A Heuristic-Based Approach to Automatically Detecting 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

A significant problem in maintenance of object-oriented software is the identification of classes that are in breach of design guidelines and are thus likely to cause particular problems when it comes to repairing or modifying systems. For large software systems, manually detecting instances of poor encapsulation can be tedious and error-prone. This paper presents an automated technique based on a set of heuristics to detect such breaches of encapsulation in Java systems, particularly with regard to god classes and data classes. A precision-recall analysis of the application of the technique on a number of open source systems demonstrates its effectiveness, and the approach is shown to compare favourably with existing metric-based 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 [15] 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 to classes (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.

Two of the common 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][7], 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 application to an open source system and also by comparison with a metrics based approach. The results indicate that it is an efficient and effective way to detect potential data and god classes in a system, although the evaluation raises some questions about exactly what constitutes a data or god class.

2. Related Work

2.1. Data Classes and God Classes

Data classes are described by Fowler [8] 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 [16] 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.

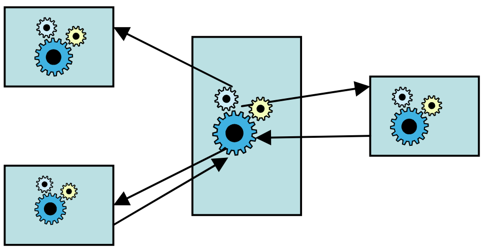

Figure 1: Ideal distribution of data and behaviour

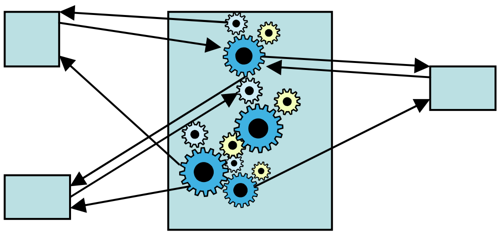

Figure 2: Impact of god and data classes

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[1] [17] [18] but there are two pieces of work which are closely related and which address the detection of god and data classes – those of Marinescu [13] and Chatzigeorgiou et al. [3].

Marinescu proposes a metric based approach to data and god class detection. Data classes are characterised by Marinescu 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 [14]. 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 of foreign data (AOFD), 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 [4] 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 [2] 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 NOPE 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 [11] (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. Detection Technique

This section describes a new approach to the automatic detection of god and data classes. It is based on a set of heuristics that are symptoms of poor design - patterns of interaction or elements within the source code of a system that are indicative of god or data classes and may be identified statically. This approach has similarities to that suggested by Marinescu but does not rely exclusively on metrics to make its diagnosis and avoids the use of threshold values - a clear drawback of such metrics-based approaches. Instead any information that can be extracted from the program text is considered if it helps detection.

3.1. Data classes

The detection of data classes relies on two things, being able to detect public fields and being able to detect accessor methods on the class interface.

To detect public fields the access modifiers for each field of a class are analysed and those which have public access are identified. Public state is widely criticised by the object-oriented design literature as it couples the implementation of a class to its users. Riel says it best when he says “it throws maintenance out the window”.

The approach used to detect accessor methods is to look for public methods that contain a single statement. In the case of a getter accessor the statement will be a return statement and it must return an attribute of the class. A setter accessor is similar except the statement will be an assignment statement which must assign a value from a method parameter to a class attribute.

It is not universally accepted that accessors are a problem. Meyer [15] advocates the use of “query” methods in the public interface of a class to return information about its state and argues for the use of setter methods to help reduce the length of parameter lists in other methods. Riel [16] on the other hand considers them to be dangerous “because they indicate poor encapsulation of related data and behaviour”. Fowler [8] agrees saying “We have lost count of the number of times we’ve seen a method that invokes half-a-dozen getting methods on another object to calculate some value”, before recommending that the data and behaviour be encapsulated together.

The literature on data classes has little to say about the content of accessor methods except to imply a get/set naming strategy [8][16] and that they are often small [13]. The chosen detection strategy was decided by studying many examples of accessor methods in student coursework submissions from a significant third-year group project [19]. These routinely follow the pattern described, specifically in terms of their manipulation of class state.

A distinction can be made between partial data classes, where the existence of any public state or accessor methods signify a problem and total data classes which require the entire class to consist of accessors or public state. This approach considers partial data classes enough of a problem to trigger detection. Total data classes are a problem because they contain no behaviour and expose their state, the solution is to move the behaviour into the class and remove the accessor [8][15]. There is no reason why classes which only have some accessors should be treated differently from classes which are entirely composed of accessors. Both exhibit the same problem and require the same solution so to distinguish between them is both impractical and potentially dangerous as it may overlook serious problems.

3.2. God classes

Riel [16] argues that god classes can be identified in four ways; using the class name, the class size/complexity, use of accessor methods or by measuring its cohesion. His advice does describe four qualities associated with god classes but only one of them, use of accessor methods, actually defines a god class, the others describe symptoms which may apply to some gods but not all.

The sharing of data and behaviour across class boundaries is at the heart of the god class / data class problem and it should be driving their detection. In contrast Riel's other three qualities are true of god classes only in certain circumstances. Class names are subjective and a controlling or procedural name may not be indicative of a god class. Similarly a god class is not necessarily large. It may contain only a small amount of controlling behaviour. This might have an impact on its method size but would have a marginal impact on the size of the class. Riel also assumes that god classes are procedural in nature, this can result in large classes and cause low cohesion but it does not necessarily mean that the class is a god. A procedural abstraction can contain all its required state and behaviour inside the class. Such a class may contain multiple abstractions but it is not a god.

God classes are detected by looking for inappropriate use of state from another class. This is controversial as it focuses on a subset of Riel's advice. Inappropriate state use is detected in a similar way to the detection of data classes. The suspect class is searched for any public field accesses or calls to accessor methods on another class. If any are found then the class is considered to be a god. What the class does with the accessed state is not considered important because the very act of having a reference couples the god to the data regardless of its use.

Polymorphism is important in the detection of god classes. Calls to data classes can be hidden by polymorphic calls to interfaces or parent classes in an inheritance hierarchy. This technique takes a conservative approach to polymorphic method resolution, where if any of the potential matches are resolved to a data method (as described above) then the call is considered to be from a data class.

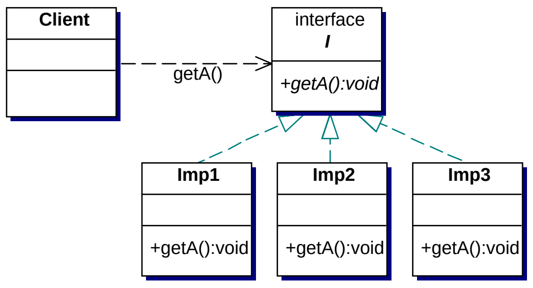

Figure 3: Resolution of Polymorphic Method calls

This is illustrated in the diagram above. The client calls a method, getA(), on the interface I. The interface lacks an implementation those provided by its children are considered. In each child the getA() method is evaluated to determine if it is an accessor method. If any of the implementations are found to be accessor methods then the interface is considered a data class (for the purposes of this method access).

3.3. Implementation

The technique was implemented as a plug-in for Eclipse and operates on Java programs. It uses abstract syntax trees to extract the static information that characterises god and data classes described above. The tool requires source code and the system must be in an executable state before being processed. The tool produces a report which lists all classes investigated and reports if any were found to be a data or god class. It also includes a debug mode which provides details about which characteristics contributed to its diagnosis. The high-level detection algorithm is described below:

Data/God Class Detection Algorithm

• First pass for each class in the system (detects data)

1. Build the AST for the class.
2. Search for class attribute nodes in the AST with public access modifiers. If any are found then the class is considered a data class.
3. Search for instances of return or assignment statements nodes in the AST
4. For each return statement check if the expression on its right hand side evaluates to an attribute access (if not found in local class check further up the hierarchy (stopping at Object or when the parent class resides outside the system))
5. For each assignment statement check if the expression on its left hand side resolves to an attribute access (as

with return) but also check that the right hand side expression resolves to a parameter node from the method containing the statement.
6. If a return or assignment statement has been found then measure its size (looking for methods with one statement).
7. If the class has been found to contain public state or accessor methods then store it in a cache along with details of the state and accessor methods that were found. If no data class characteristics have been identified evaluate its super classes. If any are found to be data classes then the current class is also data.

- Second pass for each class in system (detects gods)

1. Build the AST for this class.
2. Search for field access nodes within the AST, (exclude self sends and inherited attributes). Check that the field access is defined as public.
3. Search for instances of method invocation nodes within the class. Resolve the type of object on which the method is invoked (ignore self sends).
4. Look up type and method in the cache to determine if the method is an accessor.
5. If the type is an interface search its implementations looking for any which exist in the cache. If any of the implementations are resolved to be an accessor method then the call is considered to be to a data class.
6. If an accessor call or public state access is found in the AST then the class is considered a god. If no god class characteristics have been identified evaluate its super classes. If any are found to be god classes then the current class is also a god.

As part of the implementation the technique underwent considerable calibration and refinement using a collection of smaller systems (pieces of coursework undertaken as part of a third-year student group project). The authors selected six of these – 5 from within our own university and one from another university. The systems were relatively small, ranging in size from around 20 to 40 classes, which permitted the authors to analyse them exhaustively for signs of god and data classes. This was also made feasible by the fact that the authors were familiar with the problem domain which simplified the analysis, even though there were a wide variety of solutions. Studying these systems helped to refine the technique in a number of ways. Precision and recall scores calculated across the six systems produced an average precision of 0.99 and an average recall of 0.99 for data classes and an average precision of 0.89 and an average recall of 1 for gods. This suggests that the technique was effective at identifying the data and god classes in these systems.

4. Case Study

The goal of the case study was to determine the effectiveness of the proposed heuristic based technique for detecting god and data classes. The study was designed to identify the strengths and weakness of the approach and also to determine the usefulness of identifying god/data class based design flaws. The study was based on a precision-recall analysis comparing the results of a manual review-based detection against the automated application of the technique. Detailed analysis of false positives and false negatives provides insights into the potential effectiveness of the technique and is used to propose further refinements of the approach.

4.1. Methodology

The BeautyJ [9] system was selected from the open source community to be used as the basis of the case study[1]. The authors had encountered it before in a different context, but had no insights into its design. BeautyJ is a source code transformation tool for Java source files. It contains approximately 200 classes separated into 33 packages. The entire system was considered too large to inspect manually so a subset of 7 packages containing 50 classes and appearing to contain the core functionality of the system was selected for analysis. It was manually inspected by two reviewers to identify the baseline set of god and data classes that exist in the system. The automated technique was then applied to the system and a list of candidate god and data classes was generated. The results of the automated analysis were then compared to the results of the manual inspection using a precision-recall analysis.

4.2. Manual inspection

The manual inspection of god and data classes was performed by the second and third authors. At the time that they performed the inspection they did not have a detailed understanding of how the automated technique operated. This enabled them to categorise the BeautyJ classes independently of the technique. To assist them during the review they were provided with a checklist

[1] It must be stressed that this study is in no form intended as a criticism of the BeautyJ system and we are indebted to its author for making the source available.

which summarised the prominent literature on god and data classes.

The two reviews were performed individually and took approximately 3 hours each to complete. The reviewers agreed in the majority of cases but they were not unanimous in their opinion. Of the 50 classes inspected they disagreed on the assessment of 11 data classes and 7 god classes.

The evaluation required an authoritative categorisation of god and data classes to act as a baseline for the precision-recall analysis, so the reviewers met along with the first author to try to resolve the discrepancies. To assist the discussion, notes taken by the reviewers during their initial categorisation were used, as was the BeautyJ source code and the debug output of the tool. The sources of disagreement were debated and a consensus then reached about their classification. The authoritative list is shown in Table 1 along with the results produced by the application of the automated technique to the BeautyJ system.

4.3. Precision and recall results

A precision-recall analysis was performed comparing the results produced by the automated technique and the results from the manual inspection. Recall was calculated as the ratio of god/data classes retrieved by the automated technique versus the total number of god/data classes in the system as defined by the manual inspection. Precision was calculated as the ratio of god/data classes (as defined by manual inspection) retrieved by the automated technique versus the total number of classes retrieved.

The initial precision and recall values for god classes were 0.37 and 0.86, respectively, and for data classes 0.59 and 1. The results suggest that the technique has been successful in finding the majority of problem classes but its accuracy is low. In particular it appears that the technique detects a large number of false positives. However, it will be argued in the Discussion section that, on closer analysis, the precision results in particular are actually much better than this.

Table 1: Comparison of manual and tool results

Investigated classes	Tool Data	Manual Data	Tool God	Manual God		Tool Data	Manual Data	Tool God	Manual God
beautyjTask	Yes	No	No	No	Code	Yes	Yes	Yes	No
Beautyj	No	No	No	No	PackageMember	No	No	No	No
Main	No	No	No	No	Exception	Yes	No	Yes	No
BeautyJ	Yes	No	No	Yes	Import	Yes	No	Yes	No
Task	Yes	Yes	No	No	Class	Yes	Yes	Yes	Yes
Type	Yes	Yes	Yes	No	SourceParser	Yes	No	Yes	Yes
Package	Yes	Yes	Yes	Yes	ParseException	Yes	Yes	No	No
ImportClass	Yes	No	No	No	SimpleNode	Yes	Yes	Yes	No
MemberExecutable	Yes	Yes	Yes	No	ParserConstants	No	No	No	No
Implementation	Yes	Yes	No	No	Parser	Yes	No	Yes	Yes
SourceObjectDeclared	Yes	Yes	Yes	No	JavadocParserTokenManager	Yes	No	Yes	No
Constructor	Yes	No	No	No	JavaCharStream	Yes	Yes	No	No
Documentation	Yes	Yes	No	No	JJTParserState	No	No	No	No
NamedIterator	Yes	Yes	Yes	No	ParserTokenManager	Yes	No	Yes	Yes
ImportPackage	Yes	No	No	No	JJTJavadocParserState	No	No	No	No
ClassInner	Yes	No	Yes	No	JavadocParserConstants	No	No	No	No
DocumentationTagged	Yes	Yes	No	No	JavadocParserTreeConstants	Yes	No	No	No
Field	Yes	Yes	No	No	TextImage	No	No	No	No
ProgressTracker	No	No	No	No	Token	Yes	Yes	No	No
SourceObjectDeclaredVisible	Yes	No	No	No	TokenMgrError	No	No	No	No
SourceObject	Yes	Yes	No	No	JavadocParser	Yes	No	Yes	No
Method	Yes	Yes	No	No	Sourclet	No	No	No	No
DocumentationDeclared	Yes	Yes	Yes	Yes	SourcletOptions	No	No	No	No
Member	Yes	Yes	No	No	AbstractSourclet	Yes	Yes	No	No
Parameter	Yes	Yes	Yes	No	StandardSourclet	Yes	No	Yes	Yes

4.4. Discussion

The results of the technique are overwhelmed by the large number of false positives produced by both the data and god class detection strategies. However, a closer inspection of these false positives reveals some valuable insights and suggests that the technique may perform better than first appears.

4.4.1 Data Classes. A manual inspection of the sixteen false positive data classes detected by the evaluation surprisingly showed them all to be valid data classes! In each case the classes were found to have either public state or inherited data class characteristics from a parent class. The reasons for the initial review failing to detect so much public state are not clear. It was one of the characteristics of data classes that the reviewers were asked to detect and it was successfully detected on a few occasions. It is possible that the formatting of attributes in the source code or the relatively small size of an attribute definition may not have drawn the eye of the reviewers in the same way that method bodies did. The reason for inherited behaviour being overlooked is more straightforward. It is not part of the standard literature advice so was not included in the reviewer's checklist. In addition it is difficult to detect manually because it involves code which is disparately placed within the system (providing yet further evidence for the need for an effective manual approach). These findings strongly suggest that the false positives detected in this evaluation should be reconsidered as valid data classes.

4.4.2 God classes. Twelve false positive classes and one false negative god class were detected by the evaluation. A manual inspection revealed that all of the false positives contained access to data classes suggesting that they should have been detected during the review. However, it is possible that a number of classes were misdiagnosed during the review because of the particular difficulty of manually assessing god class behaviour. This type of analysis requires careful parsing of method bodies to detect call sites, which must then be reified into a call to a particular type and its implementation inspected to determine if it is an accessor method. Given this process it would not be surprising if a manual inspection overlooked some incidents of data class access.

It is also possible that some classes were overlooked because they did not demonstrate other more visible god class properties and conform to the stereotypical dominant procedural controller that is often described in the literature. These might be easier to detect than those which just use another class's data but this does not guarantee that they are genuine god classes.

An example of one of these false positives is the `Type` class in the BeautyJ system. This class shows strong god class qualities in the `countArrayDimension(Node)` method. This method gets a list of children from its `Node` parameter and iterates through this collection inspecting each node for a particular id which determines if it is an array. This couples the `Type` class unnecessarily to details of the `Node` class. If `Node` changes how it holds its children or how it models the identity of array nodes then `Type` will also have to change. Moving the behaviour from `Type` into the `Node` class would decouple the two classes and also allow other users of Node to benefit from the array counting functionality.

The particular feature that the false positive gods appear to lack was the notion of control coupled with a cyclic interaction of getting, manipulating and setting data. This interaction can be difficult for a manual review to detect but the dominant control structure is easier to recognise. An indication that reviewers tend to be drawn towards this style of class is given by the one false negative god class that was found during the evaluation (the BeautyJ class itself). A subsequent manual inspection of the class revealed that it was not a god class as it did not use state from another class. The reviewers had concluded that it was a god on the basis of its size name and procedural behaviour but the lack of data access was overlooked. This is further evidence of the need for an automated approach. Manual inspections find identifying data accesses difficult and can be persuaded by circumstantial evidence to declare a god.

4.5. Reassessed scores

The problems inherent in manually detecting data and god classes suggest that the precision-recall scores may not accurately reflect the performance of the technique. Adjusting the figures for the detection of data classes results in precision improving to 1 with a recall of 1. However, this may be generous for already in extensions of this research we have noticed further classes that are not canonical data classes, but nevertheless expose some of their state. This issue needs to be explored further. For god classes the adjustment is more debatable because although the technique detects encapsulation problems, they may not all be true god classes, in the sense that they don't conform completely to the published stereotype. Nonetheless if poor encapsulation is used as a criterion the performance improves producing a recall of 0.94 and a precision of 0.84. It is our belief that raising even minor breaches of encapsulation is beneficial as it may indicate a problem in its own right or the start of a bigger problem in the future.

5. A comparison with a metrics-based approach

This section compares our heuristic based approach with an existing metrics based one – that of Marinescu's [13] – with a view to analysing the strengths and weaknesses of the two strategies. The availability of a supporting toolset was the principle reason for choosing Marinescu's work as the basis of this comparison. Both tools were run over two open source systems: BeautyJ and JEdit but for reasons of space only the BeautyJ results are reported here. Table 2 summarises the results of running both tools over BeautyJ.

Table 2: Metric and heuristic results for the BeautyJ system

	# classes	Data	God
Metrics	Total	10	10
	Unique	2	0
	Common	8	10
Heuristics	Unique	63	39
	Total	71	48

Considering firstly the detection of data classes, the initial observation to make is the large difference in the number of problem classes discovered by the two approaches. The initial reaction to this is that something somewhere is seriously wrong. The explanation for this lies in the solution employed by Marinescu, which is to filter the results obtained by his technique so that it only returns the worst offenders. This reduces the number of results returned but it also hides some of the problem classes from the maintainer and presents an unrealistic view of the state of the system. Marinescu's data class cut offs allow classes to have up to three public fields and five data methods before they are considered to be a problem. He also requires an overwhelming ratio of data members to non data members before detecting a data class. This limits his technique to the detection of severe or pathological cases and also prevents the early detection of some problems when they might be easier to address.

If data exposure is a problem then each incidence of exposure contributes towards that problem. It is therefore important that when doing remedial work to repair data encapsulation no incidence should escape consideration. In addition inclusion based on the ratio of data to non data should be irrelevant. If data is exposed then the problem cannot be mitigated by considering how good the behaviour is in the rest of the class. A better solution is to rank the results by their severity. Marinescu applies a ranking to his results after they have been filtered but it would seem better to forgo filtering and simply rank all of the results returned. If this filtering is relaxed then the results become much more comparable, but there still remain two classes unique to the metrics-based approach which are both very small inner classes – something that our technique does not currently analyse.

Our approach reports both the absolute number of data leaks (public data and data methods) per class, and the proportion of the class that these leaks constitute. Ranking them by proportion most closely emulates Marinescu's ranking and leads to a similar set of classes appearing in the top ten with the highest entries clearly being canonical examples of data classes – all public data and getters and setters. There are a couple of exceptions though. Marinescu identifies the class `MapTableModel` as being the 3rd most serious data class, whereas we rank it as 56th! An investigation of this class reveals two clear data leaks in the form of a getter and a setter that both approaches detect, and two methods that access only part of the data (in the form of getting and setting an element of an array) that is included in the metric count but ignored by us. The other notable exception is the inclusion of the `Token` class which, based on the propotional ranking appears as 3rd in our approach as it has eight public data fields and two methods – one of which is a `toString()` which our approach detects as a data leak because it is seen to be exporting state. This is clearly an erroneous classification on our behalf – in an earlier version of our system we included the heuristic that getters and seters began with "get" or "set" but then removed this as it was felt to be overly restrictive. The consequence is the occasional mis-classification of `toString()` methods as data accessors. This will be rectified in the future.

Turning our attention to the god class detection the same issue arises regarding the number of classes and again the same explanation is given. Marinescu also filters the god class results to isolate what he considers to be the worst offenders. If this filtering was removed the results would be much closer. As for the data classes a ranking was created based on the proportion of god methods (those which exhibit god-like behaviour) in the class and compared with Marinescu's top ten results. Again there was a strong degree of overlap (six of his classes appear in our top ten and eight in our top fifteen). Considering the exceptions again, Marinescu ranks the class `GenericMetadata` as the 3rd god class, whereas we rank it as 36th. Investigation of this approach reveals a number of uses of data in associated data classes (our tool allows the user to see the methods responsible and also the data they are using), but nothing so significant to rank it so highly. It would appear to be the values for WMC and TCC that are promoting the class to this level. This is in line with the traditionally accepted advice, but as argued earlier in this paper, is by no means an indication of god-like bahviour. In constrast our approach ranks the Parser class as 5th, whereas it does not make it into Marinescu's filtered rankings. On our analysis, this class contains ten god methods (methods which make use of data leaks to potentially manipulate the state of data classes) and no less than forty individual manipulations of leaked data. This is a clear advantage over the metrics approach which just counts the number of external types referenced and does not consider either the number or the nature of the interactions.

To summarise it is clear that the metric based approach performs well, particularly in the simpler case of the data class, but the limitations start to become exposed when detecting data classes. The power of the heuristic approach is in detecting the subtleties of interactions and relationships between classes by employing more sophisticated analysis (such as improving the precision of the approach by resolving polymorphic calls). However, this also comes at a cost as computing these interactions can take significantly longer. However, this should not be a major issue for the maintainer of a system as the design flaws can be computed once and then the results explored, analysed and repaired over time.

6. Lessons

A number of lessons can be drawn from this study regarding the efficacy of the technique, how its findings affect what we know about data and god classes and also how this work compares to the existing literature. One initial observation is that the static analysis approach employed has considerable advantages over the existing metrics-based approaches. For example, it permits polymorphic calls to be resolved, allows the details of method behaviour to be inspected, checks that the getters and setters are operating on the sate of an object and considers individual abuses of data leaks rather than a gross count of the number of references that a class contains. These features result in a more accurate analysis than could be achieved by metrics alone.

6.1. Data class detection

The data class detector works very well and appears to be an effective way to automatically identify data classes in a system. The description of the internal behaviour of accessor methods, the use of inheritance and the detection of partial data classes appear to have captured useful characteristics for detecting data classes. Future studies should consider refining or enhancing this set.

A related finding from the study is that data and god class functionality can be forced onto a system because of a desire to reuse functionality. If a class library or code generation tool provides a data or god centric API then there is little option but to follow a similar decomposition in order to correctly reuse the supplied class. In BeautyJ this restriction is illustrated by the use of the JavaCC compiler generator [10]. This is responsible for a number of god and data dependencies found within the system.

6.2. God class detection

The god class detector works well in certain circumstances but still needs work to improve its generality. It detects classes which have encapsulation problems (which includes all god classes) but many of those detected have very small amounts of data sharing behaviour and do not have a characteristic godliness about them.

The description is still a useful platform for further work. It has helped to focus attention on to the data sharing behaviour of the god class, rather than more circumstantial indicators such as size or cohesion. It has also emphasized the importance of resolving polymorphism to detect all data accesses. Future work should continue in this direction by attempting to account for how much data access there is in a class, and using data flow analysis to try to determine how the data is used by the potential god class and identify those classes that are the really serious offenders.

7. Conclusions

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 demonstrated that it is difficult to detect these problems manually but that a relatively straightforward static analysis can identify a small set of problem areas (or 'bad smells') that can then be isolated for detailed, manual analysis.

The empirical evaluation based on the BeautyJ case study showed that high accuracy in terms of precision and recall can be achieved relative to a human analysis of the system. The benefits of the approach are in terms of time and scale. It took around 6 hours to perform the analysis manually for a system of around 50 Java classes. It is difficult to imagine human analysis being carried out on industrial-scale systems that are orders of magnitude larger than this. On the other hand, an automated approach can accurately and repeatedly identify god-data class boundaries that should be considered for redesign and refactoring in a matter of minutes.

This initial case study along with the comparison with the metrics-based approach has demonstrated that the approach has strengths but it requires further investigation and refinement. The definition of data and god classes used so far is relatively straightforward. Further work will examine the potential benefits of increasing the sophistication of the analysis e.g. by detecting accessor methods that get and set state partially or indirectly or by analysing the pattern of data usage in god classes.

8. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[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] 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.

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

[9] Gulden, "BeautyJ", http://beautyj.berlios.de/, 2006.

[10] JavaCC, "javacc", https://javacc.dev.java.net/, 2006.

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

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

[13] 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.

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

[15] B. Meyer, "Object-oriented Software Construction", Prentice Hall, Upper Saddle River, NJ, USA, 1988.

[16] A. J. Riel, "Object-oriented Design Heuristics", Addison Wesley, Reading Mass. 1996.

[17] 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.

[18] 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.

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