Tool to Measure and Refactor Complex UML Models

Tamás Ambrus and Melinda Tóth, Eötvös Loránd University
Domonkos Asztalos and Zsófia Borbély, ELTE-Soft Nonprofit Ltd

Modifying and maintaining the source code of existing software products take the majority of time in the software development lifecycle. The same problem appears when the software is designed in a modeling environment with UML. Therefore, providing the same toolchain that already exists in the area of source code based development is required for UML modeling as well. This toolchain includes not just editors, but debugging tools, version controlling systems, static analysers and refactoring tools as well. In this paper we introduce a refactoring tool for UML models built within the Papyrus framework. Beside the transformations, the tool is able to measure the complexity of UML models and propose transformations to reduce the complexity.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**] Model Validation and Analysis; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.2.m [**Software Engineering**] Miscellaneous

Additional Key Words and Phrases: model quality, UML model, metrics, refactoring, bad smell detection, Papyrus, EMF

1. INTRODUCTION

Using UML modeling for designing a software product is heavily used by the industry. However the tool support for model based development have not reached the same level as the tool support of source code based development. Our goal was to provide a tool to support refactoring and static analysis of UML models, which were developed in the open source modeling framework, Papyrus [Papyrus 2014].

There are tools, such as EMF Refactor [Arendt et al. 2010], that targets refactoring of EMF models. This tool provides an extensible framework for defining EMF model transformations and also model metric calculations. Several class refactorings and metrics have been defined in EMF Refactor. Therefore we based our tool on this framework.

The main contributions of our work are the followings. (i) We have built a refactoring tool for Papyrus models based on EMF Refactor. (ii) We have implemented several state machine based refactorings. (iii) We have defined well-known model complexity metrics for state machines and introduced some new metrics to measure the models. (iv) We have implemented bad smell detectors and refactorings to reduce the complexity of the models.

The rest of this paper is structured as follows. In Section 2 we briefly introduce EMF Refactor and Papyrus. Section 3 illustrates the usage of our tool with an example at first, and then Sections 4, 5 and 6 present all of the features. Finally, Section 7 presents some related work and Section 8 concludes the paper.

This work is supported by the Ericsson-ELTE Software Technology Lab.
Authors' addresses: Tamás Ambrus, Melinda Tóth, Eötvös Loránd University, Faculty of Informatics Pazmany Peter setany 1/C, H-1117 Budapest, Hungary email: ambrus.thomas@gmail.com, tothmelinda@elte.hu Domonkos Asztalos, Zsófia Borbély, ELTE-Soft Nonprofit Ltd, 1117 Budapest, Pazmany Pater setany 1/c email: asztalos.domonkos@eltesoft.hu, leona.thet@gmail.com

In: Z. Budimac, Z. Horváth, T. Kozsik (eds.): Proceedings of the SQAMIA 2016: 5th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Budapest, Hungary, 29.-31.08.2016. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

2. BACKGROUND

We chose our product to be an Eclipse extension due to several decisions. Therefore it can build upon two other extensions: EMF Refactor and Papyrus. Since EMF Refactor is open source we can contribute our improvements when we reach a bigger milestone in this project.

2.1 Papyrus

[Papyrus 2014] is an open source model-based Eclipse extension. It can show the UML diagrams in a view in Eclipse. It also provides an other view for the semantic elements only, this is a kind of outline named model explorer. These two help users to edit all types of UML diagrams.
Model explorer and GUI editor work synchronously, meaning:
- clicking on an element on GUI should select the same element in model explorer,
- selecting an element in model explorer should select the same element on GUI,
- the appearing context menu should equal for elements in both views.

Modifying the underlying model programatically can cause differences in the separate views that must be handled manually.

2.2 EMF Refactor

[EMFRefactor 2011] is an open source tool environment for supporting model quality assurance process. It supports metrics, refactorings and smells of models based on EMF (Eclipse Modeling Framework). It basically builds upon the **org.eclipse.ltk.core.refactoring** Java package which supports semantic preserving workspace transformations [Arendt et al. 2010].
There are many predefined metrics, refactorings and smells for class diagrams [Arendt and Taentzer 2010]. This results a stable, useful tool to design reliable, easy-to-understand class diagrams. Preference pages are provided to the refactorings and also to the metrics and smells. These preference pages contain a list about the defined refactorings, etc. For the metrics, each item is related to a category. Since categories come from UML elements (like 'Class'), it is also easy to expand the tool with self defined ones (e.g. 'State'). The aim of the preference pages is that users can choose items they want to work with. For example, marking a refactoring means that it can occur in the suitable context menu of a model element. Context menu appears if the user clicks (with the right mouse button) on an element. The context menu filters accessible items based on the selected elements automatically. For example, while editing a state chart, class diagram refactorings are omitted from the list. It does not guarantee passing preconditions though, it makes only suggestions depending on the type of the selected elements.
The result of the selected metrics appears in a view named Metric Result View. This view does not follow the model changes, in order to have up to date measurements users need to re-run metrics on the changed model. Users can run metrics from the context menu of a selected element: in this case a subset of the selected metrics (based on the type of the selected element) will be evaluated. The result is a list which contains the name and value of the metrics.
New metrics, refactorings and smells can be added using the proper extension points.

3. TOOL DEMONSTRATION

It is hard to decide whether the quality of a model is adequate. Although we can estimate the understandibility and maintainability by looking at it, metrics are essential tools of quality measurement. For a UML model, we can define numbers that describe it in details, such as number of elements, number of connections, etc., and have conclusions by examining the connections between them. This may be a very exhaustive process, thus it seems to be a good idea to use a tool for that.
The tool we have been developing is an extension of EMF Refactor. By using our tool, the user can use

predefined metrics that may contain information about the quality and complexity of a model. Metrics may show us bad design if the number is too large: by defining a threshold, besides detecting smells in the model, we also can eliminate them as they are in connection with specific refactorings, this way we can improve the quality of the model.
This tool also gives us proof that we improved the quality as the predefined metrics may show lower numbers and smells may disappear.

3.1 Example

To demonstrate our tool we use the example presented in [Sunyé et al. 2001]. A phone state diagram was described where the user can start a call by dialing a number, the callee can answer it if not busy and the two participants can talk until hanging up the phone. In the example, we got a flat state machine, therefore after we created the diagram that can be seen on Figure 3, we noticed that its quality can be improved: there are a lot of transitions with the same trigger that all goes into state Idle. This problem can also be detected by using smells. If we select the project and open Properties window, we can set the proper thresholds of smells in EMF Quality Assurance/Smells configuration (Figure 1). A suitable smell is Hot State (Incoming Transitions), that marks those states that have more incoming transitions than the threshold. If we set up the configuration as in Figure 1, then calculate all smells

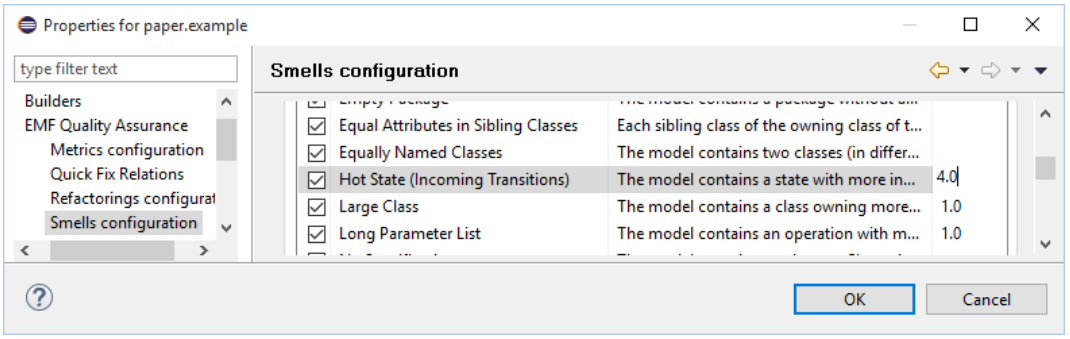

Fig. 1. Configuration of model smells. Default threshold is 1.0, we set the threshold for Hot State smell to 4.0.

(right click on the model, EMF Quality Assurance/Find Configured Model Smells), it finds Idle as a smelly state (Figure 2). We can eliminate it in two refactoring steps: group the belonging states into a

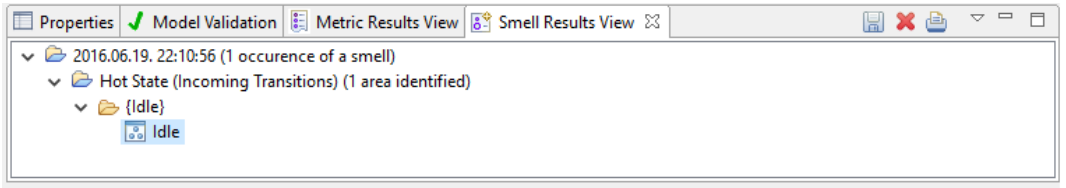

Fig. 2. Smell Result View of the mobile phone call state machine. Number of the incoming transitions of state Idle is above the threshold.

composite one and fold their transitions into a single one. You can see the result of these steps in Figure 4. By eliminating the similar transitions, we have got a simpler, clearer diagram. Moreover, the result can be measured: although the deepness of the state chart has increased, less transitions means less cyclomatic complexity, which is a good approximation for the minimum number of test cases needed.

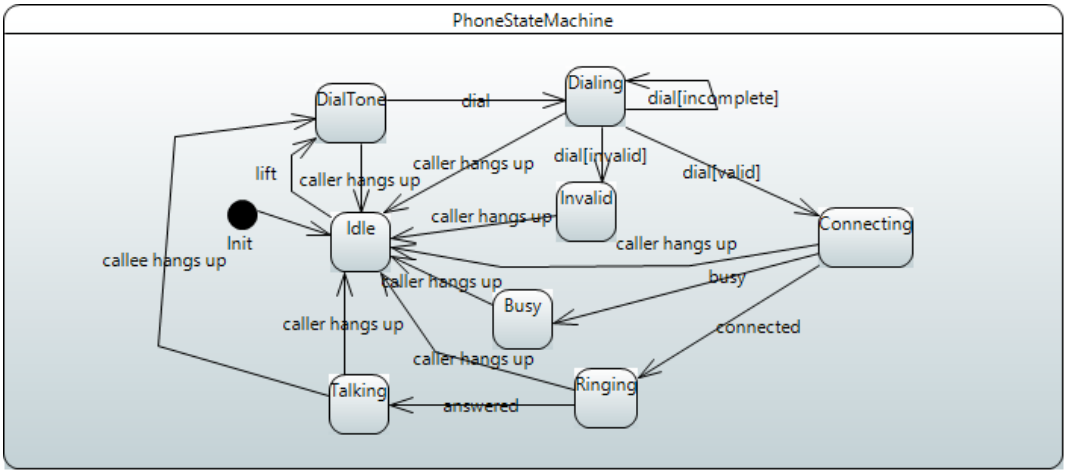

Fig. 3. Flat state machine of mobile phone calls.

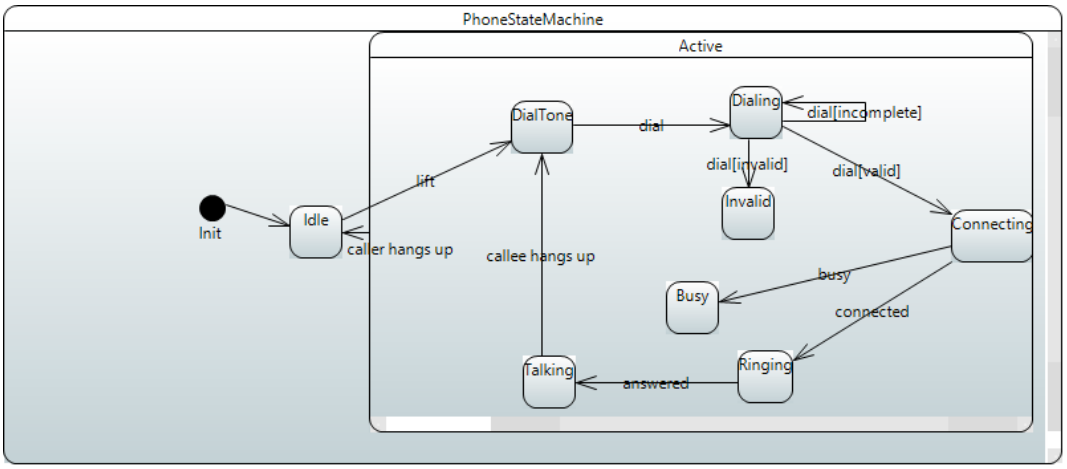

Fig. 4. State machine of mobile phone calls after a **Group states** and a **Fold outgoings** refactoring. Active states are grouped together and all transitions of the substates of Active composite state are folded into a single transition.

4. REFACTORINGS

Refactorings are operations that restructure models as follows: they modify the internal structure of the models, though the functionalities of the models remain the same. Models before and after refactoring are functionally equivalent since solely nonfunctional attributes change.

A fully defined refactoring consists of preconditions, postconditions, main-scenario (changes of the model) and a window for additional parameters.

Many refactorings are provided by EMF Refactor. All of them can be used on class diagrams, e.g. *add parameter*, *create subclass*, *create superclass*, *move attribute*, *move operation*, *pull up attribute*, *pull up operation*, *push down attribute*, *push down operation*, *remove empty associated class*, *remove empty subclass*, *remove empty superclass*, *remove parameter*, *rename class*, *rename operation* and several compositional refactorings. One of our goals was to extend these and visualize them properly.

4.1 Visualization

The existing class diagram refactorings modify only the EMF model, the result is not visible in the Papyrus diagram, therefore our first goal was to add this feature. This involved programmatically creating and deleting views of model elements simultaneously with the EMF model changes. The main aspects were not only to refresh the Papyrus model, but also to support undoing in a way that every change can be reverted in one step. To achieve that, transactions are supported in EMF Refactor which means that it detects the changes during the refactoring process and stores them in a stack, providing an undoable composite command. Unfortunately, EMF and Papyrus changes cannot be made in the same transaction due to multithreading problems, thus we implemented a solution where atomic actions (add, remove) are caught, and we modify the diagram by handling these. We try to keep the consistency of the model and the corresponding diagram.

4.2 New refactorings

Since EMF Refactor defines refactorings only for class diagrams, our other goal was to create refactorings for state machines. State machines provide many opportunities to refactor model elements causing small and consistent changes. Most of the refactorings we implemented may be found in [Sunyé et al. 2001], that contains the pre- and postconditions of all refactorings. In the article, postconditions differ from the ones defined in EMF Refactor, they must be checked after the refactoring process to guarantee that the refactoring made the specific changes.
In order to refactor successfully, our refactorings first check the preconditions, then pop up a window, that contains a short description of the selected refactoring and the input fields for the parameters – some of the refactorings needs additional parameters, e.g. name of the composite state which will be created. After that, it checks the conditions that refer to the parameters, then executes the proper changes. If any of the conditions fails, the execution is aborted and the error list is shown.
The added state machine refactorings are:
- *Group States*: it can be used by selecting many states to put them into a composite state instead of moving them and their transitions manually,
- *Fold Entry Action*: to replace a set of incoming actions to an entry action,
- *Unfold Entry Action*: to replace an entry action by a set of actions attached to all incoming transitions,
- *Fold Exit Action*: to replace a set of outgoing actions to an exit action,
- *Unfold Exit Action*: to replace an exit action by a set of actions attached to all outgoing transitions,
- *Fold Outgoing Transitions*: to replace all transitions leaving from the states of a composite to a transition leaving from the composite state,
- *Unfold Outgoing Transitions*: the opposite of the *Fold Outgoing Transitions*,
- *Move State into Composite*: to move a state into a composite state,
- *Move State out of Composite*: to move a state out of a composite state,
- *Same Label*: copy the effect, trigger and guard of a selected transition.

The refactorings are executed regarding the predefined conditions to keep semantics and they also modify the Papyrus diagram.
We also implemented two new important class diagram refactorings:
- *Merge Associations*: associations of class A may be merged if they are of the same type and they are connected to all subclasses of class B,
- *Split Associations*: the opposite of the *Merge Associations* refactoring.

5. METRICS

Metrics are able to increase the quality of the system and save development costs as they might find faults earlier in the development process. Metrics return numbers based on the properties of a model from which we may deduce the quality of the model. For example, a state machine with numerous transitions may be hard to understand as the transitions may have many osculations. On the other hand, states embedded in each other, with a deep hierarchy, might also be confusing. According to this, by calculating the metrics we get an other advantage: we can detect model smells, furthermore, some of them can be repaired automatically. We describe this in Section 6.
In EMF Refactor there are many class, model, operation and package metrics defined. Our goal was to create state and state machine metrics. State metrics measure the state and the properties of its transitions, while state machine metrics calculate numbers of the whole state machine. We added 10 state (Table I.) and 16 state machine (Table II.) well-known metrics, all of them measure the complexity of the model. These metrics can be easily calculated as they have simple definitions: they describe the model by the number of items and connections in the model.

Table I. Defined state metrics

Name	Description	Name	Description
st_entryActivity	Number of entry activity	st_doActivity	Number of do activity (0 or 1)
st_exitActivity	Number of exit activity (0 or 1)	st_NOT	Number of outgoing transitions
st_NTS	Number of states that are direct target states of exiting transitions from this state	st_NSSS	Number of states that are direct source states of entering transitions into this state
st_NDEIT	Number of different events on the incoming transitions	st_NITS	The total number of transitions where both the source state and the target state are within the enclosure of this composite state
st_SNL	State nesting level	st_NIT	Number of incoming transitions

Table II. Defined state machine metrics

Name	Description	Name	Description
NS	Number of states	NSS	Number of simple states
NCS	Number of composite states	NSMS	Number of submachine states
NR	Number of regions	NPS	Number of pseudostates
NT	Number of transitions	NDE	Number of different events, signals
NE	Number of events, signals	UUE	Number of unused events
NG	Number of guards	NA	Number of activities
NTA	Number of effects (transition activities)	MAXSNL	Maximum value of state nesting level
CC	Cyclomatic complexity (Transitions - States + 2)	NEIPO	Number of equal input-parameters in sibling operations

6. BAD SMELL DETECTION

As mentioned earlier, model quality is hard to measure, but with simple heuristics – smells – we can detect poorly designed parts. Moreover, in some cases we can offer solutions to improve them using specific refactorings.
Useful smells consist of a checker part and a modification part. Checker part may contain metrics and conditions: we can define semantics for the values of the metrics. Categorizing the values means that

we can decide whether or not a model is good or smelly. The modification part may contain refactorings to eliminate the specified smells.
EMF Refactor defines 27 smells for class diagrams, about half of them is rather a well-formedness constraint than a real smell: unnamed or equally named elements. Most of them provides useful refactorings to eliminate the bad smells.
In our extension, we implemented four important metric-based smells for state machines:
- *Hot State (Incoming)*: a threshold for the number of incoming transitions,
- *Hot State (Outgoing)*: a threshold for the number of outgoing transitions,
- *Deep-nesting*: a threshold for the average nesting of states,
- *Action chaining*: a threshold for transitions, its main responsibility is to recognize whether too many entry and exit actions would be executed in a row.
We can also detect unnamed or unreachable states and unused events.
Defining the thresholds of the smells is not easy ([Arcelli et al. 2013]), they may vary in the different projects. We defined the smells and the default thresholds based on the experience of our researchers and the reference values used in the state-of-the-art. If the users find these values inappropriate to their models, they can modify them in our tool manually.

6.1 Smell based refactorings

It is not always obvious which refactorings can help to eliminate bad smells. As we presented in Section 3, a state with a large number of incoming transitions can be simplified in two steps: group the states where the transitions come from, then fold the outgoing transitions of the new composite state.
Having a large number of outgoing transitions is more complex. An idea is to describe the smelly state more precisely, this way the substates and details may explain the different scenarios, but unfortunately it also increases the complexity. In this topic further research is needed.
Deep-nesting can be improved by using specific Move State out of Composite refactorings. A further step could be to detect ”unnecessary” composite states which increase the complexity more by nesting than decrease it by folding transitions. In connection with that, an important aspect is that by using composite states, code duplication is reduced as common entry and exit actions do not have to be duplicated in substates.
Finally, action chaining is a very complex problem. It depends on the specific actions if it can be reduced or fully eliminated. Though detection is useful and shows a possible bad design, it may be better to be handled manually.

7. RELATED WORK

The literature regarding the metrics and refactorings of UML models is extensive. Since our interest is tool developments for executable UML models, our review of the related work is focused on the tool-related topics.
The most well-known model measurement tool is [SDMetrics 2002]. It is a standalone Java application having a large set of predefined metrics for the most relevant UML diagrams. SDMetrics also supports the definition of new metrics and design rules, and is able report the violation of design rules. Although several metrics mentioned earlier in this paper were first implemented in SDMetrics by our project team, we have decided to use EMF Refactor because of the refactoring support and the easy integration with Eclipse.
The recent developments show an increased interest in the combination of metrics evaluation and refactoring services in a single toolchain. A good starting point for the review is the survey published by [Misbhauddin and Alshayeb 2015]. The survey refers to 13 publications (including EMF Refactor) about state chart measurements and refactorings. An other publication dealing with state charts and

providing full automatic refactorings is [Ruhroth et al. 2009]. It is based on RMC Tool, a quality circle tool for software models. [Dobrzanski 2005] presents five different tools that describe or implement model refactorings. Refactoring Browser for UML focuses on correctly applied refactorings, while SMW Toolkit describes new refactorings without validation. The goal of Odyssey project is improving understandibility by defining smells for class diagrams.
Compared to these researches our tool aims to support model driven development in the Eclipse framework based on Papyrus and EMF Refactor. We want to provide a tool that is built into the daily used modeling environment of the users, and there is no need to use a separate tool: the users can develop, maintain, refactor, measure and analyse their models in the same environment. One more reason that made us to choose EMF Refactor is the txtUML toolchain developed by our research group. In the txtUML toolchain executable UML models can be defined textually and the framework is able to generate executable code and Papyrus diagrams as well [Dévai et al. 2014]. Naturally, we can use our tool with the generated diagrams, nevertheless it would be a great advancement to measure and refactor them before the generation, using only the textual definition of state machines. We want our tool to be an important part of the txtUML toolchain as well.

8. SUMMARY

We presented a tool that is able to measure the complexity of state charts and execute transformations to reduce their complexity. Besides implementing metrics, smells and refactorings in connection with state machines, we extended the original functionality of the EMF Refactor tool with the feature of Papyrus visualization.
We plan to implement more refactorings and smells in order to improve automation of model quality assurance. An important point of view is that we defined only metric-based smells, but in EMF Refactor, graph-based smells are also supported. Our plan is also to validate these refactorings to ensure the consistency of the model.

REFERENCES

D. Arcelli, V. Cortellessa, and C. Trubiani. 2013. Influence of numerical thresholds on model-based detection and refactoring of performance antipatterns. In *First Workshop on Patterns Promotion and Anti-patterns Prevention*.

T. Arendt, F. Mantz, and G. Taentzer. 2010. EMF Refactor: Specification and Application of Model Refactorings within the Eclipse Modeling Framework. In *9th edition of the BENEVOL workshop*.

T. Arendt and G. Taentzer. 2010. *UML Model Smells and Model Refactorings in Early Software Development Phases*. Technical Report. Philips and Marburg University.

Gergely Dévai, Gábor Ferenc Kovács, and Ádám Ancsin. 2014. Textual, executable, translatable UML. Proceedings of 14th International Workshop on OCL and Textual Modeling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), pages 3-12, Valencia, Spain, September 30. (2014).

Lukasz Dobrzanski. 2005. UML Model Refactoring: Support for Maintenance of Executable UML Models, Master Thesis. (2005).

EMFRefactor 2011. EMF Refactor. https://www.eclipse.org/emf-refactor/. (2011). Online; accessed 16 June 2016.

M. Misbhauddin and M. Alshayeb. 2015. UML model refactoring: a systematic literature review. *Empirical Software Engineering* 20 (2015), 206–251. DOI:http://dx.doi.org/10.1007/s1066401392837

Papyrus 2014. Papyrus. https://eclipse.org/papyrus/. (2014). Online; accessed 16 June 2016.

T. Ruhroth, H. Voigt, and H. Wehrheim. 2009. Measure, diagnose, refactor: a formal quality cycle for software models. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 360–367. DOI:http://dx.doi.org/10.1109/seaa.2009.39

SDMetrics 2002. SDMetrics. http://www.sdmetrics.com/. (2002). Online; accessed 16 June 2016.

G. Sunyé, D. Pollet, Y. Le Traon, and J.M. Jézéquel. 2001. Refactoring UML Models. In *UML '01 Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn (Eds.). 134–148. DOI:http://dx.doi.org/10.1007/3-540-45441-1_11