○	○	ist so gestaltet, dass der Benutzer beeinflussen kann, wie und welche Informationen am Bildschirm dargeboten werden.
<E3.5>	erzwingt unnötige Unterbrechungen der Arbeit.	○	○	○	○	○	○	○	erzwingt keine unnötigen Unterbrechungen der Arbeit.

Erwartungskonformität

<E4>	**Kommt die Software durch eine einheitliche und verständliche Gestaltung Ihren Erwartungen und Gewohnheiten entgegen?**								
	Die Software...	---	--	-	-/+	+	++	+++	
<E4.1>	erschwert die Orientierung, durch eine uneinheitliche Gestaltung.	○	○	○	○	○	○	○	erleichtert die Orientierung, durch eine einheitliche Gestaltung.
<E4.2>	lässt einen im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.	○	○	○	○	○	○	○	lässt einen nicht im Unklaren darüber, ob eine Eingabe erfolgreich war oder nicht.
<E4.3>	informiert in unzureichendem Maße über das, was sie gerade macht.	○	○	○	○	○	○	○	informiert in ausreichendem Maße über das, was sie gerade macht.
<E4.4>	reagiert mit schwer vorhersehbaren Bearbeitungszeiten.	○	○	○	○	○	○	○	reagiert mit gut vorhersehbaren Bearbeitungszeiten.
<E4.5>	lässt sich nicht durchgehend nach einem einheitlichen Prinzip bedienen.	○	○	○	○	○	○	○	lässt sich durchgehend nach einem einheitlichen Prinzip bedienen.

Individualisierbarkeit

<E6>	**Können Sie als Benutzer die Software ohne großen Aufwand an Ihre individuellen Bedürfnisse und Anforderungen anpassen?**								
	Die Software...	---	--	-	-/+	+	++	+++	
<E6.2>	lässt sich von dem Benutzer schlecht an seine persönliche, individuelle Art der Arbeitserledigung anpassen.	○	○	○	○	○	○	○	lässt sich von dem Benutzer gut an seine persönliche, individuelle Art der Arbeitserledigung anpassen.
<E6.3>	eignet sich für Anfänger und Experten nicht gleichermaßen, weil der Benutzer sie nur schwer an seinen Kenntnisstand anpassen kann.	○	○	○	○	○	○	○	eignet sich für Anfänger und Experten gleichermaßen, weil der Benutzer sie leicht an seinen Kenntnisstand anpassen kann.
<E6.4>	lässt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer schlecht für unterschiedliche Aufgaben passend einrichten.	○	○	○	○	○	○	○	lässt sich - im Rahmen ihres Leistungsumfangs - von dem Benutzer gut für unterschiedliche Aufgaben passend einrichten.
<E6.5>	ist so gestaltet, dass der Benutzer die Bildschirmdarstellung schlecht an seine individuellen Bedürfnisse anpassen kann.	○	○	○	○	○	○	○	ist so gestaltet, dass der Benutzer die Bildschirmdarstellung gut an seine individuellen Bedürfnisse anpassen kann.

Lernförderlichkeit

<E7>	Ist die Software so gestaltet, dass Sie sich ohne großen Aufwand in sie einarbeiten konnten und bietet sie auch dann Unterstützung, wenn Sie neue Funktionen lernen möchten?								
	Die Software...	---	--	-	-/+	+	++	+++	
<E7.1>	erfordert viel Zeit zum Erlernen.	○	○	○	○	○	○	○	erfordert wenig Zeit zum Erlernen.
<E7.2>	ermutigt nicht dazu, auch neue Funktionen auszuprobieren.	○	○	○	○	○	○	○	ermutigt dazu, auch neue Funktionen auszuprobieren.
<E7.3>	erfordert, dass man sich viele Details merken muss.	○	○	○	○	○	○	○	erfordert nicht, dass man sich viele Details merken muss.
<E7.4>	ist so gestaltet, dass sich einmal Gelerntes schlecht einprägt.	○	○	○	○	○	○	○	ist so gestaltet, dass sich einmal Gelerntes gut einprägt.
<E7.5>	ist schlecht ohne fremde Hilfe oder Handbuch erlernbar.	○	○	○	○	○	○	○	ist gut ohne fremde Hilfe oder Handbuch erlernbar.

II. UTAUT Fragebogen zur Nutzung und Akzeptanz (in Englisch)

The following questions are based on the UTAUT (Unified Theory of Acceptance and Use of Technology).

<U1> Performance expectancy	**Agree**				**Disagree**		
<U1.1> I would find the system useful in my job.	○	○	○	○	○	○	○
<U1.2> Using the system enables me to accomplish tasks more quickly.	○	○	○	○	○	○	○
<U1.3> Using the system increases my productivity.	○	○	○	○	○	○	○
<U1.4> If I use the system, I will increase my chances of getting a pay raise.	○	○	○	○	○	○	○

<U2> Effort expectancy	**Agree**				**Disagree**		
<U2.1> My interaction with the system would be clear and understandable.	○	○	○	○	○	○	○
<U2.2> It would be easy for me to become skillful at using the system.	○	○	○	○	○	○	○
<U2.3> I would find the system easy to use.	○	○	○	○	○	○	○
<U2.4> Learning to operate the system is easy for me.	○	○	○	○	○	○	○

<U3> Attitude toward using technology	**Agree**				**Disagree**		
<U3.1> Using the system is a good idea.	○	○	○	○	○	○	○
<U3.2> The system makes work more interesting.	○	○	○	○	○	○	○
<U3.3> Working with the system is fun.	○	○	○	○	○	○	○
<U3.4> I like working with the system.	○	○	○	○	○	○	○

<U4> Facilitating conditions	**Agree**				**Disagree**		
<U4.1> I have the resources necessary to use the system.	○	○	○	○	○	○	○
<U4.2> I have the knowledge necessary to use the system.	○	○	○	○	○	○	○
<U4.3> The system is not compatible with other systems I use.	○	○	○	○	○	○	○
<U4.4> A specific person (or group) is available for assistance with system difficulties.	○	○	○	○	○	○	○

<U5> Self-efficacy	**Agree**				**Disagree**		
<U5.1> I could complete a job or task using the system…	○	○	○	○	○	○	○
<U5.2> If there was no one around to tell me what to do as I go.	○	○	○	○	○	○	○
<U5.3> If I could call someone for help if I got stuck.	○	○	○	○	○	○	○
<U5.4> If I had a lot of time to complete the job for which the software was provided.	○	○	○	○	○	○	○
<U5.5> If I had just the built-in help facility for assistance.	○	○	○	○	○	○	○

III. Weitere Anmerkungen, Kritik, Verbesserungsvorschläge …

… zur Farbgebung, Strukturierung der Informationen, Navigation

… zur Anreicherung von Erfahrungen mit Lernelementen (Integration von Wissensmanagement und E-Learning)

… zu Lernelementen

…

Document Information

Title:	Dissertation Eric Ras - Annex 2: Materials of the Empirical Studies
Date:	January 20, 2009
Report:	IESE-002.09/E
Status:	Final
Distribution:	Public