Java Implementation Verification Using Reverse Engineering

David Cooper, Benjamin Khoo, Brian R. von Konsky, Michael Robey

Curtin University of Technology
Department of Computing
GPO Box U1987
Perth, Western Australia 6845

{cooperdj, khoobks, bvk, mike}@cs.curtin.edu.au

Abstract

An approach to system verification is described in which design artefacts produced during forward engineering are automatically compared to corresponding artefacts produced during reverse engineering. The goal is to automatically determine if an implementation is consistent with the original design. In the system described, XML Metadata Interchange (XMI) representations of Unified Modelling Language (UML) class diagrams are recovered from compiled Java class files. These are automatically compared with the corresponding diagrams produced during forward engineering by software engineers using CASE tools. Examples are provided in which reversed engineered UML class diagrams differ from those produced during forward engineering but are still faithful to the original design intent. Such differences are often due to more abstract system representations being captured in forward engineered design artefacts, the inclusion of design attributes and annotations that are not retained in the final implementation, and issues associated with the use of weakly typed containers. In other cases, differences indicate a deviation from the intended design. It is this latter type of difference that this paper is particularly interested in identifying. We advocate that an automated comparison of forward and reverse engineering artefacts should be performed during formal code inspection preparation and used to guide human review of the identified differences.

1 Introduction

The role of verification is to determine if a system conforms to its specification. Verification typically focuses on detecting defects through dynamic testing (Sommerville 2001, pp 440-455), static analysis (Ogasawara *et al.* 1998), and team-based code inspections (Fagan 1986, Ackerman *et al*. 1989). A secondary purpose of code inspections is to improve system quality and maintainability by ensuring that an implementation conforms to corporate standards and accepted industry best practice.

Most approaches to verification are based upon the premise that if an implementation conforms to its specification, then it must also conform to its design, and that defects can be minimised by adhering to agreed upon style and coding conventions.

However, for a given specification there may be many valid designs with corresponding implementations. If an implementation conforms to its requirements but deviates from the documented design, this suggests that system maintainability may be compromised. Therefore, there is a need for an automated auditing tool to prove consistency.

This paper describes the algorithm used by REV-SA - a tool for Reverse Engineering and Verification using Static Analysis. The principal goal of REV-SA is to identify implementation deviations from the documented design. This is achieved by comparing UML design artefacts to corresponding artefacts recovered through reverse engineering. Differences are filtered prior to comparison when design information has not been retained in a recoverable manner or when a certain type of item is known to be unique to the forward engineered artefact.

Classes specified by the user may also be filtered prior to artefact comparison. It is common for a designer to choose not to include some standard classes on a class diagram, such as those from the Java Development Kit (JDK).

Following comparison, the remaining differences identified by REV-SA are then flagged for further human scrutiny.

It is proposed that justifying artefact differences identified through reverse engineering should form part of the code-inspection process and be included on inspection checklists.

2 Background

UML is a design tool prevalent in industry, and is capable of capturing both static and dynamic system attributes. Tools supporting UML are available as commercial software packages. These include Rose (IBM 2003) and

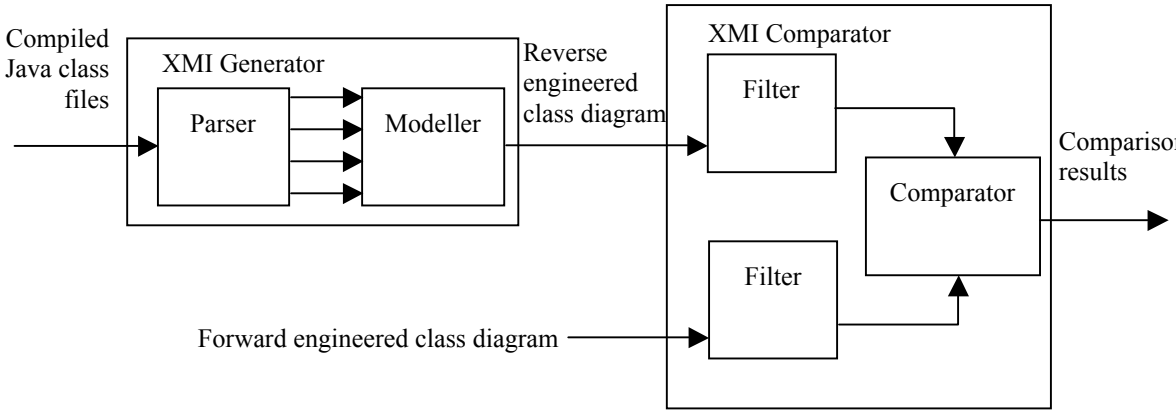

Figure 1: The structure of REV-SA showing the flow of data through the system.

Together (Borland 2003). Non-commercial open source UML tools like ArgoUML (Robbins and Redmile 2000) are also widely available.

UML is not specific to a particular implementation language. This sometimes leads UML tools to model language specific features like Java inner classes in different ways (Kollmann *et al.* 2002b).

In most UML tools, XML Metadata Interchange (XMI) format is generally available either directly or indirectly as a textual means of data archival. However, XMI tags used by a given UML tool are sometimes unique to that tool. This makes information interchange and inter-tool model comparison a challenge. Although not addressed in this paper, incompatible XMI formats could be translated to the one expected by REV-SA using Extensible Stylesheet Language Translation (XSLT) (w3.org 2003) as a preliminary step.

Kollmann *et al.* (2002b) reported on the results of a study comparing supposedly equivalent UML class diagrams that were reverse engineered using standard features available in Rose, Together, Fujaba (Paderborn 2003) and Idea (Bremen 2003). Differences found in the reverse engineering diagrams produced by each tool were automatically determined. However, the algorithm employed to compare the reversed engineered diagrams was not described in any detail. The comparison algorithm used by REV-SA is a contribution of the work reported here.

The difficulties associated with design recovery are numerous (Kollmann *et al.* 2001, 2002a, 2002b; Tonella and Potrich 2001).

Kollman and Gogolla (2001) discuss an approach to source code pattern identification for adorning UML associations, including representations of unidirectional versus bi-directional associations, multiplicity, aggregation and composition, n-ary associations and qualifiers.

Tonella and Potrich (2001) present an algorithm for recovering the class of items stored in weakly typed containers, providing a C++ example. In their algorithm, a flow graph is built describing all potential sequences of assignments, casts, and accessor and mutator gets and sets to infer the types of items stored.

The recovery of design documents where none are otherwise available has also been a major area of previous research (Kollmann and Gogolla 2002a; Systä *et al.* 2000) This includes the broader goal of software visualisation, in which information extracted from source or compiled code is presented in any number of graphical representations. In part, this has sometimes been achieved by extracting class coupling metrics (Kollmann and Gogolla 2002a) and more traditional object-oriented metrics like the depth of the inheritance tree (Systä *et al.* 2000).

Metrics are often collected for the purposes of managing quality (Chidamber *et al.* 1998). In addition to the use of software metrics, it is often suggested that automated static analysis output can be used in preparation for code-reviews (Sommerville 2001, p. 431).

Dunsmore *et al.* (2003) have suggested that code inspection checklists should include object-oriented questions. They argue that the following object-oriented question (and others) should be added to code inspection checklists for object-oriented systems:

"Is all inheritance required by the design implemented in this class?"

REV-SA is a static analysis tool that could be used to analyse an implementation for design conformance and to answer this question and others during code inspection preparation.

3 Methodology

As shown in Figure 1, REV-SA consists of two tools, an XMI Generator and an XMI Comparator. The XMI Generator reverse engineers Java class diagrams, generating XMI files readable by the public domain CASE tool ArgoUML (Robbins and Redmiles 2000). Input to the XMI Comparator takes two ArgoUML compatible XMI files as input and compares them.

Both the XMI Generator and ArgoUML use the Novosoft UML library (NSUML) from Novosoft (2003) to generate compatible XMI files.

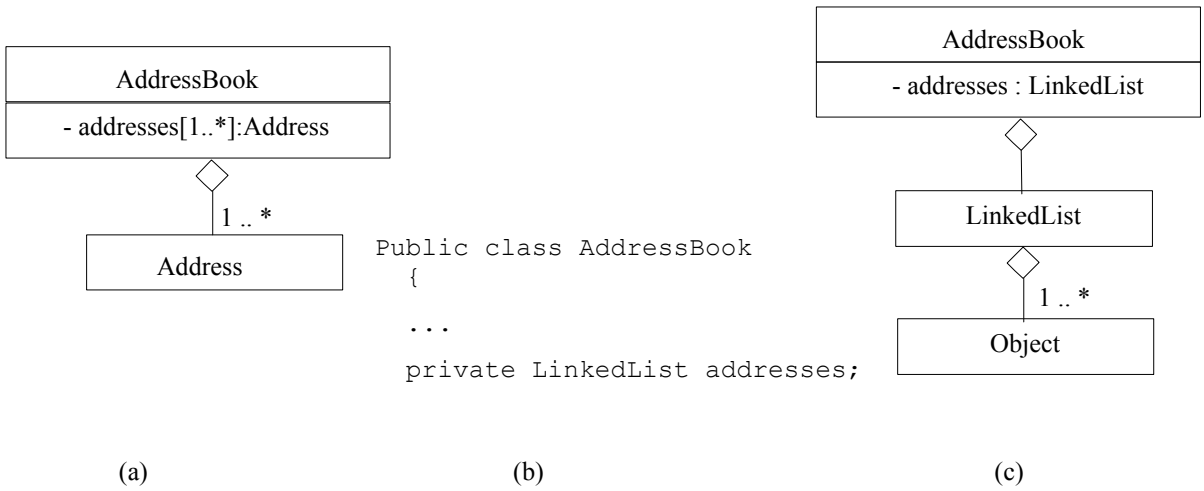

Figure 2. An example of container class inconsistency. (a) shows the original design, (b) the resulting java and (c) the UML class diagram produced by REV-SA.

Unfortunately, NSUML generates XMI tags that may not be compatible with other UML tools. However, this limitation could be addressed using XSLT (w3.org 2003) to translate a given XMI file into the format expected by NSUML.

REV-SA parses compiled class files instead of source code so that class relationships can still be recovered even when source code is not readily available for some classes.

More significantly, REV-SA is the first stage of a more extensive tool that will reverse engineer dynamic UML state chart and sequence diagrams from execution traces produced from running byte code. Dynamic analysis of running byte code is required in this case, and consequently byte code is also used for REV-SA. Doing so allows the same artefact to be used for both static and dynamic analysis.

3.1 XMI Generator

The REV-SA XMI Generator contains two core classes. The first of these - *Parser* - parses a set of compiled Java class files. The second class - *Modeller* – uses the NSUML API to output an XMI representation of the model.

The XMI Generator uses the Java Reflection (java.lang.reflect) package, as it contains the classes and methods necessary to extract information from a compiled Java class. This enables the XMI Generator to recursively parse a set of Java class files, adding extracted information into the object model as it is recovered.

As each class is parsed, the extracted information is passed to the modeller, which adds the information and relationships to the NSUML model.

Once parsing is complete, the model is converted into the XMI format using the NSUML library.

When parsing a class, any other class encountered by REV-SA will also be parsed by default. In some circumstances it might not be desirable to represent this on the generated UML document, as it might be part of a standard library. Consequently, input to REV-SA consists of the list of classes that are not to be parsed and the initial list of classes that are to be parsed.

When a class file is to be parsed, all classes that have a relationship with the class are recursively parsed unless previously specified to the contrary. In that case, it is represented in the reverse engineered model without any attributes or operations. No further relationships from the ignored class are considered.

The XMI Generator was created to capture inheritance, realisation and aggregation relationships. All three types of relationships are critical in ensuring that an implementation matches the system design.

Extracting the superclass and the list of interfaces implemented by each class makes it possible to recover inheritance and realisation relationships.

Extracting class fields implemented by each class also allows aggregation relationships to be recovered. In other words, a given class is considered to be an aggregate of its non-primitive fields.

The multiplicity attribute of the aggregation relationship is recovered through examination of the fields extracted from a class. The default multiplicity used when creating a new aggregation relationship is 0..1. If there is more than one field of the same type, then the multiplicity of the aggregation relationship is increased appropriately. When an array is encountered, the multiplicity is immediately set to 0..*.

This technique works when dealing with multiple fields of the same type. A problem arises however, when container classes are encountered.

During the design phase, the exact data structure for a particular container class is often not known. For example, in an address book design there is an obvious aggregation between a class responsible for the address book (i.e. AddressBook) and a class responsible for a single address (i.e. Address). However, how the collection of addresses will be managed may not be

known during the early design phase. Consequently, a UML class diagram as shown in Figure 2(a) might result.

At a later point during design or implementation, a suitable abstract data type such as a linked list is chosen, resulting in the Java code shown in Figure 2(b). REV-SA would reverse engineer the Java code to produce the UML class diagram shown in Figure 2(c).

The use of the linked list container class (i.e. java.util.LinkedList) has caused an inconsistency with the original design. Even so, both versions of the class diagram convey the intent that an address book will contain a collection of addresses, albeit at different levels of abstraction. REV-SA cannot resolve the difference further using data that can be accessed via the reflection package. However, REV-SA will mark it as an inconsistency, but one that the designer can easily resolve by replacing Object with Address on the class diagram.

The inconsistency in the address book example is caused because the UML design in Figure 2(a) is an abstraction of the actual final implementation. Once such an inconsistency is identified by REV-SA, the original design can be amended to make it consistent with the actual implementation. Such a change might be particularly warranted as a product reaches the maintenance phase.

Perhaps more significantly, REV-SA will also identify unintended variations from the original design.

3.2 XMI Comparator

Comparing two files in the same XMI format is conceptually simple. Any given item in one file should exist in the other in a manner that is semantically equivalent. A comparison is done by translating both files into another common representation such that syntactical redundancy is removed and the ordering of particular elements is not significant.

The general solution is to determine what cannot be recovered or is consistently missing from a given forward or reverse engineered artefact. The two XMI input files are filtered accordingly, removing such information if found. Filtered items are listed in the appendix at the end of this paper.

The comparison process itself, as distinct from the filtering process, is entirely generic. To support this, the filtering process builds two tree structures, one for each model.

Each tree node has:

- a set of "defining information", such as a class name or a method signature;
- a set of "features" or atomic properties, such as modifiers and visibility; and
- several sets of child-elements, where each set is called a "group", such as class attributes and class realisations.

Once the filtered trees have been constructed, the comparison process uses the information contained in them without further regard to specific element types. The XMI Comparator focuses on generic comparison of node content only.

The XMI Comparator generates a third tree containing the comparison results. A set of recursive routines then produce console output listing all inconsistencies and ambiguous elements.

3.2.1 Handling Container Classes

For container class fields and method parameters, the comparison procedure is somewhat different. Since the entire purpose of a container is to store a number of other objects, it should be represented on a reverse-engineered class diagram as multiplicity attached to the relevant attribute and/or association. However, this requires that the actual class of object stored in the container is known.

In the reverse engineered address book example shown in Figure 2c, a designer would examine the class diagram produced by REV-SA and make the determination that Object should be Address.

A more sophisticated byte code parsing algorithm could enable REV-SA to make the same determination. At the present time REV-SA does not have this ability.

More generally, it is possible that objects of many different types could be stored in the container object and hence the determination of what belongs on the class diagram is totally subjective. At this time REV-SA can only raise this as an inconsistency on the list of issues for the designer to resolve.

The XMI Comparator first determines if a particular class is a container. It considers a container to be any class or interface that:

- is java.util.Collection or java.util.Map; or
- extends or implements a container class/interface.

In order to make this determination, the XMI Comparator attempts to use the Java Reflection package to load the class/interface. If it cannot be loaded because it does not exist or has been misnamed, then it is not a valid container. Otherwise, its superclasses and implemented interfaces are checked recursively until the top of the hierarchy is reached, or one of them is discovered to be java.util.Collection or java.util.Map.

Such classes and interfaces are flagged as "ambiguous", as described in the next section.

3.2.2 Comparison Ambiguity

The XMI Comparator lists those cases for which there is a chance that human scrutiny could account for an ambiguity or inconsistency.

3.2.3 Element Filters

The filtering process aims to remove any information from a model that cannot be recovered from compiled Java class files. Most major UML class diagram elements present in an XMI file are mirrored in the XMI

Comparator, where they serve to filter information pertaining to either one or a small number of closely related UML class diagram elements. From here on, the word "element" will be used in the context of the filter itself and not to refer to UML or XMI elements.

For convenience, each element has a name, a set of modifiers (eg. abstract, static, etc.), and a visibility setting. Not all element types make use of these features. Features not used by a given element are ignored by the XMI Comparator.

The element at the root of each tree is referred to as the "model element". There is only one of these for each model.

A model element is the only element type that can contain data type elements, in deference to Java where primitive data types exist only in the global scope. However, model elements are a type of package element. Thus, they can also contain class elements, interface elements and other package elements (except for model elements).

In addition, class elements contain sets of attributes, realisations (implemented interfaces), and nested classes and interfaces. It is worth noting that while Java allows interfaces to contain attributes, nested classes and nested interfaces, UML does not support any of these, and so interface elements have no provision for them.

Attribute and operation elements are straightforward. Neither have any child elements. Like most elements, attributes are defined by their name only.

Operations, however, are defined by their name plus their parameter list, which is an ordered list of classifiers. The names of the parameters are not compared or recorded since this information cannot be extracted by the XMI Generator. Likewise, parameter directions such as in, out or in-out are not recorded or compared.

In REV-SA, each association element is bound to a single classifier and represents a unidirectional relationship with another classifier. Thus, one UML association may be represented by two separate association elements (i.e. one for each classifier).

At the present time, the only relationships recognised by REV-SA are inheritance, realisations and aggregations. UML compositions are considered to be aggregations; the distinction is lost in implementation and consequently cannot be recovered by the XMI Generator.

A caveat exists with the implementation of multiple associations between a pair of classifiers. The purpose of each association can easily be shown on the UML diagram by means of role names, but this is not recoverable from Java class files. The XMI Generator will amalgamate such associations on the basis that the programmer could have been implementing a single association with a multiplicity greater than one. The XMI Comparator takes similar measures to ensure that it can correctly deal with the situation. Only one association element to a given classifier is allowed to exist. If additional associations to the same classifier are given in the XMI file, the multiplicities are added to the original association, effectively merging them.

Finally, the generalisation and realisation elements are used to represent inheritance and interface implementation respectively. Neither contains child elements. Their defining information is taken to be the name of the parent classifier (the superclass, superinterface or implemented interface), and each is bound to the child classifier. That is, each classifier element contains the set of generalisations and (in the case of classes) realisations in which it is the child, and the generalisations and realisations themselves point to the parent.

3.2.4 Comparison structure

The comparison process is generic and provides a means of comparing hierarchical structures without references to specific UML constructs. The tree that results from the filtering process can be traversed and compared without any knowledge of the semantics of the particular nodes.

In all, three types of comparisons are performed and nested within each other: element comparisons, feature comparisons and group comparisons.

Element comparisons involve two corresponding elements, one from each model. Two elements are deemed to correspond if their defining information (class name, method signature, field names, etc.) is identical. Corresponding features are also compared.

Similarly, a group comparison is invoked for each pair of corresponding child elements. An element comparison is successful and reports no inconsistencies if all feature and group comparisons are successful.

The primary function of the group comparison is to match elements to each other by comparing their defining information. Given a set of classes from one model and another corresponding set of classes from the other, a group comparison will report any elements that are in one but not the other. An element comparison is also made for every pair of corresponding classes.

A group comparison is successful if all element comparisons are successful and there are no missing or extra elements in either model. If unsuccessful, the item is listed as a candidate for human scrutiny.

4 Results

REV-SA was tested by a team of two designer-implementers, both working with two different software specifications. These were contrived problems of limited size that could be designed and implemented quickly while attempting to cover the range of software design and implementation attributes that could be used to verify REV-SA.

Figure 3 shows a portion of a forward engineered class diagram created using ArgoUML by one of the designer-implementers for one of the test cases.

Following implementation, the class diagram was reverse engineered using REV-SA. The resulting diagram is shown in Figure 4, following use of the auto-placement feature available in ArgoUML.

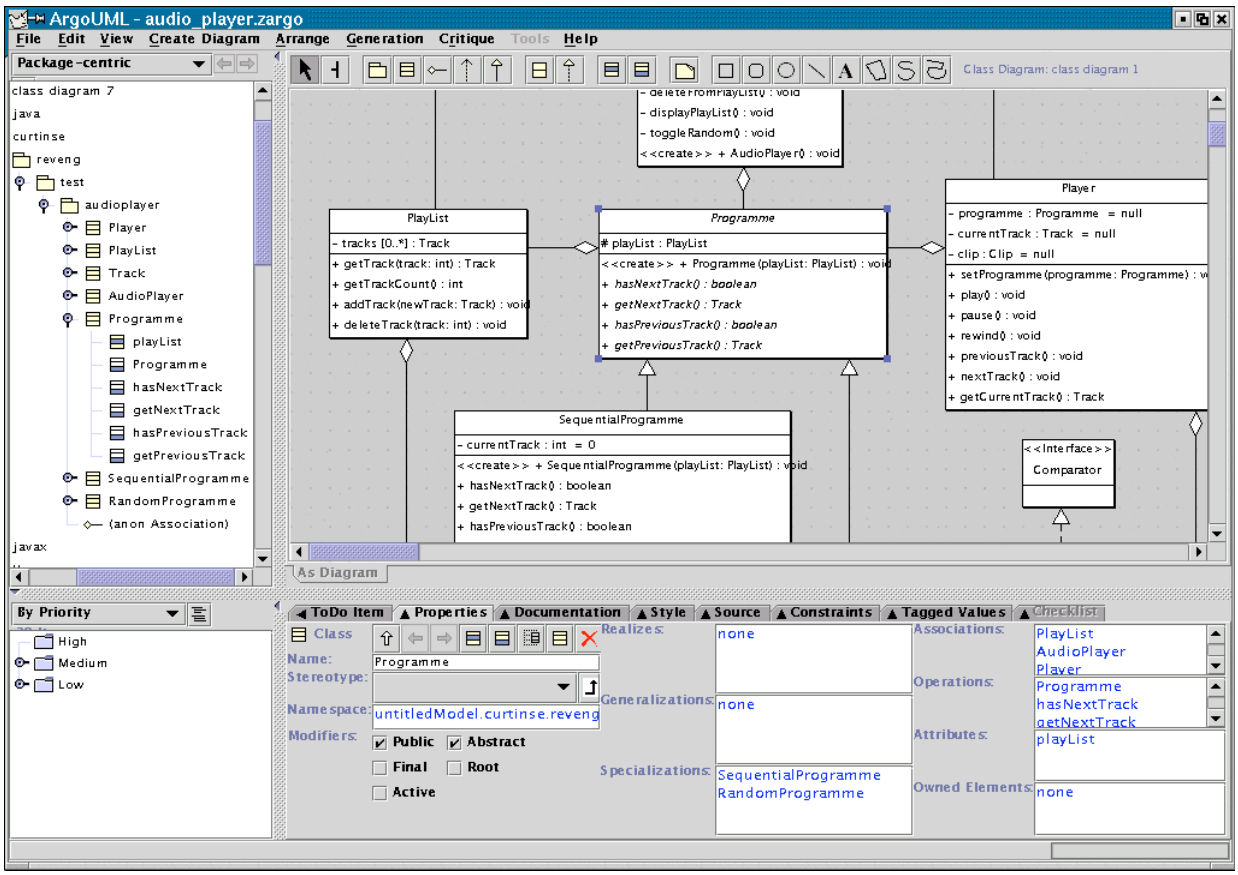

Figure 3. A forward engineered test case produced by a human user in ArgoUML.

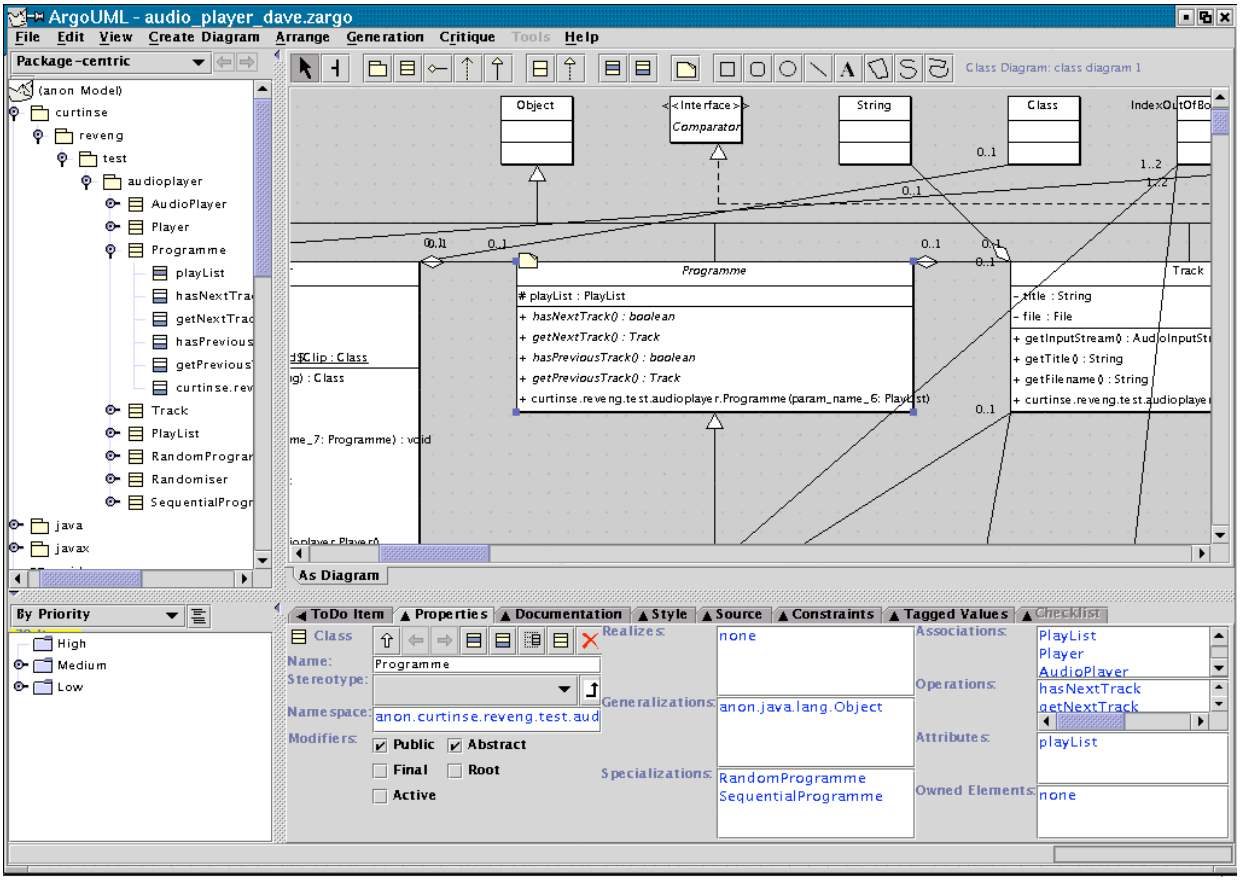

Figure 4. Reverse engineered diagram equivalent to Figure 3 following auto-placement.

While the auto-placement feature is not perfect, it does provide an automated first attempt at symbol placement. Later, the diagram can be manipulated manually, if deemed necessary.

In this particular example, there were 8 designer-defined classes and 1 non-designer defined interface in the original forward engineered diagram. The reverse engineered diagram contained each of these, plus 13 additional non-designer defined classes (String, etc.) and 3 non-designer defined interfaces (List, etc.). For the particular test case shown in Figure 3, the forward engineered class diagram contained 8 aggregation relationships between designer-defined classes, 2 classes were derived from other designer-defined classes, and 1 class implemented a non-designer defined interface.

In the equivalent reverse engineered diagram shown in Figure 4, there were only 6 aggregation relationships between designer-defined classes instead of the expected 8 due to the weakly typed container problem. There were 7 aggregation relationships between designer-defined and non-designer defined classes. Additionally, each designer-defined class was shown to inherit from Object.

The results shown in Figures 3 and 4 were typical of the other test cases from both designer-implementers.

4.1 Implementation-Design Deviations

Design deviations identified by REV-SA in the test cases include the following:

- Missing classes
- Missing methods/constructors
- Spelling inconsistencies
- Additional methods or differing method signatures

Most of the identified errors in the test cases can be attributed to carelessness on the part of the designer-implementers and were not a sign of any larger design or implementation flaw.

REV-SA serves as a mechanism for locating all such deviations so that they can be rectified.

4.2 CASE Tool Introduced Inconsistencies

CASE tool introduced inconsistencies were the most numerous difference detected by REV-SA during testing.

For example, the ArgoUML CASE tool used the Void class as the return type for “void” methods. The primitive type void is not the same as the class Void and was therefore flagged as a difference by REV-SA.

ArgoUML created classes that were used internally, but never deleted them when no longer used. This created a situation where numerous additional classes existed in the human designed model when output in XMI format.

Another inconsistency occurred when one designer made assumptions about the default settings of UML structures such as classes. The designer incorrectly assumed that classes in the CASE tool were public unless otherwise stated. However, since the default setting for class visibility in ArgoUML is private, REV-SA flagged all classes as having a visibility mismatch.

4.3 Implicit Compiler Actions

A number of reported inconsistencies were found to be the result of additional elements introduced into the compiled class file by the compiler itself. These include generalisations to java.lang.Object, default constructors and synthetic fields and methods (Sun 2003).

All Java classes inherit from the class Object except for Object itself. This is not explicitly represented in Java source code and is of little relevance in the normal course of implementation. During the design phase, it is also of virtually no consequence. However, the reverse engineered class diagram will show every class as inheriting from Object, since this information is automatically inserted into Java class files by the compiler.

The REV-SA tool can handle this issue by optionally allowing the user to explicitly instruct the tool not to place specific classes such as java.lang.Object on the diagram. However, one designer did not take advantage of this feature during testing.

Another situation in which the Java compiler will add to a class file is for a class that includes no explicit constructor. In such situations, the Java compiler will automatically install a trivial default constructor that invokes the superclass constructor.

Also not represented in the forward engineered diagram during testing, the Java compiler added class initialiser methods to compiled class files and were therefore shown on reverse engineered diagrams. The XMI Comparator flagged these as differences between the two artefacts.

Additionally, the test cases contained synthetic fields and classes. These facilitate access to the members of inner and nested classes (Sun 2003). These appeared in the reverse engineered design as extraneous attributes and operations which neither appeared in the original design nor in the source code.

5 Future Work

Improvements can be made to the REV-SA tool with respect to several major areas.

An obvious improvement would be to detect and warn when compiler introduced differences are detected instead of flagging such differences as errors.

For example, REV-SA should be modified to detect default constructors and class field initialiser methods, introduced into the Java class file by the compiler.

Java byte code could be parsed directly instead of relying on the Java Reflection package. Doing so would enable REV-SA to resolve the weakly ordered container problem. This can be achieved by detecting the types of objects that are assigned to the container, thereby allowing the specific object type to be shown on the class diagram (Tonella and Potrich 2001).

Additional graph-theory research could also be undertaken to improve auto-placement of symbols and associations on the reverse engineered diagram.

6 Conclusion

Testing of REV-SA has not been exhaustive given contrived test cases utilising only two test subjects. This makes it impossible to generalise the results or quantitatively analyse the efficacy of the described approach as an aid to code inspection.

However, our experience suggests that the use of REV-SA to reverse engineer UML class diagrams from compiled Java class files and compare them to the equivalent forward engineered design artefact has arguably been shown to be a useful approach for verifying software implementation against a given design.

Experience has shown that many design deviations such as spelling inconsistencies, missing methods and classes or object-oriented hierarchies not found in the original design are detected by REV-SA, allowing corrections to be made.

REV-SA testing showed that the CASE tool introduced many differences during forward engineering. Since only one tool was tested, this cannot be said to apply to all CASE tools. However, this was not considered to be a major source of error in designing REV-SA. It was with some surprise that the majority of differences were related to issues of the CASE tool as described in Section 4.2.

REV-SA has demonstrated promise in the verification of Java implementations using the UML metamodel as a basis for comparing forward and reverse engineered artefacts. A description of the algorithm used by REV-SA to compare XMI representations of class files is the principal contribution of this paper. The approach used to reverse engineering class files using the Java Reflection package was also described.

We believe that using a static analysis tool like REV-SA to identify those locations where an implementation deviates from the design should become a regular part of code-inspection preparation (Dunsmore *et al.* 2003).

Reconciling and amending design documents with an implementation is also deemed advisable to avoid later difficulties during code maintenance, ensuring the consistency of all software development artefacts.

7 References

Ackerman, AF, Buchwald, L.S. and Lewski, F.H. (1989): Software Inspections, *IEEE Software*, 6(3):31-36.

Borland: Borland Together Control Center Main Page, http://www.borland.com/together, accessed 3 Sept. 2003.

Bremen: University of Bremen, Idea Homepage, http://dustbin.informatik.uni-bremen.de/projects/idea/, accessed 3 Sept. 2003.

Chidamber, S.R., Darcy, D.P. and Kemerer, C.F. (1998): Managerial user of metrics for object-oriented software: an exploratory analysis, *IEEE Trans. Software Eng.*, August, 1998, 24(8):629-639.

Dunsmore, A., Roper, M., and Wood, M. (2003): The development and evaluation of three diverse techniques for object-oriented code inspection, *IEEE Trans. Software Engineering*, 29(8):677-686.

Fagan, M.E. (1986): Advances in software inspections, *IEEE Trans. on Software Eng.*, 12(7):744-751.

IBM: Rational Rose Home page, http://www.rational.com/products/rose/index.jsp, accessed 3 Sept. 2003.

Kollmann, R., and Gogolla, M. (2001): Application of UML associations and their adornments in design recovery, *Proceedings of the Eighth Working Conference on Reverse Engineering*, October 2001, pp. 81-90.

Kollmann, R., and Gogolla, M. (2002a): Metric-based selective representation of UML diagrams, *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, March 2002, pp. 89-98.

Kollmann, R, Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. (2002b): A study on the current state of the art in tool-supported UML-based static reverse engineering, *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 22-32.

Novosoft: *Metadata Framework and UML Library*, http://nsuml.sourceforge.net/, accessed 3 Sept. 2003.

Ogasawara, H., Aizawa, M., and Yamada, A (1998): Experiences with program static analysis, *Proceedings of the 5th International Software Metrics Symposium*, 20-21 Nov. 1998, pp. 109 –112.

Paderborn: University of Paderborn, Fujaba, http://www.fujaba.de, accessed 3 Sept. 2003.

Robbins, J.E. and Redmiles, D.F. (2000): Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Journal of Information and Software Technology*. 42(2):79-89.

Sommerville. I. (2001): *Chapter 20- Software Testing* in *Software Engineering*, 6th ed., Addison-Wesley, 2001, ISBN 0-201-39815-X, pp. 440-466.

Systä, T., Ping, Yu, and Muller, H. (2000): Analyzing Java software by combining metrics and program visualization, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pp. 199-208.

Sun: *The Java Virtual Machine Specification,* Sun Microsystems, http://java.sun.com/docs/books/vmspec/, accessed 3 Sept. 2003.

Tonella, P., and Potrich, A. (2001): Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers, *Proceedings of the IEEE International Conference on Software Maintenance*, Nov. 2001, pp. 376-385.

w3.org: The Extensible Stylesheet Language Family, http://www.w3.org/Style/XSL/, accessed 3 Sept. 2003.

8 Appendix

The problem posed by Java's container classes is discussed in section 3.2.

The use of the term XMI here refers specifically XMI as supported by the Novosoft UML API (NSUML) and ArgoUML.

The use of the term Java refers to the combination of the standard Java 2 language and the Java Virtual Machine class file format, with no custom extensions.

The following lists information not recoverable from compiled Java class files through the Java Reflection package (java.lang.reflect).

Stereotypes- XMI supports labelling each element with an optional stereotype. Java does not support general stereotyping.

"**abstract" modifier-** In addition to classes and operations, XMI allows packages, interfaces, datatypes and associations to be abstract. Java does not.

"final" modifier- In addition to classes, operations, attributes and associations, XMI allows packages, interfaces and datatypes to be final. Java does not.

Datatype generalization- XMI datatypes can be generalised, whereas in Java they have no place in any strict inheritance hierarchy.

Datatype operations- XMI datatypes can have operations, whereas in Java they are merely acted upon by external operations.

Query operations - XMI allows the designer to flag an operation as a query operation if it makes no change to the state. Java does not.

Operation concurrency- XMI allows an operation to be either "sequential", "concurrent" or "guarded". Java allows a method to be "synchronized", which is equivalent to guarded, but makes no distinction between sequential or concurrent methods.

Parameter name- Both XMI and Java give names to operation parameters, but there is no requirement for a Java compiler to preserve them during compilation.

Parameter direction- XMI allows parameters to have a direction ("in", "out", "in-out" or "return"). Java only makes a distinction between a return value and non-return parameters.

Parameter default values- XMI allows parameters to have default values, but there is no way to achieve this in Java since all parameters must be included in a method call. C++ does not require this. C++ does require default values for parameters not included in function calls to be specified in the function signature.

Attribute initial value- The Java Reflection package cannot distinguish between initial values of zero, "null", "false" and Unicode 0000, and the absence of an initial value.

Association name- XMI allows associations to be named. Java does not.

Association ends- XMI associations have two distinct entities at either end - association ends. In Java, a field, parameter or variable in only one class will often suffice to implement an association.

Association end role- An XMI association end may be supplemented with a role name.

Attribute and association end multiplicity- XMI allows attribute and association multiplicity to take on a range of integers with specified lower and upper bounds. Whether implemented through multiple fields or by an array, Java provides no method for capturing anything but the upper bound. The lower bound is always zero.

Aggregation and composition- XMI makes the distinction between aggregation (a weak whole-part relationship) and composition (an exclusive whole-part relationship). There is no way to extract such information from the Java Reflection package.

Association end ordering- XMI allows an association end to be "unordered", "ordered" or "sorted", depending on the significance of object ordering. The Java Reflection package provides no means to determine this.

Association end changeability- Changeability refers to whether objects can be dynamically added or removed from an association end. XMI supports "changeable", "add only" and "frozen" changeabilities, meaning, respectively, that no objects may be added or removed. Java possesses a limited mechanism for specifying frozen changeability, through use of the "final" keyword, but this cannot be enforced in the general case, especially when associations are implemented using container classes.

Names of generalisations and abstractions- XMI allows textual names to be assigned to generalisation and abstraction relationships. Such names have no counterpart in Java.

Multiple inheritance- XMI supports arbitrary multiple inheritance. Java supports single inheritance only, in addition to the implementation of multiple interfaces.