Measuring design diagrams for product quality evaluation

A. Inkeri Verkamo, Juha Gustafsson, Lilli Nenonen, Jukka Paakki

Abstract

Many important product quality characteristics, such as performance and maintainability, cannot be added on late in the development, and hence early warnings of poor quality expectation would be very useful to allow early corrective measures. To produce such information, we base product quality prediction on measurements and calculations of the UML design diagrams of the software. Some design metrics may be used as direct predictors of quality; as an example, cohesion and coupling metrics are generally considered good predictors of the maintainability of the product. On the other hand, certain quality characteristics, e.g., performance, require more complex analysis of the design by computing predictions of the execution time of a typical use case with user provided information of the input size and attributes.

Another way of taking advantage of the design diagrams is by detecting design patterns and anti-patterns in the design, and using these as predictors of the final product quality. While using design patterns is considered to have a positive effect on the quality and productivity of the development process, their effect in product quality is more complex and may in fact be positive or negative, depending on the context. We suggest using the pattern library to store information of the previous uses of the patterns and the quality of the corresponding systems, to predict the quality effects of the design patterns.

In our paper, we present a tool for measuring UML diagrams of the product to predict three central product quality characteristics, size, understandability, and performance. The tool also detects all occurrences of a previously defined set of design patterns and anti-patterns from the diagrams. Our paper reports industrial experience in using the tool to detect patterns and measure quality of design diagrams.

1. Introduction

While software development has long been mostly focused on improving product time to market, recently a growing interest has also been taken in quality issues. But there is a fundamental difference in the way that product quality concerns affect the software development process, as opposed to improving development efficiency. Development time may often be decreased by means of corrective actions in a single phase of the lifecycle, e.g., by using improved methods or better tools in code generation or testing. But if we aim at improving product quality this goal must be kept in mind throughout the development process, as it is likely to require changes in several phases of the lifecycle.

A typical example of a quality characteristic that cannot be added on in a late phase is performance [21]. Contrary to what many textbooks claim (see, e.g., [4]), tuning of the fully implemented system is not the winning strategy to ensure adequate performance, but rather performance requirements must be taken into account in early stages of software development. Similarly, maintainability requires much more than producing documentation in a late phase of software development, and satisfactory usability cannot be guaranteed simply by including a package of GUI routines to the software distribution.

The development of a new software product seldom starts from scratch. In particular with the object-oriented approach to software development, the idea of reusing parts of old products has gained wider acceptance. The reused parts may be anything from chunks of actual code to design ideas or design solutions to well defined problems. But this way of producing software may also affect the quality of the final product. To take full advantage of the reuse based approach, previous experience of the quality characteristics of the reused parts should be made available and used to predict the quality of the new product.

In this paper we address the problem of product quality prediction during an object-oriented software process. In Section 2 we take a look at measuring and predicting quality of the software product, and in Section 3 we study some quality measures that can be obtained during the design phase. In Section 4 we discuss design patterns, which have been suggested as a means of improving the quality of the design phase, and address the connection between design patterns and product quality. In Section 5 we describe our approach of including product quality prediction as part of the design phase, and in Section 6 we give some examples of using our tool for quality predictions in industry level software development. Finally Section 7 is a short conclusion.

2. Quality measurement and prediction

Various sources describe important aspects of software quality and attempt to define metrics for product quality evaluation. Several software quality models have been defined (some of the early references are [3, 15]), and while the structure and other details of such models differ, there is wide agreement that certain factors, e.g., usability, efficiency (or performance), maintainability, and reliability, should be included in our notion of product quality. As the emphasis of the various quality characteristics may differ widely from one product to another, part of the requirement analysis phase should be an explicit definition of the quality factors that are relevant to the product [8]. To evaluate the existence or lack of quality in our product, we then need to find suitable metrics for the given quality factors; a useful aid for this purpose is the GQM approach [2].

Finding an appropriate set of metrics for measuring a certain quality factor of the product is often a difficult task. Usability is certainly very tightly coupled with the needs of the intended users, and thus can hardly be evaluated without a set of usability tests taken by a representative group of users. Similarly, running performance tests requires that we have a clear understanding of the typical workload of the system, descriptions of the use cases and their portion in the job mix, and a set of typical inputs for each case. Reliability testing again requires extensive runs or simulations of the software to ensure that failure density remains below the acceptable level.

The problem with this type of quality measurement is the need for test runs on the final product, which is very intensive on the final testing phases of the development lifecycle. However, if product quality is found unacceptable, improvement often requires changes in the design of the system and thus may result in extensive rework. It would be very helpful to find predictive metrics of product quality that could be applied earlier in the development lifecycle, so that corrective actions could be taken immediately if the predicted quality does not meet the requirements. The design phase is particularly important in this respect, since many of the crucial decisions stem from the product design. To diminish the amount of unnecessary rework, our purpose is to define design metrics that can be used especially to predict the quality of the final product.

3. Measuring in the design phase

To obtain useful quality information early enough for improvement purposes, we base our quality predictions on measurements and calculations of the high-level design diagrams of the software. In defining a software architecture, various views on its different aspects are used, such as the four views and the set of scenarios suggested by the "4+1 model" [11]. A widely used presentation for describing these views is the Unified Modeling Language (UML) [5], which is fast developing into a de facto standard of the field.

Depending on the quality factors that we are interested in, we may need to study various types of UML diagrams. For predicting software size or run time space requirement we use class diagrams and object diagrams. For performance prediction we need more process-oriented diagrams such as activity diagrams and sequence diagrams. Some of the parameter data may be obtained from diagrams presenting the physical view of the software, such as deployment diagrams. Scenarios typically presented as use case diagrams are necessary to track the behavior of the system during a typical or particularly interesting use case.

Some design metrics may be directly useful as predictors of a certain quality characteristic. For instance coupling, as measured by the CBO (coupling between objects) metric of Chidamber and Kemerer [7], is considered a good predictor of the changeability of the product [10], while the DIT (depth of the inheritance tree) metric has been shown to be a good indicator of fault-proneness [1]. On the other hand, predicting certain quality characteristics requires more elaborate analysis of the design. For performance prediction we need a fairly detailed description of the execution structure and also user provided information of the input size and characteristics, to be able to compute best case and worst case execution times, or the average execution time of a particular use case. Suitable forms of presentation for the execution structure are extended sequence diagrams, activity diagrams or execution graphs [19].

4. Design patterns

Predicting quality characteristics is radically improved by taking advantage of the experience gained in previous projects. It is particularly useful to recognize the reuse of earlier design solutions and take into account their observed quality effects. In the object-oriented field the wide acceptance of UML makes it easy to reuse the work done by others, and active collection of reusable design solutions has already emerged. High quality design solutions to frequently occurring software problems, generally known as *design patterns*, have been described and stored in pattern libraries using a standard format [9]. Correspondingly, bad solutions, or *anti-patterns*, have been detected and they can likewise be stored together with a description of their disadvantages [6].

Using patterns has been advocated because of their positive effect on the software structure, and their potential to speed up the design process and to improve its quality. So far, little is known about the effect of patterns (and anti-patterns) on product quality, although recently some interesting observations on performance anti-patterns have been published [20]. In fact, from the point of view of product quality, some of the commonly used design patterns are not necessarily good or bad per se, as their quality effects may depend heavily on the situation where they are used. Therefore the pattern library should be augmented by recording user experience on previous uses of the patterns and related quality information, to improve the common understanding of their applicability.

Let us take an example from the area of performance concerns. Here it is quite possible that a certain design pattern may have either positive or negative effects on product quality depending on the characteristics of the application area and the intended workload [22]. For example the scalability of the solutions may be different and a design pattern suitable in one environment may be quite inappropriate in another one. Hence the difference between a "good" pattern and a "bad" one may be in the parameters of the system, and storing this type of dependency information in the pattern library can aid the designer in predicting the quality of the software.

As an example, let us look at the *Mediator* design pattern in Figure 1, where all communication between the partners is handled by the DialogDirector [9]. This may be very efficient if the communication is infrequent and the messages are complex, but it turns out to be disastrous if the number of communicating partners is large and each of them frequently sends short messages to the others. Evaluating the actual performance effects of this pattern requires knowledge of the hardware environment and the projected workload, and maybe even some prediction of how the workload may change during the lifetime of the software.

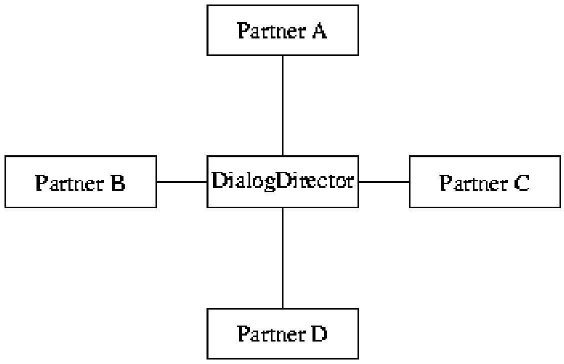

Figure 1. Mediator pattern.

Finally we must remember that not all pattern usage is preplanned [13]. In fact, many of the good practices and patterns of software design have been created long ago and they may frequently recur in software products without the developers being aware that their design solution is an identified “pattern” at all. Even more probably, the occurrence of an anti-pattern may be unknown to the developer. Again the experience data stored in the pattern library could be used, provided that the pattern (or anti-pattern) embedded in the design can be recognized. It is therefore useful to detect all the recurring patterns in the design, regardless of their origin, and to apply the information in the pattern library to predict the quality effects of the detected patterns.

5. The Maisa approach to quality prediction

Maisa (Metrics for Analysis and Improvement of Software Architectures) is an ongoing research project for developing methods and tools for automatically analyzing the software architecture and for predicting some central quality factors based on this analysis [17]. At present the quality factors that are examined are size, complexity and predicted performance, but the concept is suitable for various kinds of other quality factors as well. Some measures (such as design size measures) can be directly computed from the original UML diagrams. For more complex measures such as performance prediction, we need small extensions to UML to allow user provided information such as execution times and branching behavior; this can be accomplished by using annotations, such as UML stereotypes, tagged values and constraints [5].

In the Maisa project, the strategies used for quality prediction rely on two basic ideas: pattern discovery and hierarchical measures. In the following sections, we study each of these ideas in more detail. Section 6 then presents some industry examples of measuring and predicting product quality based on these measures.

5.1. Pattern discovery

The analysis in Maisa is based on discovering patterns from the UML diagrams that describe the software architecture. While the original purpose of design pattern collections has been to encourage designers to reuse the patterns as components of the software under development, we should not rely on the developer to provide information of the patterns embedded in the architecture. Instead we wish to discover all patterns and anti-patterns automatically from the software design, irrespective of their origin, and to evaluate their quality effects. In fact, some common patterns are very general and they may not be sufficiently well defined in a high-level design. Therefore pattern discovery may result in finding also patterns that are in fact not intended as such at all, or in failing to find patterns because of inadequate level of detail in the diagram. By interaction with the tool, the user can choose which patterns to look for and afterwards each finding can be approved or discarded by the user. The pattern discovery method is based on considering pattern mining as a constraint satisfaction problem [17]. The method that we use, AC-3, is described in more detail in [12].

In order to evaluate the quality effects of the embedded patterns, the pattern library must contain the necessary information for quality calculations. The quality calculations may also involve use case related parameters that must be provided by the user. Moreover, historical information about the usage and relevance of the patterns must be collected and maintained in the pattern library to improve the accuracy of the predictions. This type of data has previously not been included in pattern libraries, but it would add value even to the traditional use of the libraries as sources for reuse. At present, we are working on a suitably flexible structure for such an enhanced pattern library.

In practice each pattern covers only part of the entire UML diagram, but even detecting a single anti-pattern (or unsuitable use of a design pattern) may pinpoint a crucial problem. Some elements may be part of several patterns, while some parts do not belong to any pattern. Hence quality prediction of the entire system is not entirely covered by the pattern mining approach, but depending on the comprehensiveness of the pattern library we may gather useful information of the potential pitfalls of the architecture.

5.2. Hierarchical quality evaluation

Many quality measures are hierarchical in the sense that the measure for the entire system can be computed as a function of the measures of its subsystems, taking into consideration the sequencing and nesting structure of the architecture [8]. Typical examples of hierarchical measures include size metrics such as the number of classes, complexity measures such as the McCabe cyclomatic complexity measure [14], and performance measures such as response time [18].

We assume that the software is presented as a collection of UML diagrams. Some of these typically form a hierarchy representing the same view (e.g., a hierarchy of class diagrams), while others represent different views of the software (e.g., class diagrams for the logical view, and interaction diagrams for the process view). Quality prediction may require combining data from several diagrams of different kinds. For example, a use case diagram may describe a set of possible actions and their probabilities, and a set of activity diagrams may describe possible flows of control in more detail, annotated with the resource requirements of the activities. To obtain an average response time of the software, we need to compute the weighted sum of the response times of the use cases, while the response time of each use case in turn is computed from the execution times of the activities of the activity diagram (for details, see [18]).

While many software quality factors may be described using a single number (e.g., McCabe's complexity measure, software size measured in lines of code or in function points) some quality factors such as performance require using several numbers. To predict performance, the user typically wants worst case and best case predictions, and often also the performance for some typical cases within the domain [22]. This may require interactive use of the tool, where the user gives parameter values for each of the interesting cases, after which the tool computes the required predictions. Similarly, we may want to study the software on various level of detail, and our measuring tool should produce measures for the entire system as well as smaller subsystems (such as subdiagrams, or classes).

Metric	Per project	Per diagram	Per class

	avg	min	max	avg	min	max	avg
NOA (nr of associations)	62	14	30	20.7	0	7	1.09
NOC (nr of classes)	57	15	26	19.0			
NOMA (nr of all methods)	5	0	5	1.7	0	2	0.09
NOOB (nr of objects)	0						
NOAttr (nr of attributes)	47	0	26	15.7	0	7	0.82
DIT (depth of inheritance tree)		1	3	2.0			
NOI (nr of interfaces)	0	0	0	0			
NOM (nr of public methods)	5	0	5	1.7	0	2	0.09

Table 1. Metric results for the CASE tool architecture.

6. Examples of pattern-based measurement

At present, the Maisa tool has been tested both with small artificial examples and with example cases of industry level software. Our first example studies the high-level design of an industrial CASE tool consisting of three class diagrams. The architecture of the CASE tool was examined with Maisa, and Table 1 presents some size metrics of the diagrams (see [7] for details of the metrics).

Maisa produces size metrics for the entire architecture (or project) and separately for each diagram and each class (where appropriate). At the high-level design phase the architecture contains 57 classes but only 47 attributes and very few methods. This is typical of high-level design diagrams where only the most important attributes and methods are included. However, this has some implications when searching for design patterns, since typically design patterns are defined using code level concepts. As an example, an association is a design level concept that can be presented at the code level in many different ways. High-level diagrams typically do not contain this level of detail, and in order to detect patterns in high-level diagrams, we need to relax the rules describing some of the patterns.

But if we relax the rules too much, we may not be able to distinguish between related patterns. Another problem with more relaxed rules is that we may find false positives, i.e., pattern candidates that are not real pattern instances. The possibility of false positives increases with higher level diagrams. In these cases the user must interactively select the correct instances from those detected by Maisa.

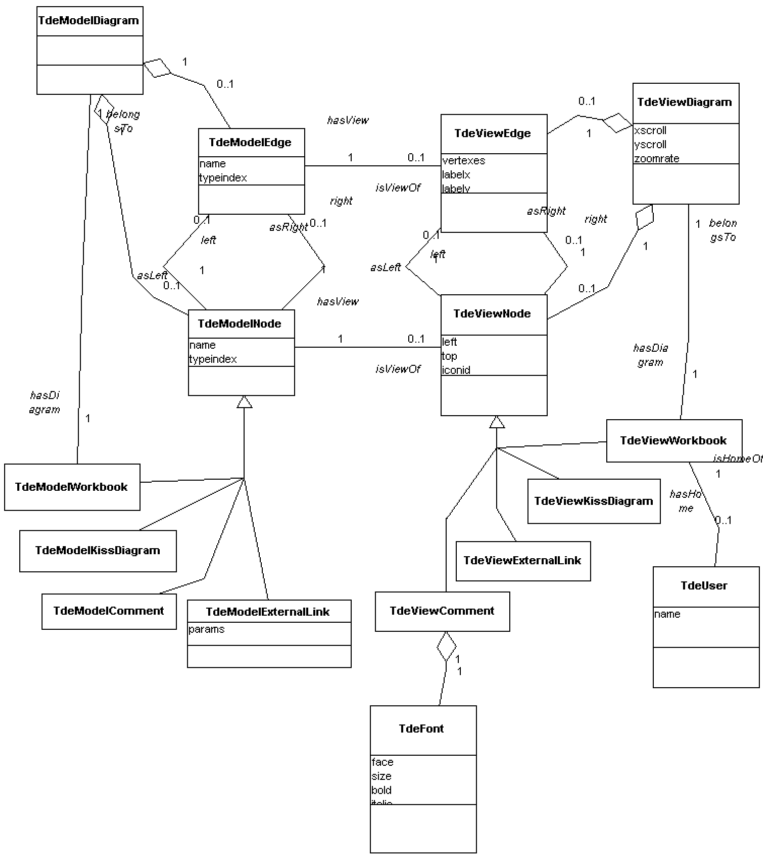

Figure 2. Design diagram for part of the CASE tool.

The diagram in Figure 2 is one of the design diagrams for the CASE tool architecture. It has just a few attributes or methods and, in consequence, only some of the basic pattern elements such as referencing attributes and derived attributes can be found in it. These are detected in the same way as the more complex patterns. As a result, Maisa found 41 instances of the *Derived Attribute* pattern element, but no other patterns.

As a second example, the diagram in Figure 3 shows part of the class diagram for the Maisa tool itself. In principle, the enclosed area in the picture could quite well represent the *Factory Method* or even part of the *Abstract Factory* pattern [9]. When using Maisa to search for *Factory Method* instances from this class diagram, we found four correct instances of the pattern. The following two descriptions give the name of the pattern role on the left and the name of the corresponding diagram element on the right (note that all the methods and elements in the example are not in Figure 3):

Product = CElement
ConcreteProduct = CClass
ConcreteCreator = CClassDiagram
Creator.FactoryMethod() = CDiagram.NewElement()
Creator = CDiagram

and

Product = CElement
ConcreteProduct = CState
ConcreteCreator = CStateDiagram
Creator.FactoryMethod() = CDiagram.NewElement()
Creator = CDiagram

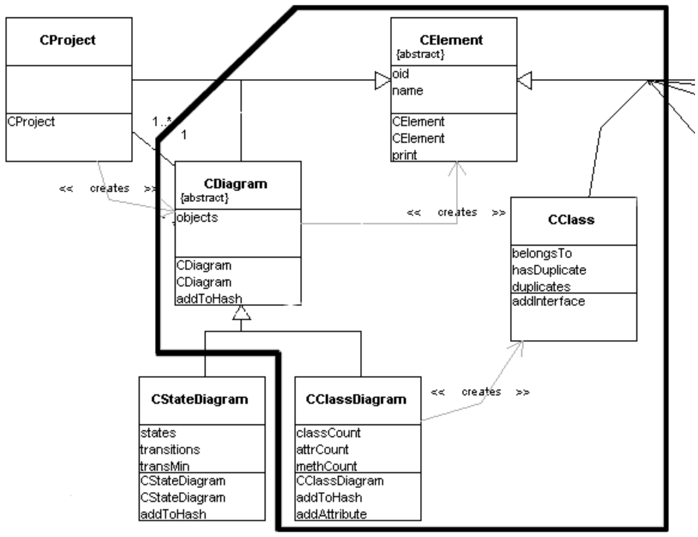

Figure 3. Pattern instance found in the class diagram of Maisa.

If the NewElement method is not included in the diagram or the CClass instance is not used as a return value, these instances will not be found. If we define the methods too loosely, Maisa finds twelve instances, eight of which are incorrect. (Actually the classes depicted here are not even supposed to form *Factory Method* instances. This will, however, become evident only when we use actual program files as basis for our measurement. The diagram classes are containers rather than factories.)

Our third example considers hierarchical evaluation of quality metrics, which is particularly important in performance evaluation. The (extended) activity diagram in Figure 4 shows a simple use case for an Internet shopping system. Execution times of the activities are presented as annotations over the corresponding nodes the activity diagram. (The units are not explicitly specified, but typically they would be milliseconds.) A lower limit for the execution time can be computed by selecting a path consisting of the fastest subroutines and using the minimum execution times for each node. In the minimum case the user buys nothing and hence the node *buy products* is not entered at all; the minimum execution time is 1530 units (= 450 + 800 + 280). This lower limit assumes that no product is bought and also that every other activity occurs as quickly as possible, which is clearly unrealistic. A more realistic lower limit would be the average execution time for the minimum case which is 3250 units (= 800 + 2000 + 450).

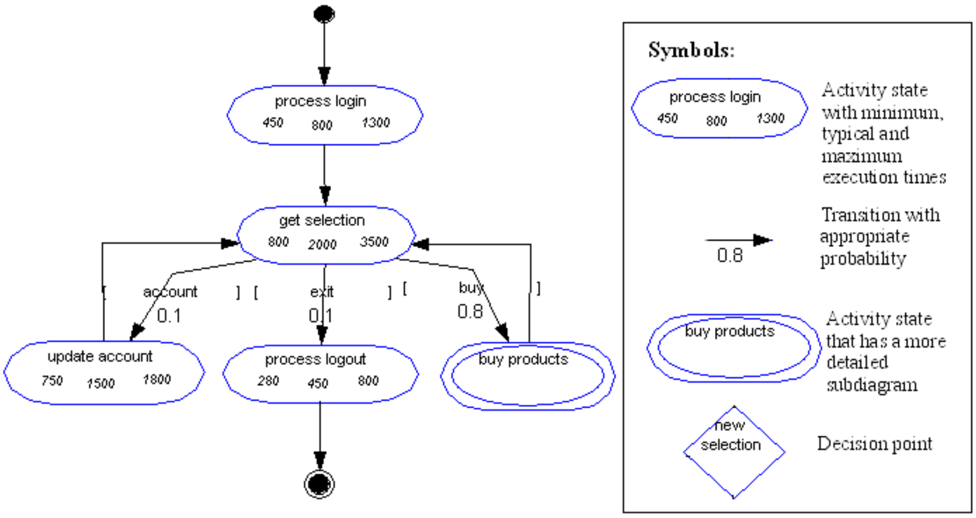

Figure 4. Shopping system activity diagram with annotations.

To obtain execution times for use cases where the customer buys at least one item, requiring the activity *buy products*, we need an execution time for this node. The node has been expanded in Figure 5 and its total execution time is computed from the execution times of the nodes in the subdiagram. For the case where the customer buys one item and pays by account, we get a total average execution time of 4625 units (= 800 + 2000 + 1375 + 450). The value 1375 for the node *buy products* results from the path through the diagram in Figure 5 where one product is purchased and paid by account. (Note that the number of products is represented by the branching structure of the node *new selection*.) Finally, to compute the execution time for a typical case we use the respective branching probabilities to weight the execution time for each path. According to the branching probabilities of Figure 5, an average customer buys four items. Half of the customers pay by account, 40 per cent use a credit card and 10 percent pay by invoice. The average execution time of the typical case is 6079 units (= 800 + 2000 + 0.1*1500 + 0.8*3855 + 0.1*450), where again the value 3855 is the average execution time for the *buy products* subdiagram.

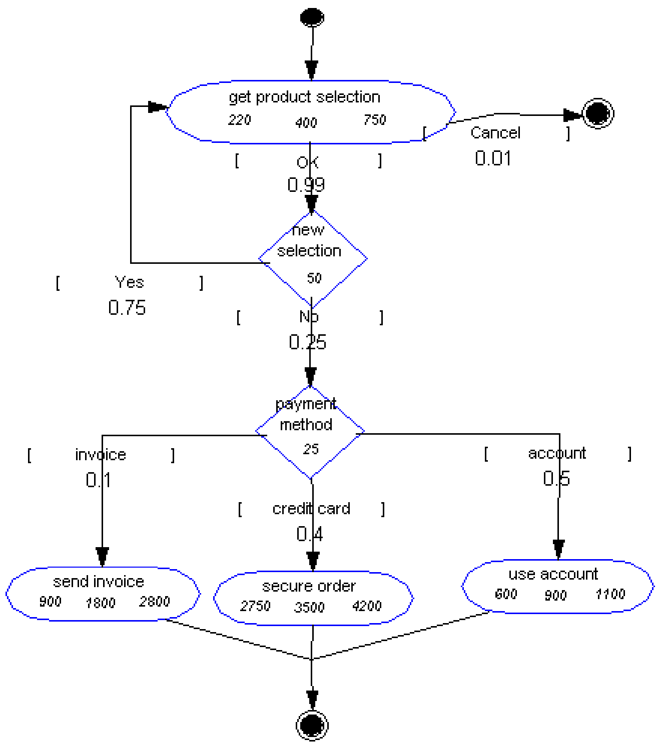

Figure 5. Subdiagram for activity “buy products”.

7. Conclusion

In this paper we have considered the possibilities of measuring software design quality and predicting product quality based on the design patterns embedded in the architecture. By discovering recurring patterns and anti-patterns in the architecture and using previously recorded knowledge of their quality effects, we expect to arrive at reasonably accurate predictions of the final product quality already during the design phase. The success of the method depends heavily on the comprehensiveness of the pattern library, and particular care must be taken that both successful and unsatisfactory attempts of pattern usage are stored in the library, with adequate detail to refine our understanding of the scope and effects of the patterns. In particular, most of the published design patterns have been class diagrams, and as such not useful for performance prediction. It seems possible that performance related patterns are more domain specific than most design patterns; certainly the performance effects of patterns should be evaluated with the intended application domain in mind.

The tool developed in the Maisa project produces both simple metrics for software size and complexity [16] as well as more complex predictions of performance, based on a collection of published patterns [9]. Our next task is to augment the pattern library with specific patterns of our industrial partners, along with experience of pattern usage that will serve as the basis of quality prediction. We are confident that an extensive selection of such domain specific patterns will emerge as the users gather experience of the gains of including their own patterns in the library.

Bibliography

[1] V. Basili, L.C. Briand, and W.L. Melo, "A validation of object-oriented design metrics as quality indicators", IEEE Transactions on Software Engineering 22, 10, 1996, 751-761.

[2] V.R. Basili and D. Weiss, "A methodology for collecting valid software engineering data", IEEE Transactions on Software Engineering 10, 16, 1984, pp. 728-738.

[3] B.W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, and M.J. Merritt, "Characteristics of Software Quality", North-Holland Publishing Company, 1978.

[4] G. Booch, "Object Oriented Design with Applications", Benjamin/Cummings, 1991.

[5] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1998.

[6] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. III McCormick, and T.J. Mowbray, "Antipatterns: Refactoring Software, Architectures, and Projects in Crisis", John Wiley & Sons, 1998.

[7] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object-oriented design", IEEE Transactions on Software Engineering 20, 6, 1994, pp. 475-493.

[8] N. Fenton and S.L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach", International Thompson Computer Press, 1996.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[10] H. Kabaili, R.K. Keller, F. Lustman, and G. Saint-Denis, "Class cohesion revisited: an empirical study on industrial systems", in Proceedings of the Fourth International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Cannes, France, June 2000.

[11] P.B. Kruchten, "The 4+1 view model of architecture", IEEE Software 12, 6, 1995, pp. 42-50.

[12] A. Mackworth, "Consistency in network of relations", Artificial Intelligence 8, 1, 1977, pp. 99-118.

[13] R. Martin, "Discovering patterns in existing applications", in J.O. Coplien and D.C. Schmidt, editors, Pattern Languages of Program Design, Addison-Wesley, 1995.

[14] T.J. McCabe, “A software complexity measure”, IEEE Transactions on Software Engineering 2, 4, 1976, pp. 308-320.

[15] J.A. McCall, P.K. Richards, and G.F. Walters, “Factors in software quality;” Technical Report RADC TR-77-369, Rome Air Development Center, 1977.

[16] L. Nenonen, J. Gustafsson, J. Paakki, and A.I. Verkamo, “Measuring object-oriented software architectures from UML diagrams”, in Proceedings of the Fourth International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Cannes, France, June 2000.

[17] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A.I. Verkamo, “Software metrics by architectural pattern mining”, in Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), Beijing, China, August 2000.

[18] C.U. Smith, “Performance Engineering of Software Systems”, Addison-Wesley, 1990.

[19] C.U. Smith and L.G. Williams, “Performance engineering evaluation of object-oriented systems with SPE-ED”, in R. Marie et al., editor, Computer Performance Evaluation: Modelling Techniques and Tools, Springer-Verlag, 1997.

[20] C.U. Smith and L.G. Williams, “Software performance antipatterns”, in Proceedings of the Second International Workshop on Software and Performance (WOSP'2000), Ottawa, Canada, September 2000.

[21] C.U. Smith and M. Woodside, “Performance validation at early stages of software development”, In E. Gelenbe, editor, System Performance Evaluation: Methodologies and Applications. CRC Press, 1999.

[22] A.I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki, “Design patterns in performance prediction”, in Proceedings of the Second International Workshop on Software and Performance (WOSP'2000), Ottawa, Canada, September 2000.