Refactoring UML Models

Using OpenArchitectureWare to measure UML model quality and perform pattern matching on UML models with OCL queries *

Twan van Enckevort
Xebia BV
twan@roto.org

Abstract

In object oriented software development, the Unified Modeling Language (UML) [20] has become the de-facto modeling standard. UML plays an important role for software factories, in which a high quality abstract UML model is the primary source of input used to generate a working system. While there are many tools that enable assisted refactoring of source code, there are few tools that enable assisted refactoring of UML models.

In order to determine UML model quality for UML models used in code generation projects, a selection of quality metrics has been made. While there are a large number of metrics available to determine code quality, there are only a limited number of metrics applicable to UML models. Most model quality metrics have been derived from code quality metrics [16]. Syntactic and semantic model check rules have been implemented, that allow detection of undesirable model properties. The syntactic model checkers have been derived directly from the UML specification. The semantic model checkers have been derived from a range of anti-pattern descriptions.

We have delivered a prototype that detects undesirable model features in order to test the model improvement capabilities. The prototype contains selected model quality metrics, syntactic and semantic model check rules. Both metrics and rules have been formulated in the Object Constraint Language (OCL) [21], which operates on UML models. The system is built using Open Source tools, allowing easy extensions of the prototype. The effects of suggested repair actions on the model are measurable through the selected model quality metrics and by subjective comparison. The prototype was able to improve model quality for four industry models both by metrics and subjective comparison.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—object-oriented design methods, computer-aided software engineering (CASE)

General Terms Design, Experimentation

Keywords Metrics, model quality, OCL, semantic rules, syntactic rules, UML

1. Introduction

In software development an analysis of requirements leads to a model that describes the system being developed. The Unified Modeling Language currently is the de-facto language used for modeling object oriented software systems [10]. Software Factory driven projects, that follow a Model Driven Architecture (MDA) [19] approach, use the model as the primary source from which the system is generated. This leads to the situation where the quality of the input model directly influences code quality in terms of maintainability and risks of introduction of defects. Software engineering economics dictate that bug prevention is cheaper than bug detection [2]. Modifying a UML model before any code is generated is cheaper than modifying both the model and the generated source code. While there are a multitude of methods and tools available that allow assessing code quality and enable assisted refactoring of code in order to improve its quality, there are few methods and tools to assess and improve UML model quality [4].

The UML class diagram is most frequently used in object oriented software development. It is the best researched diagram in terms of UML model quality metrics [13]. This project adopts well defined quality metrics for the UML class diagram and defines syntactic and semantic rules that can be applied to a UML class model in order to verify the model. A prototype software system was built that can measure the quality of a given UML class model based on selected quality metrics and helps improve the model by im-

* The prototype is available via URL `www.roto.org/public/Pub/prototype.zip`

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

plementing syntactic and semantic model check rules. Evaluation was done against the selected quality metrics and by subjective comparison. The rules set evaluated proved to be successful in improving the UML models.

2. Background

2.1 Software factory

A software factory can be compared with a car factory. In a car factory the prototype is carefully crafted and once finalized it is mass produced on a production line, leaving freedom for individual modifications [11]. Software factories try to achieve the same in software development. Domain specific knowledge is used to create a blueprint for building similar applications all working within the same domain. The UML input model describes the system to be built and is input to the code generation process. For such model to code transformations high quality UML models are required. The UML class diagram is the most frequently used UML diagram in the generation of software using software factories. Therefore this report focuses on these most important models.

2.2 UML model quality metrics

UML models are used to describe a system and focus on certain aspects of that system. For example a model can focus on the services provided or on the user interaction. Thus abstraction is typical for UML models, which implies models are less complete and less precise in describing a system when compared to source code [16]. High quality models can help in preventing undesirable different interpretations of the model. In order to be able to make statements on model quality, which are needed to review the effects of the refactoring effort, suitable metrics should be selected.

The ISO25000:2005 quality model identifies a number of primary uses for software artifacts: Operation, Transition and Maintenance. Operation deals with all aspects of the software being in operation. Transition deals with issues related to moving a system from one environment such as development into another such as production. For both operation and transition uses, the UML model is not used. During the maintenance phase, the UML model is consulted to gather information for the system and updated to reflect changes being made to the system. Since the UML model itself is a design artifact, which is being used in the development phase of a software development project, another high level use emerges: Development [16]. Thus there are two primary uses of interest when selecting metrics to define quality for UML models: Development and Maintenance.

Lange and Chaudron identified a number of different uses for UML models in software development [16]. This paper will focus on UML model quality to support a selected set of purposes: code generation, testing, communication and comprehension. For each modeling purpose Lange and Chaudron define, they listed a number of characteristics for which metrics can be implemented. The selected UML model metrics focus on these characteristics, thus allowing quantification of UML model quality. The selected UML model metrics are listed below:

- The CK metrics suite consists of six different metrics: the weighted methods per class (WMC), the depth of inheritance (DIT), the number of children (NOC), response for a class (RFC), coupling between objects (CBO) and lack of cohesion in methods (LCOM).[6]
  - The WMC metric quantifies the complexity of a class and is defined as the sum of the complexities for each method in a class. In the calculation of this metric, the complexity of each method in a class is considered unity, reducing this metric to a count of the number of methods of a class. A higher number for the WMC metric makes a class more fault prone thus defects are more likely to occur. [23]
  - The DIT metric is defined as the maximum length from a class to the root of the inheritance tree. In Java multiple inheritance is not allowed, thus the calculated DIT is by default the maximum value of the DIT. The opinions on the interpretation of the DIT value differ with regard to fault proneness, speed of development and quality. What is clear is that low values for the DIT are an indication that the re-usability was sacrificed for understand-ability [12]. More research is needed though, to explore the true effect of the DIT metric on model quality.
  - The NOC metric counts the immediate number of sub-classes for a class. The observations for the DIT metric equally apply to the NOC metric. Coad and Yourdon defined a set of object oriented design principles: coupling, cohesion, clarity of design, generalization-specialization depth and simplicity of objects and classes. The use of these Object Oriented design principles leads to a better object oriented design [3]. This is reflected in the values of the DIT and NOC metrics [12].
  - The CBO metric calculates the number of classes a class is related to. This metric is difficult to calculate as it requires code, but it can be estimated by calculating the sum of all class dependencies. It is a measure of class coupling for which the same observation applies as for the NOC metric [18].
  - The RFC metric is defined as the number of methods a class can call. This metric cannot be calculated as part of this dissertation project, as it requires behavioral diagrams.
  - The LCOM metric cannot be calculated as it requires the intersection of instance variables accessed by a method, which is not available in a class diagram.

- The Fan-in and Fan-out metric was proposed by Henry & Kafura and is a measure of reliance [15]. This metric is very similar to the NDepIn and NDepOut metric [12]. For a specific class both metrics are indicators for the time required to understand and modify the class.
  - Fan-in describes how often a class is used in terms of incoming uses, thus being a measure for the amount of classes that rely on this class.
  - Fan-out describes the number of outgoing uses for a class, thus being a measure how much this class relies on other classes.

2.3 UML model quality improvement

For source code there are a number of tools that allow assisted refactoring. In general these tools provide metrics and use syntactic and semantic model checks at the source code level. Improvement of a UML model can be done at the level of syntactic improvements, semantic improvements and pragmatic improvements, similar to the model quality levels defined by Lindland [17]. Syntactic improvements detect violations of the model against its language specification and suggest improvements. For a UML model violations against the UML specification need to be detected. Semantic improvements detect violations of the model against its domain and suggest improvements. Since UML models are generally used for modeling Object Oriented systems, the semantic improvements deal with detecting bad practices in Object Oriented design. Pragmatic improvements deal with correspondence of the model with the target audience' s interpretation and is much more difficult to automate. UML modeling tools in general sometimes provide metrics, partially enforce UML syntax and generally ignore semantics. The UML refactoring tool prototype will implement metrics, syntactic and semantic rules to help improve the UML class model. Using the selected UML class model metrics, it is possible to calculate the initial model quality and track the quality as the model is being re-factored.

Pragmatic model quality is assessed using the Coad and Yourdon design principles [3]. These principles have been verified and lead to a better object oriented design if adhered to. The principles are cited below:

- "Coupling: First, interaction coupling between classes should be kept low, something which can be achieved by reducing the complexity of message connection and decreasing the number of messages that can be sent and received by an individual object. Second, inheritance coupling between classes should be high, achievable by ensuring that each specialization class is indeed a genuine specialization of its generalization class." [3]
- "Cohesion: First, a service in a class should carry out one, and only one, function. Second, the attributes and services should be highly cohesive, i.e., no unused attributes and services and they should all be descriptive of the responsibility of the class. Third, a specialization should actually portray a sensible specialization. It should not be some arbitrary choice which is out of place within the hierarchy creating a less cohesive class due to unrelated inherited features." [3]
- "Clarity of design: First, use of a consistent vocabulary is important. The names in the model should closely correspond to the names of the concepts being modeled. Second, the responsibilities of a class should be clearly defined and adhered to. Furthermore, the responsibilities of any class should be limited in scope." [3]
- "Generalization-Specialization depth: It is important not to create specialization classes which are conceptually not a real specialization, e.g., created for the sake of reuse. Rather an inheritance hierarchy should capture a conceptual taxonomy used to model the problem at hand. Keeping objects and classes simple: First, avoid excessive numbers of attributes in a class. An average of one or two attributes for each service in a class is usually all that is required. Second, *fuzzy* class definitions should be avoided. A class should map to a type of entity in the problem description. All definitions should be clear, concise, and comprehensive." [3]

3. Requirements specification

In order to satisfy the immediate goals of the project, the UML model refactoring tool requires the following primary abilities:

- Calculate metrics for a model, allowing model quality to be established;
- Present the calculated metrics, allowing model quality to be reported;
- Allow querying a model for certain model features, allowing bad model constructs to be found and suggest improvements.

In addition to the primary abilities the tool needs to provide basic model operations. This comprises loading a UML model, editing a UML model, saving a modified model and presenting a graphical representation of the loaded model. The UML model refactoring tool prototype should be able to work with models written in the UML 2 specification. In order to be able to repair the model, graphical UML model manipulation should be possible.

4. Tool selection

The UML model refactoring tool prototype will focus on the ease of creation of a set of selected UML metrics and the ease of creation of a set of model check rules. Therefore a programming language such as Java is less suitable, as it lacks support for querying UML models. A logical choice is OCL, which is an OMG language that allows querying UML models. In the typical sequence of events, a UML model is

loaded from the file system for processing. The calculated metrics are reported in the form of a metrics report. Any model check rules triggered will be reported in a message console. In the course of the writing up of requirements it became apparent that a generation tool would be suitable for implementing the set requirements. Typical MDA tools offer the supporting functionality that is needed for this project. The model is queried in order to generate code for model parts. This functionality can be used to calculate the selected metrics and to search for certain model features. The code generator is used to generate source code by means of a template. A template can also be written to generate a document presenting the calculated metrics. A number of open source MDA tools were investigated: AndroMDA, Eclipse using the Model Development Tools and Eclipse using OpenArchitectureWare.

AndroMDA is a MDA tool that allows generation of complete applications from a UML model. It has an elaborate set of patterns that can be used for code generation. At the time of the investigation it supported UML 1.4 rather than the latest UML 2 standard. Its focus is on code generation and therefore relied on integration with third party UML modeling tools, which requires continuous switching. At the time of the investigation it did not offer Eclipse integration, making the tool less extendable and making it difficult to use the Eclipse modeler. Therefore this tool was dropped from the list of candidates.

The Eclipse IDE with the Model Development Tools plug-ins is a second alternative that was investigated. It supports UML2 and offers integrated UML modeling as well as Eclipse integration. While it offers true OCL queries compliant with the OCL specification, it is not the mainstream solution nor does it offer code generation facilities required for generating a metrics report. The Eclipse IDE with OpenArchitectureWare like the Model Development Tools offers UML2 support as well as integrated modeling and Eclipse integration. Although the OCL implementation deviates from the OCL specification, it is a full implementation with very similar syntax. It is the mainstream modeling tool in Eclipse.

Taking the above in consideration a trial revealed that the OpenArchitectureWare Eclipse plug-in was easier to work with compared to the Model Development Tools Eclipse plug-in, allowing more focus on writing OCL metrics and rules queries. Thus the prototype was developed using Eclipse and the OpenArchitectureWare Eclipse plug-in. OpenArchitectureWare contains all the required basic functionality including the basic functionality that enables metrics calculation, rules processing and report generation.

4.1 Component view

The different tools the prototype depends on are shown in figure 1.

- **Eclipse** is an integrated development environment from the Eclipse foundation, released under the Eclipse Public License. [9] The IDE uses a plug-in mechanism to offer integration with the various projects run under the umbrella of the Eclipse Foundation. The prototype was built using Eclipse platform 3.3 (Europe Edition) using the plug-ins mentioned below.
  - **Eclipse Modeling Framework (EMF)**: The EMF plug-in allows modeling for XMI models and code generation for those models. It also enables interoperability for other plug-ins. Open Architecture Ware makes use of functionality offered by the EMF plug-in
  - **Unified Modeling Language version 2 (UML2)**: UML2 is an open source implementation of the UML2 meta model based on the EMF plug-in. The prototype uses the UML2 plug-in to load sample UML2 models into an EMF repository. This repository is used by Open Architecture Ware to query the model
  - **Graphical Modeling Framework (GMF)**: The Graphical Modeling Framework enables GUI based model manipulation. It is based on the EMF plug-in.
- **Open Architecture Ware**: Open Architecture Ware is a framework for building Model Driven Architecture / Model Driven Software Development tools and is part of the Eclipse Generative Modeling Technologies project. [22] It is built on the Eclipse IDE and supports UML2 models through EMF. Check, Xtend and Xpand are packages that are part of the Open Architecture Ware framework that are used in the prototype to perform metrics calculation, rule checking and report generation

The prototype makes use of a workflow manager, which controls report generation and rules checking. The report generator controls metrics calculation.

As a MDA tool, OpenArchitectureWare supports model editing, model checking and code generation. These actions are controlled through the Workflow Manager, which uses the workflow engine by defining the actions to be executed for a model. The Xtend package provides an editor offering type checking and code completion for OCL constraints and queries. OCL is an OMG standard that allows querying and modifying UML models by means of OCL queries, that operate on the UML meta model. EMF models can be checked using the check package. It performs those checks by means of OCL queries that operate on the loaded UML2 model using the UML2 meta model. In the prototype system, check is used to calculate individual model check metrics and validate model check rules that have been written in Xtend. This is represented in the ModelChecker and MetricsCalculator components. The Xpand package offers an editor for writing code generation templates. These templates are normally used to generate software code for a loaded UML2 model.

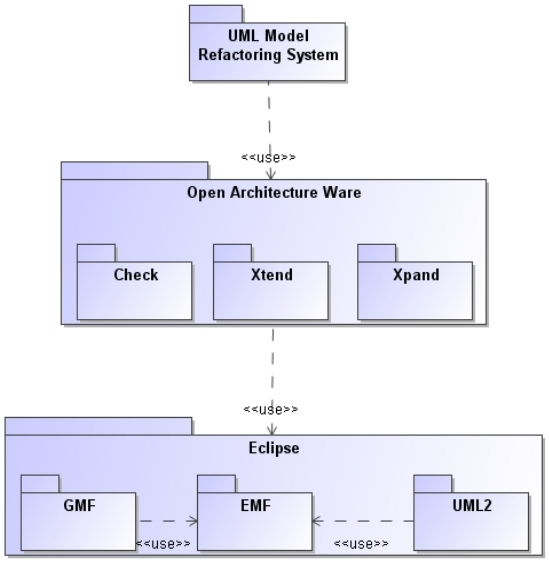

Figure 1. UML Model Refactoring Prototype implementation diagram

The code generation facility can also be used to generate reports by means of writing a report template. The ReportGenerator is a code generation template, where the generation process will result in a report rather than software code.

The models to be used in this project generally were created with the aid of a graphical UML modeler and occasionally by reverse engineering Java code. MagicDraw is a UML modeling tool, which supports reverse-engineering a UML model from Java code as well as fully supporting EMF XMI-import and export. It was used in this project to reverse-engineer Java code into a UML2 model. Since these models cannot be read natively by EMF, they have to be converted into XMI, which is also supported by MagicDraw.

4.2 Code view

Both metrics and rules were implemented in the form of OCL queries. OCL allows constraints and queries to be defined specifically for UML models. Constraints typically are pre-conditions, post-conditions, invariants and guards for a UML model, thus adding information to a model. This makes a model more precise. An example of such a constraint is the limitation of a range for an attribute, such as 'TrainNumber' which should have a value between 1 and 99999. Queries can be used to retrieve information from the model. An OCL query is executed in a context that defines where the statement is valid. An example of such a context is `eAllContents`. The dot operator can be used to access properties of the object. For example `eAllContents.typeSelect(class)`, which selects all classes in a model. Operators similar to Java (such as `+, -, *, /, <, >, and, or`) are present. The `->` operator can be used to access collections such as `size()`, `isEmpty()`, `notEmpty()` and `select(expr)`. These result in a (set of) value(s), such as `object.allFeatures.size();`, which returns the number of features for object. OCL also supports conditional expressions through keywords such as `if`, `then`, `else`, `not` and `or`. These language features are sufficient to implement the selected set of metrics as well as semantic and syntactic model check rules in the form of a set of OCL queries.

The sample metric below calculates the number of methods (wmc) for a class, where the weight of each operation is unity. This metric is part of the CK metrics suite and is an example of a count based metric.

```
int wmc(Class c) :
c.ownedOperation.size;
```

The sample syntactic model check below checks if the class' name is unique within its scope.

```
context Class ERROR "Duplicate class " + name :
namespace.ownedMember.typeSelect(Class).select
(c|c.name == name).size == 1;
```

The model check operates on all classes and checks if within the namespace the name of each class only occurs once.

Semantic model checks in general have been based on adherence to the object oriented design principles defined by Coad and Yourdon [3]. This requires searching the model for certain undesired model fragments: pattern matching. If a fragment is found, the model check rule is triggered. OCL is a single expression language, which supports first order predicate logic using an object oriented syntax on a UML model. It is possible to chain OCL expressions. If a model fragment that needs to be found in a model is written in the form of a decision tree, it is possible to implement such a decision tree in the form of an OCL query. This enables pattern matching on model fragments, which is needed to implement semantic model check rules. An example query shown below finds duplicate operations in two lists of operations by making recursive calls to itself:

```
List[Operation] findOverriddenMethods
(List[Operation] child, List[Operation] parent,
List[Operation] matches) :
    let targetOperation = child.first():
    targetOperation == null ? (matches)
    : (parent.select(o|o.name == targetOperation.name)
    .isEmpty ? (findOverriddenMethods(child.
    withoutFirst(), parent, matches)): (parent.select
    (o|o.name == targetOperation.name).exists
    (o|o.ownedParameter.reject(l|l.direction.toString()
    .toLowerCase()=="return").name ==
    targetOperation.ownedParameter.reject
    (l|l.direction.toString().toLowerCase()=="return")
    .name && o.ownedParameter.reject(ll.direction.
    toString().toLowerCase()=="return").type.
    toString() == targetOperation.ownedParameter.
    reject (ll.direction.toString().
    toLowerCase()=="return").type.toString())
        ?(matches.add(targetOperation)
        ->findOverriddenMethods
```

```
(child.withoutFirst(), parent, matches))
 : (findOverriddenMethods
 (child.withoutFirst(), parent, matches))));
```

The findOverriddenMethods method returns a list of matches to the caller, where the list is empty if no matches were found in the model. It uses a list called matches, which initially is empty, to maintain state while recursively searching for matches by comparing each element in the child list to all elements in the target list until the child list has been completely searched.

While the OCL specification contains a rich function set, these are insufficient. Therefore extensions were added to the project, that extend the base functionality offered in the OCL. For example the OCL specification enables traversing an inheritance relationship to find the generalizations for a class. But the OCL specification does not enable traversing an inheritance relationship to find all children for a class. Since this is a frequently used function, it was added as an extension, so it can be called from other OCL queries easily. Thus extensions add frequently used additional functionality required in metrics calculation and rules detection, which can be called more easily. Enabling the search for children of a class is enabled by the following extension:

```
cached Set[Class] children(Class c) :
  c.eRootContainer.eAllContents.typeSelect
(Class).select(c1| c1.parents().contains(c));
```

Here the root container for a class is searched to find all classes that have the target class as its parent, thus returning a collection containing all children for the target class.

5. Implementation

5.1 Metrics

In section 2 the selected metrics have been listed. The code generator creates a formatted report, making use of a report template. From within the template calls to the individual metrics are made while traversing the input UML model. The report shown in figure 2 shows the generated output for a test model.

5.2 Syntactic rules

Table 1 lists the implemented syntactic rules. The element column indicates the model element to which a rule applies: (P)ackage, (C)lass, (O)peration, (A)ssocation/Generalization, (I)nterface, (D)atatype and (E)numeration. The reference column reflects the (M)andatory or (R)ecommended practice from the UML specification or rules drawn from my own experience with code generators (O). A number of syntactic rules have been derived from own experience and therefore are discussed more extensively:

- There are several rules that check if a model element is a programming language identifier. In code generation projects, the name of model elements is used in code generation. In case a model element is a reserved keyword in a specific programming language, using such a reserved keyword as the name of a model element may lead to compilation errors in the generated code. While a code generator may provide name mangling functionality, this cannot be relied upon and thus it is best to not use any reserved keywords when naming model elements.
- There are a number of rules, that check if model elements are named. For most model elements, with the exception of operations, associations and generalizations, the name is used in code generation. Having an empty name may lead to compilation errors in the generated code. While a code generator may assign a random name, this cannot be relied upon and thus it is best to always name model elements.
- Rule P5 states a package name should be all lowercase. This rule is specific to Java code generation projects, where it is a convention. While not checked as part of this rule set, the Java language also recommends using top level domain names in package names to avoid name clashes.
- Rules A1.5 and O3.5 ensure attributes and parameters are always typed. For typed languages, such as C++ and Java, not providing a type will lead to compilation errors in the generated code.
- Rule AG4 states that associations that are not navigable in any direction have no practical use and should be avoided. In my work on code generators I have seen that there is first of all no technical reason to have non-navigable associations. A non-navigable association links two classes together, where the individual classes are not to have any knowledge of the other class. Thus instances of classes that take part in such a relationship

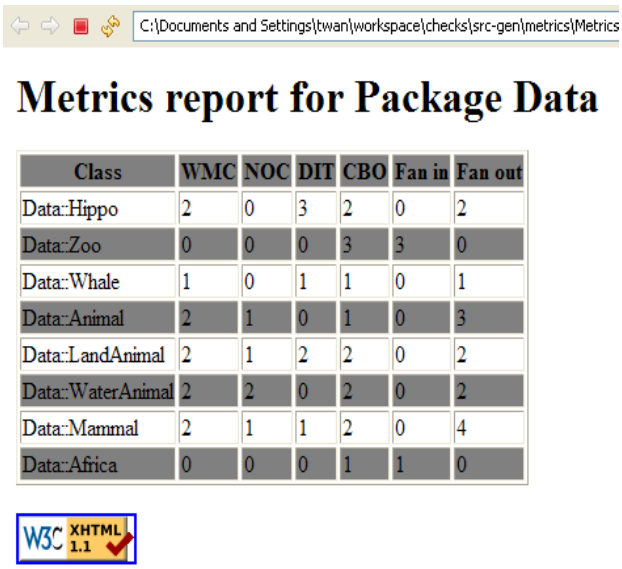

Figure 2. UML Model Refactoring Prototype sample report

cannot be reliably maintained without violating this principle. In addition I have not seen a sound business case for non-navigable associations, hence this warning.

5.3 Semantic rules

Table 2 lists the semantic rules that have been implemented in the UML model refactoring prototype.

From these rules, rules P2, P18, P19 and P20 require more effort to be calculated than the others and thus these are discussed separately.

Rule P2 calculates whether a class is part of a cyclic dependency. In order to determine cyclic dependencies, the model is traversed starting from the start class, walking through all navigable associations, until the path that can be walked over navigable associations ends or the start class is encountered again. The navigableAssociations (class c) function returns a set of associations which can be navigated away from the source class. Recursive calls to the traverse function are used, to traverse these associations and visit the target class with the help of the targetClass function. Self references are valid samples of circular dependencies hence care is taken not to invalidate models having self references.

Rule P18 searches for cyclic dependencies in packages. It employs a similar search mechanism as used for rule P2, but instead of making use of the navigableAssociations function, it makes use of a packageDependencyCount function, which returns the number of dependencies between a source and a target package. The package dependency count is defined as the number of UML dependencies between packages, the number of class attributes and class operation parameters in Package P' that have a type set to a class that is contained in package P, the number of associations, aggregations and compositions from a class in package P to a class in package P', the number of UML dependencies and usage links between classes in package P and classes in package P', the number of classes in package P that are a child of a class contained in package P' and the number of classes in package P that implement an interface contained in package P' [24].

Rule P19 verifies if a package P depends on a less stable package P'. The package stability is calculated as the ratio of the afferent coupling divided by the sum of the afferent coupling and the efferent coupling [24]. Afferent coupling is calculated as the total number of dependencies from all other packages in a model to a package P. The efferent coupling is defined as the total number of dependencies from package P to all other packages in the model using the packageDependencyCount function used in rule P18. Rule P20 checks if a package P depends on a less abstract package P. The package abstractness is calculated as the ratio of abstract classes and interfaces in the package to the total number of interfaces and classes in the package [24]. Dependant packages are found using the package dependency count function used for rule P18.

During testing performance problems were encountered, where large models would take more than 1 hour in order to be processed on a machine with a 1.3 GHz centrino processor and 1.5 Gb of memory. Repeating the test on a 2.6 GHz intel core duo processor and 4 Gb of memory reduced this processing time to 30 minutes. Since a large number of queries is actually performed more than once, caching was enabled for those queries that do not modify the state. This alleviated the performance problem encountered during testing.

6. Evaluation and results

The aim of this project is to improve UML models based on syntactic and semantic model check rules. For the evaluation of this project, a number of models were subjected to metrics calculation and rules validation using a prototype. Based on the rules triggered, the models were improved and again subjected to metrics calculation. Thus UML model validation rules were evaluated by comparing the metrics results before and after model improvement. Additionally the refactored models were evaluated by subjective comparison of the models before and after refactoring by a UML expert. For subjective comparison of the models before and after refactoring, the models were assessed keeping the Coad and Yourdon design principles listed in section 2.3 in mind.

6.1 Evaluation of model 1: a social security benefits application

The UML class model was reverse-engineered from a Java application that supports social security benefits calculation for a governmental organization. The reverse-engineered model was used for metrics calculation and rules evaluation. Since the model was reverse-engineered, it contained a large number of classes from the default Java library, which were erased from the metrics report. The model was updated using the hints provided by the triggered rules. Relevant results are summarized in table 3.

The before and after results from table 3 show that the WMC metric for classes *NormatieveAangifte* and *Inkomstenperiode* have been significantly reduced due to refactoring, making these classes more maintainable. As a result of this refactoring CBO, fan-in and fan-out increased by 1, making it marginally more difficult to understand the class due to one additional association. For the classes *InkomstenVerhoudingInitieelEvent, InkomstenVerhouding, TijdvakCorrectie* and *InkomstenVerhoudingCorrectieEvent,* time required to understand the classes decreased. This is opposite for parent class *AbstractSafeOrUpdateInkomstenVerhoudingEvent* The same applies to classes *TijdvakAangifte* and *NatuurlijkPersoon* which improve against *AbstractTijdvakEvent.* The inheritance structure was not changed as part of the refactoring, as indicated by the DIT metric showing no changes. The maximum value of the DIT metric in the model is 2 and generally being 0 for most classes. While it is pos-

Table 1. Implemented syntactic rules

Rule	Element	Reference	Description
P1	P	M	A package should at least contain one or more classes
P2.1	P	M	The name of a class should be unique in the class' namespace
P2.2	P	M	The name of an interface should be unique the interface's namespace
P2.3	P	M	The name of an association, if named, should be unique in the association's namespace
P3	P,C,O,A,I,D,E	R	An element should have a name
P4.1	P,C,O,A,I,D	O	The name of an element should not be a Java keyword
P4.2	P,C,O,A,I,D	O	The name of an element should not be a C++ keyword
P5	P	O	The name of a package should be all lowercase
C1	C	M	A class should not have duplicate attributes
C2	C	M	A class should not have duplicate operations
C4.1	C, I	R	The name of an element should start with a capital
C5	C	M	A class marked as leaf should not have any children
C6	C	M	Cyclic inheritance is not allowed
O1.3	O	R	If the name of the operation does not match the class' name (thus not being a constructor), its name should not start with a capital
O2	O	R	An operation should have at most one return parameter
O3.1	O	M	The parameter name for an operation should be unique
O3.2	O	M	All operation's parameters should be named
O3.5	O	O	All parameters of an operation should be typed. Untyped parameters leads to compilation errors for code generation projects
A1.2	A	R	Each attribute should start with a lowercase character
A1.5	A	O	Each attribute should be typed. Untyped attributes leads to compilation errors for code generation projects
AG1	A	M	For a binary association there should be no more than one composite or shared aggregation end
AG2	A	M	All association ends must be connected
AG3.1	A	M	All generalizations must have a parent
AG3.2	A	M	All generalizations must have a child
AG3.3	A	M	The parent and child for a generalization should have a matching meta type
AG4	A	O	Associations that are not navigable in any direction have no practical use and should be avoided
I1.5	I	M	An interface should only contain public operations
I1.6	I	M	An interface should only have public attributes
E1.1	E	M	An enumeration should at least have one literal

sible that there are no more inheritance relationships in the model, it may also be the case that the model and originating code is structured for readability rather than maintainability. Based on the metrics, the model was improved slightly.

Looking at the model subjectively, the following observations can be made: Some classes contained an excessive number of attributes- and methods, which partially was alleviated by refactoring the individual classes. Unfamiliarity with the model made it difficult to further refine the model by creating genuine inheritance relationships. Hence the inheritance coupling is lower than might have been achievable in this model. The cohesion and clarity of design in the model was high, as each service based on the naming performed one function only, there were no unused attributes, attribute naming was clear for those fluent in the business domain and inheritance relationships seemed genuine. The generalization-specialization was sufficient, with each generalization-specialization seeming genuine. The naming of services' and class' attributes was clean and hence not changed. There were classes with excessive number of attributes. These were adjusted as far as possible. The classes mapped seemed to map to entities in the business domain as far as it was possible to judge using the business domain knowledge. Summarizing, the model from a subjective point of view was improved in terms of class size.

6.2 Evaluation of model 2: Albatross Airlines Fleet Management model

The UML model describes the fleet management module of an airline. The model was updated according to the triggered rules. Relevant results have been summarized in table 4

Table 2. Implemented semantic rules

Rule	Reference	Description
P1	[5]	A God class is a class that has more than 60 operations and/or attributes. Such a class has too many responsibilities and is hard to create, test and maintain and thus should be refactored
P2	[1]	Classes should not be part of a cyclic dependency cycle. This leads to code that is difficult to understand and maintain
P3	[7]	Classes that have no instance variables nor any associations that are navigable away from itself, this class could be replaced by a singleton. The use of a singleton is advised for classes that for which only one instance should exist, thus the class is responsible for its own creation and providing a global access point
P4	[14]	If an abstract class only contains abstract public operations and final static public fields, it could be replaced by an interface
P5	[14]	A child class should not hide an attribute of a parent class
P6	[24]	An abstract class must have at least one concrete child class, unless the class is part of a framework, where the user of the framework is expected to extend this abstract class
P7	[14]	A concrete method should not be overridden by an abstract method
P8	Experience	Class has more than one parent, which is allowed in C++, but is not allowed in Java. Note the UML specification allows multiple inheritance
P9	[24]	An abstract class has a parent class that is not abstract
P10.1	[24]	An abstract class should have abstract methods
P10.2	[14]	An abstract class should not have public constructors
P11	[24]	A non constant attribute is public. This violates the information hiding OO principle, as uncontrolled read / write access to the class' attribute is given
P12	[24]	An operation is abstract while the owning class is not abstract
P13	[24]	An operation has five or more parameters, which could destabilize the operation's signature
P14	[24]	A class contains get/has/is methods, that probably should be marked as query
P15.1/2	[14]	Two or more direct sub-classes of a class or interface define an attribute having the same signature, which is not defined on the parent and could be refactored to lift this attribute to the parent
P16.1/2	[14]	Two or more direct sub-classes of a class or interface define an operation having the same signature, which is not defined on the parent and could be refactored to lift this operation to the parent
P17	[14]	Inherited static methods should not be overridden in the child class
P18	[24]	There should not be cyclic dependencies on the package level
P19	[24]	A package depends on a less stable package. This is a maintenance risk and refactoring is recommended
P20	[24]	A package depends on a less abstract package. Refactoring is recommended to make the less abstract package more abstract

Table 3. Summarized findings for the Social Security Benefits Application model

Class name	$\mathbf{WMC_{before}}$ $\mathbf{WMC_{after}}$	$\mathbf{NOC_{before}}$ $\mathbf{NOC_{after}}$	$\mathbf{DIT_{before}}$ $\mathbf{DIT_{after}}$	$\mathbf{CBO_{before}}$ $\mathbf{CBO_{after}}$	$\mathbf{Fan\text{-}in_{before}}$ $\mathbf{Fan\text{-}in_{after}}$	$\mathbf{Fan\text{-}out_{before}}$ $\mathbf{Fan\text{-}out_{after}}$
NormatieveAangifte	100 / 42					
Inkomstenperiode	74 / 52					
Inkomstenverhouding Initieelevent				3 / 2	1 / 0	2 / 0
InkomstenVerhouding					37 / 35	
TijdvakCorrectie					2 / 1	
InkomstenVerhouding CorrectieEvent				2 / 1	2 / 1	3 / 2
AbstractSafeOrUpdate InkomstenVerhoudingEvent	8 / 9					5 / 9
TijdvakAangifte					2 / 1	
NatuurlijkPersoon					21/20	
AbstractTijdvakEvent						3 / 4

The before and after results show that with one exception the WMC metric was not changed for any of the classes. The WMC change was attributed to the removal of an occurrence of multiple inheritance. There were no excessive

Table 4. Summarized findings for the Social Security Benefits Application model

Class name	WMC_{before} WMC_{after}	NOC_{before} NOC_{after}	DIT_{before} DIT_{after}	CBO_{before} CBO_{after}	Fan-in$_{before}$ Fan-in$_{after}$	Fan-out$_{before}$ Fan-out$_{after}$
SeatsUI			5 / 4			
CompleteOverhaulUI			3 / 2			
LightOverhaulUI			3 / 2			
MaintenanceUI			2 / 1			
SeatReconfigurationUI			3 / 2			
EmergencyUI			3 / 2			
SeatConfigurationUI			4 / 3			
Maintenance				16 / 14	16 / 15	8 / 7
Aircraft				16 / 11		
SeatConfiguration	5 / 3		6 / 7	12 / 13		

amount of attributes and services in any of the classes. The NOC metric did not change for any of the classes. The DIT metric did change, indicating that in a number of occasions generalizations were removed, leading to a more understandable model. This was the case for classes *SeatsUI, CompleteOverhaulUI, LightOverhaulUI*, *MaintenanceUI*, *SeatReconfigurationUI*, *EmergencyUI* and *SeatConfigurationUI*. In 25 percent of the classes, values for CBO, fan-in and/or fan-out were reduced, leading to a more understandable model. This was most significant in classes *Maintenance* and *Aircraft*. An exception was the *SeatConfiguration* class, where values for CBO and fan-in were slightly higher. Based on the metrics, the model was improved slightly.

Looking at the model subjectively, cohesion was less high than the previous model: while each service seemed to carry out one single function within the abstraction level the business is presented in the model, the model in a real life situation could have been a bit more fine grained considering the size of the operation represented by the model. Although naming was clear, it was not ready for code generation, which is why the model (class names, service names and attribute names) was heavily renamed. It would have been beneficial if name checking rules would have been more extensive towards code generation. The deletion of attributes and moving of attributes to more suitable classes contributed to keeping classes simple and improving the clarity of the design as well as cohesion. In the model, an occurrence of multiple inheritance was removed as it was unnecessary, thus improving the coupling and cohesion of the model. Summarizing subjectively the model has improved.

6.3 Evaluation of model 3: Protein Identification Application model

The UML model describes a protein recognition program. The model was updated according to the triggered rules. The summarized results are shown in table 5.

The metrics show that the WMC metric did not change. The reason for this is the fact that the model did not contain any operations, effectively making this model a data model. The NOC metric changed for the *AnnotableData* class, due to the removal of an occurrence of multiple inheritance. Based on the metrics this makes the model slightly less object oriented. The DIT metric did not change, indicating the structure of the model did not change. For seven classes, values for CBO, fan-in and/or fan-out were reduced, leading to a more understandable model. On the other hand in three cases values for CBO, fan-in and fan-out increased, making these classes more difficult to understand. This was most significant in the *CombinedPeakLists* class, which was affected due to the removal of a circular dependency in the model, explaining the sacrifice in understandability. Based on the metrics, the model was improved slightly.

Looking at the model subjectively, cohesion was not very high, as there were no operations defined in the model. For code generation purposes, the model should be more fine grained, reflected in the absence of operations. Naming of the model was clear, which was confirmed by a low number of naming rules being triggered by the model. Coupling could be better. The model has a large number of specializations, which seem genuine for the problem domain. In the model, an occurrence of multiple inheritance was removed, thus improving the coupling and cohesion of the model. The *CombinedPeakLists* class became less understandable, due to the removal of circular dependencies in the model. The clarity of the design could be improved by adding more detail to the model in the form of operations. Summarizing subjectively the model has improved.

6.4 Evaluation of model 4: Delivery model

The UML model describes a delivery model. The model was updated according to the triggered rules. The summarized results are shown in table 6. While the model triggered a large number of rules, a lot of rules were actually triggered by other diagrams than the class diagram. Thus appendix O only lists those rules triggered that are applicable to the class model. Based on the metrics, the model did not improve as is seen by increasing values of fan-in and fan-out for a

Table 5. Summarized findings for the Social Security Benefits Application model

Class name	WMC_{before} WMC_{after}	NOC_{before} NOC_{after}	DIT_{before} DIT_{after}	CBO_{before} CBO_{after}	$Fan\text{-}in_{before}$ $Fan\text{-}in_{after}$	$Fan\text{-}out_{before}$ $Fan\text{-}out_{after}$
PeptideHypothesis				4 / 3		
PeakList				6 / 5	6 / 5	11 / 9
IdentificationHypothesis						5 / 6
ResultRef				2 / 1	2 / 1	3 / 2
PeakListFile						10 / 8
ResultsRef					2 / 1	
UserModification						6 / 7
CombinedPeakLists				3 / 5	3 / 5	4 / 9
AnnotableData		5 / 4		5 / 4	0 / 1	

number of classes. Thus based on metrics, the model did not improve.

Table 6. Summarized findings for the Social Security Benefits Application model

Class name	$Fan\text{-}in_{before}$ $Fan\text{-}in_{after}$	$Fan\text{-}out_{before}$ $Fan\text{-}out_{after}$
InsertDeliveryNote		3 / 4
DeliveryItems		3 / 5
DeliveryNote	5 / 3	4 / 6
Customer	4 / 3	2 / 4
Product		1 / 2

Subjectively, the cohesion of the model has improved. While attributes and method parameters were named properly, most were not typed, making the model unclear. Based on the rules triggered, all attributes and parameters were typed. The typing of method parameters explains the increased values for both fan-in and fan-out. A cyclic dependency between two classes was removed, leading to slightly better coupling. Thus subjectively this model did improve.

6.5 Rules evaluation

For the syntactic rules implemented, most were of limited value for the test models. This was expected since the models were created in MagicDraw; Either by modeling or by reverse engineering from code. MagicDraw enforces cross diagram consistency and forbids UML constructs that are forbidden according to the UML specification from being created. This is not the case for all modeling tools on the market and hence these syntactic rules could be valuable for models created by these modeling tools. For code generation projects, the naming rules are of particular value and could in fact be extended for specific languages as well as coverage of more detailed language specific conventions.

For the semantic rules implemented, rules P3, P7, P11, P12 and P17 were not triggered. In practice rules P1, P2, P13, P18, P19 and P20 provided valuable hints during refactoring. For rule P2 and P18 to be used effectively, it would have been beneficial to print the cyclic dependency rather than printing each class / package separately. Rules P19 and P20 could be improved to print values for stability and abstractness. While naming and typing subjectively improved the model, this was not reflected in metrics values. Thus it is important to assess a model not purely on metrics alone. Based on the test results, larger models may benefit more from metrics calculation and rules validation than smaller models.

While the selected rules were able to improve all class models, they also resulted in a large amount of unwanted rules being triggered for these models. The reason for those unwanted rules was the presence of other diagrams in the UML model. These diagrams were also parsed in rules validation. Some extensions could be fine tuned to ensure interfaces of the UML class are rejected. This would for instance have prevented all class' interface types from being included in metrics calculation and rules validation.

From a development perspective, writing metrics and rules in a technology that is very much related to the UML 2 meta model, makes it easier to focus on the problem at hand. Once a significant set of OCL extensions was written, creating new metrics and rules became significantly easier due to the possibility of re-use. Performance problems were alleviated using the cached statement for all read-only queries, thus abstracting the performance problem from the technology very effectively. Some extensions could be further improved for performance, for instance by short circuiting searches as soon as a positive match is encountered.

7. Conclusions and further work

Refactoring a model by means of rules has proven an effective method of refactoring a UML class model as described in section 6. The concept of using OCL queries to write up both metrics and rules has proven an efficient method of implementing both metrics and rules. The before and after refactoring metrics generated, needed to be manually compared. For the model developer it would be better to provide a visual comparison of metrics before and after refactoring. This would require additional research to find desirable values for the selected quality metrics. The number of metrics

could also be extended to include those convenient for refactoring (package stability, class abstractness) as well as and those operating on the other UML diagram types [12].

Since this project was limited to the UML class model, it was not possible to detect anti patterns for the full set of UML design patterns. Including metrics and rules for all UML diagrams would have enable detection of most anti patterns [4]. While stereotypes exist for UML design patterns, these are not standardized nor part of the UML specification. It would be beneficial if there was formal support in UML for design patterns, as this would make the alternative (name based) searches more reliable [8]. This could be used to selectively search the model for anti-patterns based on initial searches for patterns.

The tooling was effective in enabling writing metrics and rules once the rule developer is familiar with the UML 2 meta-model. The rules developed could be re-written for inclusion in the Eclipse Model Development Toolkit. This Eclipse plugin is used by most Eclipse modeling plugins and hence it would enable not only OpenArchitectureWare users but all Eclipse plugins connecting to the Model Development Toolkit to make use of them out of the box in modeling projects. It would be beneficial if, for those rules where this is possible, to enable automated model repair actions. The rules could in future be extended to include other diagrams, effectively enabling both searches for general design patterns and cross-checking different diagrams for consistency.

Acknowledgments

This project was performed in partial fulfillment of the requirements for the degree of Master of Science at the University of Liverpool. I thank all individuals who were and / or are involved in providing the open source libraries that I used in this project. I thank Peter Vermeulen from Capgemini for sponsoring this project.

References

[1] S. Ambler. *The elements of UML style*. Cambridge University Press, New York, NY, 2003.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, Upper Saddle River, NJ, 1981.

[3] L. Briand, C. Bunse, and J. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *Software Engineering IEEE*, 27(6):513–530, June 2001.

[4] L. Briand, Y. Labiche, and A. Sauve. Guiding the application of design patterns based on uml models. In *22nd IEEE International Conference on Software Maintenance*, pages 234–243. IEEE Computer Society, September 2006.

[5] W. Brown, R. Malverau, H. McCormick III, and T. Mowbray. *Anti-patterns - refactoring software, architectures and projects in crisis*. Wiley Computer Publishing, New York, NY, 1998.

[6] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE transactions on software engineering*, 20(6):476–493, June 1994.

[7] Cunningham and Cunningham. *Portland pattern repository*. http://c2.com/ppr/, 2005.

[8] M. Elaasar, L. Briand, and Y. Labiche. A metamodeling approach to pattern specification and detection. In *Model Driven Engineering Languages and Systems*, pages 484–498. Springer, Berlin / Heidelberg, September 2006.

[9] E. Foundation. *Eclipse IDE*. http://www.eclipse.org/downloads/, 2008.

[10] L. Fuentes-Fernandez and L. Vallecillo-Moreno. An introduction to uml profiles. *Upgrade*, 5(2):6–13, April 2004.

[11] V. Fuller and D. Upton. *Wipro Technologies: The Factory Model*. Harvard Business Online, Boston, MA, 2005.

[12] M. Genero, M. Piattini, and C. Calero. A survey of metrics for uml class diagram. *Journal of Object Technology*, 4(9):55–92, November 2005.

[13] M. Genero, M. Piattini-Velthuis, J. Cruz-Lemus, and L. Reynoso. Metrics for uml models. *Upgrade*, 5(2):43–48, April 2004.

[14] Gronback. Model validation: Applying audits and metrics to uml models. In *BorCon 2004*. Borland Corporation, 2004.

[15] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE transactions on software engineering*, 27(5):510–518, September 1981.

[16] C. F. Lange and M. R. Chaudron. Managing model quality in uml-based software development. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP 05)*, pages 7–16. IEEE Computer Society, September 2005.

[17] O. Lindland, G. Sindre, and A. Sølvberg. Understanding quality in conceptual modeling. *Software IEEE*, 11(2):42–49, March 1994.

[18] J. McQuillan and J. Power. On the application of software metrics to uml models. In *Models in Software Engineering, Workshops and Symposia at MODELS 2007 Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, pages 217–226. IEEE Computer Society, October 2007.

[19] Object Management Group. Technical guide to model driven architecture: The mda guide v1.0.1, omg/03-06-01 version. June 2001.

[20] Object Management Group. Unified modeling language, formal/2005-07-05 version. July 2005.

[21] Object Management Group. Object constraint language specification, formal/2006-05-01 version. May 2006.

[22] OpenArchitectureWare. *Platform for model-driven software development*. http://www.openarchitectureware.org/, 2008.

[23] R. Subramanyam and M. Krishan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE transactions on software engineering*, 29(4):197–310, April 2003.

[24] J. Wüst. *SDMetrics: The software design metrics tool for UML*. http://www.sdmetrics.com/, 2005.