Fraunhofer Institut
Experimentelles
Software Engineering

Comparing Design Alternatives from Field-Tested Systems to Support Product Line Architecture Design

Authors:
Jens Knodel
Thomas Forster
Jean-François Girard

Submitted for publication at
WCRE 2004

This work is partially funded
by the European Commission
EUREKA 2023/ITEA-ip00009
'FAct based Maturity through Institutionalization Lessons-learned an
Involved Exploitation of System-family engineering' (FAMILIES).

IESE-Report No. 078.04/E
Version 1.0
June 30, 2004

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)
Sauerwiesen 6
67661 Kaiserslautern

Abstract

This paper introduces an approach to mine field-tested design solutions when defining the architecture of a new product line. The design comparison approach (DCA) compares design solution alternatives implemented in existing systems and analyze the impact of the solutions on these systems. This explicit comparison and analysis offer the architect of the to-be-built product line to develop an architecture of higher quality by incorporating field-tested, proven concepts and strategies. We show the applicability and usefulness of the approach in two case studies concerned with the architectural design of Eclipse plug-ins.

Keywords: architecture evaluation and reconstruction, design comparison, reverse engineering, product lines.

Table of Contents

1 Introduction

Developing software systems is a complex and difficult task. Reuse of existing solutions and artifacts is a promising solution for the challenges for today's software-developing organizations and their needs for reducing cost, effort and time-to-market, the increasing complexity and size of the software systems, and increasing demands for high-quality software and individually customized products for each customer. This is especially true for software product lines where proactive reuse is one of the main concepts.

The architectural design phase is the central phase for decisions about how the software system will achieve its requirements [17]. The architecture of a software system is defined in [14] as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. Usually architects tend to reuse successful solutions based on their background and their experiences. When several software systems are available, the architecture design phase would benefit from the experience gained in field-tested solutions. The goal of the design comparison approach (DCA) is to explore existing alternatives and to learn about different solutions applied to similar problems, to identify advantages and drawbacks of the solutions, and to rate them with respect to their applicability in the context of the development of new systems. During a design comparison, we apply reverse engineering techniques to mine the existing systems for information needed to perform the actual comparison. DCA does not aim at comparing the complete system architectures and designs. Instead it focuses on finding the solutions to specific design problem addressed in different systems and assessing them.

Design comparisons are initiated by requests from the architects and they produce views that highlight certain aspects of the analyzed systems along with a ranking of the solutions. A request thereby addresses a specific aspect of interest in architecture development. DCA is expert-driven, iterative and highly context-sensitive to the current design problems. The architects cooperate closely together with reverse engineers that know about the capabilities and the strengths of different reverse engineering techniques and methods. DCA operates in a request-driven mode to solve concrete design problems on demand.

The remainder of the paper is structured as follows: Section 2 discusses typical usage scenarios. A detailed description of our approach is provided in section 3. Then section 4 presents two case studies centered on design problems we faced while developing Eclipse plug-ins. Section 5 summarizes related work, while section 6 draws conclusions and gives an outlook on future work.

2 Typical Usages

DCA supports iterative scenario-based architecture design of a new system (especially the design of a product line architecture) like PuLSE-DSSA [4, 5] or ADD [2]. It aims at answering high-level requests about existing solutions relevant for design of the new system. We consider the following four types of high-level requests about architectural alternatives:

- **Patterns**: Patterns address recurring design problems that arise in specific design situations, and present a solution to them [7]. Request about patterns may be what patterns are used and what is their impact, what is the solution to a requirement or a specific design problem, or why was a pattern applied.

- **Features**: Features represent functionality visible to the user of a software system. Requests concentrating on features are for instance what common or variable features are or how a feature is realized.

- **Quality attributes**: A quality attribute is a non-functional requirement presenting a property of the software system. Quality attribute requests deal with the following question: does a system achieve a quality attribute (e.g. performance, maintainability) and if so, what are the means applied to achieve it?

- **Strategies**: Strategies in architectural design represent design rationales and manage the interaction of patterns, features, and quality attributes and prioritize them. Strategies explain the fundamental architectural design decision within a system and deal with requests like: what are the consequences, difficulties and drawbacks of an applied strategy, or which patterns are involved to realize a strategy.

The main case where the design comparison approach can improve the quality of the architecture is migration towards product line engineering. Clements [9] defines product line engineering as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. The migration from single system development towards product line engineering is a challenging task for software development organizations. The commonalities of the existing, individual systems have to be merged into the product line infrastructure, and the product line architecture has to support the derivation of the future members. When designing this architecture, we recommend reusing the field-tested, proven solution implemented in the existing systems. The question that arises now is which of the different alternative solutions fit best to the requirement and the qualities attributes of the product line architecture to be built. Applying the architecture comparison approach contributes in the migration case by giving guidance in how to select the most suitable of the given alternatives.

3 The Design Comparison Approach

When the architect of a product line faces a problematic design decision for the product line architecture, a possible source to solve the problems are solutions given in existing systems. To learn about benefits and consequences of such solutions, the potential reuse candidates (either concepts or implementations) have to be compared and rated. In order to understand the solutions and their properties the product line architect demands information about them, a so-called high-level request. Responses to requests enable learning about successful solutions and their consequences.

The design comparison approach (DCA) is initiated by such requests. Together with evaluation criteria (elicited from stakeholders) and optionally a product map these inputs are processed into a response, which summarizes the various characteristics of the solutions embodied in selected systems and rate them with respect to the given criteria. To analyze the best solutions, the product line architect consults the rating produced in the final of 5 steps of the approach (see Figure 1 for an illustration of the DCA approach). These steps exchange feedback either within or across iterations. The result of DCA, a response, can result in follow-up iterations or new requests. To accomplish its goals, DCA involves three major roles:

- **Product line architect**: The architect designs the architecture of new systems. He decomposes it into components, decides how these components interact, determine commonalities, provides mechanisms to handle variabilities and their resolution, and formulates the system’s design principles.

- **Expert**: A system expert knows his system very well and understands the strategies, patterns and concepts applied in the existing system. The original architect of a system constitutes often a good system expert.

- **Reverse engineer**: The reverse engineer applies techniques to extract the information needed to answer a request from existing systems.

The role of the architect and of the expert can be performed by the same persons in cases where the architect of the new system is also an expert for one of the existing systems being compared.

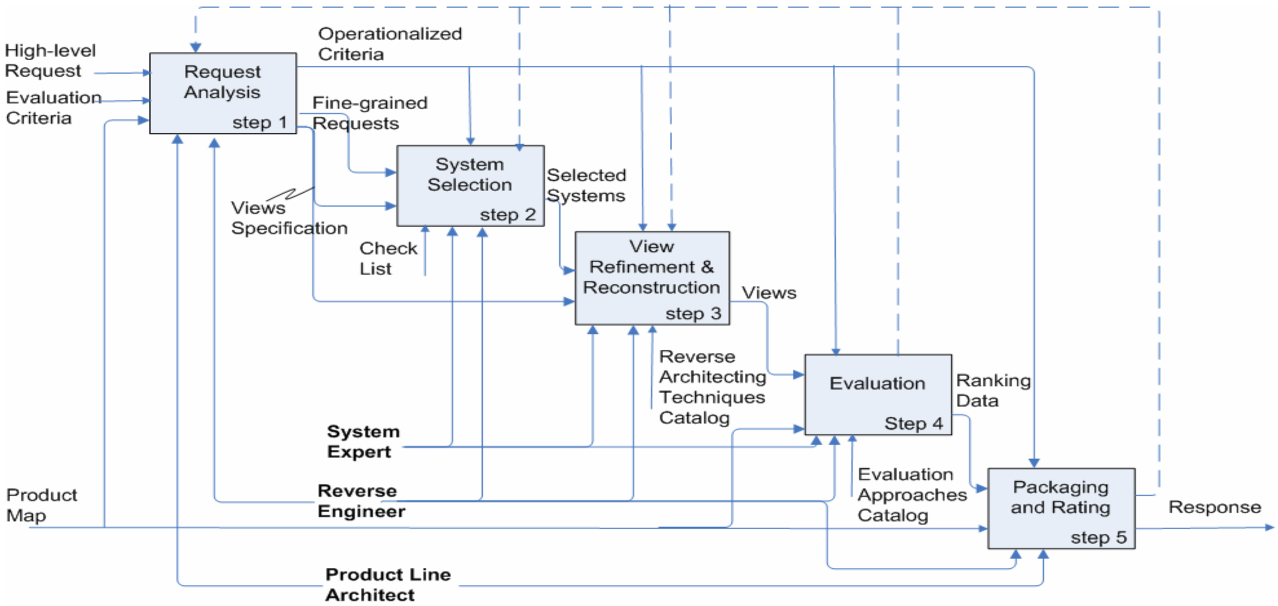

Figure 1 Design Comparison Approach Overview

3.1 Request Analysis

This step refines the high-level request, operationalized the evaluation criteria, and produces specifications for the views that should be available to support the analysis and comparison of solutions as well as strategies implemented by the selected systems.

In most cases, the architect initially provides high-level requests that cannot be directly answered because they are too abstract, too broad, or too complex. Therefore the approach starts with a request analysis that breaks request down into operational pieces, which can be processed by the subsequent steps. The final step “Packaging and Rating” delivers responses to overall, high-level requests. To evaluate the success of each system in fulfilling these detailed requests, the architect and the reverse engineer refine and operationalize the evaluation criteria as needed (e.g., performance or maintainability). The concrete parameters of a criterion are often strongly linked to the application context (e.g., the operation should have a response time of at most 5 seconds). For a product line, this context can be expressed by a product map relating the envisioned application instances of the product line and their features [21].

Given the detailed requests and the operationalized criteria, the reverse engineer produces a specification of the views that would represent the aspects covered by the request and support their evaluation of criteria. These specifications are the main input of view refinement and reconstruction step. The reverse engineer and the architect often trade-off between how well these views support the evaluation and the effort needed to obtain them.

The architect prioritizes the fine-grained requests and the criteria to schedule in which order they will be considered and which views should be produced first. This is done because the evaluation of some requests may terminate the further processing of the high-level requests; while others can result in new concrete, fine-grained requests.

3.1.1 Refinement Example

Assuming that the architect of a product line considers using the model-view-controller (MVC) pattern within his design, but he is uncertain about its impact and the level of flexibility needed. He could make a high-level request about the experiences with the MVC pattern in the existing systems. The request cannot be answered directly; it has to be refined. The derived fine-grained requests are:

1. In which systems is the MVC pattern applied?
2. What are consequences of using MVC with respect to certain criteria (e.g., performance, maintainability)?
3. How many views, controllers, and model layers are present in the implementation?
4. Are the views hierarchical?
5. What are the consequences in terms of performance, complexity and maintainability of the answer to request 3 & 4?
6. Do the pattern instances among the different system aim at solving the same problems? Are these instances comparable?

Some requests aim at obtaining context information that support further request requests (e.g., 1, 3, 4, and 5). For them, it is not necessary to develop concrete criteria. For other requests the architect should provide concrete evaluation criteria. To evaluate maintainability, for example (e.g., request 2 & 5), he describes modification scenarios. In order to identify acceptable performance, the architect indicates the maximal response time for the critical operations.

Both for purely informative and criteria-bound requests, the reverse engineer specifies the views that are available for the evaluation or that will be reconstructed for it. For evaluating the MVC’s maintainability, a description of the involved classes' interface is enough. For the performance evaluation, interaction diagrams or message sequence charts might be needed.

3.2 System Selection

In this step, the reverse engineers and the experts select the systems they will compare to respond to the given request. The goal is selecting few valuable systems that will bring most insights during a comparison for a limited effort.

When code reuse is an important factor, then only systems that are compatible with the existing implementation will be considered. When the design solutions are more relevant than code reuse, then other system can also be considered.

A checklist helps the reverse engineers and the expert to select the systems. The checklist reflects the main factors, which influence the systems selection strategy. These factors are available architectural views, presence or absence of a dominant system, the design stage of the new system, the market success, and the properties of the system. The type of the requests and the effort available for comparing systems are cross-cutting selection factor.

The effort available for comparing systems and the effort required to reconstruct the views and analyze them constraint the systems can be evaluated.

The actual system selection is a human-based activity that depends strongly on the context. There are no fix rules how to combine the different factors. However, the reverse engineer and the expert have a better chance to select the right system if they use the proposed checklist.

3.3 View Refinement and Reconstruction

For each of the systems selected, this step produces views that fulfill the specification produce during request analysis. It either reconstructs new views or finds and adapts existing views provided by the documentation of the system. To support this task, DCA takes advantage of a reverse architecting techniques catalogue. The views capture the aspects of the system needed to evaluate the operationalized criteria properly.

To act as an effective basis for comparison, two software systems should be described at the same level of abstraction. Often the descriptions available for the selected systems are at very different level of abstraction. In this case one description is either refined or abstracted. When the effort for refining them is too high and the information contained in the view is not essential for answering the requests, the detailed views are abstracted. When answering a request requires information not present in the available system description, we apply reverse engineering techniques on the different systems to reconstruct the views containing this information or completing existing views. Such goal-driven techniques are performed to exploit just the required information from the existing software systems and to focus only on the aspects relevant to the requests.

A reverse engineering infrastructure provides a reverse architecting techniques catalog. The techniques from this catalog are selected based on the view specification and the criteria coming from the request analysis. The selection of the appropriate technique is done by the reverse engineer, while the execution of a specific technique in most cases requires involvement of system experts.

For instance, if a request aims at comparing aspects of different implementations of the same feature, then techniques like feature location [12] or feature reverse engineering [13] have to be applied. Another example is two components providing the same functionality: to evaluate them their interfaces have to be documented [16] and the dependencies on other components have to be revealed [5]. Techniques like the reflexion model [18] and can be used to reflect the mental model of expert with the actual implementation. For the MVC example, the reconstructed views show the classes composing the MVC patterns and their relationships. In addition, object diagram can captured typical scenarios at run-time. Reverse techniques applied in this case are for instance pattern matching [20], pattern completion [5], or dynamic techniques capturing the object instances at different points in time [19].

The reverse engineers drives the view refinement and reconstruction step because he knows of the power and capabilities of the available techniques, he can asses the expected value of their output, and he is able to estimate the required effort to apply a technique.

In summary, the view refinement and reconstruction step produces the views on the selected systems required to evaluate them. Thereby will different request results in different views on the systems, and the reverse engineer may decide not to reconstruct a view for a selected system because the expected value does not justify the effort.

3.4 Evaluation

In the fourth step, the expert evaluates the solutions used in the selected system by applying different techniques on the set of views capturing each system's solution. The expert evaluates the solutions with respect to one criterion at a time so that he can rank the results of the evaluation and express them into joint abstractions.

3.4.1 Evaluation Techniques

The expert (supported by the reverse engineer if appropriate) applies one or more of the following techniques to evaluate the solutions with respect to the criteria developed during the request analysis:

- **Simulation**: Within a simulation, the reverse engineer observes or measures the system characteristics relevant to the request. The presence of deadlocks, for example, can be detected during simulation.

- **Instrumentation**: In this technique, part of the code related to a request is instrumented and run to gain insight into the working of each system as it is currently implemented.

- **Prototyping**: Prototyping is a technique that executes a simplified version of a system to observe and measure some property of the final system. In our approach, modifying and adjusting the existing implementation to the given request is one efficient way to produce a prototype. In this case, only the aspects related to the request and supporting it are kept in the system. Everything else is removed.

- **Context analysis**: This analysis reveals the dependencies of a key element of the implementation of a solution targeted by the request. The goal of this activity is to get a better understanding of how the architectural alternatives are embodied in the system and to identify possible side-effect that may occur when reusing it in other contexts.

- **Scenario analysis**: This technique gathers and applies scenarios that exercise different aspect of a request (e.g., different quality attributes) on each of the selected software systems. The expert applies the scenarios and identifies risks associated with certain scenarios [10].

- **Model sensitivity analysis**: For this technique the expert or the reverse engineer build a numeric model of the solution and analyze the relevant criteria while varying different parameters.

The success of a technique depends on the type of request, the information available and the system studied. Each technique, however, produces information about only one individual system at a time. Therefore the next activity within the composition step is the creation of a joint abstraction.

3.4.2 Creation of Joint Abstractions

The creation of a joint abstraction combine the information found in the individual systems. Here the results of the existing systems are aggregated and brought in the context of the product line space. We propose the following means to represent joint abstractions:

- **Comparison matrix**: A comparison matrix lists the different, individual systems selected as columns and the criteria related to the requests as rows. The responses for each system (or the subjects of the analysis) are then filled into the proper cells. The reverse engineer and the expert can now reason about advantages and disadvantages of the different alternatives and refine results by evaluation of other criteria in further iterations.

- **Product line views**: The product line views capture information about all selected systems and focus on the complete product line. These views support identifying variabilities and commonalities among systems.

After building joint abstractions, the expert produces a partial solution ranking that indicates how well each solution fulfilled the given criterion.

3.5 Packaging and Rating

Whereas the previous step evaluated single, operationalized criteria, this step compose the information gained about the selected system in order to derive a final response to the product line architect's high-level request. The rating can be regarded the opposite step to the break-down of the request done step one of DCA. The low-level results are combined to give one response to the high-level request. The operationalized criteria and the product map thereby guide the rating. The reverse engineers support the product line architects in interpretation of the ranking data. The final decision is made by the architects having the goals and the requirements of the to-be-designed product line architecture in mind.

The recording of the results includes the responses together with the initiating requests, the selected systems and reasons for their selection, and if available, integration and usage information about the gained responses. The recording supports potential reuse of results in further iterations and documents decisions and rationales. The response recording step should be supported by a knowledge repository that facilitates the access, the manipulation, the storage and the management of the responses gained so far.

In summary, the rating done by the product line architects can initiate additional requests to obtain further information, or they decide whether to accept the best rated solution or to reject the potential reuse candidates provided by the selected systems.

3.6 Summary

DCA is an iterative, expert-driven approach to learn about alternatives, solutions and strategies embodied in existing systems. DCA uses reverse engineering techniques to obtain information from existing systems. During all the steps, feedback, learning effects, and insights gained are used to improve the results of later or ongoing iterations. In short, DCA seeks at mining solutions (concepts and/or implementations) from existing system in order to support the product line architects in their work.

4 Eclipse Plug-In Case Studies

To support its PuLSE method research, the software product line (SPL) department at IESE develops product line tools based on Eclipse. Eclipse is a basic tool integration platform where additional functionality is provided by plugins.

The two case studies presented here fall in the context of our effort for building the tool infrastructure from which individual product line tools are derived. The first case study takes place in the context of defining a graph representation supporting efficient reverse engineering techniques within our Eclipse product line infrastructure. The goal is to identify the best graph representation for our product line from multiple reverse engineering tools previously developed offering solutions: different internal representations, implemented in different languages, designed for different purposes. The second case study involves addressing a request to understand "internal model management" in Eclipse IDE plug-ins. Our IDE tools should share an infrastructure that is compatible and easy to integrate together with other plug-ins not developed in-house. The goal in this case was to reuse both, concepts and code.

4.1 Reverse Engineering Graph Representation

IESE SPL decided to integrate multiple reverse engineering and metric tools into its infrastructure. These tools often manipulate large hierarchical graphs and require efficient representation. Before defining a new graph representation with its interface, the architect formulates a request about determining the best field-tested solutions. He also defines that these solutions should be considered from the criteria performance, representation size, as well the maintainability and extendibility. He also provides a list of existing and envisioned tools and the features they need (i.e., a simple form of a product map).

Starting DCA with **request analysis**, the architect refined the request into the following questions:

1. Which representation offers the most compact memory representation?

2. Which representation offers the fastest access and manipulation?

3. How scalable is each representation?

4. Which API supports to application that is easy to use and maintain?

5. Which representation is easier to maintain?

6. Does the implementation offer a regression test?

7. Does it have a cache or could a cache be added easily to increase scalability?

He binds the refined requests to the given criteria and makes them operational by adding concrete values or evaluation mechanisms. For the representation size (request 1), for example, he asserts that a typical small graph containing 10 000 nodes, 40 000 edges, and 2 500 annotations should occupy less than 5 Meg. For the performance (request 2), he identifies specific operations that are typically performed on the graph and for some of them he specifies an upper bound on their execution time. He identifies operations like accessing neighbors of a node, collapsing node into components and computing the edges among lifted components, or identifying if pairs of nodes belong to the same components. He also specifies, for example, that traversing the graph above through a depth-first search should take less than 1 second on a reference machine. When he has operationalized the criteria, the architect prioritizes them.

Next, the architect and the reverse engineer specify the views needed to answer the fine-grained requests based on the operationalized criteria. In this case, they decided to include the following views:

- a conceptual view semantic net representing the relations among key concepts in the representations

- a data model of the representations annotated with storage requirements

- an API description

In the **system selection**, the reverse engineer selected two prior graph implementations that serve the same purpose in other contexts and one internal representation used in Eclipse. For each implementation, an expert was available and was briefly consulted to check if he believed that the representation was adequate for the envisioned purpose. The first prior implementation (*R-Rfv*) was written in Refine and had been used as a basis for many research tools. It was developed with a focus on compact representation and flexibility. The second representation (*P-Rsf*) had been developed in Perl, to be portable. The focus was on simplicity and efficiency. The third representation (JavaE) is part of the internal model management of JDT rooted at the JavaElement. It does not constitute a generic graph representation, but rather one specific to the entities represented in the java development environment. Further representations in Eclipse were considered, but not selected for the evaluation because no expert was available for these representations and more effort would go in the reconstruction of the view than what was available.

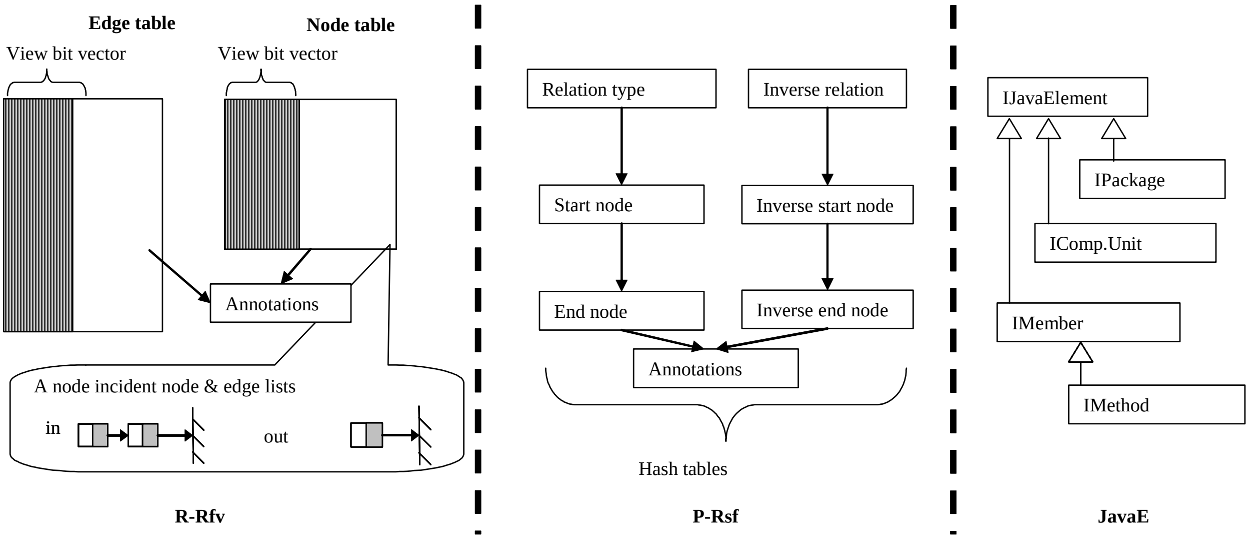

Figure 2 Simplified Data Model

The reverse engineer performed **view reconstruction** by interviewing expert and using information from the source code browsers. The conceptual views were constructed by interviewing the system experts. The reverse engineer reconstructed the data model using code browsers and the API by consulting the source files. Figure 2 depicts a simplified data model for the graph representation of the three solutions considered.

The **evaluation** relied on scenario analysis and model sensitivity analysis. For evaluating the API's maintainability, the system experts applied change scenarios concerning tool development (e.g., reflexion model, dependency analysis, coupling metrics calculation) using the API. R-Rfv and P-Rsf were roughly equal for the understandability. JavaE's API is more complex due to the fact that the model is build only when it is needed.

To evaluate the space efficiency of each representation, the reverse engineer derived formulae computing the representation size in terms of the number of nodes, edges, and annotations as well as the average data size. This model sensitivity analysis allowed him for different scenarios to analyze the scalability of each representation and to estimate the graph size that would fit in memory.

The reverse engineer considered that building a prototype sufficient to evaluate the performance of each representation would require more time than they had. Instead the expert of each system simulated the data accesses that would be required to perform crucial scenarios as first approximation. A minimal prototype was built to analyze if the performance of the best solution would fulfill the performance criteria.

criteria		R-Rfv	P-Rsf	JavaE
space efficiency	reference graph	1.3 Meg < 5 Meg	4.9 Meg < 5 Meg	> 5 Meg
	scalability	smallest	3.7 * Refine	keep nodes only while needed in memory
maintainability	API	~ Perl (more functions)	~ Refine	only basic functions
understandability	explicit concepts	node, edge, views	relation as tuples	fix sed of nodes
	API	good	good	indirect & complex
cache		none	none	already present (LRU)

Figure 3 Comparison Matrix

As joint abstraction, the reverse engineer produced a comparison matrix summarizing for each criterion the results of the analysis. Figure 3 presents an excerpt from this comparison matrix. The reverse engineer ranks the solutions for each criterion. In terms of space efficiency and pure size scalability, for example, R-Rfv is considerably smaller. The API of R-Rfv and P-Rsf are roughly as maintainable and understandable.

The results of the analysis showed that R-Rfv and P-Rsf are easier to understand and to maintain than the JavaE. When the complete graph should be keep in less than 5 Megabytes in memory, then the R-Rfv representation is better and can accommodate graphs with up to 35 000 nodes and 140 000 edges. However, for larger graph, an on-demand approach is needed and JavaE already provides a good solution, which is integrated in Eclipse. The recommendation is to extend the R-Rfv with a cache to face the demand of larger graphs.

Following the DCA process, the architect obtains a more objective answer to his request that systematically exploit the experience from past systems. He used this response as basis for his design decision and easily articulated the rationales for his decision.

4.2 The Plug-In Internal Model Management

To support our own infrastructure of software product line tools, we decided to develop a series of Eclipse plug-ins. In particular, we had the goal to develop a component browser plug-in that is able to display information about KobrA-components [1] within the Eclipse platform (models capturing the specification and the realization of a component available in XML format had to be visualized by the component browser plug-in). A second plug-in, the frame processor, dealt with the realization of product line implementation technology, called frame processing [3]. Thereby variable parts are captured in separate implementation units (i.e., frames) and the frame processor takes over the resolution of the frames when instantiating a member of the product line. Furthermore the goals were to visualize the frame hierarchy and to have an editor for the frames.

In the beginning the architect recognized that both to-be-developed plug-in required an internal model management. We investigated other plug-ins of the open source communities that had the same design problem. We found three programming language development tools, the JDT [15] (for Java: ~3 000 classes), the CDT [8] (for C/C++: 2 200 classes) and the CobolDT [11] (for COBOL: 1 200 classes) having a solution to our design problem. Because of immaturity of the CobolDT, it was decided in the system selection step to leave it out.

So the high-level request the architect had was how is the internal model management realized in the JDT and CDT and can he reuse the solution for his own design purposes. The architect and the reverse engineer then broke down this request and had several iterations of DCA to derive an answer. The next paragraphs will describe some of the main points in applying DCA.

The architect and the reverse engineer started to refine the request and were first interested in feature and classes that implemented the model management. The reverse engineer specified to reconstruct behavioral views for opening, saving and creating a project with sequence diagrams. Dynamic technique provided the method invocation for each of the features, while static technique enabled to collapse the methods into classes, packages, and in the end into components. The request was further refined and the information was used to build a conceptual view with the help of the system expert (see Figure 4). The conceptual view for both, the JDT and the CDT, contains only one variation point (stereotype <<variant>>), the compilation unit for Java source code files, and the translation unit for C/C++ source code files. The other conceptual components were common for both systems. Through inheritance the Openable class adds the aspect of being saveable to model elements that have a physical representation in the file system and can therefore be manipulated and saved.

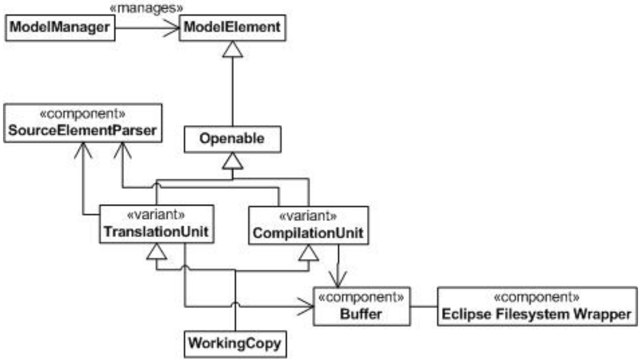

Figure 4 Conceptual view of the save feature

The fact that both alternatives realized the save feature in almost the same manner gave us confidence for the reuse of that solution for our purposes. The architect therefore derived requests to investigate the classes responsible for model management in more detail. The reverse engineer refined the views on the object model, a key concept of both JDT and CDT. The Eclipse platform wraps the physical file system with its abstraction for files, folders, projects and the workspace, which can be seen as container for those entities. Build upon the Eclipse data model the IDEs provide a further abstraction to access these entities of a programming language through the API. This means that the IDE models add programming language specific semantic to the Eclipse model elements. For instance, a folder corresponds to a package in Java. From an architectural point of view these object models are part of a layered architecture where the physical file system is the basis, on which the Eclipse platform data model is built, on top of are the data models of JDT and CDT.

The architect then wanted to know what the reasons were for introducing an elaborated object model as it would have been also possible to work with the abstractions provided from the Eclipse platform. To get a basis for our comparison we refined our interest in the request analysis and asked how these models are managed.

Starting with a context analysis to retrieve the structure of the IDE object models, the reverse engineer learned where the particular object models are physically located in the plug-in package hierarchies. From existing architectural descriptions [22] he found out that inheritance was a key concept in these models. Based on this information the reverse engineer traced the root classes of both object models and queried the fact base to discover the inheritance hierarchy. Figure 5 shows an excerpt of the JDTs class and interface model hierarchy, which is similar to that of the CDT.

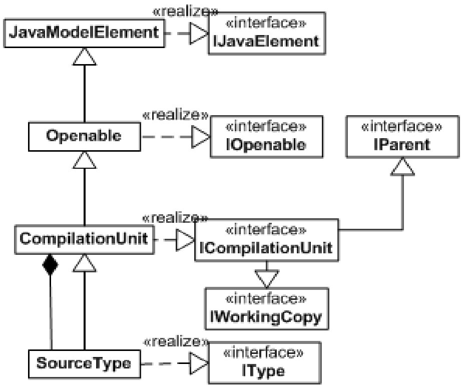

Figure 5 JDT Data Model Excerpt

From the static structural information of the object model it was not obvious, how the model management including typical operations like adding, removing, moving and finding elements was realized. Therefore the reverse engineer traced these functionalities by several dynamic analyses using instrumentation to have a basis for comparing the model management in both IDEs.

In the scenario analysis the system expert evaluated the reasons why the object models are part of the design and implementation strategies in both IDEs. Eclipse provides a model view controller framework, where the hooks enforce the usage of an object model to ease the integration of a plug-in with the Eclipse platform. This goes together with the fact both IDEs make use of the JFace GUI framework (i.e., a set of components like tree viewers and table viewers) that expects a model based input. To visualize the data the plug-in deals with, it would be enough to provide simple proxies for the real world entities like packages or class files in the JDT and feed them to the GUI framework. As an object model is a good concept to deal with the complexities of real world entities both models are far more elaborated (i.e., they provide handles to the programming language specific concepts and constructs, so the entities can be accessed without having to deal with the underlying physical resources or intermediate abstractions and dependencies between those entities can be computed). The model elements add additional semantics to entities like files and folders and provide services which are specific to these entities. The use of an object model is a good choice if meta-models for the problem domain exist where the meta-model elements can be directly mapped onto classes within object models and computation of element dependencies can be realized by querying the object model.

Commonalities between both implementations are a least-recently-used (LRU) cache (removes unused model elements if the cache size is exceeded) on demand loading of model elements (model elements are not loaded, until a user opens an element containing them).

Differences are concerning key features like adding and removing model elements, which are implemented slightly different. The data for the comparison are based on several operations that add and remove elements to the object model (e.g., a new project was created in the workspace). In both IDEs, we examined the process flow to analyze the execution complexity. Here the reverse engineer considered the number of involved classes, method calls and the lines of code for the analogue operations. Understandability was a subjective criterion for the system expert, which to some extend goes together with complexity but also with the code structure, method naming and code documentation.

The result of the analysis showed that both models have high reuse potential whereby the better understandability is given by the CDT. The JDT solution is more complex but provides a fine-grained event propagation mechanism.

DCA provided a systematic analysis of the existing systems and provided us with useful results for concepts, features and patterns for the development of our own plug-ins. Based on the above results the architect ranked the CDT alternative higher than the JDT solution since he do not need a fine-grained event mechanism that would have introduced unnecessary complexity into the development. Therefore the CDT solution was selected as basis for our plug-ins under development.

Although the initial investment of time to analyze the existing systems and the subsequent adaptation for our purposes was higher than a development from the scratch, the architect profits from reusing sound design concepts in a stable architecture that fulfills our requirements with no need for major redesign (something that was experienced in other projects starting development from scratch). The analysis not only provided tested artifacts from mature software systems, which increases the quality of the plug-ins under development but moreover we learned about different alternatives for solving similar problems. Based on the information gained from the analysis the architect could compare the systems systematically and choose the solutions he rated best with respect to the requirements.

5 Related work

The software architecture analysis method (SAAM [10]) evaluates the modifiability of software architectures with respect to a set of representative change scenarios. The architecture tradeoff analysis method (ATAM [10]) is also a scenario-based method, which extends SAAM to address further quality attributes. Its goal is to analyze whether the software architecture satisfies given quality requirements and how the satisfaction of these quality requirements trade off against each other. In contrast to the design comparison approach, the two above methods focus on evaluating the complete architecture of a system using expert-driven scenario-based analysis. Our approach evaluates design alternatives using information obtained through reverse engineering techniques of field-tested systems. Furthermore, our approach strongly relies on evaluation techniques beyond scenario analysis.

The software architecture comparison analysis method (SACAM) by Stoermer et al. [23] aims at comparing the fitness of architecture candidates to be used in an envisioned system (i.e., a goal similar to ours). Whereas they rather rate the fitness of architectures as a whole with respect to quality attributes and business goals, we focus rather on smaller design parts that are reusable in the context of the new product line. In the evaluation phase they focus on scenarios, while we propose as well other means (e.g., simulation, instrumentation or prototyping). The SACAM is organized in a workshop-oriented manner. Our design comparison approach differs in this point because we emphasize on a close cooperation of architects, experts and reverse engineers because of the benefits of a common understanding of ongoing activities and intermediate results.

Bosch [6] presents four architecture assessment techniques (i.e., scenario-, simulation-, mathematical model- and experience-based assessments). These techniques aim at the evaluation whether a system fulfills its quality requirements or not. In contrast to our approach Bosch’s approach focuses on the architecture of a single system. The scenario- and simulation-based assessments are similar to some of our evaluation techniques.

6 Conclusion

In this paper, we presented DCA, an approach to answer the request of a product line architect facing hard design decisions by extracting and comparing the design alternatives present in field-tested systems. In the two case studies, DCA's systematic analysis of the selected systems provided useful insights about concepts, features and patterns that helped the architect to design a product line architecture from which two successful instances were derived.

6.1 Lessons Learned

Although applying DCA requires additional effort, in our case studies the architects appreciated the support it provided and estimated that it helped them produce a better architecture, with a broader acceptance, and a shared understanding. The better architecture can be in part attributed to the addition of more objective inputs and the analysis of mature solutions. More people accept the resulting design, since they directly contributed as expert or because solutions from their systems made their way into the new design. We expect that this facilitates the transition of an organization towards product line engineering. DCA explicitly supports the documentation of the rationales for key design decisions.

In the case studies, we observed that creating conceptual models of distinct solutions contributed to the common understanding, shared vocabulary, and considering a broader design space. Based on this experience we are confident that the additional effort required by DCA pays off for hard decisions. This hypothesis will be the subject of a future experiment.

6.2 Future Work

Ongoing work will perform further case studies and we currently plan to apply the approach in industrial migration projects where organizations want to migrate from single system development towards product line engineering.

In future, we will broaden and refine our catalog of reverse engineering techniques, questionnaires to accelerate the request analysis, and extend our evaluation approach catalog. We will formalize the insight gathered through the case studies into hypotheses and validate them experimentally where possible and appropriate.

7 References

[1] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J.: *Component-based Product Line Engineering with UML*, Component Software Series, Addison-Wesley, 2001.

[2] Bass, L., Klein, M., and Bachmann, F.: *Quality Attribute Design Primitives and the Attribute Driven Design Method*, in Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), 2001.

[3] Bassett, P.: *Framing Software Reuse. Lessons From the Real World*, Yourdon Press, 1997.

[4] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M.: *PuLSE: A Methodology to Develop Software Product Lines*, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), ACM, Los Angeles, CA, USA, May 1999.

[5] Bayer, J., Forster, T., Ganesan, D., Girard, J.-F., John, I., Knodel, J., Kolb, R., and Muthig, D.: *Definition of Reference Architectures based on Existing Systems*, Fraunhofer IESE, March 2004.

[6] Bosch, J.: *Design and Use of Software Architectures*, Addison-Wesley, 2000.

[7] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[8] CDT: *C/C++ Development Tools, http://eclipse.org/cdt/*, October 2004.

[9] Clements, P., and Northrop, L.: *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, August 2001.

[10] Clements, P., Kazman, R., and Klein, M.: *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

[11] CobolDT: *Cobol Development Tools, http://eclipse.org/cobol/*, October 2004.

[12] Eisenbarth, T., Koschke, R., and Simon, D.: *Aiding Program Ccomprehension by Static and Dyamic Feature Analysis*, in ICSM, Florence, Italy, 2001.

[13] Eisenbarth, T., Koschke, R., and Simon, D.: *Locating Features in Source Code*, IEEE Transactions on Software Engineering, March 2003.

[14] IEEE1471: *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems*, IEEE Computer Society, IEEE Computer, 2000.

[15] JDT: *Java Development Tools, http://eclipse.org/jdt/*, October 2004.

[16] Knodel, J.: *On Analyzing Interfaces of Components*, in 6. Workshop Software Reengineering, Softwaretechnik-Trends, 2004.

[17] Kruchten, P.: *The Rational Unified Process: An Introduction*, Addison-Wesley, March 2000.

[18] Murphy, G., and Notkin, D.: *Software Reflexion Models: Bridging the Gap between Source and High-Level Models*, in ACM SIGSOFT~'95: Proceedings of the Third Symposium on the Foundations of Software Engineering (FSE3), Washington, D.C., October 1995.

[19] Riva, C., and Rodriguez, J. V.: *Combining Static and Dynamic Views for Architecture Reconstruction*, in CSMR, Budapest, Hungary, March 2002.

[20] Sartipi, K., Kontogiannis, K., and Mavaddat, F.: *A Pattern Matching Framework for Software Architecture Recovery and Restructuring*, in IWPC, June 2000.

[21] Schmid, K.: *A Comprehensive Product Line Scoping Approach and Its Validation*, in Proceedings of the 24th International Conference on Software Engineering (ICSE'02), May 2002.

[22] Shavor, S., D'Anjou, J. and Fairbrother, S.: *The Java Developer's Guide to Eclipse*, Addison-Wesley, June 2003.

[23] Stoermer, C., Bachmann, F., and Verhoef, C.: *SACAM: The Software Architecture Comparison Analysis Method*, CMU/SEI, SEI-2003-TR-006, December 2003.

Document Information

Title:	Comparing Design Alternatives from Field-Tested Systems to Support Product Line Architecture Design
Date:	June 30, 2004
Report:	IESE-078.04/E
Status:	Final
Distribution:	Public