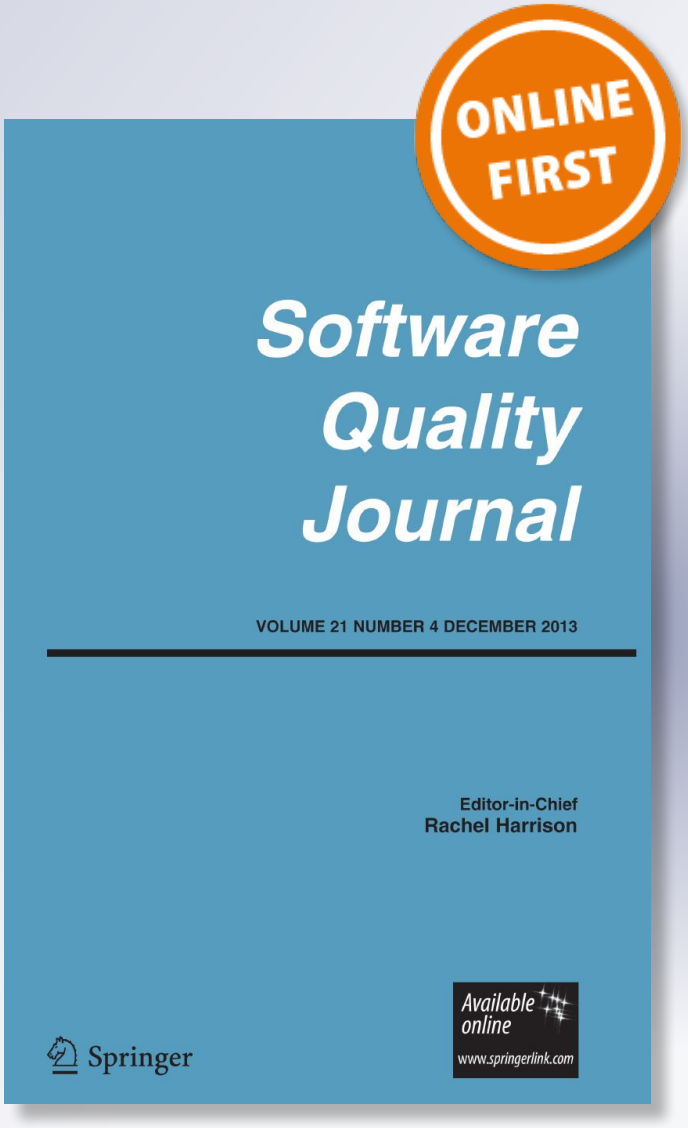
Round-trip engineering with the Two-Tier Programming Toolkit

A.H. Eden, E. Gasparis, J. Nicholson & R. Kazman

Software Quality Journal

ISSN 0963-9314

Software Qual J
DOI 10.1007/s11219-017-9363-9



Springer

Software Qual J
DOI 10.1007/s11219-017-9363-9

Round-trip engineering with the Two-Tier Programming Toolkit

A.H. Eden[1] · E. Gasparis[2] · J. Nicholson[3] · R. Kazman[4,5]

Abstract A major impediment to the long-term quality of large and complex programs is inconsistency between design and implementation. Conflicts between intent and execution are common because detecting them is laborious, error-prone, and poorly supported, and because the costs of continuously maintaining design documents outweigh immediate gains. A growing inconsistency between design and implementation results in software that is unpredictable and poorly understood. Round-trip engineering tools support an iterative process of detecting conflicts and resolving them by changing either the design or the implementation. We describe a Toolkit which supports a round-trip engineering of native Java programs without interfering with any existing practices, tools, or development environments, thereby posing a minimal barrier on adoption. The Toolkit includes a user-guided software visualization and design recovery tool, which generates Codecharts from source code. A "round-trip" process is possible because Codecharts visualizing source code can be edited to reflect the intended design, and the Verifier can detect conflicts between the intended and as-implemented design. We demonstrate each stage in this process, showing how the Toolkit effectively helps to close the gap between design and implementation, recreate design documentation, and maintaining consistency between intent and execution.

Keywords Design verification · Tool support · Software evolution · Java · Codecharts

Categories and Subject Descriptors (ACM)—D.2.10 Software Design, D.2.2 Design Tools and Techniques, D.1.5 Object-Oriented Programming, K.6.3 Software Maintenance

✉ A.H. Eden
eden@sapience.org

[1] Sapience.org, London, UK

[2] Singularity Labs, Voudouri Street, 341 00 Chalkida, Evia, Greece

[3] ZiNET Data Solutions, 25 Russell Street, Hastings TN34 1QU, UK

[4] Software Engineering Institute, Carnegie Mellon University, Fifth Avenue, Pittsburgh, PA 15213-2612, USA

[5] University of Hawaii, Maile Way, Honolulu, HI 96822, USA

Published online: 11 May 2017

1 Introduction

It has been argued that supporting human understanding must be the primary goal of the software engineering tools (Jackson 2008). Understanding the discrepancy between the intended and as-implemented design of a complex software system is particularly crucial for managing software evolution. Improving on this goal is the purpose the tools described in this paper and of the round-trip engineering process they support.

Maintaining consistency between designs and implementations is one of the discipline's central challenges. Consistency needs to be maintained for several reasons. First, because programmers need a current and accurate view of the as-implemented design for the purpose of software maintenance, evolution, and reuse. The absence of such a view inevitably leads to a growing conflict between design and implementation, aka *architectural drift*, which may ultimately result in software that is poorly understood and unpredictable (Biggerstaff 1989; Koschke 2001). To provide this view, the program's design must be documented and kept current and accurate throughout its lifecycle. This poses a particular challenge for software projects developed using iterative or agile processes where changes in design decisions and in the implementation occur regularly throughout the software lifecycle. Further, detecting design violations (Mo et al. 2015) is important for ensuring the quality and integrity of software, and crucially so for high-security and safety-critical software.

To keep the design and implementation synchronized, practitioners have used a range of forward and reverse engineering tools. However, without proper integration between the tools, this task can be laborious and potentially error-prone, hence often neglected. Consequently, design documents increasingly become inconsistent with the current version of the implementation, leading to software that is poorly understood and unpredictable.

This paper describes an integrated set of tools that support the goal of round-trip engineering, the Two-Tier Programming (TTP) Toolkit—henceforth, *the Toolkit*. The unique contribution of the Toolkit is in *closing* the round-trip cycle. That is, information generated by the reverse engineering tool directly feeds into the forward engineering tool and vice versa. Crucially, diagrams generated by our design recovery tool are indistinguishable from modeling diagrams manually generated by the programmer. In other words, diagrams which were reverse engineered from the implementation can be edited just like diagrams that a programmer created modeling their design decisions. This capability is critical for the process to be called "round-trip".

Further, we show that, unlike any other set of tools, the Toolkit not only generates diagrams visualizing the as-implemented design from plain source code of any Java program but also provides a useful view of programs at *any* level of abstraction. As a result, programmers can continuously redesign and re-implement the program without having to re-generate either diagrams or source codes. Any inconsistencies between design and implementation can therefore be discovered immediately and eliminated, either by changing the design or by changing the implementation.

In our previous work (Eden and Nicholson 2011), we sketched the operation of each tool in the Toolkit. The conformance-checking tool (the Verifier) in Nicholson 2011; Eden et al. 2013 and the reverse engineering tool (the Design Navigator) is described in Gasparis et al., 2008a; Gasparis et al., 2008b; Gasparis 2010. The theory underpinning these tools is discussed in Eden and Nicholson 2011; Eden et al. 2013; Nicholson 2011. This paper focuses on *round-trip engineering*—the task of maintaining consistency between design and implementation throughout the software lifecycle. It shows how the Toolkit supports a seamless, integrated

cycle of modeling-implementation-visualization-modeling activities in practical settings, facilitating the propagation of changes from design to implementation and back at all stages of the lifecycle. Significantly, unlike code generation tools, the Toolkit allows programmers to continue using (native) source code, which means that it does not preclude the use of any other tool or software development environment.

2 Forward, reverse, and round-trip engineering

2.1 Forward engineering

Forward engineering tools support that part of the software development process which is concerned with the transition from design to implementation (Fig. 1). One approach to forward engineering is code generation, which seeks to represent the entire program at a level of abstraction higher than that offered by third-generation programming languages (Schmidt 2006) and *only* at that level. Code generators therefore ensure that the implementation conforms to design decisions by automating the transition from design to code. Such tools represent the application at a higher level of abstraction and create executable code from this representation. But code generators tend to "lock in" the project to tools specifically tailored to integrate with the respective tool.

Yet another approach, referred to as Automated Round-trip Engineering (Assmann 2003) assumes that there is a functional relation which maps the implementation to the design and seeks to automate the transition from implementation to design by computing this function. Unfortunately, to date, no such functional relation has been shown to map the constructs of object-oriented programs to object-oriented design, nor has it been proven that such a function can be defined at all.

Our interest lies with a different approach to forward engineering. It assumes that the transition from design to implementation is largely a manual undertaking. Such tools support program development and not seek to replace programming.

Some modeling-only tools help programmers to conceptualize and represent design decisions. Visual specifications are encoded using diagrams in notations such as UML Class Diagrams, Dataflow Diagrams, and Statecharts. Programmers however feel that modeling is not of high priority because diagrams often fail to represent design decisions clearly and precisely, and because there are very few tools that can enforce conformance to these diagrams. Furthermore, even if the implementation is consistent with the design during early stages in the software lifecycle, little priority is given to maintaining this consistency. The reasons are, among others, that daily changes to the implementation are mostly small and incremental, that conflicts with the design are not always immediately apparent, and that present-day tools do not support propagation of changes from implementation to design and vice versa. Consequently, diagrams created during the early stages are nearly always discarded and go unmaintained as the program evolves.

To overcome these difficulties, *design verification* tools check whether a given implementation conforms to the design and report any inconsistencies (Guttag et al. 1982; Wing 1990). Creating such tools however is difficult (Assmann 2003) because specifications are difficult or impossible to verify fully automatically (Eden et al. 2013). Further, source code contains so many implementation details that the process of conformance checking becomes particularly tricky.

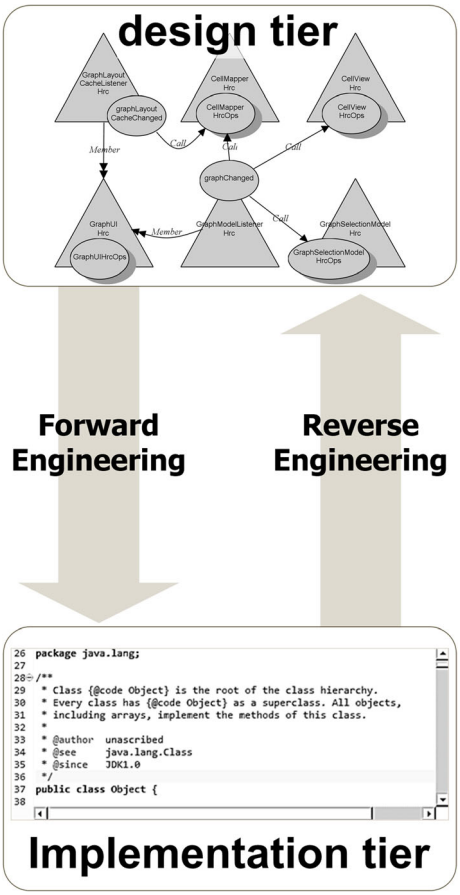


Fig. 1 Forward and reverse engineering

2.2 Reverse engineering

Reverse engineering tools (Müller et al. 2000) help programmers understand programs by generating representations that represent the program at a higher level of abstraction (Chikofsky & Cross 1990). Design recovery tools (Biggerstaff 1989) are reverse engineering tools that analyze source code and generate a picture or a diagram that represents the entire program or parts thereof "as-implemented" design (Kazman & Carrière 1999).

Unfortunately, while visualization tools help comprehension and maintenance, they leave the task of manually comparing the visualizations with the original design decisions to the programmer.

Compounding this problem, reverse engineering tools rarely integrate well with forward engineering tools. Apparently, the reason is because design decisions are frequently represented in a different language from the language of design recovery. For example, modeling tools that generate class diagrams will not communicate with design recovery tools that visualize programs as bitmaps. Hence, many reverse engineering tools simply ignore any preexisting diagrams, even if the diagrams were generated by the same tool and create a new diagram each time they are used (Assmann 2003).

Another problem is that reverse engineered visualizations can rarely be edited (with Rigi (Müller & Klashinsky 1988) as one notable exception). Updating visualizations with changes in the design or implementation therefore remains a manual task, which is time-consuming and error-prone. Therefore, visualizations are of little use during software evolution.

Furthermore, design recovery tools must scale with the size of the program. They must therefore be able to discover ever larger sets of correlated classes and methods and utilize powerful abstraction mechanisms to visualize them effectively. Some commercial integrated development environments (Nickel et al. 2000) (Guéhéneuc 2004) can generate UML class diagrams from source code. However, there is considerable doubt as to whether UML is the best choice for program visualization (Demeyer et al. 1999).

2.3 Round-trip engineering

The shortcomings of forward and reverse engineering tools place the onus to maintain consistency between design and implementation on the programmers. In reality, project resources are rarely allocated to this task. This issue has its greatest effect on software that evolves frequently, in particular in projects using agile and iterative development process. Consequently, an increasing gap is created between design and implementation, the results of which include architectural erosion and architectural drift (Perry & Wolf 1992), poor understanding of the system, and eventually losing control over it. This process contributes to a system's technical debt; specifically, this is a form of architecture debt (Xiao et al. 2016). But this debt is preventable, as we will show.

Round-trip software engineering tools support the task of reconciling design and implementation (Sendall & Küster 2004). They can integrate with software at any stage of lifecycle, in particular even if the code has no meaningful documentation.

A round-trip engineering process involves the following activities (Fig. 2):

- *Modeling*, which helps generate diagrams that specify design decisions about a program under development
- *Implementing*, aka programming, needs no special support, and is carried out using a traditional, general-purpose programming tool or development environment
- *Design verification*, henceforth *verification*, which checks whether the implementation conforms to the design, detects and reports inconsistencies between them
- *Visualizing*, which helps generate diagrams depicting the as-implemented design of an existing program

This approach to round-trip engineering requires tools that support a seamless, iterative cycle of modeling-implementing-verifying-visualizing-modeling. In particular, the tools must allow the user to reuse the same diagrams that were generated from the implementation and edit them to reflect changes in the design; verified to ensure consistency with the implementation; recovered again, and so forth. The transition from design to implementation and back closes the cycle (Fig. 2), hence "round-trip" engineering.

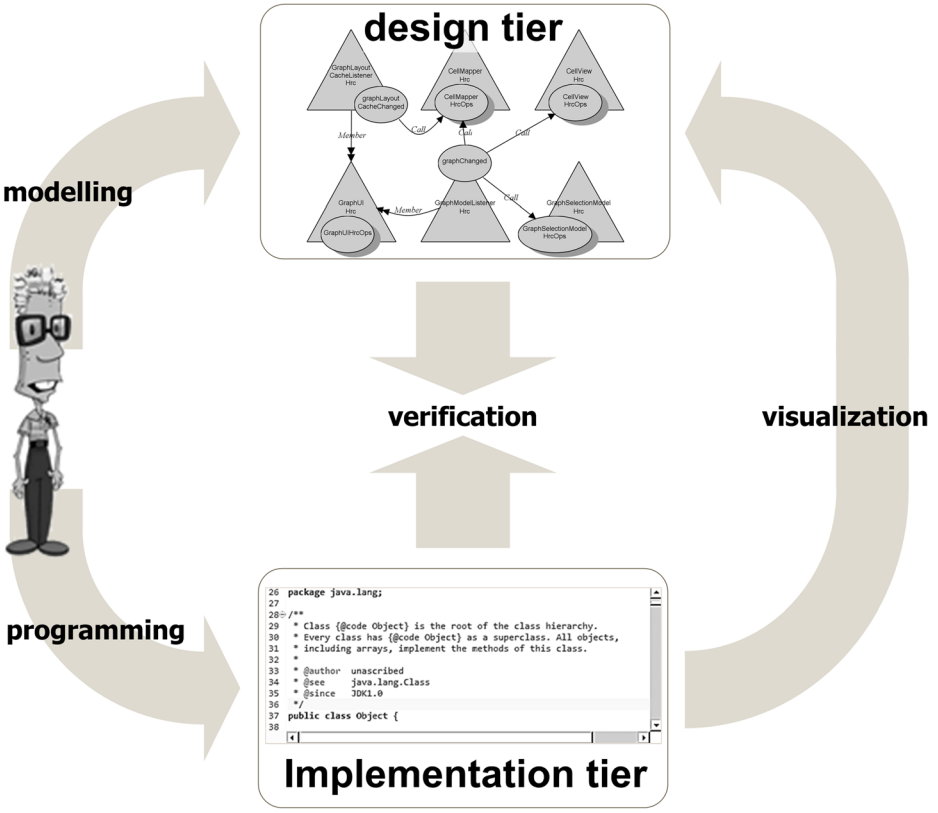


Fig. 2 Round-trip software engineering

Sendall and Küster (Sendall & Küster 2004) observe that supporting round-trip engineering is difficult and that most development tools support it only to a limited extent. This difficulty motivated us to develop the Toolkit.

3 The TTP Toolkit

The Toolkit supports round-trip engineering using several tools. The modeler is a visual editor used for creating Codecharts specifying the design of object-oriented programs, design patterns, and application frameworks. The Verifier is a *design verification* (c.f.) tool that checks the conformance of the source code to Codecharts and reports any conflicts between them. To support visualization, a design recovery tool called the Design Navigator can generate Codecharts representing the source code at any level of abstraction. Diagrams generated by the Design Navigator can be edited by the modeler, verified by the Verifier, and so forth, supporting the round-trip process.

The Toolkit provides tools that represent and maintain the association between the design and the implementation—a requirement commonly referred to as *traceability*. Traceability is supported by the Design Model Editor and the Assignment Editor. The information about the design and the implementation of a program are bundled using the "project" container. Users begin by

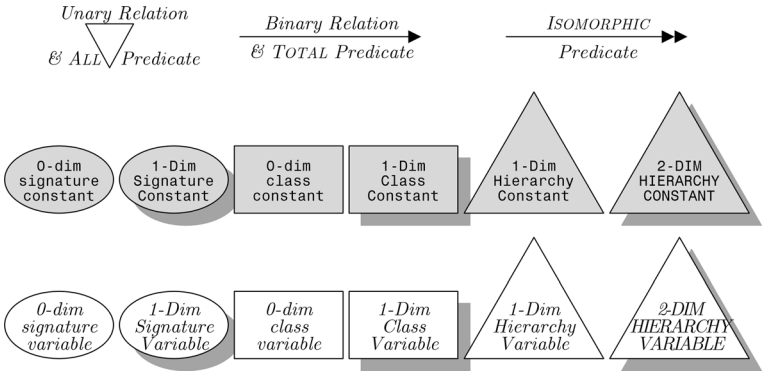


Fig. 3 The vocabulary of Codecharts

creating a project, to which they can add Codecharts, any number of implementations (source code file/folder locations), and all remaining resources.

The analyzer parses plain (native) source code of any program in standard Java 1.5[1]. The analyzer identifies entities (e.g., classes, methods) and relations (e.g., *inherit from*) and stores them in a repository called the Design Model of the program.

LePUS3, the language of Codecharts (Eden and Nicholson 2011; Eden et al. 2013), is a design description language whose vocabulary (Fig. 3) is tailored to represent the building blocks of object-oriented design, such as class hierarchies, sets of dynamically bound methods, and correlations between them. A Codechart is a formal expression in LePUS3 (1982) (1990): each Codechart is a formula in first-order predicate logic. The notion of conformance is defined in terms of the conformance of a model-theoretic structure generated from the source code (by the analyzer) to the formula a Codechart represents. Consistency between a given program and Codecharts can be automatically verified by means of static analysis since Codecharts are *fully decidable*—namely, satisfaction to such specifications can be determined by a *recursive function* (Sipser 1997).

In the remainder of this article, we demonstrate how this set of tools supports round-trip engineering by undertaking various typical activities in software design, implementation, and evolution.

4 Designing

In this section, we demonstrate how the modeling process fits with the forward engineering leg of the round-trip process.

4.1 Detailed design

The modeler (Eden and Nicholson 2011; Eden et al. 2013) supports creating diagrams that represent detailed design decisions about individual classes, methods, and their relations. For example, Fig. 4

[1] (Although the tools we implemented currently support only programs implemented in Java, analyzers of source code in other class-based programming languages can easily be integrated since no other component of the Toolkit interacts directly with the analyzer or with the source code. Proofs of concept for such analyzers have already been produced for C++, C#, and PHP5.

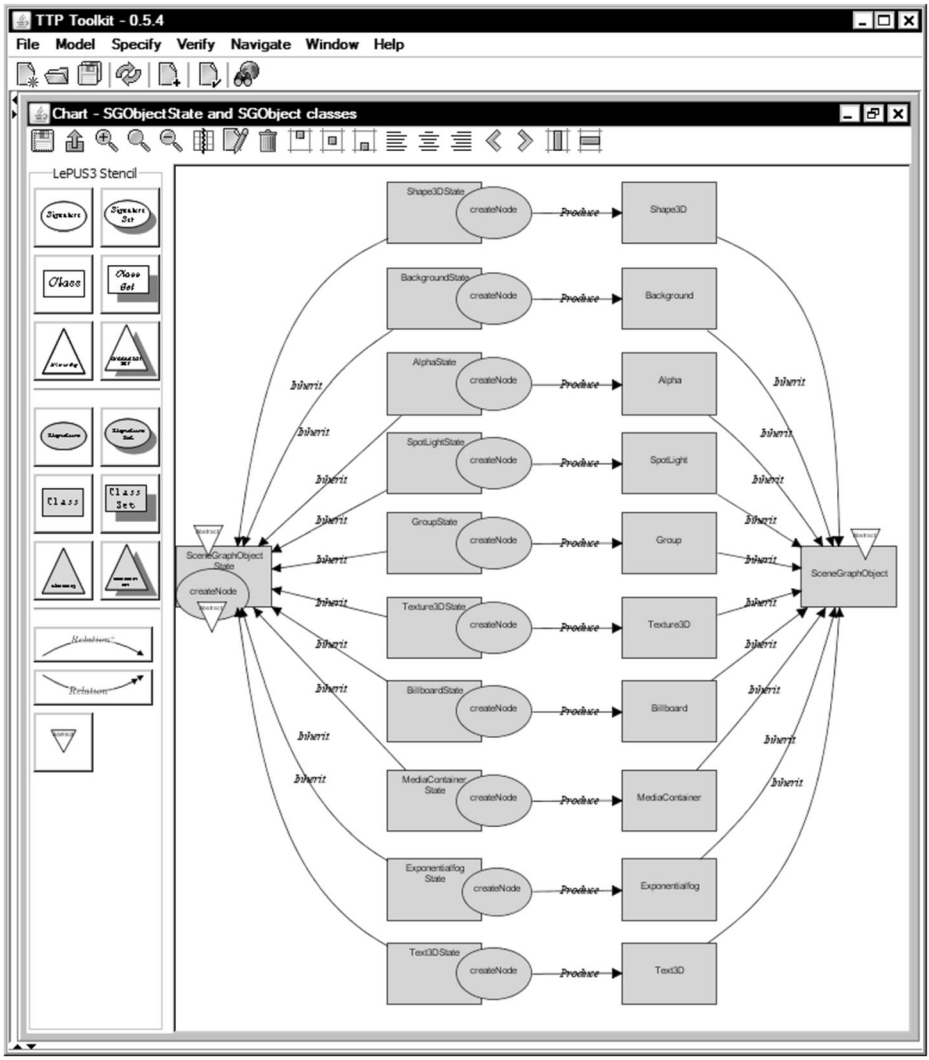


Fig. 4 Modeling detailed design. Codechart modeling classes Scene Graph Object (SGObjectHrc) and Scene Graph Object State (SGObjectStateHrc), their subclasses, and the relations between them

depicts a Codechart which was created by dragging tokens from the stencil (left pane) and dropping them onto the canvas (right pane).

Methods are represented by superimposing a signature term (ellipse) over a class term (rectangle). Relations between entities are represented using arrows labeled with the respective (binary) relation symbol. For example, the formula *Produce* (createNode⊗Shape3DState, Shape3D) specifies that the body of the method Shape3DState.createNode() contains a statement which creates an instance of class Shape3D and returns it. Properties are represented using inverted triangles drawn over the respective term.

4.2 High-level design

Practitioners conceive many design decisions in abstract, high-level terms, relating to packages, class hierarchies, sets of methods, and so forth. Implementation minutiae such as those expressed in Fig. 4 are unknown or irrelevant at early design stage. Consider for example this design decision about Java 3D version 1.3.1:

Design Decision 1. Java 3D high-level design (* Please insert a new line under the title)
Each Scene Graph State class defines a factory method that creates and returns the respective Scene Graph Object

Design Decision 1 articulates a generalization of the design modeled in Fig. 5: each "Scene Graph" class, namely, a class representing a type of a node in a Java 3D scene graph (e.g., shape, background, and spotLight) needs to have a twin state class (e.g., shape state, background state, and spotlight state). The reason is because state classes are factory classes: each can create instances of its respective "Scene Graph" class dynamically (for example, when restoring a scene graph from a serialized representation). Design Decision 1 is "abstract": it does not commit to how many classes count as "Scene Graph" class or a "Scene Graph State" class (there are over 200 in the version of Java 3D analyzed (Anon 2006)). Indeed, the *principle of least constraint* (Perry & Wolf 1992) requires that architectural design decisions should not commit to implementation details prematurely. The ability to carry out abstract modeling is therefore paramount for high-level design, allowing programmers to "see the forest from the trees". And programmers need tools that can model high-level design decisions at the appropriate level of abstraction.

For these reasons, the modeler also supports creating more abstract diagrams. For example, Fig. 5 shows how the modeler was used to represent Design Decision 1. The diagram depicts two *hierarchy constants* (triangles) designated SGObjectStateHrc and SGObjectHrc. (A *hierarchy* is a set of classes that includes one class that all other classes inherit from.) It also models a set of dynamically bound methods in SGObjectStateHrc, represented by superimposing the createNode symbol over the SGObjectStateHrc symbol. This combination stands for the set of

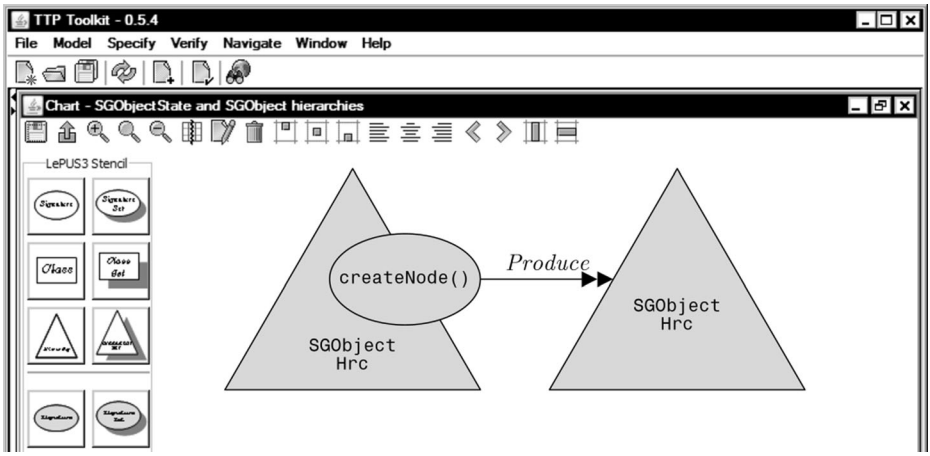


Fig. 5 Modeling high-level design. A Codechart modeling the Scene Graph Object and the Scene Graph Object State hierarchies and a set of dynamically bound factory methods

methods with signature createNode defined in the SGObjectStateHrc classes, including the abstract method SceneGraphObjectState.createNode().

4.3 Design patterns

Design patterns (Gamma et al. 1995) document design motifs in common practice. For example, the Factory Method pattern is heavily used in Java 3D. Since patterns are abstractions, they cannot be appropriately modeled using the same set of symbols that are used for modeling programs. For example, *Factory* (called *Creator* in Gamma et al. 1995) represents a role or a participant in the Factory Method pattern. This role is implemented in Java 3D by class SceneGraphObjectState. In other words, Factory is a generic placeholder which should not be confused with any specific implementation. Therefore, specific classes such as SceneGraphObjectState (Fig. 5) are modeled as *constants*, whereas pattern "participants" such as Factory are represented using *variables*, not unlike variables in standard mathematical practice. LePUS3 variables are characterized by white fill instead of gray (Fig. 6).

The modeler offers both variable and constant symbols in the same palate ("Stencil"). For example, as in the preceding figures, the Codechart modeling the Factory Method in Fig. 6 can

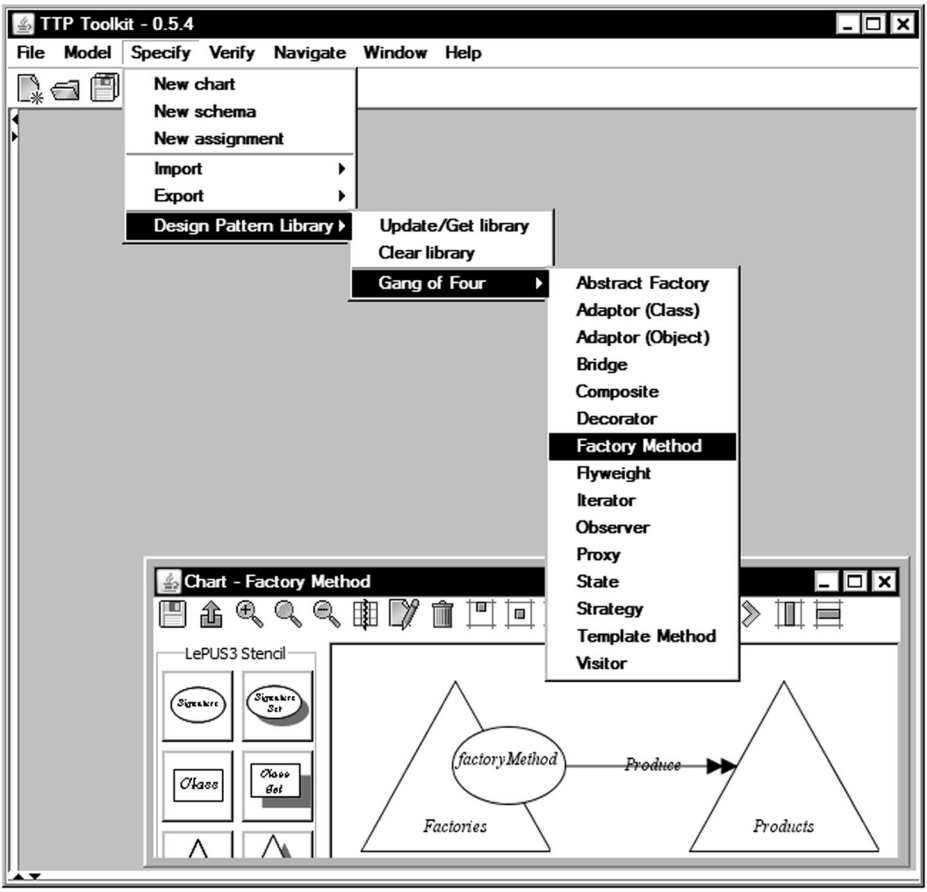


Fig. 6 Downloading the specification of the Factory Method design pattern

also be created by dragging symbols from the palate and dropping them on the canvas. Since the "gang of four" design patterns (Gamma et al. 1995) are frequently used, the Toolkit spares the programmer the effort of re-creating them by downloading their specifications as demonstrated in Fig. 6.

5 Detecting inconsistencies

Design verification (henceforth, *verification*) refers to the process of checking conformance of a given implementation to a specific set of design decisions, detecting and reporting conflicts between them. The theory underlying the process is presented in Eden and Nicholson 2011; Eden et al. 2013; Nicholson et al. 2014. For example, the Verifier can check the consistency of the Codecharts modeling the design of Java 3D (Fig. 5) with the source code of Java 3D. The result, depicted in Fig. 7, yields the result PASSED.

Verification can be carried out only after the Toolkit has analyzed the implementation. First, the Toolkit is provided with the location of the source code (left pane, Fig. 8). Next, the user clicks Analyze All and the Toolkit generates a *design model* for the program. For example, analyzing the Java 3D packages, it detects the class Shade3D and method Shade3DState.createNode(), and the relations "Shade3D *inherits from* SceneGraphObject" and "Shade3DState.createNode() *is-a-member-of* Shade3DState".

Codecharts that contain abstractions such as SGObjectHrc in Fig. 5 are verified by spelling out what each abstraction stands for. For example, we must indicate that

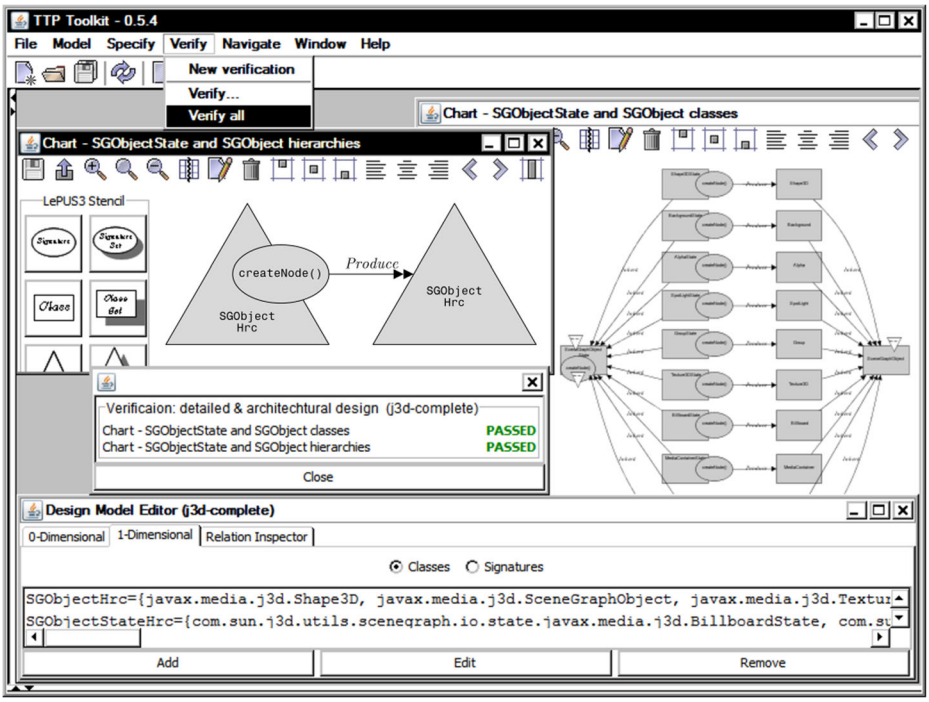


Fig. 7 Verifying the detailed (Fig. 4) and high-level (Fig. 5) design of Java 3D

SGObjectHrc (Fig. 5) stands for the hierarchy containing classes Shape, Background, SpotLight, etc., and similarly, spell out the classes in the SGObjectStateHrc hierarchy. To this end, the Design Model Editor maintains a dictionary which maps constants to the respective entities. For example, to modify SGObjectHrc, the user clicks the button "Add", creates an entry for SGObjectHrc, and indicates which classes are in Java 3D (Fig. 8). By doing so, the term SGObjectHrc becomes meaningful to the Verifier.

Verifying the consistency of an implementation with a design pattern requires a similar step because there is nothing to check in the implementation until the variables have given meaning in the program, technically referred to as the process of assigning them an *interpretation*. Therefore, we specify where the code is intended to implement the pattern—which can be in several places in the program. For this purpose, the Assignments Editor can be used to create and maintain *assignments*, which map entities to constants (Eden et al. 2013): functions that map variables to constants, as demonstrated in the left side of the screenshot in Fig. 9. For example, to define an assignment from the variables in the Factory Method pattern (Fig. 6) to the classes and methods in Java 3D, the user invokes the Assignments Editor and selects the Codechart "Factory Method" from the drop-down list (Fig. 9). The Assignment Editor then displays the list of variables in the Factory Method Codechart. The user can proceed to create the assignment depicted in Fig. 9 which maps the variable *Factories* to the constant SGObjectStateHrc, *Products* to SGObjectHrc, and *factory*

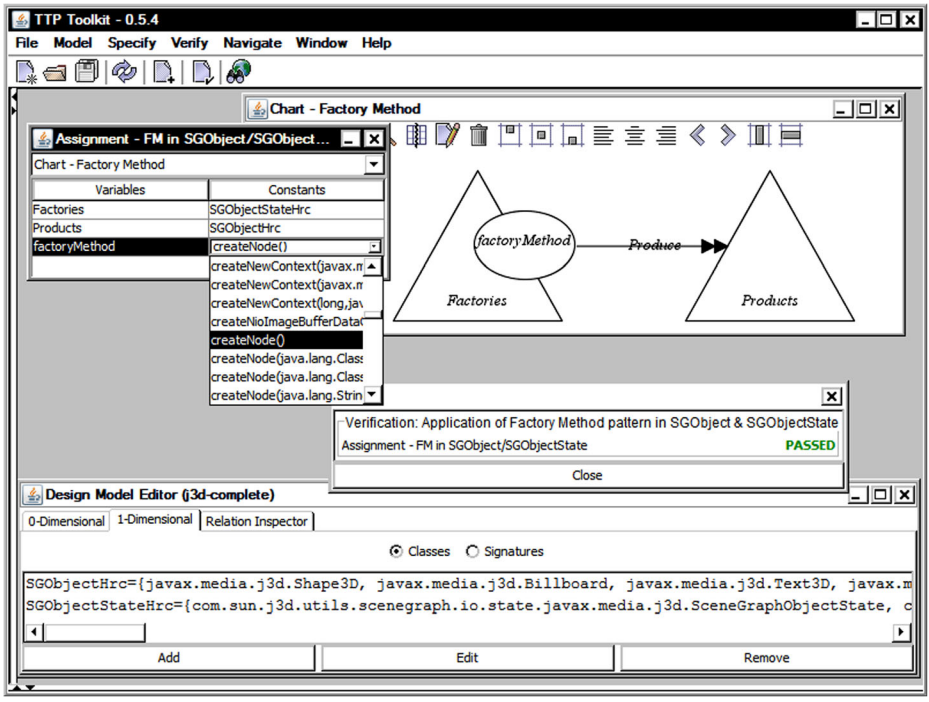


Fig. 8 Analyzing the source code and viewing the constants generated and their interpretation with the Design Model editor

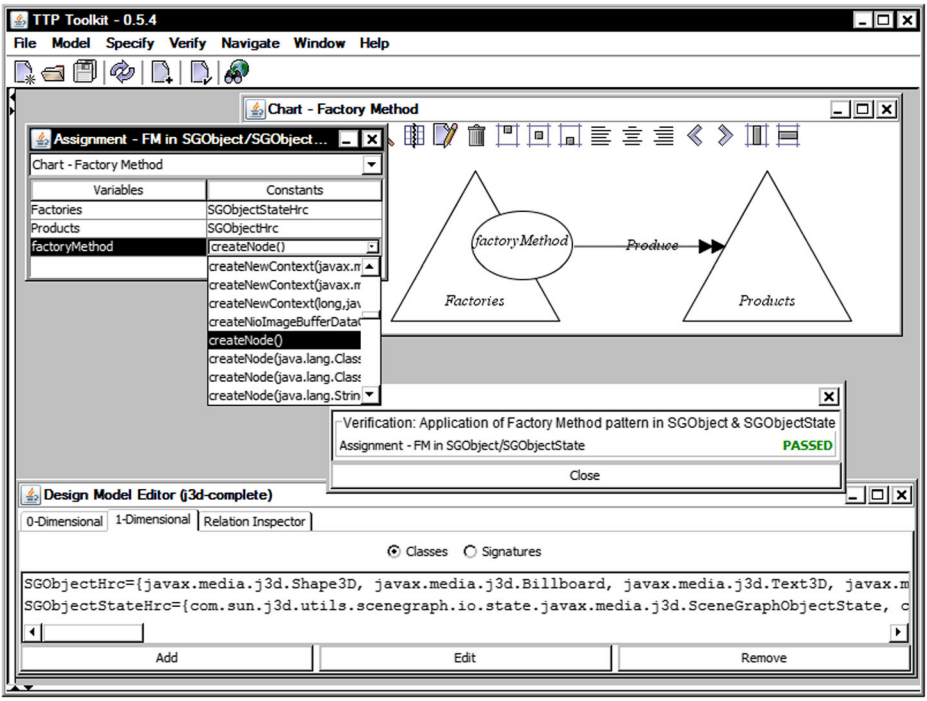


Fig. 9 Creating an assignment and the results of verifying it

Method to createNode. Figure 9 demonstrates the outcome of verifying the assignment depicted on the left, resulting in the message PASSED.

5.1 Evolving the implementation

Minor changes to the source code are continuously made during development and maintenance. Even the most innocent-looking modification may violate a design decision, often without the programmer realizing that. Of course, a conflict does not mean that the

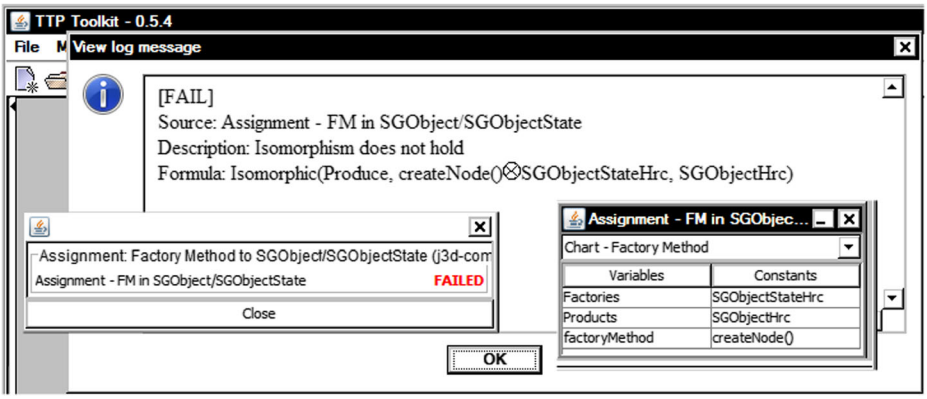


Fig. 10 Evolving the implementation. *Top*: a minor change is made to the source code. *Bottom*: an error message created the next time the Verifier is invoked

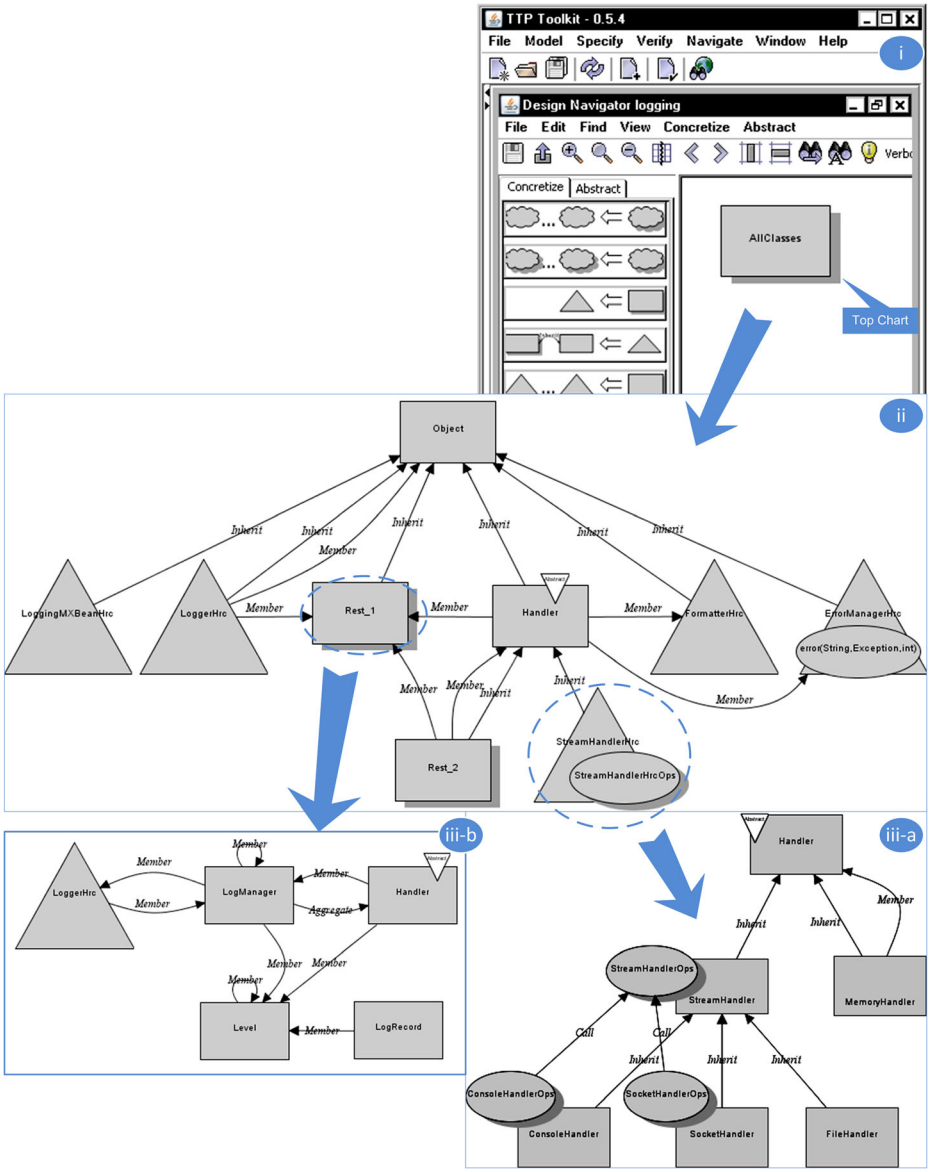


Fig. 11 A top-down visualization of the context to class Handler in the design recovery of package java.util.logging

implementation should change. Some conflicts can be reconciled by updating the design. What is important is that inconsistencies can be detected as soon as they emerge.

Figure 10 illustrates how the Toolkit can be used after evolving the implementation to detect new conflicts with the design. It depicts the source code of class Shape3DState after a small change in method createNode(). The evolved method returns an instance of class appearance instead of class Shape3D. The new implementation compiles, but Design

Decision 1 imposed by Fig. 5 and Fig. 9 was violated by this change. The reason is that the Factory Method pattern (§5) requires that each method in the set *factoryMethod*⊗ *Factories* returns an instance of a class in the *Products* hierarchy. And the assignment which the programmer created maps the set *factoryMethod*⊗*Factories* to the set of createNode() methods in the SGObjectStateHrc hierarchy, and also maps the *Products* hierarchy to the SGObjectHrc. However, the evolved factory method Shape3DState.createNode() returns an instance of class Appearance which is not in the SGObjectHrc hierarchy.

The Toolkit detects this conflict and reports it as depicted in Fig. 10 via the "FAILED" status. The detailed error message depicts the specific formula which represents the particular requirement mentioned. The programmer can now decide how to restore consistency by changing either the design or the implementation.

To resolve this conflict, the programmer could for example change the implementation. For example, the return statement in method Shape3DState.createNode() can be changed to create an instance of a class which does belong in the SGObjectHrc hierarchy. Or, class Appearance can be moved into the SGObjectHrc hierarchy.

Alternatively, the conflict can be resolved by changing the design. For example, by deciding that hierarchies SGObjectStateHrc and SGObjectHrc do not in fact implement the Factory Method pattern. Other solutions may be appropriate. Whatever solution is chosen, consistency between design and implementation can be restored.

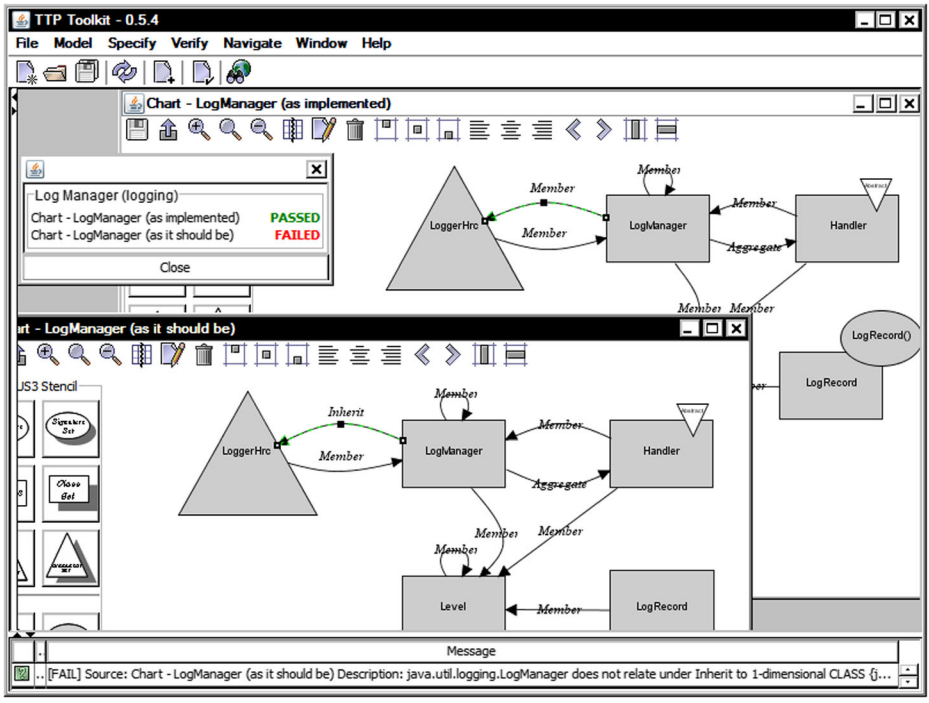


Fig. 12 Evolving the design. The programmer can open a visualization (*upper*) and edit it according to his/her changes in the design (*lower*). If the change is not implemented, it creates a conflict that the Verifier detects, returning an error message

6 Reverse engineering

Software understanding poses considerable challenges, and tools that facilitate it are of great practical value. The Design Navigator (Gasparis et al. 2008a; Gasparis et al 2008b; Eden et al. 2013) is a user-guided design recovery and visualization tool built by the principle: *overview first*, *zoom*, *and filter*, *then details-on-demand* (Shneiderman 1996), known as the "visual information-seeking mantra". Next, we briefly summarize the principles of this process and demonstrate how it fits into the round-trip process.

The Design Navigator supports a step-wise process we refer to as *design navigation*. At each step, the user selects a set of symbols in the Codechart and indicates whether to *concretize* or *abstract* them by choosing the appropriate operator (displayed on the left of Fig. 11(i)). Concretization operators generate a more detailed Codechart, for example, by splitting a set of classes to several subsets. Abstraction operators generate a more abstract Codechart, for example, by merging the representation of several classes.

Consider for example a programmer who has recently been assigned to extend the functionality of class Handler in Java's logging framework as implemented in package java.util.logging in Java's Standard Development Kit. Let us assume, not unrealistically, that the package's design has not been documented or that the documentation is unreliable because it has not been kept up to date with the evolution of the source code. The design recovery process illustrated in Fig. 11 demonstrates how the Design Navigator can help this programmer understand the as-implemented design of the package in general and of class Handler in particular.

After instructing the Toolkit to analyze the source code, this programmer can begin the visualization process by clicking Navigate ➔ From Top. This step produces the Codechart depicted in Fig. 11(i), which offers the most abstract representation of the entire package. This first Codechart, designated the Top Chart, contains only one term, AllClasses, which represents the set of all the classes in the package. To obtain a more meaningful view, the programmer concretizes AllClasses by choosing a concretization operator (left in Fig. 11(i)). A sequence of such concretization steps yields the Codechart depicted in Fig. 11(ii), which models class Handler individually and clusters the remaining classes in the package under five class hierarchies and two sets of classes. For example, Fig. 11(ii) reveals that every class in the StreamHandlerHrc hierarchy extends (possibly indirectly) class Handler. It also tells the programmer that class Handler has a data member whose type is some class in the set Rest_1. Its relation to class Object, hierarchy FormatterHrc, and the set of classes Rest_2 is also displayed.

The level of abstraction of the Codechart in Fig. 11(ii) may be useful for a number of tasks. Programmers however may wish to have more detail about the StreamHandlerHrc class hierarchy, each class in which may be affected by changes in Handler. To this end, they can delete all other constants in Fig. 11(ii) and concretize StreamHandlerHrc. The result is the Codechart depicted in Fig. 11(iii-a). Similarly, to find out more about the classes in the set Rest_1 and their relation to Handler, a programmer can concretize Rest_1, resulting in the Codechart in Fig. 11(iii-b).

This demonstrates that, compared to other reverse engineering tools which usually offer a "picture at a click", the Design Navigator requires more effort from the user. The reason is because the Design Navigator supports a process of reverse engineering that is tailored to the needs of the programmer. Every [part of every] program can be visualized in any number of ways. Depending on the programmer's needs, classes, methods, and relations may need to be

represented collectively (abstractly) or individually (concretely). In particular, for large software systems containing many thousands of classes, programmers may need answers to specific questions which concern only a small subset. To summarize, while tools that take the "one visualization fits all" approach require less manual effort, the Toolkit is geared towards answering specific questions *at the appropriate level of abstraction*, as guided by the specific interests of the user.

7 Evolving the design

We showed that Codecharts can be created manually or reverse engineered from an existing implementation. Significantly, Codecharts generated either way are indistinguishable: a reverse engineered Codechart can also be edited to reflect subsequent changes in the design and then verified against the code. Let us demonstrate how this helps detecting inconsistencies after the design has evolved. The Codechart in Fig. 12 was generated during the process of reverse engineering described previously. Consider now the following evolution step: class LogManager, which presently has a member whose type is some (any) class in the LoggerHrc hierarchy, should instead *inherit from* some (any) class in the LoggerHrc. This change in the design is made by editing the Codechart. The result is depicted in the Codechart entitled "LogManager (as it should be)" in Fig. 12. After changing the design, the programmer is free to change the implementation accordingly. Significantly, if the programmer forgets to change the implementation, or if the changes implemented are inconsistent with the revised design, the Verifier would detect the inconsistency and report it, as illustrated by the word "FAILED" in the dialog box in Fig. 12.

8 Evaluation

A pilot study compared the performance of programmers engaged in software engineering tasks using the tools described previously with those who used industry-standard tools. Next, we sketch the study's results. More details are provided in (Eden et al. 2013).

The two experiments were designed as follows. Participants were computer science graduate students (ten in the first experiment, eight in the second) who had no prior experience with the Toolkit. They were paid a fixed amount irrespective of the time it took them to complete the tasks. Variations among the participants' individual competence and skill were controlled by randomly dividing participants in each experiment into two groups. Participants in the experiment group used Codecharts and the Toolkit. Participants in the control group used Javadocs and NetBeans IDE version 6.1, which can be used to read and search in Java source files and to generate UML Class diagrams from Java source code. Following 1 h of training in the respective tools, a second step in controlling for variations in competence was to divide each experiment into two sessions: after completing the first session, participants swapped groups such that members of the experiment group became members of the control group and vice versa.

The first experiment was concerned with software comprehension, which measured the extent to which using Codecharts and the Toolkit provided answers to questions about a given implementation. In the first session of the experiment, participants were asked to find four methods in a specific class that answer to a particular description among classes in the Abstract

Windowing Toolkit in Java SDK. In the second session, participants were asked to find two methods that satisfy a particular description in java.io. The results of this experiment are very positive: they show that programmers who used the Toolkit to understand the program required on average just 23% of the time it took them to carry out the same task using NetBeans.

The second experiment was concerned with software conformance, which sought to gauge the effectiveness of answers to questions of consistency between design and implementation. Specifically, we measured the accuracy of judgments that programmers made about whether a given set of Java files from the AWT library conforms to the composite design pattern, whose description in Gamma et al. 1995 was replicated for the purpose. After swapping groups, participants in the second session were asked to judge whether a given set of classes and methods from package java.io in the Java SDK conforms to the description of the Decorator pattern as reproduced from Gamma et al. 1995. Our results of the second experiment were also positive, showing that software programmers who used the Toolkit were 1.75 times more likely to give the correct answer.

9 Related work

This paper describes the Two-Tier Programming Toolkit, an integrated set of tools that jointly support a process of round-trip engineering. Previous publications described the support of separate components in reverse (Gasparis et al. 2008a.; Eden et al. 2013) and forward (Eden et al. 2013; Nicholson et al. 2014) engineering, as well as the modeling language (Eden and Nicholson 2011; Nicholson 2011) they use. In this paper, we demonstrated the unique contribution that this combination of tools lends to the task of maintaining consistency between design and implementation throughout the software lifecycle, supporting a seamless *round-trip engineering* process.

Several articles provide a mathematical analysis of the notion of round-trip engineering (Henriksson & Larsson 2003; Hettel et al. 2008; Assmann 2003). Using precise mathematical techniques such as projection, they provide formal definition for the transformation from design to implementation and vice versa.

A few round-trip engineering tools have been described in the literature. The FUJABA project includes tools that support the representation of design decisions using story diagrams, which are described as a high-level programming language which uses UML class and behavior diagrams. The FUJABA toolset includes both forward engineering tools that generate source code from story diagrams (Fischer et al. 1999) as well as a reverse engineering tool that generates story diagrams from source code (Nickel et al. 2000). However, the literature on these tools does not describe how story diagrams representing design decisions can be compared and reconciled with story diagrams that were reverse engineered from the source code, a critical aspect of round-trip engineering. Consequently, the task of checking consistency between design and implementation past the point of code generation using the FUJABA toolset appears to be an open problem.

Rational Rose (Quatrani 1998) is a commercial tool that supports modeling design decisions by creating UML class diagrams. It can generate class and method stubs from such diagrams. It also supports the automatic generation of class diagrams from Java or C++ source code, among others. Its claim to support round-trip engineering rests on its ability to update diagrams to reflect to certain changes in the source code, and (to some extent) to propagate changes in the source code back to the diagrams. To maintain the trace information between

the representations, Rose annotates each element in the implementation with comments that associate it with its respective representation in the diagrams. The advantages of the Toolkit are therefore twofold: first, unlike the Toolkit, Rose requires changes to the source code, which would not be simple for external libraries. Second, Rose relies on the UML, whose fitness for software visualization has been questioned (Demeyer et al. 1999).

SelfSync (Paesschen et al. 2005) supports the notion of round-trip engineering using the Model-Driven Engineering method. The tool seeks to synchronize programs written in the self-programming language with their model as visualized in an extension to the entity-relationship notation called extended entity-relationship (EER). The tool however is not concerned with design abstractions. Rather, the source code and the diagrams are taken to contain equivalent information, which means that one representation can be fully automatically derived from the other. In particular, each visual token in an EER diagram visualizes an individual implementation object. Consequently, the authors take SelfSync to be an automated round-trip engineering tool (Assmann 2003). However, the restriction on the level of abstraction implies that in effect, SelfSync does not fit the purpose of maintaining consistency between design and implementation.

OPM/PL (Goldberg & Wiener 2010) is a powerful suite of tools that support modeling and visualization. The tools combine information about the source code and diagrams modeling it, representing it in a Prolog database. Design decisions about entities (e.g., classes, objects, processes, and states) and relations (e.g., "login_ok extends log_entry") can be represented declaratively, either as Prolog facts or in a notation called Object-Process Diagrams. In addition, a programming language-dependent parser, which in the first instance was built for Erlang, can detect the entities and relations in the source code and feed them back into the database. The correlation between entities in the implementation and entities in the database is also maintained using Prolog facts, e.g., implements (erl:login_ok,login_ok). Conflicts between design and implementation can therefore be detected by creating queries that check whether the entities detected in the implementation also conform to the relations encoded in the database. The main difference between OPM/PL and the Toolkit lies in the notation they employ.

10 Summary and discussion

Architectural erosion and drift (Perry & Wolf 1992) are the norms in most software development projects, which lead to poorly understood software, in particular, systems that evolve frequently, creating technical debt. The increasing gap between design and implementation, intent and execution, results in software that is unpredictable and poorly understood, which significantly depreciates its quality. Over time, the gap is so large that systems become hopelessly complex and unmaintainable. Round-trip engineering tools can help programmers maintain consistency between design and implementation by creating diagrams where none exist, by editing diagrams to reflect the designers' intent, and by flagging conflicts between diagrams and source code. The Two-Tier Programming Toolkit supports round-trip engineering of Java programs modeled and visualized using Codecharts. Software development and maintenance projects can adopt the Toolkit at any stage in the program's lifecycle. The Toolkit does not change native source code and never requires programmers to undertake expensive migration processes. It therefore poses a very low barrier on adoption, unlike many other tools. Particularly powerful is its support for design recovery, a process during which the Toolkit can generate diagrams modeling systems at any level of abstraction and, within its scope, to an arbitrary level of detail (or lack thereof).

This line of work needs be extended to pursue several demands. Further experimentation is also needed for the purpose of providing more detailed empirical evaluation of the Toolkit by a study in a realistic (industrial) setting. Planned improvements to the Toolkit include supporting conformance checking to semi-decidable and undecidable relations using runtime verification, for example, by following *rules* as described in Barringer et al. 2004 and tools such as MOP (Chen & Roşu 2007). More modest objectives are to improve the Toolkit's human-computer interface and write an analyzer for object-oriented programming languages other than Java, such as C++ and C#. Further empirical studies are planned with a focus on validating security patterns (Yoder & Barcalow 2000; Wassermann & Cheng 2003; Ryoo et al. 2012) along the lines of the work described in Alzahrani et al. 2015.

Most recently, the Toolkit has been extended to include the capability to automatically detect instances of a given design pattern (Alzahrani 2015). More precisely, detection is defined to solve the following problem: Given program p and Codechart Ψ modeling a design pattern, where (if at all) does p contain instances of Ψ? This question can be answered by automated checking of candidate participants. A critical element in providing an answer to this question involves conducting an efficient search for such candidates. Preliminary studies have demonstrated the feasibility of such a search and the correctness and accuracy of the implementation.

Acknowledgements The authors wish to thank Raymond Turner for extending considerable support throughout this research; the Research Promotion Fund, the Knowledge Transfer Innovation Fund, University of Essex, and the EPSRC for funding various parts of this project. We also wish to thank Olumide Iyaniwura, Gu Bo, Maple Tao Liang, Dimitrious Fragkos, Omololu Ayodeji, Xu Yi, and Christina Maniati for their contributions to this research.

References

Alzahrani, A. A. H. (2015). *Modelling and automated detection of design patterns using Codecharts with applications to security patterns (draft)*. Colchester, Essex: University of Essex.

Alzahrani, A. A. H., Eden, A. H., & Yafi, M. Z. (2015). *Conformance checking of single access point pattern in JAAS using Codecharts. In the world congress on information technology and computer applications*. Tunisia: Hammamet.

Anon. 2006. *Java 3D*, http://java3d.java.net/. Available at: http://java3d.java.net/.

Assmann, U. (2003). Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science, 82*(5), 33–41.

Barringer, H. et al. (2004). Rule-based runtime verification. In B. Steffen & G. Levi, eds. *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 277–306. Available at: http://www.springerlink.com/content/had48cpgyeu1lnnc/abstract/ [Accessed October 29, 2012].

Biggerstaff, T. J. (1989). Design recovery for maintenance and reuse. *IEEE Computer, 22*(7), 36–49.

Chen, F., & Roşu, G. (2007). Mop: an efficient and generic runtime verification framework. *SIGPLAN Not., 42*(10), 569–588.

Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software, 7*(1), 13–17.

Demeyer, S. et al. (1999). Why unified is not universal: UML shortcomings for coping with round-trip engineering. In *Proc. 2nd Int'l Conf. on the Unified Modeling Language*. Lecture Notes in Computer Science. pp. 630–645.

Eden, A. H., & Nicholson, J. (2011). *Codecharts: roadmaps and blueprints for object-oriented programs*. Hoboken: Wiley-Blackwell.

Eden, A. H., et al. (2013). Modeling and visualizing object-oriented programs with Codecharts. *Formal Methods in System Design, 43*(1), 1–28.

Fischer, T. et al. (1999). Story diagrams: a new graph grammar language based on the unified modelling language and java. In *Proc. Theory and Application of Graph Transformations–TAGT'98*, *LNCS 1764*. Springer, pp. 296–309.

Gamma, E. Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Reading: Addison Wesley Longman.

Gasparis, E. (2010) *Design Navigation*: *recovering design charts from object-oriented programs*. PhD Dissertation. School of Computer Science and Electronic Engineering, University of Essex.

Gasparis, E., Nicholson, J., et al. (2008a). Navigating through the design of object-oriented programs. In 15th Working Conf. on Reverse Engineering—WCRE. Antwerp, Belgium.

Gasparis, E., Eden, A.H., et al. (2008b). The Design Navigator: Charting Java Programs. In *Tool Demonstrations*, *Proc. of 30th IEEE Int'l Conf. on Software Engineering—ICSE 2008*. Leipzig: IEEE Computer Society Press.

Goldberg, M. & Wiener, G. (2010). Round-trip modeling using OPM/PL. In 2010 I.E. International Conference on Software Science, Technology and Engineering (SWSTE). IEEE, pp. 13–21.

Guéhéneuc, Y.-G. (2004) A reverse engineering tool for precise class diagrams. In CASCON '04. IBM Press, pp. 28–41. Available at: http://dl.acm.org/citation.cfm?id=1034914.1034917 [Accessed November 30, 2012].

Guttag, J. V., Horning, J. J., & Wing, J. (1982). Some notes on putting formal specifications to productive use. *Science of Computer Programming, 2*(1), 53–68.

Henriksson, A. & Larsson, H. (2003). *A definition of round-trip engineering*, University of Linköping.

Hettel, T., Lawley, M. & Raymond, K. (2008). *Model Synchronisation*: *Definitions for Round-Trip Engineering*. In A. Vallecillo, J. Gray, & A. Pierantonio, eds. Theory and Practice of Model Transformations. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 31–45.

Jackson, M. (2008). Automated software engineering: supporting understanding. *Automated Software Engineering, 15*(3–4), 275–281.

Kazman, R., & Carrière, S. J. (1999). Playing detective: reconstructing software architecture from available evidence. *Automated Software Engineering, 6*(2), 107–138.

Koschke, R. (2001). Software visualization for reverse engineering. In Lecture Notes In Computer Science. Revised Lectures on Software Visualization, International Seminar. pp. 138–150.

Mo, R. et al. (2015). Hotspot patterns: the formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA). pp. 51–60.

Müller, H. A. et al. (2000). Reverse engineering: a roadmap. In *Proc. Conf. The Future of Software Engineering*. Limerick, Ireland: ACM, pp. 47–60. Available at: http://portal.acm.org/citation.cfm?id=336512.336526&type=series [Accessed March 30, 2010].

Müller, H.A. & Klashinsky, K., 1988. Rigi—a system for programming-in-the-large. In *Proc. 10th Int'l Conf. Software engineering*. Singapore: IEEE Computer Society Press, pp. 80–86.

Nicholson, J. et al., 2014. Automated verification of design patterns: a case study. *Science of Computer Programming*, 80, Part B, pp.211–222.

Nicholson, J. (2011). *On the theoretical foundations of LePUS3 and its application to object-oriented design verification*. PhD Dissertation: School of Computer Science and Electronic Engineering, University of Essex.

Nickel, U.A. et al. (2000) Roundtrip engineering with FUJABA (Extended Abstract). In 2nd Workshop on Software-Reengineering (WSR).

Paesschen, E. V., Meuter, W. D., & D'Hondt, M. (2005). SelfSync: A dynamic round-trip engineering environment. In L. Briand & C. Williams (Eds.), *Model driven engineering languages and systems* (pp. 633–647). Heidelberg: Lecture Notes in Computer Science. Springer Berlin.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes, 17*(4), 40–52.

Quatrani, T. (1998). *Visual modeling with rational Rose and UML*. Boston: Addison-Wesley Longman Publishing Co., Inc..

Ryoo, J., Laplante, P. & Kazman, R. (2012). Revising a security tactics hierarchy through decomposition, reclassification, and derivation. In Software Secutiry and Reliability. Washington, D.C.

Schmidt, D. C. (2006). Guest editor's introduction: model-driven engineering. *Computer, 39*(2), 25–31.

Sendall, S. & Küster, J. (2004). Taming model round-trip engineering.

Shneiderman, B. (1996). The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings of the 1996 I.E. Symposium on Visual Languages*. VL '96. Washington, DC, USA: IEEE Computer Society, p. 336–. Available at: http://dl.acm.org/citation.cfm?id=832277.834354 [Accessed June 30, 2014].

Sipser, M. (1997). *Introduction to the theory of computation*. Boston: PWS Pub. Co..

Wassermann, R. & Cheng, B. H. C. (2003). Security patterns. In Pattern Languages of Programs—PLoP 2003. Robert Allerton Park, MI.

Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer, 23*(9), 8–23.

Xiao, L. et al. (2016). Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York, NY, USA: ACM, pp. 488–498. Available at: http://doi.acm.org/10.1145/2884781.2884822 [Accessed May 23, 2016].

Yoder, J. & Barcalow, J. (2000). Architectural patterns for enabling application security. In B. Foote, N. Harrison, & H. Rohnert, eds. *Pattern languages of program design 4*. Addison-Wesley.

Amnon H. Eden is the Principal Scientist of the Sapience.org thinktank. Dr. Eden's work is concerned with the application of disruptive technologies and original thought to interdisciplinary questions in the theory and practice of software design and artificial intelligence. His original work on design pattern formalization and software modeling appeared in his book Codecharts and in three of the highest ranking outlets in Google Scholar's Top Publications—Software Systems. He co-authored the first entry on the Philosophy of Computer Science in Stanford Encyclopedia of Philosophy. Dr Eden's work on singularity hypotheses led to the publication of the first peer-reviewed comprehensive volume on the subject. Dr Eden received a Master in Computer Science (Machine Learning; Cum Laude) and a Doctorate in Computer Science (Software Engineering) from Tel Aviv University. He's held posts in Tel Aviv University, Israel Institute of Technology—Technion, Uppsala University, Concordia University, and the University of Essex, and fellowships at the Center For Inquiry and the Institute of Advanced Studies, Hebrew University of Jerusalem.

Dr Epameinondas (Notis) Gasparis is the CTO at The Singularity Lab, a webhosting and content management startup which he cofounded in 2013, and a seasoned programmer and system administrator, with more than 12 years of experience in software engineering. Notis conducted his PhD research in software visualization and design recovery at the University of Essex, focusing on the problem of recovering Codecharts from object-oriented source code. Notis' publication appeared in top ranking outlets, including ICSE, Science of Computer Programming, and Formal Methods in System design.

Dr Jon Nicholson is Chief Technology Officer at ZiNET Data Solutions Limited (http://zinethq.com/), who develop novel secure data transfer and analytic solutions for use in the context of education. Previously, he was a senior lecturer in Computer Science with the School of Computing, Engineering and Mathematics at the University of Brighton. Jon's research interests lay in the fields of software engineering, the knowledge representation, and the linked data. His research to date has focused on developing novel visual languages with practical application that are more accessible than symbolic or textual notations. Jon graduated his PhD at the University of Essex, studying formal specification and verification of object-oriented design statements. Jon is a Fellow of the Higher Education Academy (FHEA).

Rick Kazman is a professor at the University of Hawaii and a Principal Researcher at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method), and the Dali and Titan tools. He is the author of over 200 publications, and co-author of several books, including *Designing Software Architectures*: *A Practical Approach*, *Software Architecture in Practice*, *Evaluating Software Architectures*: *Methods and Case Studies*, and *Ultra-Large-Scale Systems*: *The Software Challenge of the Future*.