An Introduction to
Software Mining

Richard KENNARD and John LEANEY
Faculty of Engineering and Information Technology
University of Technology, Sydney
rkennard@it.uts.edu.au john.leaney@uts.edu.au

Abstract. Software mining is related to both data mining and reverse engineering. It is focused on mining software artefacts such as code bases, program states and structural entities for useful information related to the characteristics of a system. This paper provides an introduction to the field. It first reviews a representative selection of the ways software mining has been applied. It then divides software mining into three subcategories and explains each one in detail. Finally this paper summarises some of the advantages and limitations of software mining, both now and in the future. These advantages and limitations have been informed by the author's own research applying software mining to the field of User Interface generation.

Keywords. Software mining

1. Introduction

Software mining is related to both data mining and reverse engineering. It is focused on mining software artefacts such as code bases, program states and structural entities for useful information related to the characteristics of a system [1]. The subject of software mining has a broad definition, with many different useful applications. This paper provides an introduction to the field. It first reviews representative examples from the existing literature. It then divides the discipline and studies each subcategory in detail. Finally it examines some of the advantages and limitations of software mining, both now and in the future.

This paper was informed both by the existing literature and by experiences from our own research. This research is concerned with the application of software mining to the field of User Interface (UI) generation [2].

2. Related Works

To conceptualise the field of software mining it is instructive to review a selection of the ways it has been applied. This section presents a representative, though by no means exhaustive, selection of the ways software mining has been applied to different stages of the software development life cycle. Specifically, it considers software mining's application to development, debugging and maintenance.
First, with respect to development. Grcar, Grobelnik & Mladenci [3] describe mining class names, field names and types, along with inheritance and interface information, in order to construct API documentation and domain ontologies. Their approach further mines source code comments by relying on one of the principles of 'literate programming' – keeping documentation close to the code it refers to [4]. Ma, Amor & Tempero [5] do something similar, mining the names of classes and fields to discern their semantics. Sahavechaphan & Claypool [6] analyse and demarcate sections of code as being relevant snippets for programmers to use when 'developing by example' which they describe as a "largely unwritten, yet standard, practice".

Next, with respect to debugging. Kagdi, Collar & Maletic [7] describe mining usages of groups of methods that are generally invoked together. A simple example might be a call to file.open followed sometime later by a call to file.close. By identifying such typical call usage patterns it is possible to expose atypical ones, which may be useful indicators of programming defects. As Khatoon, Mahmood & Li [8] explain "developers often reuse code fragments by copying and pasting (clone code) with or without minor adaptation to reduce programming efforts and shorten developing time... However, clone code may cause potentially maintainability problem for example, when a cloned code fragment needs to be changed, for example change requirement or additional features, all fragments similar to it should be checked for the change". Tan et al. [9] use natural language processing to compare the consistency of source code comments with the code itself, and therefore identify either misleading comments or bugs in the code. Xie & Notkin [10] use software mining to inspect classes and generate likely unit tests, which practitioners can then review for inclusion in their own test suites.

Finally, with respect to maintenance. Kim et al. [11] report on analysing software version control repositories to recognise clusters of files which are generally updated together. They are therefore able to predict which files should be updated, or at least thoroughly reviewed, following future changes. Kim et al. [12] apply machine learning techniques to crash signatures to prioritise defects. Nagappan, Ball & Zeller [13] compare different code complexity metrics along with history from defect tracking systems to predict areas of code that are likely to be defect-prone going forward. Breu, Zimmermann & Lindig [14] mine classes and methods for 'cross cutting concerns' – areas of duplicated functionality in the codebase that emerge unplanned over time and should potentially be refactored into a common subsystem.

From this diverse range of research projects, we can distil a general understanding of what is considered 'software mining'. By way of clarification it is worth noting there is a strong emphasis in the literature on mining software repositories. Indeed one of software mining's premier annual conferences is titled Mining Software Repositories (MSR 2004-present). This is arguably a misleading title because in industry parlance the phrase 'software repository' typically refers to software systems such as CVS [15] or SVN [16]. The primary feature of such systems is their versioning capability – the "V" in their names – which tracks changes over time. While some of the examples of software mining mentioned do indeed mine such version histories [11], most use software repositories simply as a convenient place to locate source code, defect reports and other documents. There are, of course, other places to locate such artefacts in which case the repository has little bearing on the software mining itself.

Note also that although statistical methods are often employed, this is not always the case. For example, Grcar, Grobelnik & Mladenci [3] employ the discovery dimension of software mining without any statistical dimension. Equally in our own
research, the authors have explored the application of software mining to the field of UI generation [17]. Analysing software mining data statistically is undesirable for UI generation because “UI tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers” and “predictability seems to be very important to [UI] programmers and should not be sacrificed to make the tools 'smarter'” [18].

Having discerned a general understanding of the field of software mining, we turn now to a more in-depth discussion. We shall divide the field into subcategories and examine each in detail.

3. Categories of Software Mining

Cerulo [19] divides software mining broadly into three categories: static analysis, dynamic analysis and historic analysis. Each of these is discussed in the following sections.

3.1. Static Analysis

Static analysis “happens when software is analysed on its descriptive dimension. It is performed on software artefacts without actually executing them” [19]. The most commonly thought of static artefacts are source code files, which are discussed in the following section. It is also important to consider files that contain externalised behaviour, which are considered in section 3.1.2.

3.1.1 Source Code

The term 'source code' is generally used to refer to the human-readable programming language code that is later compiled or interpreted into machine-readable code. The majority of an application's functionality is crystallised into its source code, making it a rich source for mining interesting software artefacts.

The techniques used to extract software mining information from source code are well established, being the same techniques traditionally used by compilers or IDEs – such as Abstract Syntax Trees (ASTs) and program dependency graphs (PDGs). Neamtiu, Foster & Hicks [20] describe using ASTs in conjunction with software version histories to mine the evolution of an application's code over time. ASTs enable practitioners to model the semantics in a language – such as its global variables, types and functions. This allows more meaningful reporting on an application's evolution than, say, comparing number of lines of code or number of files. Liu et al. [21] discuss using PDGs to compare source code for plagiarism between codebases of different applications. PDGs model the program structure of an application, allowing plagiarism comparisons on a deeper level that cannot be deceived by function renaming or statement reordering.

Whilst source code analysis is a rich source of information, not all of an application's behaviour can be determined from source code alone. Some behaviour is externalised, as discussed in the next section.
3.1.2. Externalised Behaviour

As mentioned in the previous section, the majority of an application's functionality is crystallised into its source code. Increasingly, however, significant amounts of behaviour are being externalised [22]. There are several motivating factors for this.

Firstly, the externalised behaviour can be expressed in a form more natural to its content. For example system configuration settings may be expressed in a hierarchical XML [23] file with XML Schema validation. This file can be reviewed and modified by system administrators rather than requiring software practitioners. Similarly, business rules may be expressed in a BPM language that is closer to natural language and easier for non-technical, business users to read and verify [22].

Secondly, the externalised behaviour can be updated at a different frequency to the application code. This is desirable because behaviour such as business rules are often more volatile than the rest of an application's code. Modifying the application itself requires specialised skills and carries with it the risk of introducing defects, whereas modifying configuration files or business rules is a more defined process with less margin for error and can therefore be performed by system administrators or business users [22].

Configuration files, BPML files and other such externalised representations are not generally considered 'source code', but are still valuable repositories of information for software mining. Another valuable set of information can be discerned by dynamic analysis, discussed in the next section.

3.2. Dynamic Analysis

Unlike static analysis, which centres on mining source code and externalised behaviour files, dynamic analysis “happens when software is analysed on its executive dimension” [19]. The rationale for dynamic analysis is similar to that of mining files containing externalised behaviour: whilst source code captures many properties of an application, a significant amount of its behaviour can only be determined from other places.

For dynamic analysis, a notable example is user input. What the user chooses to input can only be determined at execution time, but may have a large impact on the application's behaviour. In the most complicated case the user may be allowed to input source code itself, such as a spreadsheet formula or a scripting language macro, which can add new functionality and screens to an application.

The most common approach to dynamic analysis is reflection, which is discussed in the following section. Some programming environments further support dynamic analysis of embedded metadata, which is considered in section 3.2.2.

3.2.1. Reflection

Reflection allows an executing program to dynamically analyse itself, by inspecting not just the values but the structure of its own data. Unlike other programming paradigms such as procedural and object-oriented programming, which specify pre-determined sequences of operations, reflection allows the sequences of operations to be determined at execution time based on the data being operated on [24].

Reflection can be used to mine the characteristics of an executing program more
accurately than static source code analysis. For example a method foo may be declared in source code to accept an object of type bar. At runtime, the actual type passed to foo may be subBar, a subtype of bar with additional properties. Reflection can correctly detect the subtype, whereas static source code analysis could never make this prediction. As Cerulo [19] puts it “static analysis is affected by the undecidability problem”. Furthermore, reflection can not only detect the subtype, but can also discern its properties – even if it had no prior knowledge of the subtype existing. This makes reflection a powerful tool to handle cases where user input can dynamically add scripts and screens to an application, as discussed in the previous section.

Despite its power, reflection is limited in that it can only inspect characteristics that are pre-defined by the platform. For example a platform that supports concepts such as classes and methods may allow reflection of class names and method names but may not allow, say, reflection of the cardinality of the relationship between two classes. For example, one-to-one or many-to-one. To determine such application-specific characteristics, the platform can support an extensible mechanism such as embedding arbitrary metadata.

3.2.2. Embedded Metadata

Beyond reflection, some software environments provide explicit support for dynamic analysis of embedded metadata. For example the 'attributes' feature in the .NET Common Language Runtime (CLR) [25] and the 'annotations' feature in the Java Virtual Machine (JVM) [26]. Most languages that run atop these environments expose this capability, allowing practitioners to embed metadata into their source code. For example C# [27] and VB.NET [28] on the CLR and Java [26] and Groovy [29] on the JVM. This metadata can be dynamically extracted from instantiated objects at runtime.

Embedded metadata allows practitioners to tag domain objects with arbitrary information, beyond the normal capabilities of the platform. This can later be inspected by application frameworks to allow specialised processing. For example the Java platform does not natively support specifying a maximum length for a String [26]. This is a problem for Object Relational Mapping (ORM) frameworks which require such information when generating database schemas. Using embedded metadata, this information can be added to the source code alongside the Java field it refers to. The metadata is largely ignored by the JVM, but is significant to the ORM framework that is watching for it [30].

Whilst such metadata is valuable, its disadvantage is that it must be specified explicitly by the practitioner, adding complexity to the code. There are other forms of metadata that accrue more transparently, providing a valuable resource to tap into. These are discussed in the next section.

3.3. Historic Analysis

Historic analysis “happens when software is analysed on its evolutive dimension… on software process trails left by developers during their activities and stored in software [version] repositories” [19]. A great deal of insight into the characteristics of a piece of code can be gained by studying how it has changed over time: code that sees frequent changes may be considered less mature; code that sees changes by many different practitioners may be more prone to error; code that is often involved in defect reports may be a candidate for rework.
Historic analysis mines the 'human' element of a project more than static or dynamic analysis can. Version control is a primary resource but there are others, such as defect tracking databases and project mailing lists. Rigby and Hassan [32] describe using psychometric text analysis across mailing lists to understand the practitioner personalities behind an application and their impact on the relative success of the project.

To summarise the three forms of software mining, it can be seen they offer different perspectives on retrieving information related to the characteristics of a system (table 1). The promise of software mining is to be able to combine these perspectives. The potent combination of mining source code, embedded metadata, version histories and other artefacts brings significant advantages. These are discussed in the next section.

<table>
<thead>
<tr>
<th>Categories of Software Mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Source Code</td>
</tr>
</tbody>
</table>

Table 1: Categories of Software Mining

4. Advantages and Limitations

To conclude our overview of software mining, we consider some of its advantages and limitations. This section has been particularly informed by our own research applying software mining to the field of UI generation. Our result is an Open Source project, Metawidget, downloadable from http://metawidget.org. Briefly, the software mining component within Metawidget proceeds as in Figure 1.

It can be seen the Metawidget parent invokes a 'composite' (named after the composite design pattern by Gamma et al. [32]) inspector, which in turn invokes multiple child inspectors. Each inspector performs its own variant of software mining, whether static analysis of XML files or runtime reflection of annotations, which the composite then collates into a detailed whole. This detailed whole is used to drive the UI generation.

Consider a domain object named Hotel. Characteristics of this domain object (such as its fields; its constraints; what actions can be performed on it) are defined across various subsystems within an application. For example the database schema may define the maximum length of a field, whereas the property type subsystem may define whether a field is read-only. The inspectors in figure 1 each mine a different subsystem, as summarized in table 2.
Figure 1: Application of software mining to UI generation

Once collated, this information can be used to drive detailed UI generation, as depicted in figure 2. It can be seen that each element of the UI maps back to some metadata derived from the software mining. For example the Name field has a star because the database schema identified it as a required field. Similarly the Stars field is a slider control because the validation subsystem defined its upper and lower bounds.

The software mining component of Metawidget has proven very successful. It forms the core of an enterprise-class framework that has seen significant industry adoption. For example it has been deployed to thousands of clinics across the Spanish National Health System, and it has been incorporated into products by Red Hat, an industry leading middleware vendor.
<table>
<thead>
<tr>
<th>Domain object field</th>
<th>Application subsystem inspected by software mining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Property</td>
</tr>
<tr>
<td>Name</td>
<td>label, type</td>
</tr>
<tr>
<td>Type</td>
<td>label, type, lookup enum values</td>
</tr>
<tr>
<td>Stars</td>
<td>label, type</td>
</tr>
<tr>
<td>Rating</td>
<td>label, type, read-only</td>
</tr>
<tr>
<td>Notes</td>
<td>label, type</td>
</tr>
<tr>
<td>Save</td>
<td>label, type</td>
</tr>
<tr>
<td>Delete</td>
<td>label, type</td>
</tr>
</tbody>
</table>

**Table 2: Sources of software mining metadata**

**Figure 2: UI constructed from software mining metadata**

Building Metawidget has given us valuable insights into the advantages and limitations of software mining. We discuss these in the next two sections.

**4.1. Advantages of Software Mining**

Our research into applying software mining to UI generation has shown it to have two primary advantages over existing approaches.

First is software mining’s ability to eliminate respecification. Software practitioners traditionally have to respecify information about a system in order to drive UI
generation. For example, our domain object Hotel has a property Name which is specified by the database to be a required field. The practitioner will typically have to respecify that the field should be marked 'required' in the UI. Such respecification is error-prone and a common source of application defects [33].

Software mining can eliminate this problem by examining the existing system architecture. However, examination of any single subsystem invariably leaves some gaps to fill, because no single source of examination is comprehensive. As Schofield et al. [34] observe, “our understanding, as a community, has shown that multiple types of analyses may be relevant to understanding some aspect of [a] system… different types of evidence might be complementary, in which case their cross-referenced analysis and interpretation should increase the quality of the inferred knowledge, assuming that the computational resources for their extraction are available”. Similarly, German, Cubranic & Storey [35] stress “extracting information from most information sources is relatively straightforward. But many questions can only be answered by correlating information from multiple sources”.

The second advantage, therefore, is software mining's ability to collate information from multiple, heterogeneous sources. This is a unique proposition of software mining. Indeed without its collation and analysis dimensions, software mining would be little more than a modern, umbrella term for long established techniques such as parsing and reflection. Collation is made possible because each of the heterogeneous techniques can be applied with the understanding that it is part of a larger software mining process. It can be written to normalise and homogenise the results of its analysis with results from complementary analyses. The promise of software mining is that, by completing such a diverse and thorough analysis, it can obtain sufficient information to avoid the practitioner having to respecify information, or resorting to generalised heuristics.

Despite these advantages, some significant limitations remain. These are discussed in the next section.

4.2. Limitations of Software Mining

A first limitation of software mining is that it can only mine artefacts that are actually part of the software. In a previous journal article concerning the application of software mining to UI generation [2], we wrote “automation is difficult because UIs bring together many qualities of a system that are not formally specified in any one place, if at all”. One of our reviewers criticised: “The authors are stating that the design of the UI brings together functionalities of a system that are not formally specified. I would question this assertion as an increasing number of MNCs are hiring Human-Factors designers (HFD), with solid understanding of software development best practices, to work with potential users of the system to identify the UI needs. These HFD are part of the development team and work closely with the systems analysts and programmers to incorporate the design of the UI into the software process”. Here, the reviewer is pointing out that where we said 'formally specified' what we really meant was 'specified in machine-readable form'. There may well be detailed written documentation on, say, the correct font size for a text box – but unless this is codified in a form that a machine can interpret it is inaccessible to software mining. Tan et al. [9] have made some progress incorporating natural language processing into software mining, but this research is still in its infancy.

A second limitation is that, even if the software mining can interpret the data, it will be unable to collate it unless it can be mapped to some normalised key. Software
mining can be quite flexible in this regard, with different approaches to mapping for different sources, but the mapping must be deterministic. If a database schema cannot be mapped to a corresponding domain object type, or a domain object type cannot be normalised with some fragment of an XML configuration file, then no meaningful collation between them can occur. In practice many of these mappings are already defined in an application and can themselves be mined. For example the persistence subsystem must have a well-defined mapping between object types and database tables, and the validation subsystem must unambiguously understand which types it applies to. The issue is less clear, however, with subsystems such as business rule engines [22] whose input may be a collection of domain objects and whose output may be some rule execution. The inner workings and mappings of such subsystems may be opaque to the rest of the application and non-deterministic to mine.

A third limitation is that it may be difficult to determine when software mining has mined 'enough'. In our research on UI generation we explored using guided software mining to prevent mining too much metadata (i.e. that will not actually be needed during UI generation). But it may be impossible for the software to know whether there was additional metadata available that could have proved beneficial. Such metadata may be missed, resulting in a less functional UI, yet this may only be detectable using traditional techniques such as unit testing and Quality Assurance.

Despite these limitations, software mining has significant potential to resolve many long-standing problems. We turn to these in the final section.

5. Conclusion

This paper has provided an introduction to the field of software mining. It has reviewed a representative selection of the ways software mining has been applied; it has divided software mining into three subcategories and explained each one in detail; it has summarised some of the advantages and limitations of software mining, both now and in the future. This paper has been particularly informed by the author's own research applying software mining to the field of UI generation. Here, software mining has proven to be successful.

We believe software mining has significant potential to resolve many long-standing problems. Not only can it begin to answer previously impenetrable questions, such as which areas of a codebase are more prone to error [11,13], software mining can also dramatically improve existing approaches. For the field of UI generation, our research has shown it can provide a way to extract sufficient metadata to produce non-generic UIs, whilst at the same time not requiring practitioners to restate that metadata in repetitive and error-prone ways.

This approach has proven very popular with practitioners. We could term it 'mining over respecification'. It would have strong parallels to the industry approach of 'convention over configuration' which has also proven very popular with practitioners [30]. A philosophy of 'mining over respecification' would be directly applicable to frameworks which currently require error-prone respecification. For example, object-to-XML mapping frameworks such as Java API for XML Binding [36] could determine whether a field was a 'required' field by mining the database schema, rather than needing a framework-specific @XmlAttribute annotation. Of course we should stress this would be mining over respecification, not instead of respecification: the
practitioner could still respecify the annotation in cases where the XML mapping differed from the database schema. However this would not be the majority case.

Such insights suggest software mining is a fertile area with a promising future. It is a new field and there is much to be explored, applied and learnt. We hope this paper has provided a cogent introduction for research to come.

References

[17] Kennard, R. & Steele, R. 'Application of Software Mining to Automatic User Interface Generation', *Proceedings of the 7th International Conference on
Software Methodologies, Tools and Techniques 2008 pp. 244-254.


