Supporting OO Design Heuristics

Neville Churcher Sarah Frater Cong Phuoc Huynh Warwick Irwin

Department of Computer Science and Software Engineering
University of Canterbury,
Private Bag 4800 Christchurch, New Zealand
Email: neville.churcher@canterbury.ac.nz

Abstract

Heuristics have long been recognised as a way to tackle problems which are intractable because of their size or complexity. They have been used in software engineering for purposes such as identification of favourable regions of design space. Some heuristics in software engineering can be expressed in high-level abstract terms while others are more specific. Heuristics tend to be couched in terms which make them hard to automate. In our previous work we have developed robust semantic models of software in order to support the computation of metrics and the construction of visualisations which allow their interpretation by developers. In this paper, we show how software engineering heuristics can be supported by a semantic model infrastructure. Examples from our current work illustrate the value of combining the rigour of a semantic model with the human mental models associated with heuristics.

Keywords: Heuristics, OO Design, Metrics, Visualisation, Software Engineering, Static Analysis, Semantic Model.

1. Introduction

Software design and development involves a range of practices with varying levels of formality: examples include formal methods, coding styles, design patterns and test-driven development. The common goal is the production of high quality software.

However, quality is a sufficiently ephemeral concept that it can not be measured directly. In order to measure and understand quality, it is necessary to relate it to measurable quantities. The field of software metrics [16, 22, 21, for example] deals with the identification of meaningful quantitative measures of specific properties of software.

The most widely-used quality model is hierarchical [33]. Successive levels become less abstract until the leaves are sufficiently precisely defined that specific metrics may be employed to measure them. For example, comment density ($\frac{CLOC}{NCLOC}$) measures *Self-descriptiveness* which is one aspect of *Reusability* which is one element of *Product transition* which is one component of *Quality*.

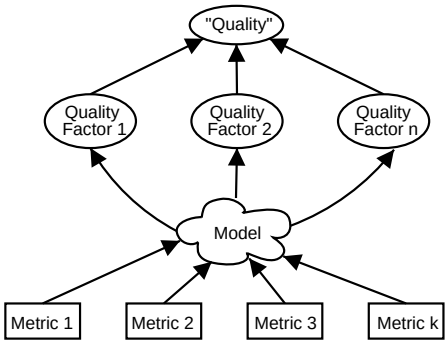

Figure 1. Modelling software quality

A hierarchical structure is, however, an artificial restriction and in general a richer model is needed to relate in a practical way the qualitative aspects of quality to directly measurable quantities, as indicated in Figure 1. Developing sufficiently robust models remains a major challenge.

Heuristics enable a 'softer' model to be constructed in order to obtain a more holistic, and subjective, view of quality. This potentially places a greater burden on the developers who must interpret this view since it consists of potentially conflicting indicators with varying degrees of precision and relevance.

Heuristics may occur as individual pieces of developers' oral tradition (e.g. "Avoid overloading operators") or may be presented as a suite covering multiple aspects of software development. The heuristics proposed by Riel are a good example of the latter [41].

Heuristics occur in many contexts: examples include the application of design patterns [20] and the selection of refactoring techniques [18].

In the current work, we are not primarily concerned with the relevance or validity of individual heuristics: our focus

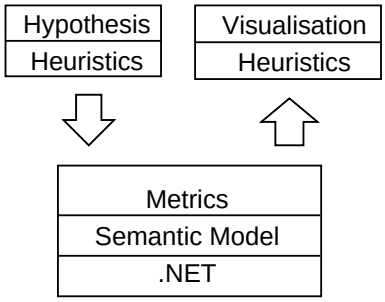

Figure 2. Supporting heuristics

is on their evaluation[1] and interpretation. Our work is intended to provide the basis for an exploratory framework in which heuristics may be postulated, explored and managed.

Two significant contributions are reported in this paper. Firstly, we show how heuristics can be refined sufficiently to enable precise formulation and quantitative evaluation.

We also present a common semantic model for OO software: the model is independent of source languages yet allows mappings to particular language constructs where required. We illustrate its rôle in supporting the formulation and evaluation of heuristics.

Our approach is based on the architecture outlined in Figure 2 and consists of a number of steps.

- A set of heuristics is identified according to their relevance to design hypotheses, activities and requirements. The selected heuristics will in general not be orthogonal—and may even have conflicts.
- Individual heuristics are parameterised so that they can be expressed in terms of precisely-defined metrics.
- The metrics employed are themselves precisely defined in terms of elements of a semantic model.
- The model is populated by extracting information from individual source files, using the .NET CLR infrastructure to maximise language independence.
- Values for the required metrics are then computed and expressed in a suitable parameterised form.
- Finally, the resulting information is presented to the user in forms appropriate to the corresponding heuristics.

The remainder of the paper is structured as follows. In the next section we outline the rationale for heuristics and discuss their relevance to software engineering. In §3 we describe our semantic model-based approach. The process of mapping heuristics to quantities which can be evaluated is discussed in §4 and some results are presented in §5. Finally, we present our conclusions and outline our ongoing work.

[1] We use *evaluation* to mean obtaining sufficient information to determine the extent to which a heuristic is satisfied

2. Design heuristics

In this section, we focus on the rôle of (sets of) design heuristics in modern OO software engineering.

Design is, in general, a difficult task because it involves finding compromises between conflicting pressures—cost and reliability, for example—and many of these pressures ultimately arise from human concerns, with all that implies in complexity, diversity and changeability. Designers must find ways to provide specific capabilities required by stakeholders, while attaining sufficient quality in emergent properties such as usability, efficiency, and flexibility.

Software designers aim to satisfy the expectations of stakeholders by meeting functional and non-functional requirements. But in order to make this possible, they must first address the needs of the software developers themselves. Keeping the complexity of the design in check is foremost among these.

Object-orientation (OO) allows software to be structured in a way that helps to manage complexity and change. However, as software reuse practitioners (and others) have discovered, realising the benefits of OO is not straightforward. Competence with the mechanisms of OO—classes and objects, attributes and methods, inheritance and polymorphism—is far from sufficient to ensure successful designs.

In this regard, OO is no different from other programming paradigms. Over some decades, software engineers have developed a number of approaches and principles to elevate design considerations above programming language mechanisms. Concepts such as abstraction, separation of concerns [13], information hiding [40], cohesion and coupling [43], and Design By Contract™ [35] provide guidance to designers.

On top of these general principles, the OO design community has developed a rich doctrine of principles, aphorisms and practices to inform designers. Examples include: the *open/closed principle* [34, 32], the *Liskov substitution principle* [28, 31], the *acyclic dependencies principle* [34, 30], *Favour composition over inheritance* [26], the *Law of Demeter* [27], *You ain't gonna need it* (YAGNI) [2] and *Tell, don't ask* [3].

This lore is supplemented by design patterns and idioms, which provide prototypical solutions to common problems. The original set of 23 Gang of Four design patterns [20] has been broadened by more design patterns [12, for example], architectural patterns [6], analysis patterns [17], antipatterns [5], and others.

[2] `http://c2.com/cgi/wiki?YouArentGonnaNeedIt`

Some authors have collated parts of this complex web of concepts into sets of heuristics[3]. Johnson and Foote (1988) provide an early example, which describes design maxims intended to promote reuse. Riel (1996) documents 61 'golden rules' for OO design, while Fowler and Beck describe 22 code smells [18].

The choice of the term 'smells' is instructive. It evokes a subjective, subtle process of perceiving something about a design. Beck and Fowler note that code smells do not lend themselves to automatic quantification [18]. The designer must form an impression of the net product of many factors at work in the design. This requires judgement and insight beyond the capabilities of simple automata.

From the architect Christopher Alexander [2, 1], the design patterns community has borrowed an illuminating way of viewing the designer's task: it is to find a balance point between all of the forces acting on the design. Design patterns are examples of such balance points, but they are not simple prescriptive solutions: the forces are identified and the factors that might influence the balance are described. A notable characteristic of design patterns is that they often break rules. For example, the Composite pattern advocates the use of methods that are overridden to do nothing, contrary to a common maxim, expressed by Riel's heuristic 5.7 (RH5.17) as "It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing." However, the Gang of Four chose to break this rule deliberately, in their words preferring transparency over safety. Many similar examples of conflicting forces can be found.

Some conflicts are so pervasive that they apply to nearly all design situations. Separation of concerns, for example, encourages decoupling portions of a design, while RH2.9, "Keep related data and behaviour in one place" often suggests the opposite.

Even within an organised set of heuristics, conflicts occur. RH5.4 says "Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better", while RH5.5 adds the qualification that "In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six".

Heuristics are a valuable tool for identifying design forces (whether conflicting or not) and evaluating design quality, but their application is not straightforward for many reasons, such as:

- **Lack of consensus on which heuristics should be adopted.** Some conflicting heuristics usefully illuminate matters of concern to the designer. Other conflicts, however, reflect differing design philosophies, and a particular designer is likely to be interested only in one side of the debate. Many of the tenets arising from software reuse culture, for example, are in opposition to more recent refactoring and agile methods approaches. The open/closed principle, for example, encourages anticipation of future needs by making the design open for extension (reusable), but without requiring modification of existing code; refactoring culture discourages anticipation of future needs and prefers modifying existing code when necessary. This cultural difference might show up in unexpected ways, such as a stronger preference for small methods in the reuse culture, so that methods constitute small overridable units.

- **Nebulous definitions.** RH2.8, for example, says "A class should capture one and only one key abstraction", but rigorously specifying the meaning of "key abstraction" is problematic. Similarly, RH3.6 "Model the real world whenever possible", is only as firm as our grip on reality.

- **Subjectivity and calibration.** Code smells require the designer to judge when some intangible threshold has been crossed. The "large class smell", "lazy class smell" and "long method smell" are obvious examples where different standards might apply. The relative importance of conflicting heuristics is also dependent on the value system of the designer. If breaking up a large class produces a lazy class, is the result better?

- **Interpretation in different contexts.** Many heuristics are expressed abstractly, in order to apply to any OO design. It may be necessary, however, to adapt a heuristic to local conditions. For example, when deciding if an inheritance hierarchy is too deep, should the root class be counted in programming languages that enforce a single root? Or, in an organisation that has adopted a refactoring approach to software development, how much emphasis should be placed on a heuristic motivated by software reuse, such as RH5.7 "All base classes should be abstract classes"?

- **Diverse levels of abstraction.** Some heuristics can be interpreted at different levels. RH2.1, "All data should be hidden within its class", might be viewed as a syntactic restriction—make attributes private—or as a semantic one, which might also discourage the use of getters. A "long method smell" could be detected at a lexical level by counting lines of code, at a syntactic level by counting statements and expressions, at a language semantic level by counting method invocations, collaborators, etc, or at a problem-domain semantic level by gauging the conceptual size of the method.

[3] We use the term "heuristics" inclusively, to describe maxims, concepts, principles, warnings, etc.

- **Information overload.** Heuristics are intended to help software engineers manage the complexity of software, but injudicious application of heuristics could compound the problem.
- **Acquiring relevant data and relating it to heuristics.** Many heuristics require substantial data gathering. RH4.4 "Minimize fanout in a class", and RH4.6 "Most of the methods defined on a class should be using most of the data members most of the time" are examples. Additionally, the correspondence between available information and heuristics is not always clear.

These issues, and the inherent fuzziness of heuristics, make automated support of heuristics difficult. In consequence, designers usually must gauge the quality of their products without assistance from tools. The designer builds a mental model of the software, and evaluates it (smells it?), according to a subjective, and perhaps even subconscious, process that is likely to be informed by heuristics, but may explicitly apply few.

Our research investigates ways to help designers with the task of understanding, evaluating and improving their products. While we view the art of design—and the judgement of how to apply heuristics—as beyond the reach of current technology, we argue that tools can provide valuable information to assist the designer with these judgements.

3. Semantic models

A key element of our approach is the use of a semantic model that exposes software structure. In this paper we extend our previous work on applications of semantic models in software metrics, software visualisation and collaborative software engineering [24, 25, 11, 39].

Object-oriented semantic models represent the semantic concepts of OO software including entities such as classes, interfaces, methods and fields together with relationships between them such as inheritance, implementation, containment, overloading and invocation. Semantic models are the result of static analysis, which is concerned with examining source code and possibly other software artifacts, such as UML diagrams, without executing the software.

In this paper we describe how semantic models can support the calculation of design heuristics. From earlier work, we have a Java [25] and a .NET semantic model [38]. Our Java semantic model captures the type system of the Java language, while our .NET model captures the type system defined by the .NET CLI standard. In effect, the .NET model gives us a way of representing the common semantic underpinnings for a wide range of programming languages.

The .NET Common Language Infrastructure allows different languages to be compiled to the same .NET Common Intermediate Language (CIL) and run on the same Virtual Execution System regardless of their individual syntaxes [37].

The existence of a language-independent semantic model confers significant advantages. It allows us to define heuristics and metrics against the common model, giving us language-independent metrics and a basis for cross-language comparisons, without loss of rigour. In addition, it makes adding new languages to the model straightforward.

It is necessary to address the problem of mapping the common semantic model to programming language-specific semantics, in order to communicate our findings back to a software engineer working in a specific programming language. We achieve this by explicitly modelling programming language-specific semantics, and mapping the relationships between the specific and common model, as shown in Figure 3. In this figure, each mapping is a bi-directional association between semantic concepts in .NET and Java. Java and C# are sufficiently similar that most mappings are 1:1. For example, a .NET class is mapped exactly onto a Java class and vice versa. Meanwhile, a Property in .NET is a combination of a setter and a getter method and is therefore mapped to two Java methods.

Our mapping approach addresses differences in the semantic models, such as the absence of a pointer type from Java. Similarly, .NET has no corresponding concepts to the scope defined by a Java source file.

With most OO languages, heuristics and metrics calculated against one model can usually be translated into the semantics of another model, as illustrated in Figure 4. For example, the Depth of Inheritance Tree (DIT) of a class in J# increases by 1 when compiled to .NET CIL. This is because every class in J# inherits from the java.lang.Object in a library assembly, which in turn inherits the root System.Object class of the .NET framework.

The latest version of .NET supports generic types and our common semantic model captures these. We have extended our earlier Java model to include the generic type features introduced in Java 1.5 and mapped them to .NET generics. Our architecture allows us to add new language-specific models and mappings as needed.

4. Evaluating heuristics

As outlined in §1, our approach is to identify heuristics of interest, and find ways of mapping them to metrics derived from our semantic model. Figure 5 shows an expanded view of the architecture summarised in Figure 2 and illustrates how this fits the pipeline-based models we have described elsewhere [23].

Heuristic statements, such as those given by Riel, are typically couched in high-level, natural language, abstract terms: they express concepts which are readily understood

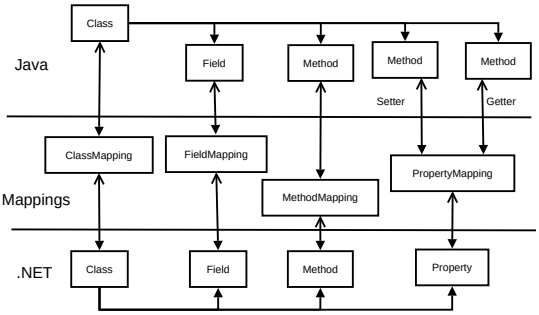

Figure 3. Mapping Java semantics to .NET.

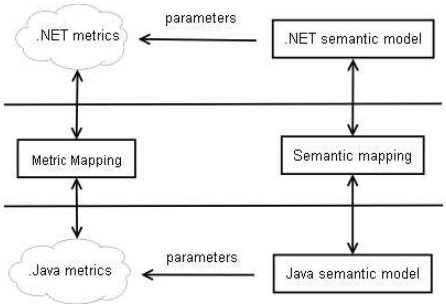

Figure 4. Mapping metrics

by developers. Unlike metrics, heuristics generally do not have precise values. They are often expressed as commands or slogans (e.g. RH2.9 “Keep data and behaviour in one place”) or in a ‘more/less is better’ form (e.g. RH3.3 “Beware of classes with many accessor methods in their public interfaces”).

Assessing whether a particular heuristic applies or is being followed is inevitably subjective. Some ‘fuzziness’ is required in order to present data to the user in a form which is consistent with the original heuristic statements.

In order to introduce objective, quantifiable elements we first derive parameterised heuristics.

This involves the identification of the relevant parameter(s) for each heuristic. For example, the occurrence of literal values (e.g. in contexts such as “inheritance should be no deeper than 6”) suggests that a constant (e.g. max_depth) might be appropriate. However, a variable whose value can be adjusted to serve as a threshold is likely to be more useful in practice. Similarly, phrases such as “no deeper than” indicate that a depth variable is involved. Parameters relating to a range of properties such as visibility, and relationships, such as the set of methods invoked, are used in the mapping process.

The parameterised heuristics are amenable to direct mapping to quantities represented directly in the common semantic model and to (measured or derived) software metrics obtained from it. This is achieved by combining metrics fil-

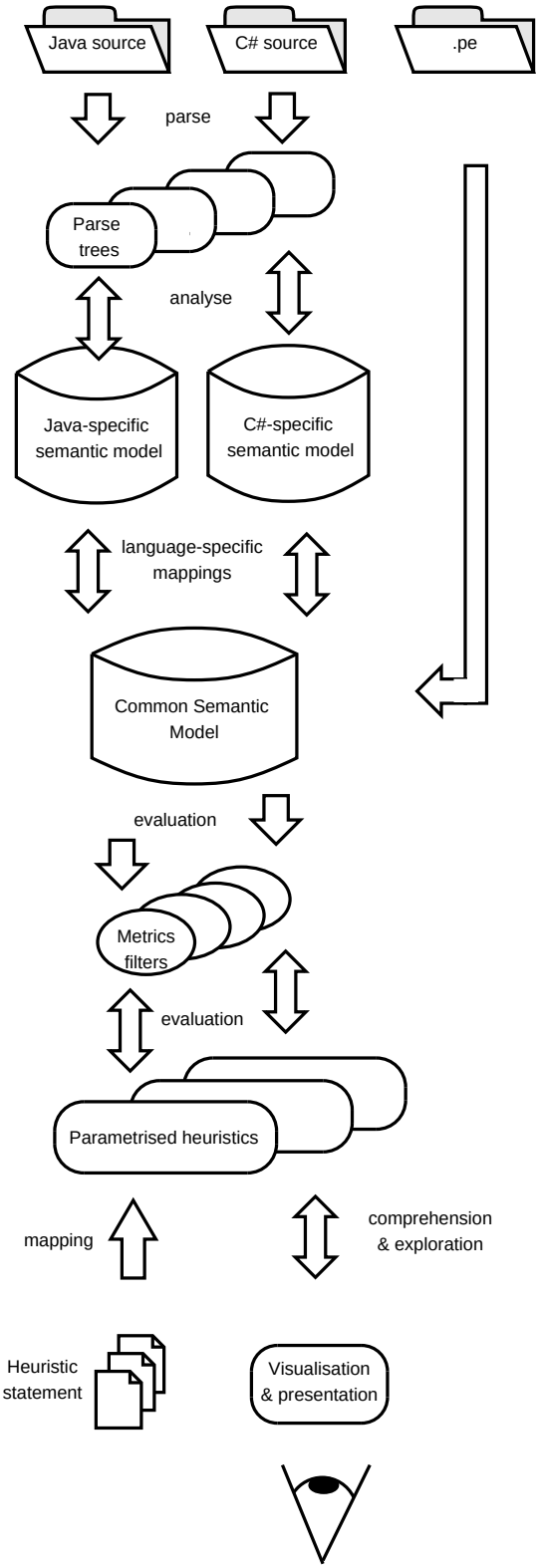

Figure 5. Heuristics in the information pipeline

ters as indicated in Figure 5.

Evaluating parameterised heuristics leads to data in various forms. For example, RH2.1 might lead to a collection containing an element for each class, together with information on the numbers of data members with public, protected, or private access mode. Alternatively, it might be presented as a collection whose elements correspond to data members. The next step is to develop ways to present this information back to the user in a form which is consistent with the original heuristic statements: it is not sufficient simply to present the metric values. This step is discussed further in §5.

The metrics and other quantities required in the evaluation of parameterised heuristics are obtained from our common semantic model. The model may be imagined as a 'symbol table on steroids' combined with a powerful cross-referencing tool: it allows queries such as *what are the methods called by the private members of my superclass* to be evaluated. Our parsing and semantic analysis tools allow language-specific models to be generated where source code is available. Additionally, it is possible to obtain data directly from the .NET virtual machine language level. Mappings from the common model back to specific languages allow data to be presented back to users in terms of the corresponding concepts and syntax of the languages used.

4.1. Example

One of the simplest of Riel's heuristics, and the first to appear in his book, is RH2.1 "All data should be hidden within its class". However, even this apparently simple heuristic requires deeper consideration before it can be applied.

While encapsulation is universally accepted as an integral part of OO, opinion is somewhat divided on the appropriate boundary. In statically typed OO languages (such as C++ and Java) the class is the boundary for encapsulation. However, in dynamically typed languages (such as SmallTalk or even JavaScript) encapsulation occurs at the object level. Consequently RH2.1, which was initially presented in a C++ context, might be regarded as somewhat controversial in some circles. In the following discussion, we assume that Riel's approach is acceptable.

Superficially, the quantified form of RH2.1 would be Boolean-valued: either all data is hidden within its class or it is not. However, the spirit of heuristics might admit softer responses such as 'some' or 'most', allowing some leeway for allowing conflicting forces to be accommodated.

In order to be useful to developers, we are likely to want richer forms of presentation of the information in order to identify refactoring opportunities. Possibilities include:

- The names of the attributes which violate RH2.1 and the classes to which they belong.
- A wider view e.g. highlighted elements on a UML class diagram.
- A histogram showing the distribution of violations of RH2.1, both in absolute terms and normalised by the number of attributes.
- A class-based view showing the proportion of attributes which violate RH2.1 and how many are inherited.

It is important not only that correct values are obtained for relevant quantities but also that the computation is demonstrably valid and repeatable. This involves, amongst other things, complete details of assumptions made and conventions used, such as the rules used to count occurrences of various quantities.

In order to specify counting rules for RH2.1 we must decide precisely what is data and what is not, by formulating and answering questions such as these:

- Will inherited data items be included?
- Does "protected" count as "hidden"?
- Do inner classes count as data?
- Do arrays and collections count as a single data item or are they weighted by their size?

The interface to this heuristic (i.e. the parameterised form) thus has Boolean parameters to control choices such as whether inherited data is included.

Software metrics suites often include some measure of attribute visibility, such as the attribute hiding factor (AHF) metric in the MOOD suite [14], and these and others are available from the semantic model.

It is not difficult to find classes which exhibit RH2.1 violations. Examples include Java's java.awt.Point and java.awt.Rectangle classes. Attributes such as `x` and `width`, which are anticipated to be accessed frequently by clients, have been made public in order to avoid the performance overhead of accessor methods. Riel gives the counter-argument that this introduces considerable risk if the representation should subsequently change (e.g. to polar coordinates) but risk management is more generally, and more properly, the responsibility of developer teams.

4.2. Discussion

Some heuristics cannot be automated at all (e.g. RH3.6 "Model the real world whenever possible") while others (e.g. RH2.1 in §4.1) are relatively straightforward. Between these extremes lie many heuristics that may not be directly measurable, but for which it is possible to measure aspects of software that might indicate whether the

```
<Heuristics program="JST.exe">
 <Heuristic name="NumberOfMessageSends">
   <Param name="max_messages" value="-1"/>
   <Param name="inc_self" value="True"/>
   <Param name="inc_ancestors" value="True"/>
   <Param name="inc_external" value="True"/>
   <Data name="symtab.Decl" value="8" />
...
```

Figure 6. Sample data

heuristics is being followed. RH2.8 "A class should capture one, and only one, key abstraction", for example, is hard to measure directly, as key abstractions are difficult to identify. We can, however, measure indirect quantities, such as LCOM [9], which might indicate the heuristic is not being followed. Our mapping process allows exploration of the consequences of such modelling decisions.

Bär and Ciupke [4] considered several sets of design heuristics from the point of view of automating their evaluation. Our approach allows a wider range of heuristics to be handled.

By basing our metrics on semantic information (corresponding to the type system of the .NET CIL), we can measure quantities that are closer to the level at which heuristics are usually expressed, and which are rigorously defined. This is an improvement over other approaches [29, for example] which attempt to map code smells to common metrics sets such as that of Chidamber & Kemerer (1994).

In our current implementation, parameterised heuristics are coded as sibling classes. The corresponding values are associated with the output data for analysis and visualisation as indicated in the sample shown in Figure 6. However, we are gradually adding support for interactive and exploratory development of parameterised heuristics.

Some of the quantities required may be available directly by querying the semantic model. Others may be computed indirectly as functions of model features. Our approach includes a framework for developing visitors, using the corresponding design pattern [20], which traverse the semantic model in order to harvest the data required. The visitor pattern enables clear separation of the heuristics and metrics from the core model representation.

Figure 8 illustrates a simplified version of part of the framework. The ModelVisitor class plays the abstract visitor rôle in the pattern and contains methods (only 3 of which are shown) for visiting each the semantic model elements which play the concrete element rôles. An example is the *visitMethodDecl()* method, shown in Figure 7 which visits MethodDecl elements. The complete model is described elsewhere [38].

Each of the fundamental metrics, such as DIT, NOAt, and NOMS, is implemented by a class which plays a concrete visitor rôle in the pattern. They traverse the semantic model and extract the raw data required for computation by overriding the appropriate ModelVisitor methods. The data is then used by the classes which play the client rôles; these are the heuristics such as NumberOfMessageSends, NumberOfAttributes and DepthOfInheritance. The relationships between metrics and heuristics shown in Figure 8 are 1:1 but, in general, these may be more complex.

```
public virtual void visitMethodDecl(MethodDecl md) {
    visitOperationDecl(md);
    // visit type parameters
    foreach (TypeParameter tp in md.GetTypeParameters()) {
        tp.Accept(this);
    }
    // navigate through constructed methods
    foreach (ConstructedMethod cm
            in md.GetConstructedMethods()) {
        cm.Accept(this);
    }
}
```

Figure 7. Using the visitor pattern

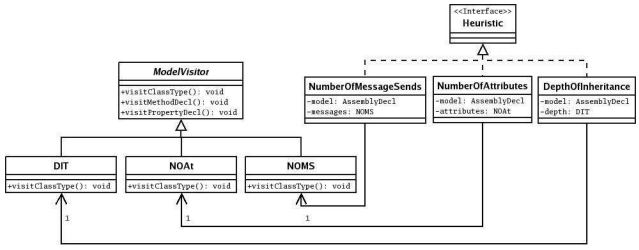

Figure 8. Heuristic evaluation using the visitor pattern

5. Visualisation

In this section we discuss the presentation of heuristics information to the user and give some examples from our current work.

Visualisation of heuristics provides many challenges. Heuristics are likely to be studied both individually and in comparison with others. Heuristics have a "soft" somewhat ambient feeling in terms of a user's mental model and the visualisation techniques used need to reflect this. It is important to convey a (possibly somewhat fuzzy) holistic impression of a heuristic rather than focus on the individual metrics which may contribute.

When dealing with heuristics, it is usually not appropriate to present information in "hard" forms such as tables,

graphs with scales or spreadsheets. We might expect that the usual 7 ± 2 [36] limit will also apply to the presentation of heuristics information and hence limit the number of heuristics which can be handled concurrently: this is further motivation for the use of visualisation techniques.

There is considerable opportunity for applying ambient information visualisation techniques which have been applied successfully in domains such as financial data [15, 19, 42, for example].

We are working towards integrating unobtrusive ambient visualisations into software development tools such as editors and diagrammers. For example, text in an editor could be underlined—in much the same way as the grammar checker in Microsoft Word—to indicate violation of heuristics. For example, a newly-added method invocation might be underlined, indicating that a violation of the *acyclic dependencies principle* has occurred. Similarly, colour could be used in a UML class diagram to highlight components which violate heuristics.

Figure 9 shows a 3D visualisation including the number of message sends and number of collaborators for classes in an application (actually our semantic model). It is presented as a 3D VRML [7] world displayed in Microsoft Internet Explorer using the Cortona plugin[4].

The metaphor employed here involves a number of peaked mountain-like shapes. The large one at the left of the figure represents the entire application while the others correspond to individual packages. Each is made up of slices corresponding to individual classes: their order is determined by sorting them according to the total number of message sends they contain.

Height corresponds to the number of message sends for the classes. The higher, transparent, surface indicates the total number of sends while the solid shape beneath indicates the number which are sent to the class itself or to its ancestors.

The solid, flatter 'hummocks' at the right of the figure also correspond to the same packages and represent the data for total messages sends normalised by the number of collaborators for each class. The shapes may be slid around the surface, allowing the user to compare and explore freely.

Figure 10 shows another approach, featuring complex glyphs inspired by Chernoff faces [8]—a well known technique for representing multivariate data in 2D. The underlying metaphor is the display of specimens as found in museums. The user is free to explore a 3D VRML bugscape: individual specimens may be moved around to assist with comparison. Each 'bug' corresponds to an element (a Java class in this case) and its characteristics (eye size, body shape, leg length, ...) indicate values of quantities in a parameterised heuristic. The unconventional metaphor adds

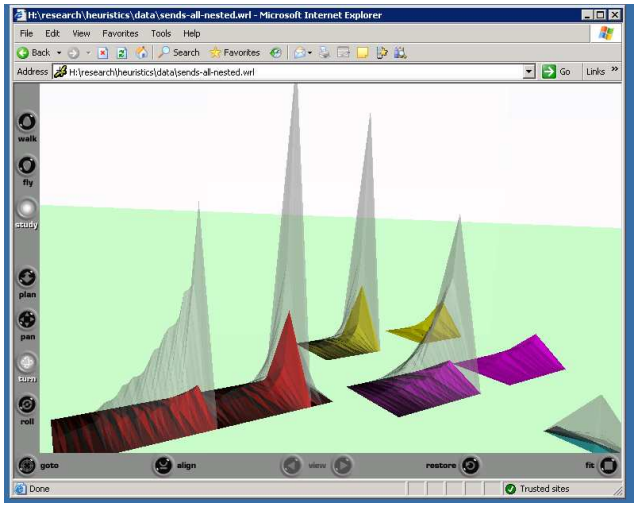

Figure 9. Visualising parameterised heuristics

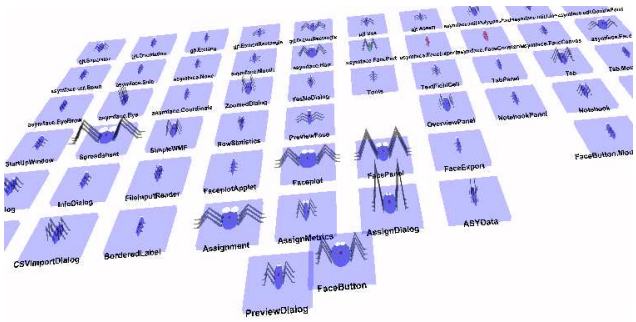

Figure 10. Exploring a VRML bugscape

to the 'fuzziness' while allowing ready comparison and inspection.

Exploratory analysis of multi-dimensional data is supported by a applications such as ggobi[5]. Figure 11 shows results for inheritance depth (DIT), number of methods (WMC) and number of collaborators (COL) metrics for the classes in one package of our application. A parallel coordinates view is shown together with a scatterplot matrix in which each pair of metrics appears separately. Techniques such as brushing enable individual data points to be compared in the various views. In previous work [10] we have described the value of such tools in *ad hoc* visualisation design. User interpretation of the fuzzy heuristics visualisations steers deeper and more specific exploration via such tools as ggobi.

[4] `http://www/parallelgraphics.com`

[5] `http://www.ggobi.org`

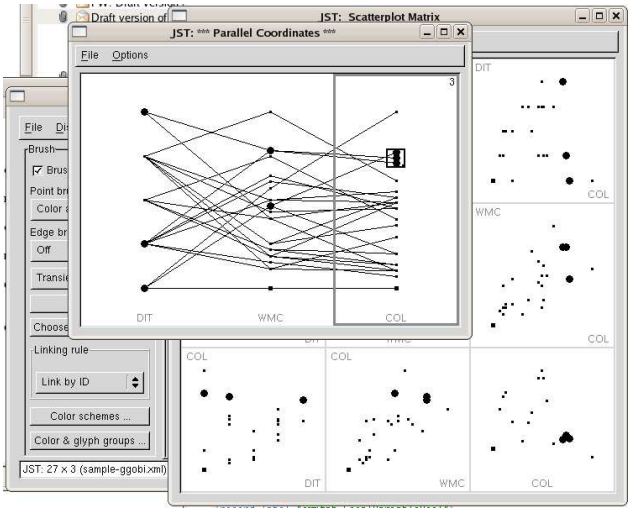

Figure 11. Exploring heuristics with ggobi

6. Conclusions and further work

Heuristics provide a link between sets of abstract design principles and quantitative software metrics. They are an important part of software design and are becoming more widely used in contexts such as refactoring.

Heuristics are informally expressed and their evaluation involves parameterisation in order to provide a bidirectional mapping to specific metrics. Effective visualisation of heuristics includes quantitative, qualitative and ambient aspects.

In this paper, we have presented a semantic model which includes the common semantics of OO languages together with mappings to and from language-specific constructs. We have shown how this common semantic model supports the evaluation of metrics required for the parameterised representation and evaluation of heuristics. Bridging the gap in this way offers considerable benefits. It not only provides a way to obtain reliable values in heuristic evaluation but also enables sets of language-independent heuristics to be managed and integrated with other software engineering tools.

We have given examples of visualisation and data exploration to illustrate the special nature of techniques appropriate to heuristic presentation interpretation.

In the present work we have not been concerned with the validity or applicability of particular heuristics but rather to establish a framework in which heuristics can be proposed, expressed and evaluated.

Our work will enable subsequent evaluation of the applications of heuristics through both focused studies, such as usability trials for particular visualisations, and longitudinal elements, such as investigations to determine which heuristics are most suitable for particular design activities. User trials will take place once our enhanced user interface is complete and longitudinal studies will initially involve studies of student software engineering projects.

We are encouraged by the results to date and are now able to deploy heuristics alongside our range of metrics and visualisation techniques. Our ongoing work includes the development of interactive tools for managing and customising heuristics. Ultimately, we anticipate that our approach will enable software engineers to propose, evaluate and test their own heuristics alongside those in the wider body of knowledge.

References

[1] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[3] D. T. Andy Hunt. The art of enbugging. *IEEE Software*, 20(1):10–11, 2003.

[4] H. Bär and O. Ciupke. Exploiting design heuristics for automatic problem detection. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 73–74, London, UK, 1998. Springer-Verlag.

[5] W. Brown, R. Malveau, H. M. III, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*. J. Wiley & Sons, 1998.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. J. Wiley & Sons, 2001.

[7] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference manual*. Addison-Wesley, 1997.

[8] H. Chernoff. The use of faces to represent points in k-dimensional space graphically. *Journal of the American Statistical Association*, 68(342):361–368, 1973.

[9] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[10] N. Churcher and W. Irwin. Informing the design of pipeline-based software visualisations. In S.-H. Hong, editor, *APVIS2005: Asia-Pacifi c Symposium on Information Visualisation*, volume 45 of *Conferences in Research and Practice in Information Technology*, pages 59–68, Sydney, Australia, Jan. 2005. ACS.

[11] C. Cook and N. Churcher. Constructing real-time collaborative software engineering tools using caise, an architecture for supporting tool development. In V. Estivill-Castro and G. Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference (ACSC2006)*, volume 48 of *CRPIT*, pages 267–276, Hobart, Australia, 2006. ACS.

[12] J. Coplien and D. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[13] E. W. Dijkstra. On the role of scientifi c thought. In *Selected Writings on Computing: A Personal Perspective'*, pages 60–66. Springer-Verlag, 1982.

[14] F. B. e Abreu. MOOD—metrics for object-oriented design. In M. Wilkes, editor, *Addendum to Proc. OOPSLA'94*, Portland, OR, Oct. 1994.

[15] P. Eades and X. Shen. Moneytree: Ambient information visualization of financial data. In M. Piccardi, T. Hintz, S. He, M. L. Huang, and D. D. Feng, editors, *2003 Pan-Sydney Area Workshop on Visual Information Processing (VIP2003)*, volume 36 of *CRPIT*, pages 15–18, Sydney, Australia, 2004. ACS.

[16] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press, 2nd edition, 1997.

[17] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[19] X. Fu and D. Li. Haptic shoes: Representing information by vibration. In S.-H. Hong, editor, *Asia Pacific Symposium on Information Visualisation (APVIS2005)*, volume 45 of *CRPIT*, pages 47–50, Sydney, Australia, 2005. ACS.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[22] D. Ince and M. Shepperd. *The Derivation and Validation of Software Metrics*. Oxford University Press, 1992.

[23] W. Irwin and N. Churcher. XML in the visualisation pipeline. In D. D. Feng, J. Jin, P. Eades, and H. Yan, editors, *Visualisation 2001*, volume 11 of *Conferences in Research and Practice in Information Technology*, pages 59–68, Sydney, Australia, Apr. 2002. ACS. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

[24] W. Irwin and N. Churcher. Object oriented metrics: Precision tools and configurable visualisations. In *METRICS2003: 9th IEEE Symposium on Software Metrics*, pages 112–123, Sydney, Australia, Sept. 2003. IEEE Press.

[25] W. Irwin, C. Cook, and N. Churcher. Parsing and semantic modelling for software engineering applications. In P. Strooper, editor, *Australian Software Engineering Conference*, pages 180–189, Brisbane, Australia, 29 March – 1 April 2005. IEEE Press.

[26] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[27] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, 1989.

[28] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), 1988.

[29] M. Mäntylä. *Bad Smells in Software—a Taxonomy and an Empirical Study*. Ph.d. thesis, Helsinki University of Technology, 2003.

[30] R. C. Martin. Granularity. *C++ Report*, 8(10):57–62, 1996.

[31] R. C. Martin. The liskov substitution principle. *C++ Report*, 8(3):14, 16–17, 20–23, 1996.

[32] R. C. Martin. The open closed principle. *C++ Report*, 8(1):37–43, 1996.

[33] J. McCall, P. Richards, and G. Walters. Factors in software quality. Technical Report (RADC)-TR-77-369, Vols. 1–3, Rome Air Development Center, United States Air Force, Hanscom AFB, MA, Nov. 1977. Available as AD-A049-014, AD-A049-015 and AD-A049-055 from: NTIS, Springfield, VA.

[34] B. Meyer. *Object-oriented software construction*. Prentice-Hall international series in computer science. Prentice Hall, New York, 1988.

[35] B. Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.

[36] G. A. Miller. The magical number 7 plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1957.

[37] J. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, 2004.

[38] B. Neate. An object-oriented semantic model for .net. Project Report HONS06/05, Department of Computer Science and Software Engineering, University of Canterbury, 2005.

[39] B. Neate, W. Irwin, and N. Churcher. Coderank: A new family of software metrics. In J. Han and M. Staples, editors, *ASWEC2006: Australian Software Engineering Conference*, pages 369–378, Sydney, Apr. 2006. IEEE.

[40] D. Parnas. On criteria to be used in decomposing systems into modules. *Commun. ACM*, 14(1):221–227, 1972.

[41] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[42] X. Shen and P. Eades. Using moneycolor to represent financial data. In S.-H. Hong, editor, *Asia Pacific Symposium on Information Visualisation (APVIS2005)*, volume 45 of *CRPIT*, pages 125–129, Sydney, Australia, 2005. ACS.

[43] E. Yourdon and L. Constantine. *Structured design : fundamentals of a discipline of computer program and systems design*. Prentice Hall, Englewood Cliffs, N.J., 1979.