The Automatic Detection of Design Flaws

Douglas Kirk, Marc Roper, Murray Wood
Department of Computer and Information Sciences
University of Strathclyde
Glasgow, UK
{doug,marc,murray}@cis.strath.ac.uk

ABSTRACT

Encapsulation and data hiding are central tenets of the object oriented paradigm. Deciding what data and behaviour to form into a class and where to draw the line between its public and private details can make the difference between a class that is an understandable, flexible and reusable abstraction and one which is not. This decision is a difficult one and may easily result in poor encapsulation which can then have serious implications for a number of system qualities. Even if a design is well encapsulated it can decay over time if the maintainers do not respect the initial encapsulation.

It is often hard to identify such encapsulation problems within large software systems until they cause a maintenance problem (which is usually too late) and attempting to perform such analysis manually can also be tedious and error prone. This paper describes the architecture of a tool which automatically detects violations of encapsulation. A tool based approach can provide a thorough and regular health check on the encapsulation of a system and help to prevent problems from occurring or identify those that must be addressed. The tool has been developed as a plug-in for the Eclipse IDE and uses heuristic descriptions of common design flaws such as data and god classes to detect areas of poor encapsulation within a system.

The technique has been evaluated in a controlled study on two large open source systems which compared its results to similar work by Marinescu, who employs a metrics-based approach to detecting such features. The study provides some valuable insights into the strengths and weaknesses of the two approaches. The paper also raises a number of questions regarding the precise definition of data and god classes, and concludes with a number of proposals to improve the approach further.

1. INTRODUCTION

Identifying a good decomposition of a system into classes when undertaking an object-oriented design is a significant challenge. As Meyer [19] states, “Finding classes is the central decision in building an object-oriented software system; as in any creative discipline, making such decisions right takes talent and experience, not to mention luck.” Even when appropriate classes have been identified there are hard decisions to be made when it comes to the attribution of data and behaviour (techniques such as CRC cards have been devised to support this activity but do not scale well). The software engineer has to try to find a decomposition which maximises desirable design qualities while minimising any negative effects.

A common manifestation of this failure to find good decompositions is poor encapsulation – where associated data and behaviour fail to be co-located – and this may take many forms (maybe a class has not been created when it should have been, or vice-versa). Two of the common encapsulation problems that can arise as a consequence of this decomposition process are data classes and god classes. Typically, these two problems occur together – data classes are lacking in functionality that has typically been sucked into an over-complicated, domineering god class instead. The boundary between the god and the data class is misplaced, the data class is poorly encapsulated, making its data accessible to the god which typically, retrieves the data, manipulates it in a way that should be done in the data class, and then stores the result back in the data class. Recent empirical work suggests that the presence of data and god classes in a system can make it harder to understand and modify [6],[9] particularly for those who have experience or education in good decomposition techniques (novice participants tended to find the centralised rather than the delegated approach easier to deal with).

It is important that design flaws such as this do not go undetected in a system as they inevitably cause problems with understanding and modification. Ideally these problems should be identified and refactored into better balanced and well-encapsulated classes as soon as possible. However, detecting the existence of these problems in large systems is extremely difficult – the scale of the software makes manual detection ineffective, unwieldy and impractical, and hence the idea of attempting to detect these design flaws automatically is very appealing.

This paper describes an approach for the automatic detection of god and data classes in existing systems. It proposes a technique which uses heuristics based on the information present in the static relationships within a program’s source code to infer the presence of god and data classes. The accuracy of the technique is evaluated by comparison with a metrics based approach and the strengths and weaknesses of the two approaches compared. It is demonstrated that our heuristic-based approach still has room for improvements but is generally more accurate and flexible than a metrics-based strategy.

2. RELATED WORK

2.1 Data Classes and God Classes

Data classes are described by Fowler [11] as “dumb data holders” which are being manipulated by the rest of the system. In the extreme case they have methods for getting and setting the data and nothing else. Data classes are a problem as they typically provide poor encapsulation of their data and lack significant

functionality. God classes are often a corollary to data classes and frequently represent an attempt to capture some central control mechanism. Riel [22] describes a god class as one that, "performs most of the work, leaving minor details to a collection of trivial classes" - these trivial classes being data classes.

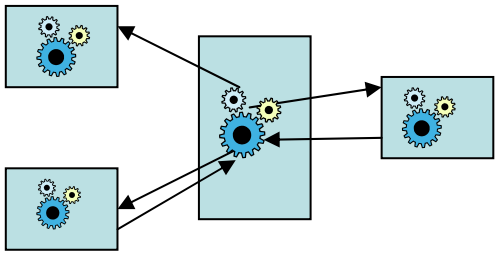

Figure 1. Ideal distribution of data and behaviour

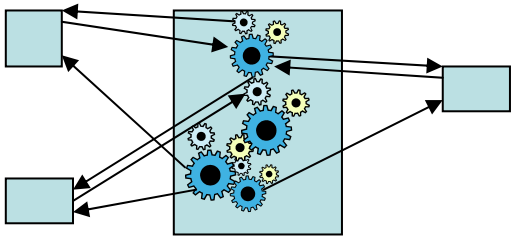

Figure 2. Impact of god and data classes

This relationship between god and data classes captures a situation where the behaviour within a system has become misplaced. Instead of being evenly distributed amongst the classes (as illustrated in Figure 1 where computations are performed locally as a consequence of requests) and closely tied to the data, it has somehow gravitated from the data classes into the god class, making the god class dominant and unwieldy, and the data classes passive and almost useless (see Figure 2, where the god class pulls the data values from the data classes, uses them to perform some computation, and pushes the results back). This is clearly an undesirable situation within a system and impacts upon a range of attributes of the design, particularly its maintainability, testability, and understandability.

2.2 Automated Detection Strategies

Many approaches have been suggested for the automatic detection of design flaws in software. They include the use of logic meta programming [24] [8], metric analysis [17], [23], invariant analysis [14] and the use of software change histories [7] [21].

This paper describes an approach using AST analysis to detect design problems. Van Emden and Moonen [25] describe an approach, similar to ours, in which they use an AST analysis to detect problems. Their approach has been implemented in a tool called JCosmo which can detect opportunities for the use of polymorphism within a design. They search software for the use of the instanceof operator, type casts or the use of switch statements and in each case provide a re-factoring which replaces the existing structure with a polymorphic equivalent.

Van Emden and Moonen's approach has been evaluated successfully in a small case study. In part their success may have been due to the small well defined nature of the entities that they sought within the AST. In fact, without being dismissive of this work, it would be quite possible to detect the same problems using the standard search tools found in any modern development environment. Our approach seeks to scale this work up to address larger elements of a program which are not so well defined and which perhaps do not map so neatly onto the structure of an AST.

Another AST based analysis is described by Dudziak and Wloka [10]. They describe the creation of a tool, J/Art, which attempts to identify a number of Fowler's bad smells and provides re-factoring support to help repair them. They do not provide any evaluation of their technique but they do address data classes so it is instructive to examine how they described the data class problem. (They do not address god classes).

They take the view that a data class is defined by the ratio of accessor methods to normal methods found in the class. If this value is greater than a given threshold (which is not specified) then the class is considered to be a data class. Inherited behaviour is not considered in this calculation. Accessor methods are described as 'getters' or 'setters'. Getters are defined as methods that only contain a return statement. The contents of the return statement must be a class field. Setters are defined as methods with one parameter and which contain an assignment statement. The assignment statement must assign the value contained in the parameter to an element of the class state.

Their description of a data class, appears very similar to our own approach (described elsewhere in this paper), but there are a number of areas in which their approach differs. For example, they do not distinguish between public and private methods when calculating the ratio of accessors to normal methods. This seems odd because private accessors are arguably a part of the implementation and therefore not subject to the same coupling issues as public methods. They also do not consider public data relevant (they do however require that the class contain some data). This is particularly unusual because public data is an even stronger source of coupling than an accessor method (one can always redefine a method body as long as the external signature stays the same but the same is not true of data). Finally their definition of getters and setters appear very exclusive and will only detect the most clear cut of accessor methods.

As far as we are aware only two other approaches in the detection literature address data and god class problems. Marinescu [17] uses metrics to detect data and god classes. While Chatzigeorgiou et al. [4] adapt the HITS algorithm to the detection of god classes. Ratiu [21] also detects data and god classes but his work is an extension of Marinescu's ideas, adding change history information to assist in their detection. This approach is not considered further because it seems orthogonal to the underlying detection strategy used. Change history information could be applied to any detection technique.

Marinescu proposes a metric based approach for automatic detection. He characterises data classes as having little functionality and exposing data through a combination of public attributes and accessor methods. These qualities are captured using three custom defined metrics; weight of class (WOC), number of public attributes (NOPA) and number of accessor methods (NOAM). The identification of a data classes relies on a combination of a low WOC value and a high NOPA or NOAM value. Weight of class is described as the ratio of non-accessor methods to accessor methods in the interface of the class. The

threshold value for detection is when more than two thirds of a class interface consists of accessor methods. Number of public attributes is a count of the attributes with public access modifiers. Classes with NOPA values greater than five are considered data classes. Number of accessor methods is a count of the accessor methods found in the class interface. Accessors are defined as methods which contain get or set in their name, are small in size and have no logical decisions [18]. Classes with more than three accessors are considered data classes.

God classes are described by Marinescu as large classes which use data from other classes and can be uncohesive. These qualities are modelled using three metrics: access to foreign data (ATFD), weighted method count (WMC) and tight class cohesion (TCC). Access of foreign data is a count of the number of data classes used by a class. Data access is determined either by access to a class's public state or by use of an accessor. Weight of class [16] is a measure of the complexity of a class. In this case the complexity of each method is assumed to be uniform so the metric becomes a count of the number of methods. Tight class cohesion [3] calculates the ratio of methods which share attributes to those which do not. The identification of a god class relies on the detection of all three metrics within a class

One of the main weaknesses of the approach would appear to be the use of threshold values (e.g. the requirement that a NOPA value of greater than five is an indicator of a data class). There is little information on neither how these threshold values are established nor how they perform across a range of systems.

Chatzigeorgiou et al. propose an adaptation of the HITS algorithm [15] (originally used to rank the authority of web pages returned by a search) to the detection of god classes. Their approach assumes that god classes are more important than other classes in the system (because they contain all the behaviour) and that this can be detected by looking at the message flow between classes. They argue that the importance of a class (its authority) is not only related to the number of classes which communicate with it (its hub weight) but also to the importance of those classes. The HITS algorithm is used to calculate authority and hub weights for each class in the system. Classes with large authority and hub weights are considered more important and therefore more god-like than other classes in the system. Chatzigeorgiou et al. claim that to determine a god one must examine both the authority and hub weights for a class. The main drawback with this approach is that it does not consider the nature of the communication between classes. There is no way to tell from the authority and hub scores alone whether the communication between classes is actually passing data inappropriately or whether it is part of a legitimate method communication.

3. THE TOOL

The smell finder tool has been developed as a plug-in for Eclipse. In general terms the tool is based on the automatic identification of a set of heuristics that are symptoms of poor design – patterns of interaction or elements within the source code that are indicative of encapsulation problems and may be identified statically. It uses a static analysis of Java source files to detect these heuristics and from this infer the presence of god and data classes. Eclipse provides a rich environment for the development of such a tool because it provides project management functions and frameworks to enable the static analysis of project artefacts.

The tool has been developed to assess the utility of our detection strategies. As it currently stands it is a relatively basic implementation which only provides a minimum amount of functionality suitable for experimental purposes. In practice any such tool would require a more sophisticated front end for specifying queries and a more detailed mechanism for reporting and visualising the problems that it discovers.

3.1 Operation

The tool operates on what Eclipse calls JavaProjects. These, as the name implies, are a collection of Java files and references to external libraries that make up a project. The tool only operates on source code so all analysis (for example searching within a hierarchy) terminates at the transition from project code to Java or external library code (assuming the source for these is not present within the JavaProject). In addition the source that the tool parses should be complete and compilable for best results.

To activate the tool a project must be selected in the PackageExplorer (a part of the Eclipse IDE). By right clicking on the project a context menu will appear which contains an option to "Detect data and god classes…" selecting this menu item will start the plug-in. The tool will then step through every Java file within the project searching within the file for various structural properties that might indicate a potential god or data class problem (these properties will be discussed in more detail below). If a problem is found the tool outputs various details about the class and the particular parts of the class which appear to be contributing towards the problem. These details are displayed in the Eclipse console and are also written out to a file for later analysis.

3.2 Implementation

For every Java file in the project the tool translates the Java source code into an abstract syntax tree representation (AST). This is equivalent to the textual representation but has the advantage of separating the description into its constituent lexical units making it easier to automatically analyse. The parsing of the text into an AST representation is performed using the Java Development Tools framework (JDT) which is provided as part of Eclipse. In addition the AST generated by the JDT includes partial symbol table information about some of its nodes. This makes it relatively easy to, for example, determine which variable a given label refers to, which can greatly simplify analysis. Where possible the tool also makes use of the Java Model framework, another Eclipse framework which is used to model various elements of a project and constituent classes. This model is less detailed but is preferred to the AST representation of a file because it is cheaper to create and work with (but has the limitation that it only contains structural information about a class – no implementation details are available).

3.3 The Analysis

The analysis identifies bad smells by recognising particular combinations of nodes, or nodes with specific values, within the AST which represent features of god or data classes. Identifying these patterns is performed using an implementation of the Visitor design pattern [12]. A visitor is created for specific nodes of interest within the tree. Whenever that type of node is visited, code is executed which analyses the node's relationship to its neighbouring nodes or to query particular values of the node itself. If the node is found to conform to a suspicious pattern of

behaviour then the containing class is identified as having a problem and evidence describing that problem is recorded.

The visitor approach has the advantage of simplifying the search for nodes within the tree. A separate visitor can be created for each type of node that has to be identified and the search strategy which moves between nodes is guaranteed to correctly visit every node within the tree (including compound expressions and other recursive structures). A disadvantage of the visitor pattern is that it can be quite inefficient as it searches every node of the tree regardless of the type of node required and where it can be legally placed. This results in a performance loss for the tool and could probably be improved by the use of a more intelligent search strategy.

The analysis requires two passes to correctly identify data and god classes. The first pass identifies data classes within the system and the second pass uses this information to help identify god classes.

3.4 Evidence

For each pattern of nodes that the tool identifies as a design problem, information is recorded to help a later manual analysis rediscover the affected parts of the class. The information recorded varies according to which problem has been detected but it includes the class name and a list of all attributes or methods of the class which contribute towards the problem. Additional information might also be present to help identify what kind of problem has been identified or to provide more information about the nature of the problem. (For example the data class test reports which type of expression within each method was ultimately responsible for affecting class state, and the god class test reports upon the types and methods of data classes that are found within its methods). Example output for the data class problem is provided in Figure 3. Similar evidence is available for the god class problem.

```
de.gulden.framework.amoda.generic.core.GenericFeature
Fields:
        enabled:    Public field
        shortcut:   Public field
        style:      Public field
        system:     Public field
        constant:   Public field
Methods:
        getShortcut[]:          Field
        setStyle[String]:       Field
        setEnabled[boolean]:            Field
        setConstant[boolean]:           Field
```

Figure 3. Example output produced by the data class detector

In addition to providing evidence the tool also provides a ranking of severity of the problems discovered. This is calculated using two alternative ranking schemes for each smell. Data classes are ranked either by the absolute number of data class features found within the class or as a ratio of the data features to non data features. God classes are ranked either by the number of types that they control or the number of discrete calls they make to methods on those types. This information is output at the end of the analysis and is intended to help guide the developer's choice of which problems to tackle first.

3.5 Detection Rules

The tool identifies data and god classes using heuristic rules derived from the software design literature. These rules are codified into sequences of nodes that can occur within the AST of a class and particular properties of those nodes within the tree. This section describes the specific details of how these rules are mapped onto the AST for data and god classes.

3.5.1 Data Class Detection.

Our approach identifies a data class if the class exposes public state or if it contains one or more data methods. Public state is recognised by looking at the access modifiers for each FieldDeclaration Node in a class, any node which responds positively to an isPublic() query is considered public state. Data Methods are slightly more complex. In our approach they are detected by searching for the presence of a return statement, which returns class state or an assignment statement which assigns a value from a method parameter to the class state. A graphical representation of this approach is provided in Figure 4 and Figure 5.

To determine if a return or assignment statement is affecting class state a separate test is applied to the relevant Expression node within the statement. The expression can be resolved into class state in one of three ways; if it is of type SimpleNode there is a chance that the expression represents the use of a field from the class. This can be tested by using the symbol table information to determine if the simple name refers to a class field (IVariableBinding.isField() must return true). The expression can also be resolved to class state more explicitly by being modelled as either a Field Access or a Super Field Access node (these represent state access which begins with an explicit 'this.' or 'super.' call).

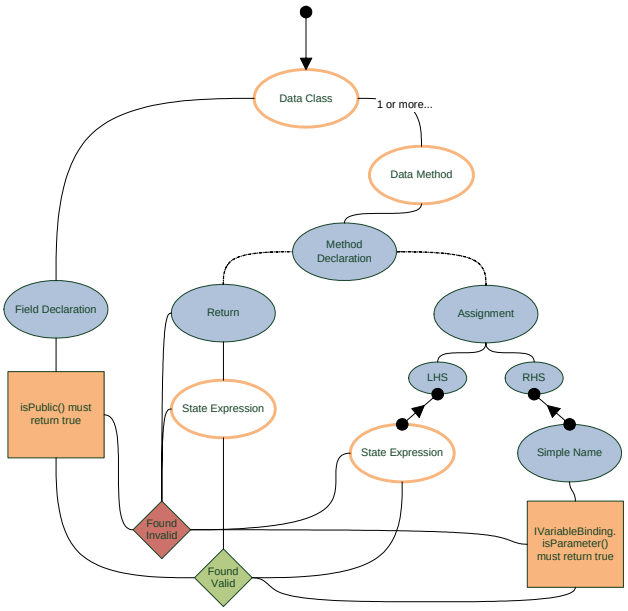

Figure 4. The Data Class Detection Strategy

The test for class state is complicated by the fact that expressions are not terminal nodes in the Java grammar. This means that in some cases an expression can be resolved into another set of other nodes which may also have to be checked for class state. In many cases such decomposition immediately rules the expression out of such resolution (for example arithmetic operations might contain many individual parts but the result of the operation will be a new value not a reference to class state). However there are several cases where an expression can be decomposed into a simpler expression that can still be resolved as class state. These include the use of parenthesis around an expression, the use of a cast operator or the use of a method call which returns part of the class state.

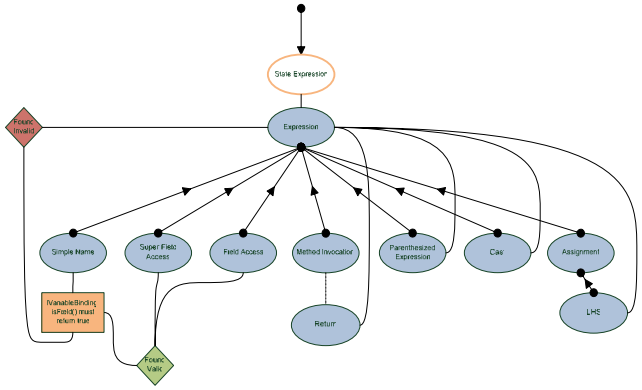

Figure 5. Detection of class state

3.5.2 God Class Detection.

In our approach god classes are detected by the presence of calls from a suspect class to a data class. This is detected in the tool by looking at the body of each method in the project searching for method calls to known data methods or field accesses to public fields (excluding self references or calls to inherited methods or fields). This approach is illustrated in Figure 6.

Method calls are easily identified as they are modelled explicitly as Method Invocation nodes within the AST. Once identified the method signature (including parameter types) is extracted and compared to a list of known data methods (provided by an earlier pass of the data class detector) if the method call is on the list then the method is considered to be a part of a god Class.

One complication about this test is that the calls to a method of a particular type might not be implemented within the type in question. It is therefore important to trace the correct implementation to test if it is a data method. For behaviour which is defined in a parent class this is not a problem as it is relatively easy to search up the hierarchy until a suitable definition is found. For behaviour which is defined below the type (i.e. polymorphic calls to an Interface) the search is not quite as simple. In this case there might be multiple compatible definitions of the method and it is not possible statically to determine which of these definitions will be invoked at runtime. The tool therefore takes a conservative position of checking each permissible definition that might be invoked and if any of those are found to be a data method then the polymorphic call is also considered to be a data method.

Remote field accesses are modelled in the AST as Qualified Name nodes. Like Simple Names these nodes are generic so an additional test is required to determine if the node represents a call to a field (using the supplied symbol table information). If the variable binding reports that the name represents a field then it is assumed to be publicly accessible (because it has been accessed from another class) and the calling class is declared a god.

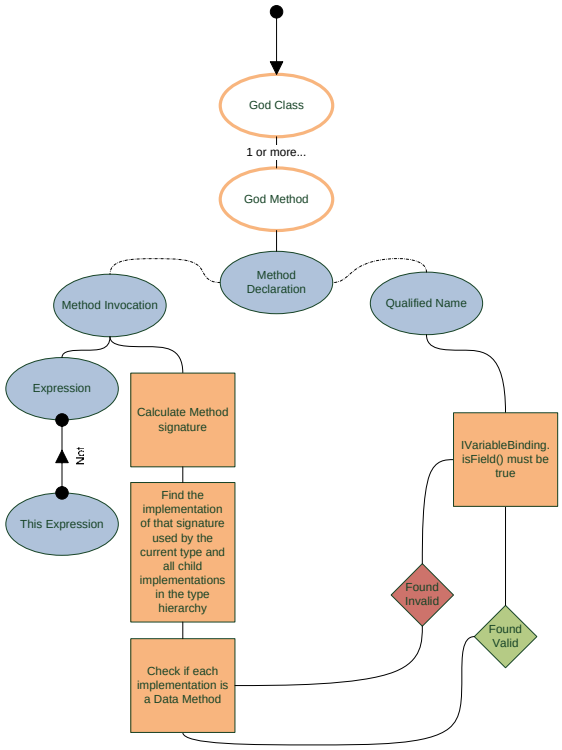

Figure 6. The God Class Detection Strategy

3.5.3 Combining Detectors.

The tool currently does not provide any end user configuration of the detection strategies but some configuration is allowed at compile time. The data and god class detector strategies are made from combinations of simpler strategies which are implemented using a pipe and filter architectural style. This allows a complicated strategy, such as the detection of a data class, to be composed of smaller and simpler strategies, such as the detection of return statements containing class state or field declarations which are publicly accessible.

Each strategy takes as input a set of classes which are processed according to the rule that the strategy is trying to detect. The classes that match the rule are then passed as output from the strategy. Strategies can be chained together in this way so that the output of one strategy forms the input of another. Chains of strategy objects can also be combined together using And, Or and Not operations. This makes it possible to assemble increasingly complicated strategies from combinations of simpler ones. It also makes it easy to alter the operation of a detector by changing the configuration of strategies used.

3.5.4 Other Detectors.

In addition to the detection strategies described above a number of other strategies were considered. They all focused on the detection of data classes.

One strategy used the heuristic of method names (similar to Marinescu's approach) to identify accessor methods. This worked

by extracting the method name from each Method Declaration node within an AST and testing to see if it began with the words ‘get’ or ‘set’.

Another strategy used method size to indicate the presence of an accessor. Once again it visited each Method Declaration within the AST but this time it extracted information about the start and end position of the method (derived from its position in the original text file). This information was used to calculate the number of lines of code used by each method. Any methods smaller than a given threshold (we found 6 lines of code worked well) were considered candidate accessors (in practice both the name and small method strategies had to be detected in order for a method to be considered as an accessor).

A final strategy that was experimented with was detecting whether public state was actually being used elsewhere in the program. The argument here is that, although the exposure is a problem it only really becomes significant once that data is used by other parts of the system. The detector operates by calculating the set of public fields that belong to a class and then searching for field accesses to them in any other class in the system (an action which is similar to the detection performed for god classes). The state is only considered to be a problem if another class in the system can be found which accesses the field (without using inheritance or self access).

The three strategies described here were used in conjunction with the other strategies to produce an aggressive variant of the detection strategy which would focus on the more obviously damaged classes in a system. Ultimately they were not used in this study because we believe that it is better for automatic detection to identify the complete range of potential problem classes in a system rather than to focus on the most pathological.

3.5.5 A Domain Specific Language Approach.

In an ideal world one would like to see the creation and alteration of queries to be included as a proper part of the plug-in available from a user interface and under the complete control of the end user. One critical step towards this vision maybe the creation of a domain specific language with which to describe and combine the various detection strategies that can be proposed.

Such a language would have the advantage of expressing the problem in terms that would be understandable to any Java developer and would remove the need to have explicit knowledge about the AST, JDT and JavaModel frameworks that are currently used to specify the search query. We are currently considering the development of such a language and while a complete description of its syntax and semantics are beyond the scope of this article an initial glimpse of what it might look like can be seen in Figure 4, Figure 5 and Figure 6.

4. EVALUATION

We have found performing an objective comparison of this tool with others difficult to perform due to the ambiguity surrounding the definitions of data and god classes.

The accepted definition of a god class as being a dominant controller of significant complexity is not entirely borne out by the literature. For example, Riel [22] begins on p32 by saying; "The behavioural form of the god class problem is caused by a common error among action-oriented developers in the process of moving to the object-oriented paradigm. These developers attempt to capture the central control mechanism so prevalent in the action oriented paradigm within their object oriented design. The result is the creation of a god object that performs most of the work, leaving minor details to a collection of trivial classes."

This sets up a position where gods are big and domineering and datas are small and functionless. He then goes on to argue (p33) that accessor methods are a symptom of god classes. "Another symptom of this problem is the creation of many get and set functions in the public interfaces of your application's classes". Crucially though he does not argue that these accessor methods must be to the exclusion of all other types of method, all he says is that there will likely be a lot of them (the use of the adjective many in the previous quote). He then goes on to describe a number of examples and it is clear from these that his concern is purely for the existence of accessor methods within other classes - not that they display an absence of behaviour.

The issue of domination is illustrated in an example of a god class (p37). In his example a heat flow regulator is accused of being a god class over three sensor classes (which are later combined into a Room class). While it is true that the regulator is controlling the behaviour of the system, the idea that it has 'centralised control' is less well demonstrated. All that must be done to correct the problem is move one method from the god class into the data class. The simplicity of the solution seems to undermine the claim that gods have to be dominant over their data (which seems to imply a much more coupled and intractable relationship).

For these reasons we have initially configured the tool to adopt a conservative approach where any data leaks are reported to be evidence of a data class and any use or abuse of such leaked data is regarded as evidence of a god class.

Clearly this is an extreme position to adopt but it is our view that any breaches of encapsulation should be identified and reported as even the most minor violation may be indicative of the start of a more serious problem. It is also our plan to augment these detectors with additional ones that sniff out evidence of the more serious problems (for example by using information such as the proportion of data leaks, the number of accesses and the way in which this data is manipulated). For this reason we have chosen to compare our system with the metrics implementation developed by Marinescu, not from a competitive point of view, but for the purposes of evaluating the strengths and weaknesses of the two approaches.

4.1 Methodology

The comparison of Marinescu’s technique and our approach was performed across two large open source systems. One system, BeautyJ [13] was selected based on previous familiarity with the system and the other, JEdit [20] was chosen because it represented a more general purpose domain and was also significantly larger than BeautyJ. It must be stressed that this study is in no form intended as a criticism of the either system and we are indebted to their authors for making the source available.

BeautyJ is a source code transformation tool for Java source files. It contains approximately 200 classes. JEdit is a text editor with a particular bias towards use as a programming environment. It

contains approximately 500 classes. Both systems are freely available in the open source community and both have been in existence for a number of years and are still actively maintained at the present time.

The comparison was performed by running both tools over each system and collecting the results produced. These results, effectively a list of data and god classes were then compared, and where different a manual inspection of the class was performed to identify why one technique had identified something and the other had not. This approach was then used to address the differences between each technique and provide evidence in support of or against the use of that particular advice.

4.2 Results

The results of running our tool and Marinescu's over the two open source systems are shown in table 1. This shows the number of total, unique, and common problems identified using both approaches. Looking at the data there is a considerable difference in the number of problems identified by each technique. Marinescu's approach detects significantly fewer problems than our technique. This immediately raises a number of questions regarding the accuracy of the two approaches. A manual examination of the data class results shows that both techniques were accurate in their detection (a small number of false positives (12) were found from our technique but in relation to the number of problems identified (163) this is quite insignificant). This suggests that the difference lies with the numbers of false negatives.

Table 1. Metric and heuristic results for the two systems

System		BeautyJ		JEdit	
	# classes	**Data**	**God**	**Data**	**God**
Metrics	**Total**	10	10	70	46
	Unique	2	1	56	15
	Common	8	9	18	31
Heuristics	**Unique**	63	83	161	114
	Total	71	92	179	145

A closer inspection of Marinescu's technique reveals that he is filtering his results quite heavily. A class is only identified as a data class if it has an overwhelming ratio (two thirds) of data exposing members to non data exposing members, it also requires more than five public attributes and three accessor methods before it is considered a data class. A god class, according to Marinescu, requires access to data features from three other classes while also being large, complex and un-cohesive. These definitions exclude a large number of classes which contain smaller amounts of data or god like behaviour. This limits his technique to the detection of severe or pathological cases.

In total, 63 suspect data classes are ignored in BeautyJ by Marinescu's technique and 161 suspect data classes are ignored in JEdit. The effects of filtering on god classes are no less significant. 83 classes in BeautyJ and 114 classes in JEdit are being ignored by Marinescu's technique, many of which are not trivial. For example, VFSFile (a class from the JEdit system) is excluded by Marinescu's filters because it has a WOC rating of 0.34 which is marginally higher than the cut off of 0.33. This is despite the fact that the class contains 9 public fields and 15 accessor methods. The classes which are being filtered are those that are beginning to degenerate into data and god classes. Addressing these is likely to be easier than dealing with pathological cases and would also enable the technique to be used as a preventative solution. While the application of filtering is attractive in that it limits the number of problems reported and presents only the worst offenders, it is also somewhat dangerous if a large number of marginally less offensive problems remain unreported. It seems preferable to detect as many problems as possible and present them all to the software engineer, who can then decide how best to tackle them.

4.3 An Analysis of Data Class Detection Strategies

The approach used to detect data classes is quite different between these techniques. Marinescu relies on a black box understanding of the method, where its name and other qualities are used to determine if it is an accessor. Our approach is more invasive and looks inside the method body to find its evidence. This section looks at these differences in more detail.

4.3.1 Naming.

Marinescu relies on the name of a method to determine if it is an accessor while our technique uses state analysis to detect if state is mutated or retrieved within a method. Marinescu's naming convention (when supported by other metrics such as size and complexity) identified approximately the same numbers of accessors as the state analysis approach (when using his unfiltered results). He detected 243 accessor methods for BeautyJ and 511 accessor methods for JEdit in comparison to 260 and 607 accessor methods detected by state analysis. The similarity in these results suggests that the convention of naming accessor methods get or set has been widely adopted in both systems.

The danger with relying on a convention, such as method naming, is that it is easily violated. A number of such violations were found in both systems during this investigation. A typical example is BeanShellAction (from JEdit) which contains three accessor methods each of which start with the prefix 'no' rather than 'get'. Marinescu's approach did not identify these methods as accessors because of their unconventional names while the state based approach detected them successfully because it works with the internal behaviour of the method. The use of convention can also cause the detection of false positives; this occurred in a number of cases where a method named 'get…' returned a null or constant value. This is arguably not part of the class state and so should not be considered an accessor but it is difficult to come to this conclusion from looking at the method signature alone.

The investigation also revealed some weaknesses in our state based approach. It cannot distinguish between trivial accessor methods and more significant ones. For example toString methods (a Java idiom to help debug and exception handling) were often detected as accessors because they returned a string held as part of a class' state. While this is technically a valid accessor method it is unlikely that it will present much of a problem during system maintenance and would most likely have been ignored by a

manual analysis. Another problem was discovered is that the evaluation of state mutation and retrieval can sometimes fail to detect legitimate problems. This occurs because the current implementation only checks for actions that are performed directly onto the class state. If actions are performed on a local variable, which is an alias for some state, or if parameters are assigned to a local variable before being used to mutate state then the change is not detected. A more rigorous form of data flow analysis will be required to address this type of problem.

4.3.2 Size.

The two approaches also differ in their evaluation of size and complexity. Marinescu only considers methods which are small and uncomplicated while our technique does not place any restrictions on the size and complexity of an accessor method.

The majority of data collected in this study supports Marinescu's position that size and complexity should both be low. A typical accessor method from BeautyJ or JEdit consists on a single line which either returns or assigns a value to some state. On one hand a metrics-based approach like Marinescu's does have one notable advantage; that it is simple to perform and yet produces relatively accurate results (given a few assumptions). However it is not hard to find examples which contradict this rule. In contrast, our technique detects the method as an accessor because it ignores its size and searches the method body for evidence of state manipulation.

4.4 An Analysis of God Class Detection Strategies

The detection of god classes in both techniques is essentially the same. Both look for calls from a god to a data class. There are some differences, in that Marinescu also considers the cohesion and complexity of his suspect god classes but the core mechanic for detection is the same. This makes it surprising that there is still considerable difference between both techniques in this comparison.

4.4.1 Inherited Data Methods.

One difference between the techniques is the use of inheritance in the detection of data classes. This was not discussed previously as it is of little use in the detection of data classes alone (as one is primarily interested in the behaviour that is physically present within each class). In fact the detection of data classes for their own sake does not use inheritance at all, only when the data classes are being used to detect gods are the inherited features considered. These features are important during god detection because a given method might hold a reference to a derived type but call data methods which belong to its super type. In this circumstance each method call must be evaluated and this requires traversing the inheritance hierarchy.

The description of Marinescu's technique makes it clear that inheritance is not used in the detection of data classes. This suggests that his approach would fail to detect this type of god behaviour. Surprisingly the investigation reveals that this was not the case. Investigations of Marinescu's filtered data suggest that he does not rely exclusively upon the same set of data classes that are detected by his data class filter. In other words classes and methods which he considers not to be data classes are used to identify and justify the existence of god methods and classes. By contrast, in our technique only material detected as belonging to a data class is used to assist the identification of gods.

4.4.2 Polymorphic Methods.

A problem related to the topic of inherited data methods is the use of polymorphic calls to base classes. In this situation there is a possibility that at least one of the derived behaviours for that method call may be an accessor method. Our technique considers this situation and performs a conservative calculation to determine if the given method should be an accessor. This calculation checks all the existing implementations for that method and if any are found to be a data method then the polymorphic call at the root of the hierarchy is similarly considered to be data (because there is a chance that the data centric implementation might be bound at runtime).

A comparison with Marinescu finds that he does not detect this form of god method. This is a potentially significant oversight as polymorphic calls are a common feature of object oriented languages. Failing to detect the presence of data methods in these situations could help to explain some of the differences observed in the detection of god classes between both techniques. However, this is not a fundamental failing of a metrics-based approach and there is no obvious reason why metrics such as ATFD could not be adapted to deal with polymorphism.

4.4.3 Cohesion and Complexity.

Another difference between techniques is Marinescu's use of cohesion and complexity measures to help determine the presence of a god class. Marinescu requires that his god classes have a poor cohesion rating and a high weighted method count (essentially that they un-cohesive and large and complex). These measures are not used to detect a god class on their own but are used in conjunction with the amount of access to foreign data (essentially the number of calls to data classes) to help detect a god.

Our argument is that while these additional measures might help to confirm a god (or to prioritise how bad it is) they are not required for a god to exist and so may be blocking otherwise legitimate gods from being detected. This can be seen clearly in Marinescu's unfiltered results where the cohesion and complexity methods are dropped (along with a threshold on AFTD) and this makes a significant difference to the number of classes detected.

4.5 Is The Problem Too Big To Fix?

The detection of data classes and god classes can both be considered separate parts of the larger problem of encapsulation. The argument goes that an ideal arrangement of data and functionality places code together with the data that it operates on, so that changes to one have only a minimum impact on the other (i.e. stay within a class).

The results of this study (if they are representative) suggest that such arrangements of code are not very common. In both cases significant amounts of the system were found to have some form of poor encapsulation. This suggests that these systems will be hard to maintain and adapt as their respective domains change and it also has implications for the practicality of automatic detection. Is detecting problems which are so pervasive actually helpful? Clearly there is a problem but if the solution requires half of the system to be rewritten then the problem may simply too big to be addressed. This is a significant problem for automatic detection

but there are a number of potential solutions which might be explored.

First there is the potential to develop the system further so that it doesn't just detect problems it also proposes a solution to them. This would involve automatically generating new abstractions from the existing structures in the code, while preserving their behaviour and also hopefully preserving the comprehension of the new system (a large scale automated re-factoring). This might require splitting existing classes and methods and recognising repetition and duplication in order to pull out new ones. A lot of the individual steps of this are feasible today – but stitching them all together into a sensible whole would be a challenging task.

The second direction is to restrict the use of the tool to that of a pedagogical aide to support students learning how to program. In this circumstance the small application development and the constant cycles of feedback might help to internalise some of the good practices which in turn would help to avoid the creation of god and data classes in real world systems (although this would need to be prefaced with some appropriate training about what gods and data classes are and how they should be avoided).

A third direction might be to emphasise the early detection of problems. This might require the tool to be adopted early in a project lifecycle, where it could give feedback as the code is being created. An obvious way to support the developer would be to provide this feedback as they type in the IDE (rather like Eclipse already does with syntax details). However there is a question about whether the detection could be responsive enough to work in such a dynamic (and only partially complete) scenario.

Whatever the situation, the fact remains that automatic detection has found a significant number of problems in both of the systems investigated. This suggests that even if the results are difficult to work with there is a benefit in detecting the problems and at least being aware of the risk lying dormant within a system.

5. CONCLUSIONS AND FUTURE WORK

A central tenet of object-oriented design guidance is information hiding that encapsulates data and functionality together in a balanced set of cooperating classes. However, achieving this design goal in practice is extremely challenging, especially for large systems that are developed and maintained iteratively over a long period of time. This paper has described an automated approach, that detects data and god class violations of encapsulation, based on a collection of 'bad smell' heuristics applied to an abstract syntax tree representation of Java program code. Data classes are detected by looking for classes that have any public state or 'data methods' - methods that get or set class state. God classes are detected by looking for classes that use data classes - either directly accessing their public state or by using their getter or setter methods. A tool has been implemented as an Eclipse plug-in which has been shown to successfully identify such potential encapsulation problems.

The tool was evaluated in a comparison with the metrics-based approach of Marinescu based on two open-source case-studies - BeautyJ and JEdit.The main finding was that, despite quite different approaches to analysis, both approaches discover a surprisingly high level of data and god occurrence in the two case studies. Our approach suggested that about 35% of the classes in both BeautyJ and JEdit had some symptoms of broken encapsulation through data classes and 46% of the classes in BeautyJ and 29% of the classes in JEdit had some symptoms of god class breaches of encapsulation. Marinescu's top-level results were much less than this, identifying 5% data classes and 5% god classes in BeautyJ, and 14% data classes and 9% god classes in JEdit. However, Marinescu heavily filters his results, only showing the worst examples of data and god class breaches of encapsulation. If the filters are removed then Marenscu's results are of the same order as those produced by our approach. The main differences that remain result from the key differences in the two approaches - Marinescu relying on method naming conventions and requiring data methods to be relatively small and simple for data class detection, and the consideration of inheritance and polymorphism in our analysis of god classes and the consideration of cohesion and complexity in Marinescu's analysis of god classes. It is our contention that although a metrics-based approach exhibits certain strengths and is attractive in its relative simplicity, our AST-based analysis suffers from fewer weaknesses and has the potential to perform a more detailed and accurate analysis particularly when it comes to the more sophisticated design flaws we plan to study (such as "Feature Envy", "Primitive Obsession" and "Data Clumps" [11]). .

The main contribution of this work is the demonstration that data and god class violations of encapsulation in object-oriented programs can be detected automatically by a heuristic-based analysis of the abstract syntax tree representation of the code. In applying the technique to two-open source case studies a surprising number of data and god classes were detected. There are technical implications of these findings - it is clear that the technique needs to be refined so that results can be ranked according to user-defined criteria and more sophistication needs to be provided in the detection of data and god classes. This can be done by incorporating some of Marinescu's ideas on class and method size and complexity together with more a sophisticated analysis of data usage in both data and god classes. There are also wider software engineering implications - if these two case studies are representative of software that is currently being developed then data and god class breaches of encapsulation are extremely common. Further work is required to investigate whether this is really the case and to determine whether breaking encapsulation in this way is having the major impact on maintenance that conventional software design wisdom would have us believe.

6. References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis, "Advanced clone-analysis to support object-oritented system refactoring", Proc. Working Conference on Reverse Engineering (WCRE 00), IEEE Computer Society, 2000, pp. 98-107.

[2] K. Beck and W. Cunningham, "A Laboratory For Teaching Object-Oriented Thinking", Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), ACM Press, New Orleans, Louisiana, USA. October 1989. pp. 1-6.

[3] J. Bieman and B. K. Kang, "Cohesion and reuse in an object-oriented system", Proc. ACM Symposium on. Software Reusability (SSR'95), ACM Press, Seattle, Washington, USA, April 1995. pp. 259-262.

[4] A. Chatzigeorgiou, S. Xanthos and G. Stephanides, “Evaluating Object-Oriented Designs with Link Analysis”, Proc. 26th International Conference on Software Engineering (ICSE'2004), IEEE Computer Society, Edinburgh, Scotland, May 2004. pp. 656-665.

[5] S. R. Chidamber, C. F. Kemerer, “A Metrics Suite for Object-Oriented Design”, IEEE Trans. Software Eng. 20(6), IEEE Computer Society, 1994. pp. 476-493.

[6] I. Deligiannis, M. Shepperd, M. Roumeliotis and I. Stamelos. “An empirical investigation of an object-oriented design heuristic for maintainability”, Journal of Systems and Software 65(2), February 2003, pp. 127-139.

[7] S. Demeyer, S. Duccasse and O. Nierstrasz, “Finding Refactorings via Change Metrics”, Proceedings of the 15th Object Oriented Programming, Systems, Languages and Applications conference (OOPSLA 00), ACM Press, Minneapolis, Minnesota, USA, October 15-19, 2000. pp 166-177.

[8] K. De Volder, “JQuery: A Generic Code Browser with a Declarative Configuration Language”, Proc. 8th International symposium on Practical Aspects of Declarative Languages (PADL06), Springer LNCS vol. 3819, Charleston, SC, USA, January 9-10, 2006. pp. 88-102

[9] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, “Does God Class Decomposition Affect Comprehensibility?”, IASTED Conf. on Software Engineering, IASTED/ACTA Press, Innsbruck, Austria, Feburary 2006. pp. 346-355.

[10] T. Dudziak and J. Wloka, “Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code”, Diploma Thesis, Technical University of Berlin, Germany. Feburary 2002.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, “Refactoring: Improving the Design of Existing Code”, Addison Wesley, Reading Mass. 1999.

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides. “Design Patterns: Elements of Reusable Object Oriented Software”, Addison Wesley, Reading, Mass. 1994.

[13] Gulden, “BeautyJ”, http://beautyj.berlios.de/, 2007.

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated Support for Program Refactoring using Invariants", Proceedings of International Conference on Software Maintenance (ICSM'01), IEEE Computer Society Press, Florence, Italy, November 6-10, 2001. pp 736-746.

[15] M. Kleinberg, “Hubs, authorities, and communities”, ACM Computer Surveys. 31(4es), ACM Press, New York, NY, USA, 1999.

[16] Lorenz and J. Kidd, “Object-oriented software metrics”, PTR Prentice Hall, Englewood Cliffs, New Jersey, U.S.A., 1994.

[17] R. Marinescu, “Detecting Design Flaws via Metrics in Object-Oriented Systems”, Proc. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), IEEE Computer Society, 2001. pp. 173-182.

[18] R. Marinescu, “Measurement and Quality in Object-Oriented Design” (PhD thesis), "Politehnica" University of Timisoara, 2002.

[19] B. Meyer, “Object-oriented Software Construction”, Prentice Hall, Upper Saddle River, NJ, USA, 1988.

[20] Pestov., “JEdit”, http://www.jedit.org/, 2007.

[21] D. Ratiu, “Time-Based Detection Strategies”, Masters Thesis, University of Timisoara, Romania. September 2003.

[22] A. J. Riel, “Object-oriented Design Heuristics”, Addison Wesley, Reading Mass. 1996.

[23] F. Simon, F. Steinbruckner, and C. Lewerentz. “Metrics Based Refactoring”, Proc 5th Conference on Software Maintenance and Reengineering, (CSMR 01), IEEE Computer Society, Lisbon, Portugal, March 2001. pp 30-38.

[24] T. Tourwé and T. Mens, “Identifying Refactoring Opportunities Using Logic Meta Programming”. Proc 7th Conference of Software Maintenance and Reuse (CSMR 2003), IEEE Computer Society, Benevento, Italy, 2003. pp. 91-100.

[25] E. van Emden and L. Moonen. “Java quality assurance by detecting code smells”, Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 02). IEEE Computer Society Press, Richmond, VA, USA, October 2002. pp 97 - 106

[26] M. Wood and M. Roper, “52.361 Group Project”, http://www.cis.strath.ac.uk/teaching/ug/classes/52.361/, 2006.