

Navigating Through the Design of Object-Oriented Programs

Epameinondas Gasparis ⁽¹⁾Jonathan Nicholson ⁽¹⁾Amnon H. Eden ^(1,2)Rick Kazman ^(3,4)[The Two Tier Programming Project](#)

⁽¹⁾ Dept. of Computing & Electronic Systems, University of Essex, United Kingdom ⁽²⁾ Centre for Inquiry, Amherst, NY, USA
⁽³⁾ Software Engineering Inst., Carnegie-Mellon University, USA ⁽⁴⁾ University of Hawaii, USA

Abstract. The Design Navigator is a tool for reverse-engineering object-oriented programs into formal charts of *any* level of abstraction. We show how the Design Navigator discovers abstract building-blocks in the design of programs and how it visualises them in terms of LePUS3, a formal Design Description Language. We demonstrate why reverse engineering programs into a formal modelling and specification language is not only possible in principle but also of practical benefit.

Keywords: Design recovery, program visualization

1. Introduction

In practice program documentation, if it exists at all, rarely delivers accurate information about programs, which leaves the source code as the only reliable source of information. The complexity of the implementation renders it inappropriate as a means to reason about a program's design. To remedy this, design recovery [1] is used to analyze programs, detect and bring to light interesting design abstractions. Existing tools (e.g. [2][3]) demonstrate that visual, language independent, semi-automatic methods for bottom-up and top-down design recovery indeed promote program understanding.

In common with these, the Design Navigator aids the user in discovering and visualizing the building blocks of O-O design. However, the Design Navigator is unique in satisfying the following additional desiderata:

- *Discovery of Design Abstractions:* Programs are semi-automatically mined for related higher-dimensional (sets of) classes, methods (e.g. dynamically-bound methods), and correlations amongst them. (The physical organisation of a program into files and directories is intentionally ignored).
- *Formal Specification:* Generated charts are sentences in LePUS3 [4], a formal Design Description Language [5], each of which stands for a formula in an axiomatized 1st-order theory in the classical predicate calculus.
- *Specification and Automated Verification:* Generated charts can be edited to create new specifications which can be verified at a click of a button.

2. Design Navigation

Having analyzed the source code, the Design Navigator operates in a user-guided, semi-automatic, step-wise process ("Design Navigation" [6][7]). Its operation is on a par with selecting a region on a geographical map and generating a new map of the same region at a lower or higher scale. Formally, Design Navigation is defined as a process of traversing the space of charts which spans the entire spectrum between the most abstract ("Top Chart") and the most concrete ("Bottom Chart") representations of a target program. At each step, the Design Navigator generates, in polynomial time, a chart that is either more abstract or more concrete than the previous, depending on the operator selected by the user (Table 1). Below we present a process of Design Navigation through package `java.util.logging` (from the Java Software Development Kit).

Table 1a – Abstraction Operators



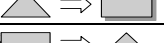

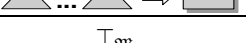
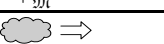




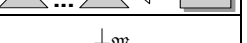
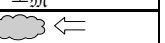
Aggregation	
Union	
Hierarchy to Set	
Collapse to Hierarchy	
Hierarchies Union	
To Top	\top_m
Elimination	

Table 1b – Concretization Operators

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

3. Case Study: java.util.logging

Initially, the source code is statically analyzed to extract information such class inheritance, class fields, method signatures, method calls, object creation etc. Then Design Navigation commences with Chart 1, the most abstract chart (“Top Chart”) modelling the entire package.

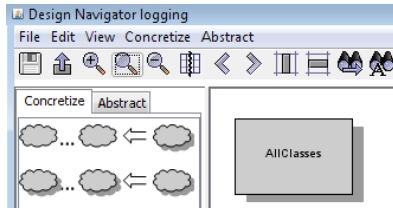


Chart 1 – AllClasses stands for the set of all classes

Concretizing Chart 1 leads the Design Navigator to discover the structure of the entire inheritance hierarchy of java.util.logging. Chart 2 formally specifies the following interesting facts:

- The package contains at least five inheritance class hierarchies, modelled as filled triangles (e.g. LoggerHrc, FormatterHrc).
- Each class in the LoggerHrc hierarchy holds one field of class Object and a field of a class in set Rest_1.
- All classes in java.util.logging inherit (possibly indirectly) from one class.

Further concretization of LoggerHrc leads to Chart 3 in which the Design Navigator discovered that class Logger has two sets of methods (modelled by superimposing signature sets LoggerOps_3 and LoggerOps_4 over Logger) such that—

- each method forwards the call to a method in LogManager with the same signature respectively

- each method produces (creates and returns) an instance of class Logger.

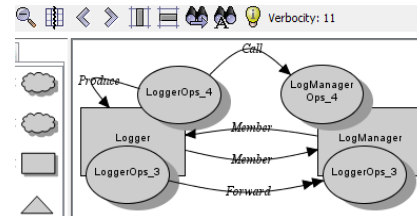


Chart 3 – Classes, methods, and correlations

References

- [1] T.J. Biggerstaff, “Design Recovery for Maintenance and Reuse,” *IEEE Computer*, vol. 22, 1989, pp. 36-49.
- [2] H.A. Müller, K. Klashinsky, “Rigi-A system for programming-in-the-large,” *Pro. 10th Int'l Conf. Software Engineering* (1988), pp. 80-86.
- [3] M. Storey et al., “SHrIMP Views: an Interactive Environment for Information Visualization and Navigation,” *CHI '02 Minneapolis, US* (2002), pp. 520-521.
- [4] A.H. Eden, E. Gasparis, J. Nicholson, “[LePUS3 and Class-Z Reference Manual](#),” CSM-474, ISSN 1744-8050, University of Essex, 2007.
- [5] E. Gasparis, J. Nicholson, A.H. Eden, “[LePUS3: An Object-Oriented Design Description Language](#),” *DIA-GRAMS 2008*, Herrsching, Germany: 2008.
- [6] E. Gasparis, A.H. Eden, “[Design Mining in LePUS3/Class-Z: Search Space and Abstraction/Concretization Operators](#),” CSM-473, ISSN 1744-8050, University of Essex, 2007.
- [7] E. Gasparis et al., “[The Design Navigator: Charting Java Programs](#),” *Companion, 30th Int'l Conf. Software Engineering* (2008), Leipzig, Germany, pp. 945-946.

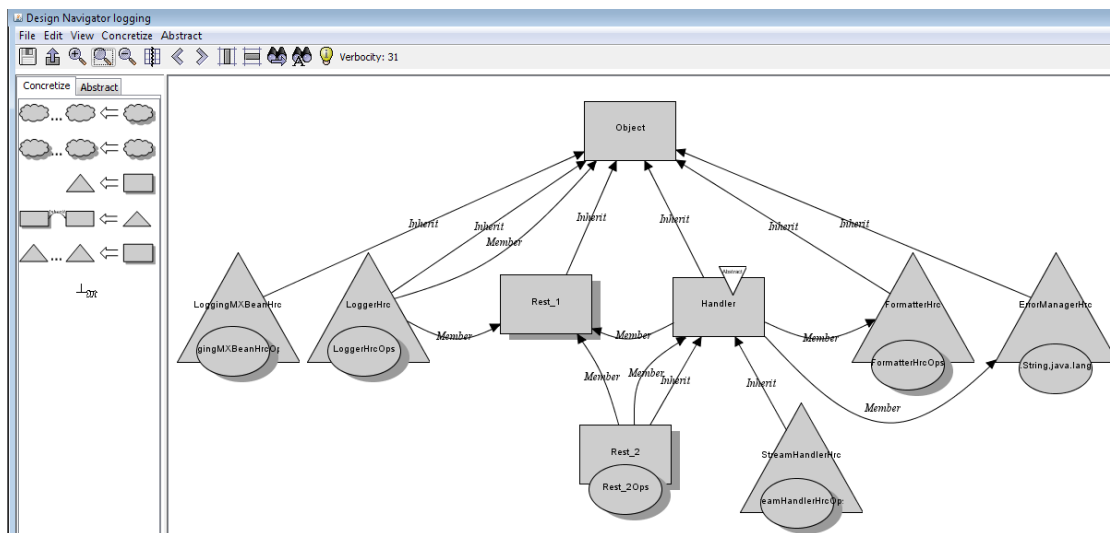
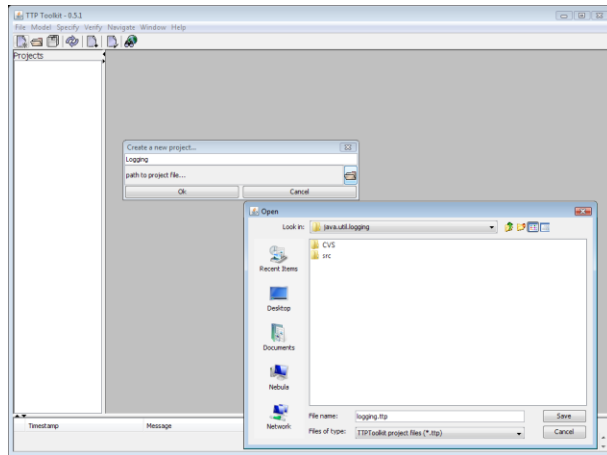
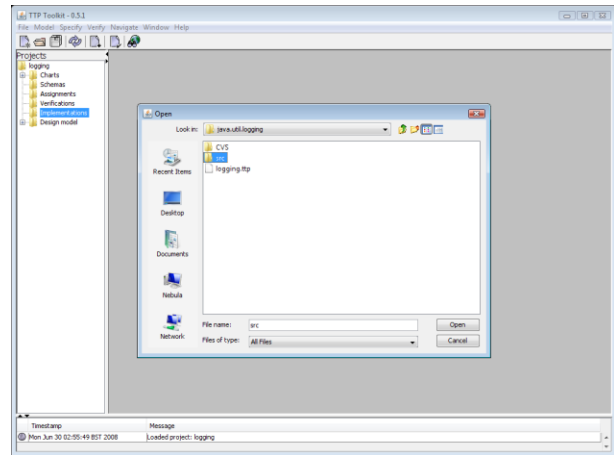


Chart 2 – Class hierarchies, class sets, dynamically-bound methods and correlations amongst them

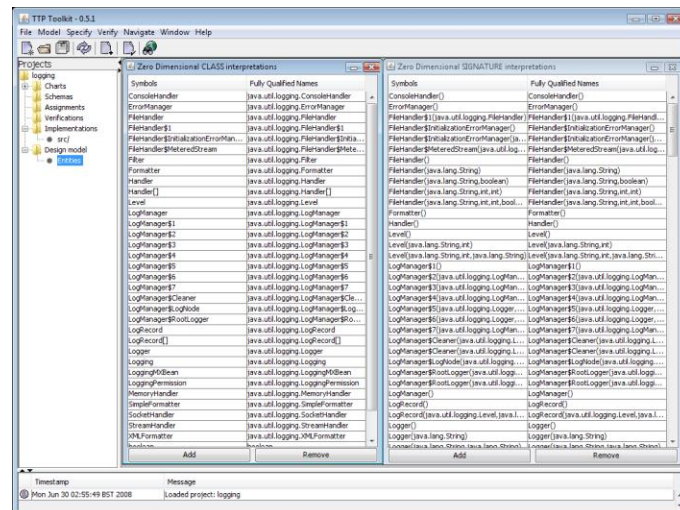
Appendix A: Static Analysis



Screenshot 1 – Creating a new project

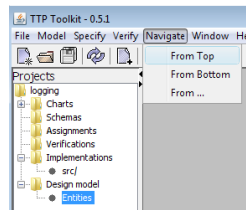


Screenshot 2 – Adding source (“implementation”) files

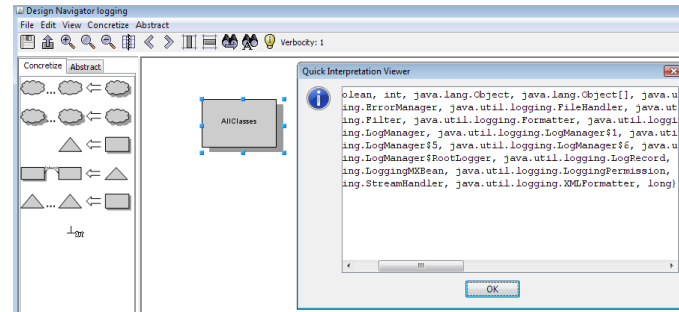


Screenshot 3 – Classes and method signatures in the database (“finite structure”) that is created by TTP-Toolkit’s static Java analyzer

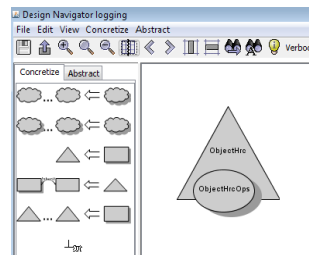
Appendix B: Design Navigation



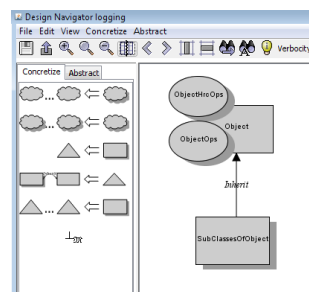
Screenshot 4 – Setting the starting point of Design Navigation to most abstract chart (“Top Chart”)



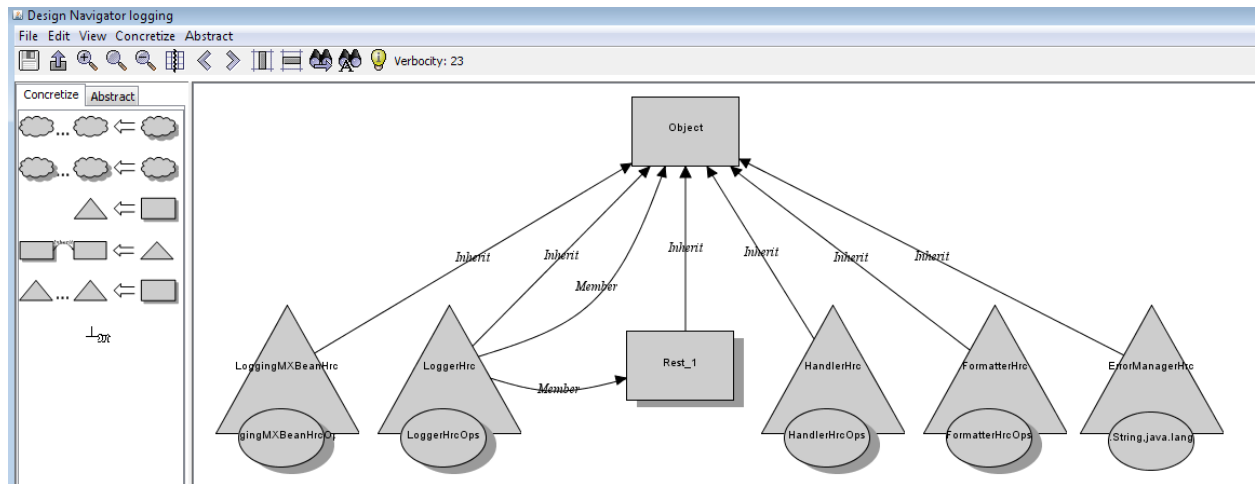
Screenshot 5 – AllClasses stands for all classes in package `java.util.logging`—Chart 1



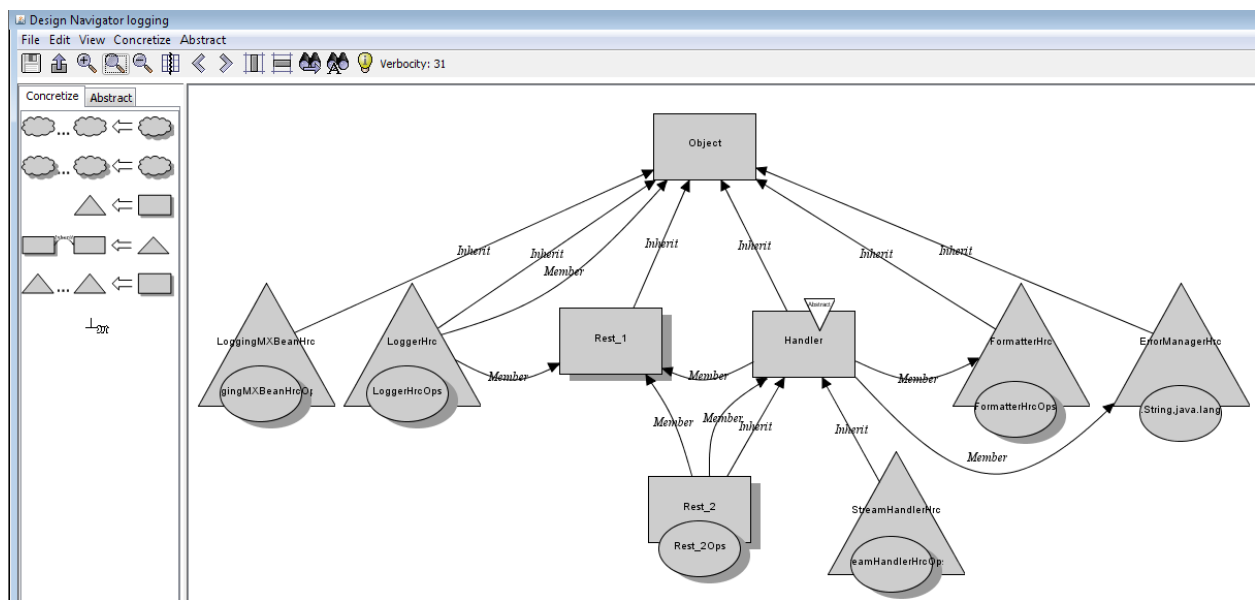
Screenshot 6 – Applying the Partition to Hierarchies operator reveals the `ObjectHrc` class inheritance hierarchy



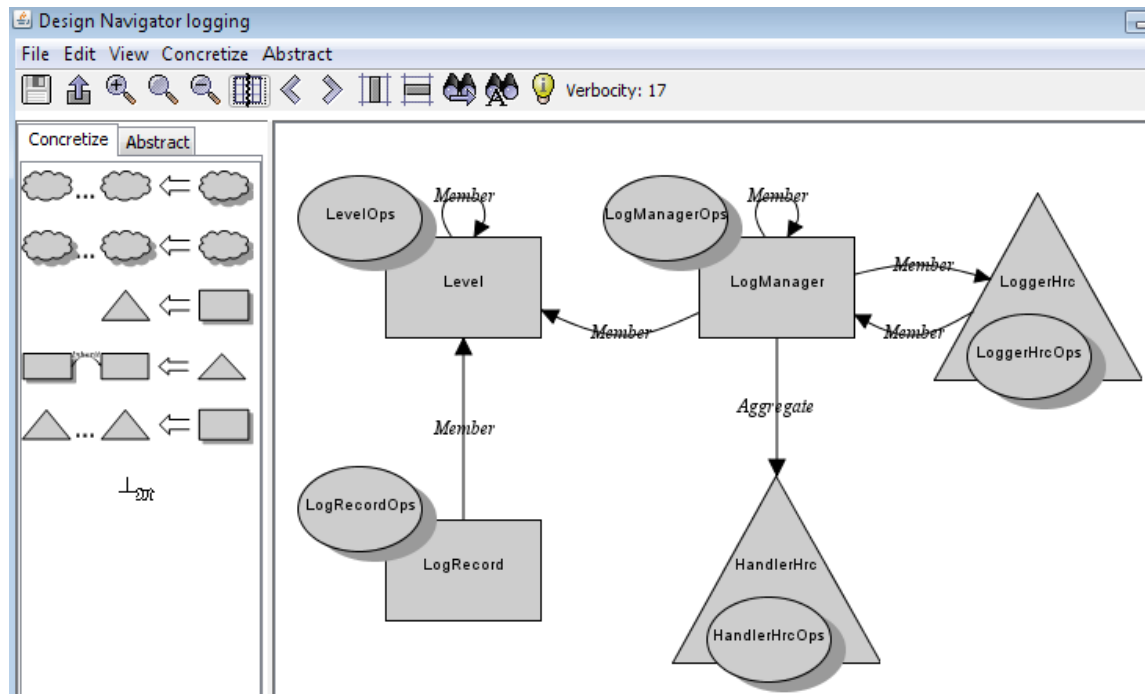
Screenshot 7 – Using the Hierarchy Expansion operator reveals the set of subclasses of class `Object`



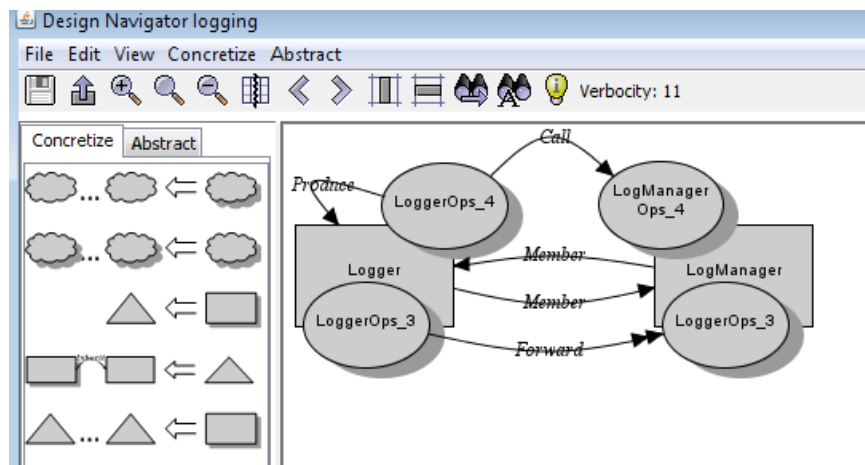
Screenshot 8 – Applying the Partition to Hierarchies operator reveals more class inheritance hierarchies



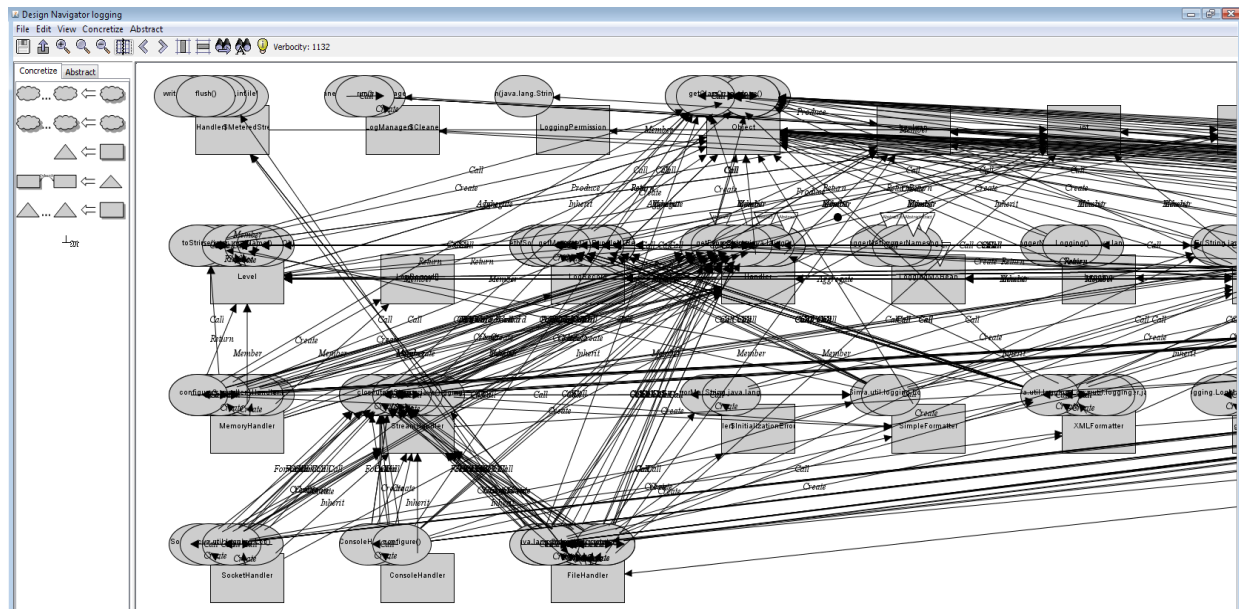
Screenshot 9 – Concretizing HandlerHrc reveals another set of classes (Rest_2) and another hierarchy (StreamHandlerHrc)—Chart 2



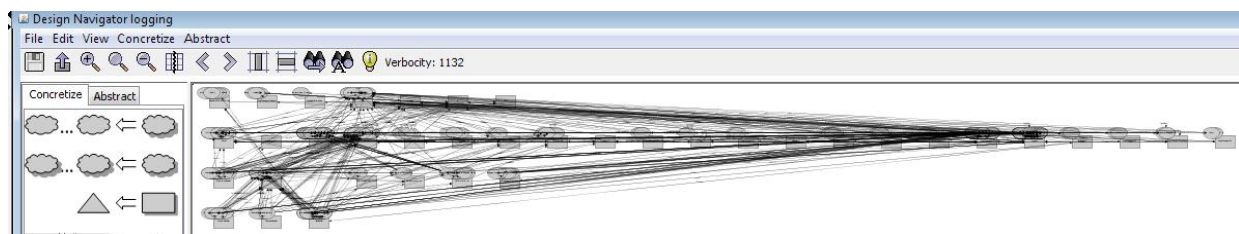
Screenshot 10 – Further application of the Elimination operator and concretization operators allows us to focus (“zoom-in”) on classes and hierarchies of more interest



Screenshot 11 – Even further concretization steps provide more fine grained results—Chart 3



Screenshot 12 – Is there a design?



Screenshot 13 – Clearly pixel scaling is not a solution