

The Design Navigator: Charting Java Programs

Epameinondas Gasparis

Department of Computing and
Electronic Systems, University
of Essex, UK

Amnon H. Eden

Department of Computing and
Electronic Systems, University
of Essex, UK and Center for
Inquiry, Amherst, NY, USA

Jonathan Nicholson

Department of Computing and
Electronic Systems, University
of Essex, UK

Rick Kazman

Software Engineering Institute,
Carnegie-Mellon University, and
University of Hawaii, USA

Abstract. Reverse engineering is largely concerned with charting that unknown territory which is the source code of an unfamiliar program. The Design Navigator is a semi-automated design mining tool which reverse engineers LePUS3 design charts from arbitrarily-large Java™ 1.4 programs at any level of abstraction in reasonable time. We demonstrate the Design Navigator’s step-wise charting process of the Java Foundation Classes, generating at each step decreasingly abstract charts thereof and discovering more building-blocks in the design of this class library.

Keywords. Reverse engineering, design mining, object-oriented design, software modelling

1. INTRODUCTION

It is a scenario that every experienced programmer dreads: use a large program whose documentation is inaccurate, lacking relevant details, out-of-sync with the implementation, or altogether nonexistent [7]. Commercial reverse engineering tools can generate unreadable diagrams cluttered with hundreds or thousands of visual tokens. If the source code is taken to be the terrain of the program then class diagrams are on a par with 1:1 maps: very detailed but not very useful. What is needed to navigate in an unfamiliar territory is a scaled (‘architectural’) roadmap. Most useful would be a design mining tool which, not unlike Google Earth [5], has the ability not only to chart unknown programs at any scale of abstraction but also to zoom-in on and zoom-out from any part of the chart, thereby concealing or revealing details about it. The Design Navigator is designed to do exactly that: analyze an arbitrarily-large (native) Java™ 1.4 program, discover its conceptual building-blocks, and chart (any part of) them at any level of abstraction in the entire spectrum between maximum information (a minimally-abstract or 1:1 map) and minimal information (a maximally-abstract or 1:∞ map).

The literature demonstrates that design mining is difficult but not impossible [6][7][8][9][12]. The Design Navigator’s approach to the problem is distinguished from most existing tools by the following:

- *User-guided:* Charts are generated interactively using zoom-in (‘concretization’) and zoom-out (‘abstraction’) operations
- *Formal:* Charts are statements in a mathematically defined specification and modelling language (LePUS3)

- *Programming-language independent:* The Design Navigator is not tailored to a specific language (only the analyzer is). It is suitable for design mining programs written in any class-based programming language (Java, C++, C#, etc.)

The Design Navigator generates charts in LePUS3 [1], an object-oriented architecture description language (ADL), the relevant features of which can be summarized as follows:

- *Rigorous:* LePUS3 is defined in first-order mathematical logic and its abstract semantics is defined in model theory
- *Scalable:* Scale in LePUS3 is achieved by abstraction mechanisms representing a small set of building-blocks
- *Object-Oriented:* LePUS3 charts capture and convey the building-blocks of object-oriented design, e.g. inheritance class hierarchies, sets of dynamically-bound methods, and total and isomorphic correlations between them

In this paper, we briefly demonstrate the capabilities of the Design Navigator in charting a small subset of classes from package `java.awt` of the Java Foundation Classes 1.4, henceforth *JFC*. We discuss the abstraction/concretization operators (Appendix) used and how the design navigation process leads naturally from one chart to another. The case study and our overall experience using the tool with other class libraries, shows how the Design Navigator gradually brings to light details of the design of Java™ programs that are otherwise hard to spot, thereby promoting their understanding.

2. DESIGN NAVIGATION

Design Navigation is a user-guided approach to design mining which generates on demand visual representations (LePUS3 charts) of the target program. Since LePUS3 is mathematically defined [1] and the abstract semantics of Java programs are also mathematically defined [10], the hypothetical ‘set of charts of *JFC*’ (mathematically: the set of charts that *JFC* implements) is also well-defined (Figure 1):

$$\text{Charts}(JFC) \triangleq \{C \mid \text{JavaAbsSemantics}(JFC) \models C\}$$

Design Navigation in this example is thus a user-guided, tool-assisted, step-wise process of traversing $\text{Charts}(JFC)$ and generating charts of *JFC*.

Theoretical analysis of this set [4] reveals that it has the mathematical properties of a lattice (Figure 1): a diamond-shaped grid of interconnected nodes (charts). At the bottom is the chart with maximum information about *JFC*, called the Bottom Chart of *JFC* (Chart 7). At the top is the (non-empty) chart with the minimal information about *JFC*, called the Top Chart of *JFC* (Chart 1).

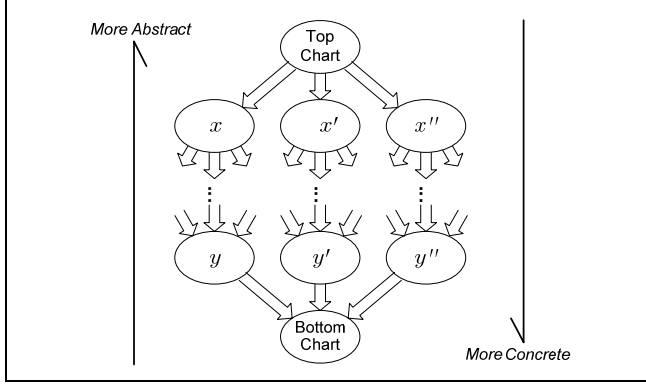


Figure 1 – Charts(*JFC*): The (hypothetical) set of charts of *JFC* has the shape of a lattice

Design Navigation is therefore a tool-assisted, step-wise process of traversing the (hypothetical) set in Figure 1. Each step of Design Navigation is either an *abstraction* (‘zooming-out’, or ‘up’ in Figure 1) or a *concretization* (‘zooming-in’, or ‘down’ in Figure 1) step. Since each chart *C* can be concretized or abstracted in many ways, depending which [set of] term[s] is concretized/abstracted, each node *C* in Figure 1 represents a chart in *Charts(JFC)* which is connected with several nodes above and below it such that—

- a node connected above *C* represents an *abstraction* of *C*
- a node connected below *C* represents a *concretization* of *C*

In the course of this paper we demonstrate a sample Design Navigation process in the set of charts of *JFC*, illustrated in Figure 2, consisting of a sequence of (mostly) *concretization* steps. It begins with the Top Chart of *JFC* (Chart 1), proceeding to produce (using the Design navigator) increasingly more detailed charts of *JFC*, and concluding with the Bottom Chart of *JFC* (Chart 7), which is the most detailed (‘concrete’) chart of the program.

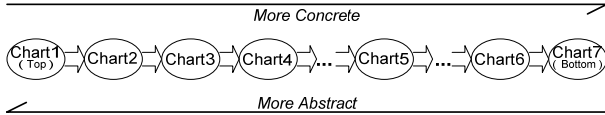


Figure 2 – The Design Navigation taken in this paper

The role of the Design Navigator is therefore, at each step, to take chart *C* and produce as instructed a chart *D* which is either an abstraction or a concretization of a [specific term/set of terms in] chart *C*. The set of abstraction/concretization operators the Navigator supports is discussed in the Appendix.

3. CASE STUDY: CHARTING *JFC*

What can we learn about *JFC* using the Design Navigator? In this section we describe the building-blocks of *JFC* we discovered in the course of our Design Navigation process (Figure 2). All charts were produced by the Design Navigator, but some terms have been re-arranged to improve readability, and the Elimination abstraction operator has been judiciously applied to remove irrelevant terms.

3.1 Birds eye-view

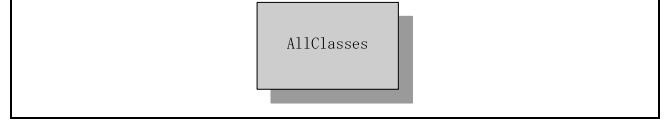


Chart 1 – Top chart.

AllClasses stands for the set of all classes in *JFC*

The Top Chart of *JFC* (Chart 1), which the Design Navigator generated by analyzing *JFC*, only reveals that there exists a set of classes in our program. Chart 1 is the least (non-empty) LePUS3 statement that can be said about *JFC*. To discover if there are any inheritance hierarchies in *JFC* we apply the Partition to Hierarchies concretization operator to the term *AllClasses* in Chart 1, producing Chart 2.

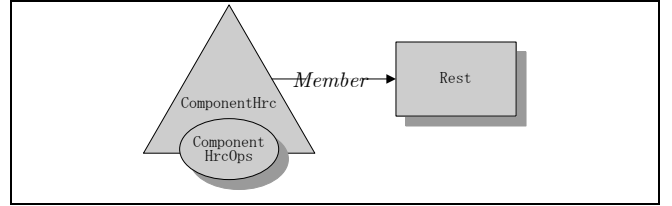


Chart 2 – Hierarchy *ComponentHrc* represents a class inheritance hierarchy

In Chart 2 the Design Navigator discovered that there exists an inheritance hierarchy in *JFC*, denoted *ComponentHrc*, and—

- each class in *ComponentHrc* has (at least) one field (‘Member’) of some class in the set of classes *Rest*.
- each class in *ComponentHrc* has a (possibly inherited) set of methods, each of which has a method signature in *ComponentHrcOps*. In other words, there exists a set of sets of dynamically-bound methods in *ComponentHrc*.

To discover more about the classes and methods in the *ComponentHrc* hierarchy we apply the Hierarchy Expansion concretization operator to *ComponentHrc*, which results in Chart 3. From this we only learn that class *Component* is at the root of this class hierarchy. To learn more about these classes and methods we must further concretize Chart 3.

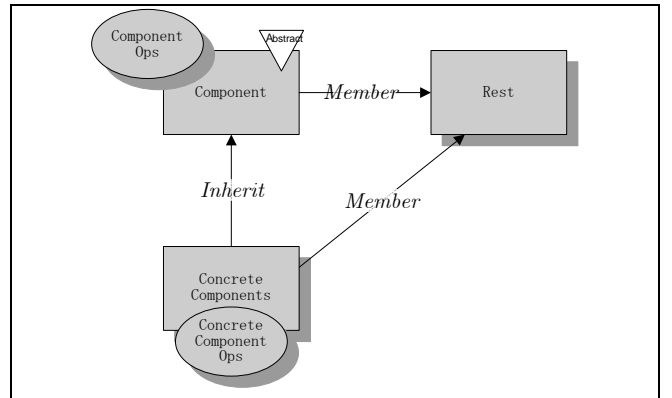


Chart 3 – Expanding *ComponentHrc* reveals its root class

3.2 Zooming in

To discover exactly which classes extend (‘inherit from’) class *Component* we apply the Enumeration concretization operator to the term *ConcreteComponents*, producing Chart 4.

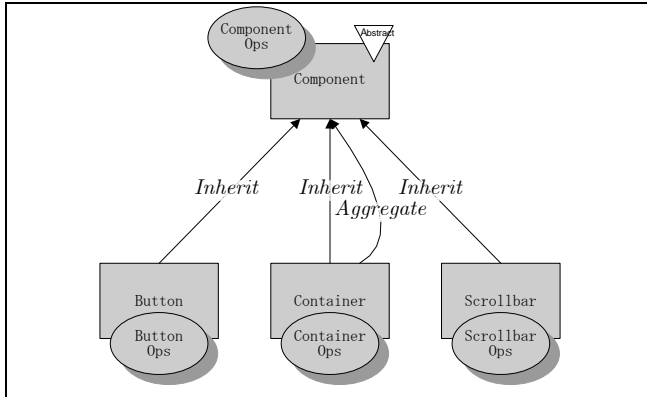


Chart 4 – Enumerating the subclasses of Component

Chart 4 shows that the ComponentHrc hierarchy consists of four classes, and that—

- the inheritance tree is not too deep, that is, Button, Container and Scrollbar directly inherit from Component
- Container ‘aggregates’ (has at least one array or collection of field holding instances of [subtypes of]) Component

Seeking to discover more about the methods of Button and Scrollbar, we apply the Partition concretization operator to ButtonOps and ScrollbarOps. The result appears in Chart 5.

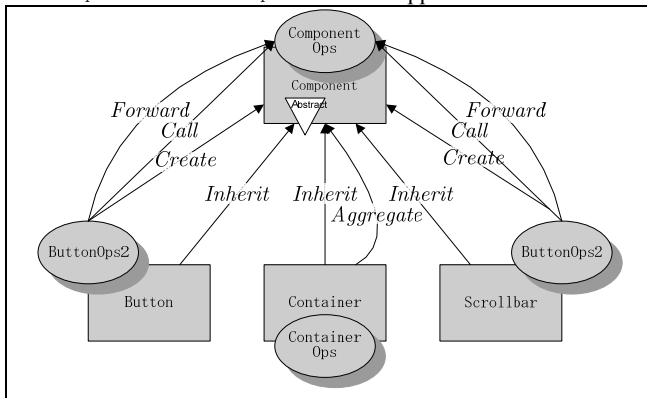


Chart 5 – Discovering some symmetry in the implementation of Button and Scrollbar

Chart 5 reveals an interesting symmetry in the implementation of Button and Scrollbar:

- each class defines a set of dynamically-bound methods sharing the same set of method signatures (ButtonOps2).
- each method in the sets mentioned above forwards the call to the respective method (with same signature) in Component
- each method in both sets creates at least one instance of class Component.

We continue the navigation by applying the Partition and Enumeration concretization operators to ComponentOps, ButtonOps2 and ContainerOps and the Aggregation abstraction operator to the resulting signatures. These actions lead to Chart 6.

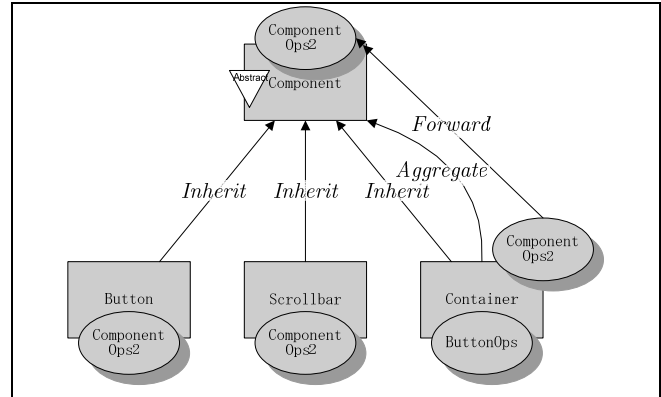


Chart 6 – Charting an instance of the Composite in JFC

From Chart 6 we discover that—

- classes Component, Button and Scrollbar define a set of dynamically-bound methods which share the same set of method signatures (ComponentOps2)
- each method in Container denoted by set of signatures ComponentOps2 forwards the call to a method in class Component

This chart contains enough evidence to convince us that classes Button, Component, Container and Scrollbar participate in an instance of the Composite [3] design pattern. In fact it completely matches the description of the Composite pattern in LePUS3 [2].

Design Navigation could proceed in this manner until all signatures in JFC have been concretized and all methods have been spelled out, leading to the Bottom Chart of JFC (Chart 7). This can be done in one step using the To Bottom concretization operator.

3.3 Ground View

The Bottom Chart of JFC (Chart 7) can be easily produced but it is impossible to read. While it serves as an indication of the complexity of O-O programs, it also suggests that potent abstraction mechanisms are essential for making sense or ‘navigating’ even in small programs. Any other modelling policy is bound to violate the Feynman-Tufte Principle [11]: “a visual display of data should be simple enough to fit on the side of a van”.

4. CONCLUSION

We have presented the Design Navigator, a tool that allows one to reverse engineer concise, precisely defined LePUS3 charts from arbitrarily large Java™ 1.4 programs in polynomial time. The tool generates increasingly more detailed charts of JFC, discovering in the process some of the building-blocks in the design of the selected classes which otherwise are difficult to detect.

Even for large programs, the Design Navigator produces legible charts within reasonable time (at worst in computational complexity squared in the number of classes and methods) when compared with the computational complexity of the most general form of the design mining (an NP-complete problem).



Chart 7 – Bottom chart. Is there a design here?

Table 1a – Abstraction (‘zoom-out’) operators



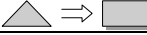





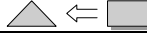



Aggregation	
Union	
Hierarchy to Set	
Collapse to Hierarchy	
Hierarchies Union	
To Top	\top_m
Elimination	

Table 1b – Concretization (‘zoom-in’) operators

Enumeration	
Partition	
Set to Hierarchy	
Hierarchy Expansion	
Partition to Hierarchies	
To Bottom	\perp_m
Introduction	

The example in §3 demonstrates the ability of the Design Navigator to mine the source code of programs, detect properties that are otherwise too hard to spot, and to chart them at the appropriate level of abstraction. It also illustrates how the stepwise application of a small set of navigation operators (Appendix) is enough to navigate in an unknown implementation and discover the overall structure of *JFC* (Charts 1,2), some of its classes (Charts 3,4), and correlations between methods (Charts 5,6). We also learned that certain classes of *JFC* participate in a instance of the Composite pattern (Chart 6). The fact that these results were accomplished without any prior knowledge of the design of *JFC* or the contents of its source code suggests that the Design Navigator is a useful tool which promotes the understanding of Java™ programs.

The Design Navigator is part of the Two-Tier Programming (TTP) Toolkit [13] which provides tool support for formal specification, visual modelling, and verification [10]. We believe that the principles of Design Navigation as demonstrated by version 0.5 of the Design Navigator can be useful to many programmers who wish to better understand unfamiliar Java™ programs and possibly discover discrepancies between the documentation and the actual design of these programs.

APPENDIX: ABSTRACTION/-CONCRETIZATION OPERATORS

The Design Navigator supports the set of ‘zoom-in’ (concretization) and ‘zoom-out’ (abstraction) operators listed in Table 1. For example, Chart 2 is the product of the application of the Partition to Hierarchies concretization operator to the term `AllClasses` in Chart 1. Note also that the set of operators is symmetric. For example, the application of the Hierarchies Union abstraction operator to Chart 2 produces Chart 1.

REFERENCES

- [1] Eden, A.H., Gasparis, E., and Nicholson, J. 2007. LePUS3 and Class-Z Reference Manual. Tech. Rep. CSM-474, ISSN 1744-8050, University of Essex. www.lepus.org.uk
- [2] Eden, A.H., Nicholson, J. and Gasparis, E. 2007. The ‘Gang of Four’ Companion: Formal specification of design patterns in LePUS3 and Class-Z. Tech. Rep. CSM-472, ISSN 1744-8050, University of Essex. www.lepus.org.uk
- [3] Gamma, E., Helm, R., Johnson, and R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Addison-Wesley.
- [4] Gasparis E., and Eden, A.H. 2007. Design Mining in LePUS3/Class-Z: Search Space and Abstraction/Concretization Operators. Tech. Rep. CSM-473, ISSN 1744-8050, University of Essex. www.lepus.org.uk
- [5] earth.google.com. Retrieved Nov. 2007
- [6] Guéhéneuc, Y. 2004. A reverse engineering tool for precise class diagrams. In H. Lutfiyya, J. Singer, and D. A. Stewart, (Eds.) *Proc. Conf. Centre For Advanced Studies on Collaborative Research* (Markham, Ontario, Canada). IBM Press, 28–41.
- [7] Kazman, R. and Carrière, S. J. 1999. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Egg*. 6(2) 107–138.
- [8] Korn, J., Chen, Y., and Koutsofios, E. 1999. Chava: Reverse Engineering and Tracking of Java Applets. In *Proc. Sixth Working Conf. on Reverse Engineering*. WCRE. Washington, DC, IEEE Computer Society, 314.
- [9] Murphy, G. C., Notkin, D., and Sullivan, K. J. 2001. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Trans. Soft. Eng.* 27(4) 364–380.
- [10] Nicholson, J., Eden, A.H., and Gasparis, E. 2007. Verification of LePUS3/Class-Z Specifications: Sample Models and Abstract Semantics for Java 1.4. Tech. Rep. CSM-471, ISSN 1744-8050, University of Essex. www.lepus.org.uk
- [11] Shermer, M. 2005. The Feynman-Tufte Principle: A visual display of data should be simple enough to fit on the side of a van. *Scientific American* (March 2005).
- [12] Storey, M., Best, C., Michaud, J., Rayside, D., Litoiu, M., and Musen, M. 2002. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems* (Minneapolis, Minnesota). New York: ACM, 520–521.
- [13] tpp.essex.ac.uk. Retrieved Nov. 2007

Part II: Description of the Presentation

1. OVERVIEW

Our presentation will focus on the Design Navigator. We will describe what it means to perform Design Navigation and use a small example adopted from package `java.awt` of the Java Foundation Classes 1.4 to demonstrate the results we get using the tool. The Design Navigator uses LePUS3 a formal specification and modelling language. While certain features of the language will need to be introduced, we will intentionally avoid presenting its rigours formal background.

2. APPROACH

The presentation will make use of both the projectors that will be available. On the first, static slides will be presented featuring support material and background information and on the second the live demonstration of the tool will run presenting actual real-time results of the Design Navigator being used with the selected classes from the JFC 1.4

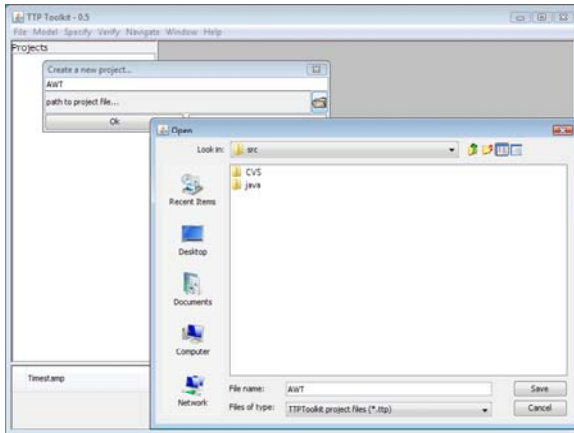
3. CONTENT

Slides only contain the slide title and the main points to be covered

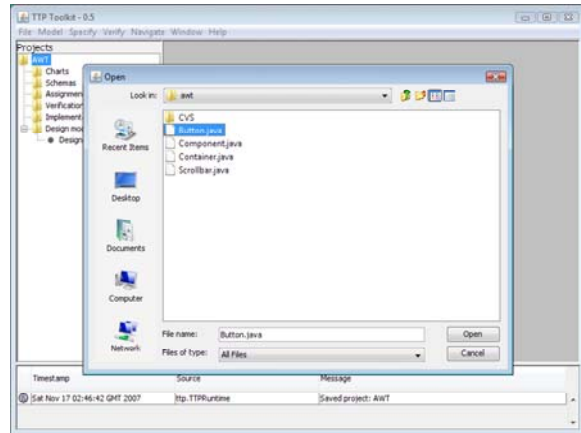
<i>Static Slides</i>	<i>Live Tool Presentation</i>
Slide 1: Reverse Engineering vs. Design Navigation <ul style="list-style-type: none"> User-guided Formal Programming-language independent 	<i>Design Navigator is idle</i>
Slide 2: LePUS3 <ul style="list-style-type: none"> Rigorous Scalable Object-Oriented Present Figure 3 (Part III) 	
Slide 3: Example Overview <ul style="list-style-type: none"> Adopted from JFC 1.4 Classes: Button, Container, Component, Scrollbar 	Preparation: <ul style="list-style-type: none"> Screenshots 1, 2, 3
Slide 4: Design Navigation <ul style="list-style-type: none"> Most abstract: 'Top Chart' Most concrete: 'Bottom Chart' Search space is a lattice (Figure 1) Design Navigation Operators (Table 1) 	Beginning the Navigation: <ul style="list-style-type: none"> Screenshot 4
Slide 5: Design Navigation 'Path' <ul style="list-style-type: none"> (Figure 2) 	
Slide 6: Design Navigator in Action <ul style="list-style-type: none"> Design Navigation Operators 'Design navigation path': (Figure 2) 	Birds eye-view <ul style="list-style-type: none"> Screenshots 5, 6 (that both show Chart 1) Screenshots 7, 8 (that shows Chart 2)
	Focusing on the Components Hierarchy <ul style="list-style-type: none"> Screenshot 9 (that shows Chart 3) Screenshot 10 (that shows Chart 4) Screenshots 11, 12 (that shows Chart 5) Screenshot 13 (that shows Chart 6)
	Ground View <ul style="list-style-type: none"> Screenshots 14, 15 (that both show Chart 7)

PART III: Screenshots

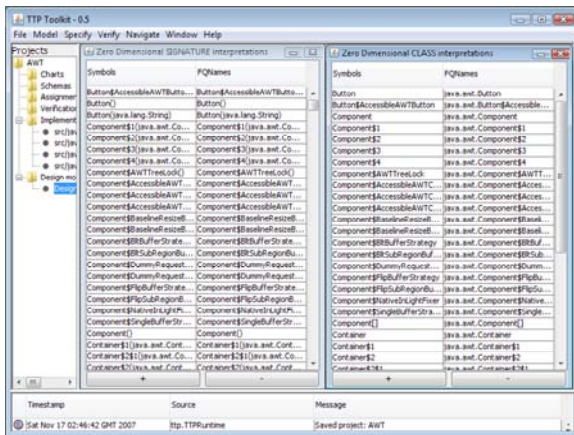
1. PREPARATION



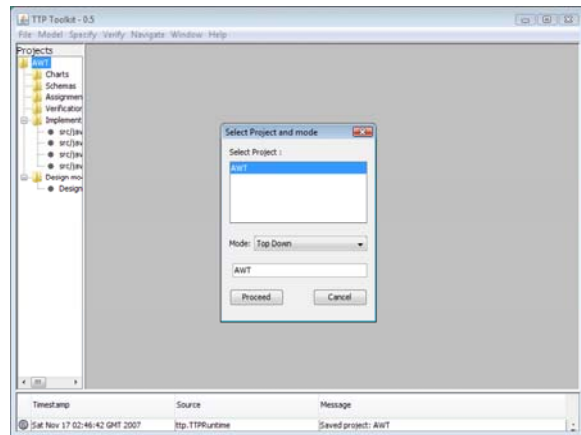
Screenshot 1 – Creating a new project



Screenshot 2 – Adding source ('implementation') files



Screenshot 3 – Classes and method signatures in the automatically generated database ('finite structure')



Screenshot 4 – Setting the starting point of Design Navigation to most abstract chart ('Top chart')

2. THE MODELLING LANGUAGE

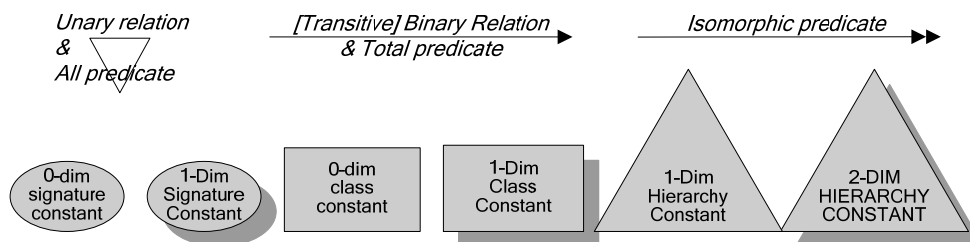
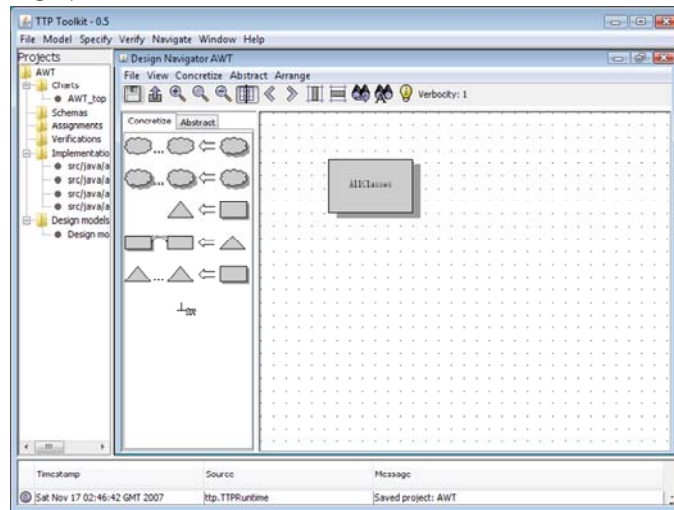
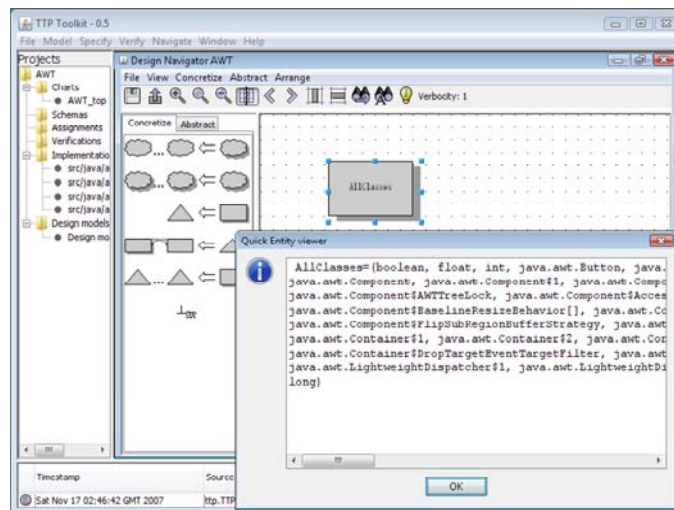


Figure 3 – LePUS3 Vocabulary

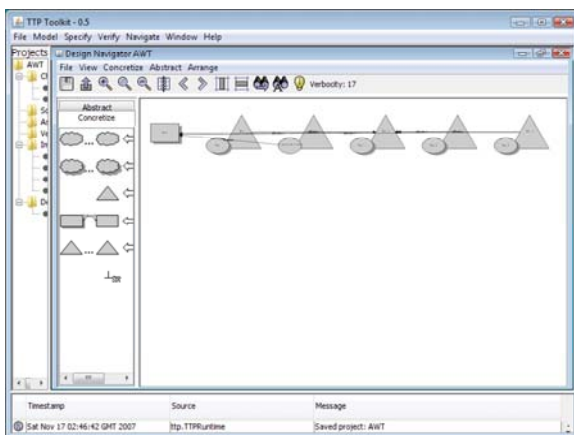
3. DESIGN NAVIGATION



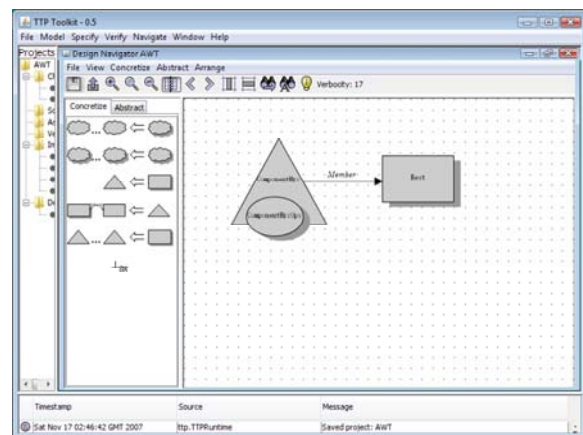
Screenshot 5 – Most abstract (‘Top Chart’) and the basic Graphical User Interface of the Design Navigator—Chart 1



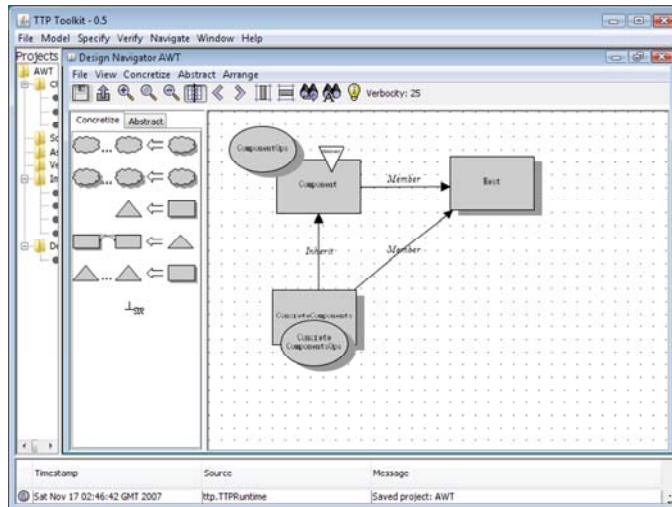
Screenshot 6 – Looking up what AllClasses stand for reveals a collection of classes from package java.awt



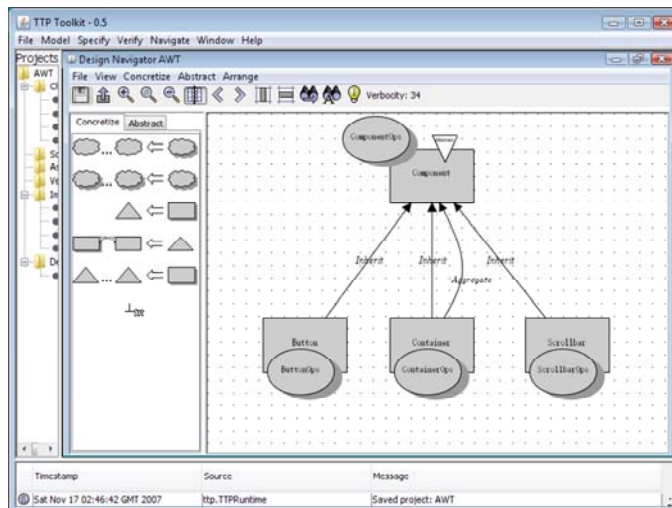
Screenshot 7 – Applying the Partition to Hierarchies operator reveals a series of inheritance hierarchies



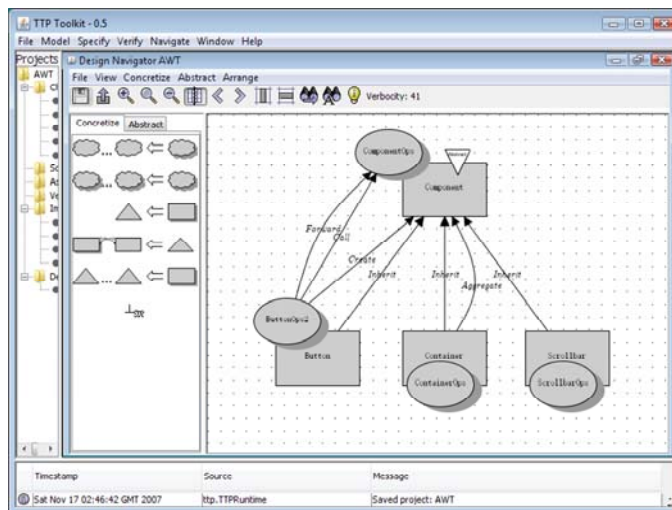
Screenshot 8 – The Elimination operator has been applied to unwanted hierarchies—Chart 2



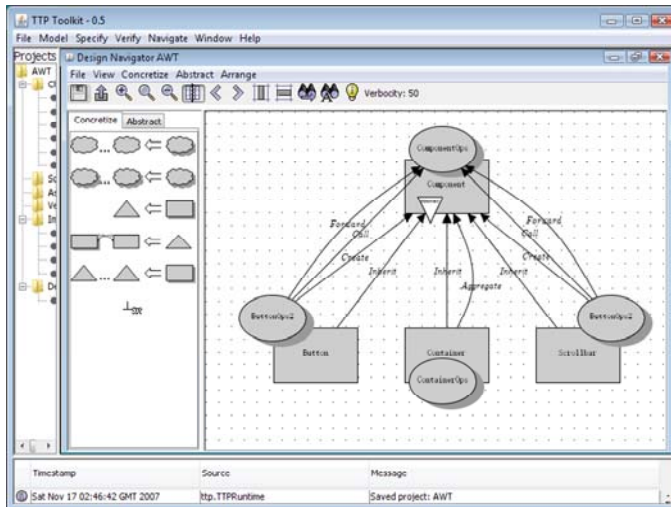
Screenshot 9 – Applying the Hierarchy expansion operator on ComponentHrc unveils Component as its root class—Chart 3



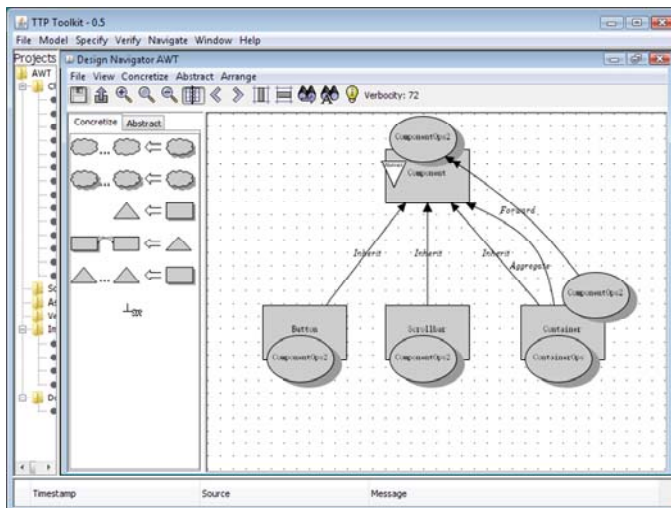
Screenshot 10 – All subclasses of Component are listed after using the Enumeration operator—Chart 4



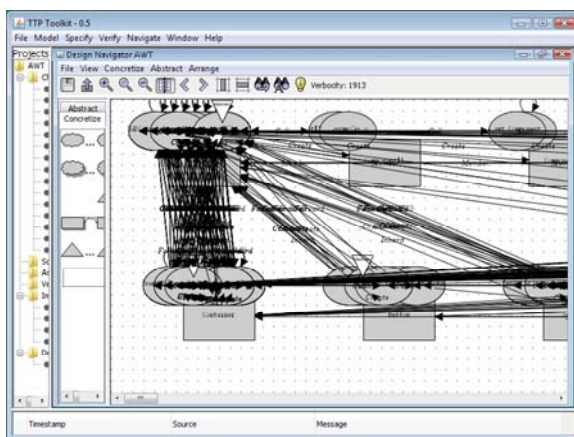
Screenshot 11 – Application of the Partition operator on ButtonOps results signature ButtonOps2



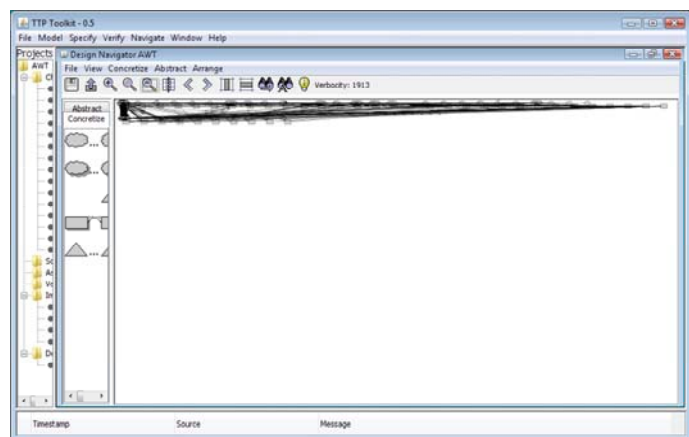
Screenshot 12 – Application of the Partition operator on signature ScrollbarOps leads again to signature ButtonOps2—Chart 5



Screenshot 13 – Further combinations of abstraction and concretization operators tease out a instance of the Composite pattern in JFC—Chart 6



Screenshot 14 – Most concrete ('Bottom Chart') is too verbose to be readable



Screenshot 15 – ('Bottom Chart') Pixel-scaling does not solve the problem—Chart 7