On the classification of first-class changes

Peter Ebraert[*] and Theo D'Hondt

Computer Science Department
Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
{pebraert,tjdhondt}@vub.ac.be

Abstract. Feature-oriented programming (FOP) is the research domain that targets the encapsulation of software building blocks as features, which better match the specification of requirements. Recently, we proposed change-oriented programming, in which features are seen as sets of changes that can be applied to a base program, as an approach to FOP. In order to express features as sets of changes, those changes need to be classified in different sets that each represent a separate feature. Several classification strategies are conceivable. In this paper we identify three kinds of classification strategies that can be used to group the change objects. We compare them with respect to a number of criteria that emerged from our practical experience.

1 Introduction

Feature-oriented programming (FOP) is the study of feature modularity, where features are raised to first-class entities [1]. In FOP, features are basic building blocks, which satisfy intuitive user-formulated requirements on the software system. A software product is built by composing features. Recently, we proposed a bottom-up approach to FOP which consists of three phases [2, 3]. First, the change operations have to be captured into first-class entities. Second, those entities have to be classified in features (= separate change sets that each implement one functionality). Finally, those feature modules can be recomposed in order to form software variations that provide different functionalities.

In previous work, we already elaborated on two techniques to capture change objects. A classic way is to take two finished versions of a software system and to execute a Unix `diff` command on their respective abstract syntax trees [4], revealing the changes. This approach, however, only works *a posteriori*, and at a high level of granularity (the version level). A more subtle alternative is to log the developer's actions as he is performing the changes. The latter approach is based on change-oriented programming (ChOP) and was proven to provide a more complete overview of the history of development actions [5].

In this paper, we focus on the classification of changes into features. Classification has two aspects: the classification model and the classification technique, which is embodied by the different software classification strategies.

[*] Research funded by the Varibru research project initiated in the framework of the Brussels Impulse Programme for ICT supported by the Brussels Capital Region

2 Classification model

The classification model is a metamodel that consists of two parts: the change model and the actual classification model. Each part focuses on another level of granularity. The change model describes how the changes are modeled. Figure 1 shows that the change model separates between four kinds of changes, which can be composed. Atomic changes have a `subject`: the program building block affected by the change and defined by the Famix metamodel [6].

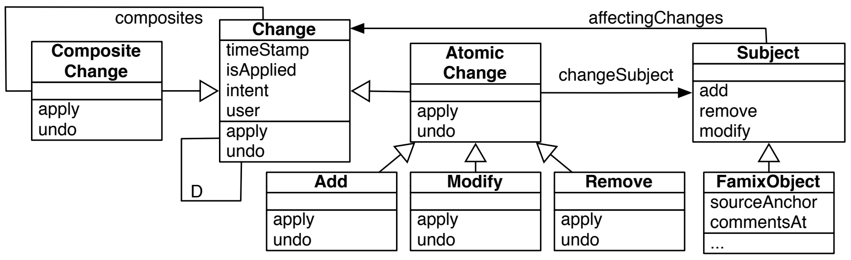

Fig. 1. Change Model

The actual classification model defines and describes the entities of the superstructure which is a flexible organisational structure based on feature and change objects. Figure 2 shows that the model contains three relations: D (the structural dependencies between the change objects), $CF4$ (which changes are grouped together into which feature) and Sub (which features are contained within another feature). The *cardinality* of $CF4$ and Sub specifies whether or not the sons (changes and/or features) have to be included in a composition that includes the parent (a feature). This information can afterwards be used to validate feature compositions (as in Feature Diagrams [7]).

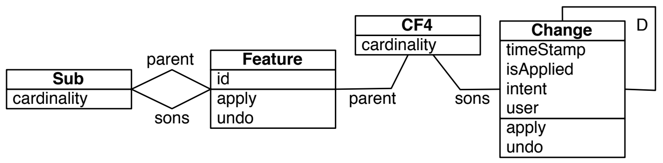

Fig. 2. Classification Model

3 Classification techniques

A classification strategy is a method for setting up classifications. Many classification strategies can be devised ranging from setting up classifications manually to generating classifications automatically. We present three classification strategies: *manual* classification, *semi-autmatic* classification through clustering and *automatic* classification through forward tagging.

3.1 Manual classification

Manual classification is the simplest classification strategy: manually putting change objects in features. The strategy can be used by the software engineer to group changes according to his wishes. Since our classification model states that a change can only be classified in one feature, this strategy should be supported by a tool which enforces that rule.

The advantages of this strategy are twofold. First, it is a very straightforward technique which can easily be implemented. Second, it can be applied on change objects that were obtained both with a *diff* and *logging* strategy. The main disadvantage is the tediousness that comes with the manual effort of this strategy.

3.2 Semi-automatic classification

Change objects contain information about *by whom*, *when*, *why* and *where* the operations they reify were carried out. Using clustering techniques [8] based on metrics on these properties, change objects can be grouped. This classification is basically a manual classification strategy. Based on the the clusters of changes, the developer decides on how the changes must be classified.

The main advantage of this strategy is that it can be used to assist the developer doing a manual strategy. The disadvantages of this strategy are threefold. First, it is more difficult to implement (clustering should be supported). Second, different parameters in the metrics might give different clustering results. Extra research is required to find adequate parameters. Third, this success of this strategy depends on the amount of information available in the change objects and is consequently not recommended to be used in combination with a *diff* strategy.

3.3 Automatic classification

In many cases, a manual classification strategy is not a feasible option. For large software systems it would take a long time to classify all classes by hand. Often classification of a software system is an activity that cannot be done by one software engineer alone since one software engineer seldom knows the whole system. When manual classification is not a valid option for the classification problem at hand automatic classification may provide a solution.

The idea behind automatic classification is that when software engineers carry out a development operation, for example implementing a new or changed specification or fixing a bug, they usually know the context in which changes are made. Moreover, the IDE knows the exact time and in what part of the software, the operation is performed. In stead of keeping this knowledge implicit in the heads of the developers, it is *tagged* into the changes partially by the developer and partially by the IDE. Afterwards, these tags can be processed automatically to generate tag-based classifications. Since software engineers are usually lazy when source code documentation is concerned, relying on discipline is not realistic. It is up to the IDE to make sure that classification knowledge about the software is recorded.

Advantages of this approach are that it can be used on the biggest of systems as it does not require manual labour, and that it is relatively easy to implement.

The sole inconvenience is that it can only be used in combination with a *logging* strategy, which enforces developers to do forward tagging.

4 Conclusion

We introduced a model and three strategies to classify changes and/or features in sets that represent features. The model consists of two parts which respectively model the change objects and the actual classification. The first strategy is straightforward: *manual* classification is a strategy to put together classifications manually. *Semi-automatic* classification is based on clustering changes together based on properties such as *by whom*, *when*, *why* and *where* the changes were applied. *Automatic* classification is based on forward tagging, and automatically groups changes together. Our findings are summarised as follows:

	Manual	Semi-Auto	Automatic
Capturing Changes	diff, logging	logging	logging
Amount of manual labour	high	average	low
Error probability	high	high	low

Only the automatic strategy is usable in a context of large-scale software systems. As that strategy requires logging as a technique to capture changes, we conclude that the development environment should support logging and enforce forward tagging, so that the changes can automatically be classified in recomposable feature modules.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 187–197
2. Ebraert, P., Van Paesschen, E., D'Hondt, T.: Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel (2007)
3. Ebraert, P., Vallejos, J., Costanza, P., Van Paesschen, E., D'Hondt, T.: Change-oriented software engineering. In: ICDL '07: Proceedings of the 2007 international conference on Dynamic languages, New York, NY, USA, ACM (2007) 3–24
4. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proceedings of the 20th International Conference on Automated Software Engineering. (2005)
5. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (2007) 93–109
6. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne (1999)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
8. Romesburg, C.: Cluster Analysis for Researchers. Number 978-1-4116-0617-3. Krieger (1990)