Feature-oriented programming based on first-class changes

Peter Ebraert*, Theo D'Hondt
Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
pebraert@vub.ac.be, tjdhondt@vub.ac.be

Abstract

A growing trend in software construction advocates the encapsulation of software building blocks as features which better match the specification of requirements. Feature-oriented programming (FOP) is the research domain that targets this trend. We argue that the state-of-the-art approaches to FOP are not satisfactory because they only provide top-down methodologies to FOP. We propose to specify features as sets of first-class change objects which can add, modify or remove building blocks to or from a software system. We present ChEOPS, a proof-of-concept implementation of this approach and use it to show how this enables bottom-up FOP.

1 Feature-oriented programming

Feature-oriented programming (FOP) is a programming paradigm that targets the separation of concerns [15]. In FOP, every concern is modularised as a separate feature: a first-class entity that forms the basic building block of a software system [4]. Features satisfy intuitive user-formulated requirements on the software system and can be composed to form different variations of the same software product [2, 3].

Aspect-oriented programming (AOP) [12] is another such programming paradigm. Aspects focus on the quantification – by specifying predicates that identify join points at which to insert code. Feature implementations are much closer to framework designs. That is, to add a feature to a framework, there are predefined building blocks that are to be extended or modified. In such designs, there is little or no quantification, but there are indeed "cross-cuts". *Mixin Layers* [17], *AHEAD* [5], *FeatureC++* [1], *Composition Filters* [6] and *Delegation Layers* [14] are state-of-the-art approaches to FOP that implement features by cross-cuts.

The commonality between all these approaches is that they all (a) allow a *top-down methodology* to FOP. This means that, in order to do FOP with one of those approaches, the development methodology has to foresee feature modularisation from the start. We propose an approach that provides a bottom-up methodology to FOP which can be applied on any programming language and interactive development environment (IDE). We believe that this might popularise FOP in industry, where people do not want to deviate from their development methodologies and environments.

2 Change-oriented programming

In [9] and [10] we propose change-oriented programming (ChOP): an approach that centralises change as the main development entity. Some examples of developing code in a change-oriented way can be found in most IDEs: the creation of a class through interactive dialogs or the modification of the code by means of an automated refactoring. ChOP goes further than that, however, as it requires all building blocks to be created, modified or deleted in a change-oriented way (e.g. adding a method to a class, removing a statement from a method, etc).

Together with us, other researchers pointed out the use of encapsulating change as first-class entities. In [16], Robbes shows that the information from the change objects provides a lot more *information about the evolution* of a software system than the central code repositories. In [7], Denker shows that first-class changes can be used to define a scope for dynamic execution and that they can consequently be used to *adapt running software systems*. In this section, we first explain how ChOP works in practice. Then, we show how ChOP enables a bottom-up FOP methodology that allows a language-independent de- and re-composition of software systems.

*Research funded by a doctoral scholarship of the IWT Vlaanderen and by the Varibru research project initiated in the framework of the Brussels Impulse Programme for ICT supported by the Brussels Capital Region

2.1 First-class change objects

In [9], we explain how a software system can be specified by a set of first-class change-objects that represent the development actions that have to be taken to produce that system. A classic way to obtain the changes between two versions of a software system is based on the executing of the Unix `diff` command on the abstract syntax tree of the source code of both versions. Xing et al describe how this is done in [19]. Another way is to log the developer's actions in the IDE. In order to do that, the IDE needs to be instrumented with functionalities to capture these actions in first-class change objects. We opt for the latter approach since it provides a more complete overview of all development actions [16].

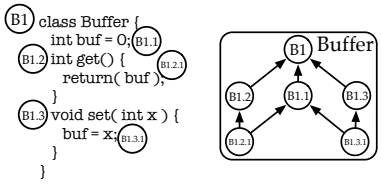

Figure 1. Buffer: (left) source code (right) change objects

Figure 1 shows the source code (on the left) of a `Buffer` application. The change objects that represent the incremental development actions taken to implement that version of the `Buffer` are depicted on the right. Those objects are identified by a unique ID: `B1` is a change that adds a class `Buffer`, `B1.2.1` is a change that adds an access of the instance variable `buf`.

The dependencies between change objects are also maintained: `B1.2.1` depends on the change that adds the method to which `buf` is added (`B1.2`) and on the change that adds the instance variable that it accesses (`B1.1`). We distinguish between two kinds of dependencies: *syntactic* dependencies – imposed by the meta-model of the used programming language and exemplified above – and *semantic* dependencies – that depend on domain knowledge and exemplified in the following section.

2.2 Specifying features

The implementation of a functionality, yields a set of change objects that are related by the functionality they implement. Consequently, the specification of a feature boils down to the grouping change objects. Figure 2 shows two functionalities that we add to the buffer: `Restore` allows the buffer to restore its previous value, `Logging` makes sure that all methods of the buffer are logged when executed. The change objects of each functionality are surrounded with a line: The red changes represent a feature that implements the `Restore` functionality, the blue changes represent a feature that implements the `Logging` functionality. Note that some dependencies do not reside within one feature. The number of such dependencies might provide a representative metric to measure the coupling between the features of a software system.

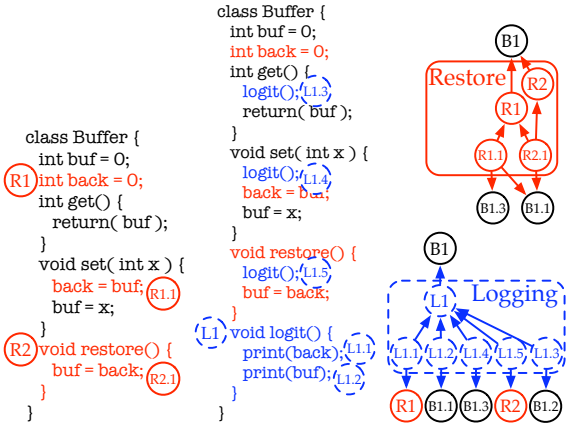

Figure 2. (left) Restore source code, (middle) Logging source code, (right) change objects

The dashed line grouping `Logging`'s changes does not only denote that these changes implement the `Logging` feature, but also that `Logging` is a *flexible* feature. The difference between *flexible* (dashed line) and *monolithic* features (full line) is that the latter can only be composed as a whole, while the former can be composed partially as elaborated on below.

2.3 Software composition

In FOP, variations of a software system may be produced by specifying a composition of the required features. In our model, a feature composition is valid if the union of the change sets that implement the features in that composition does not contain a change that has a dependency to a change object that is not in the composition. Following this definition, adding a flexible feature (like Logging) to a composition is always possible, as the change objects from such feature could be excluded from the composition in case they would have a dependency to a change object that is not in the composition.

Change objects and the dependencies amongst them can be visualised by a *directed acyclic graph*. The left part of Figure 3 shows the graph of a `Buffer` with the `Restore` and `Logging` features. Change objects with a red full line belong to the monolithic `Restore` feature. Change objects with a blue dashed line belong to the flexible `Logging`.

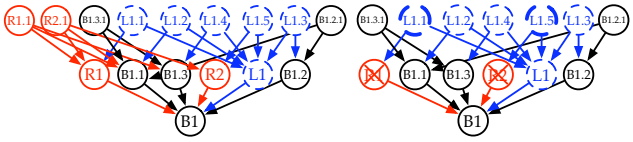

Figure 3. Composition based on first-class changes

The right part of Figure 3 presents a composition of a `Buffer` with `Logging`. `L12` and `L15` would cause the composition to be invalid (according to the definition of validity given above). The problem is that those changes respectively depend on `R1` and `R2` which are not in the composition. The semantic information stating that `Logging` is flexible, allows the exlusion of `L11` and `L15` from the composition. This results in a valid composition that specifies a buffer with a logging feature. The information about what changes are not going to be applied (`L11` and `L15`) and the information about what actions can be taken to include those changes (`R1` and `R2`) can be presented to the developer to assist in fixing composition bugs.

3 Proof-of-concept implementation

Change and evolution-oriented programming support (ChEOPS) is an IDE plugin for VisualWorks, which we created as a proof-of-concept implementation of ChOP. ChEOPS fully supports ChOP but also has the capability of logging developers producing code in the standard OO way. Behind the scenes, ChEOPS produces fine-grained first-class change objects that represent the development actions taken by the developer. The UML class diagram of the model's core is presented in Figure 4.

We identify three possible actions a developer can take to produce software systems: the addition, the removal and the modification of software building blocks. We model those commands with the classes `Add`, `Remove` and `Modify` respectively. Together, they form the concrete commands of the Command design pattern [11]. The `Atomic Change` class plays the role of the abstract Command class in the Command design pattern. Next to that, it also fullfils the responsibilities of the Leaf participant in the Composite design pattern [11]. A `Composite Change` is composed of `Changes` (which can in their turn be of any change kind), that have to be applied as a transaction.

The syntactic dependency relation between changes is modeled as a many-to-many relationship from `Change` to `Change`. For speed optimisation, syntactic dependencies are maintained in both directions. Every `Change` maintains a reference to all changes that it syntactically depends on and to all changes that syntactically depend on it. As for the

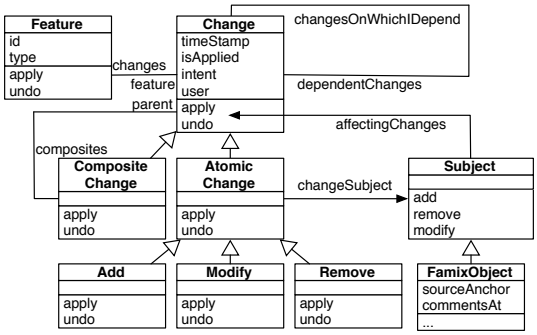

Figure 4. ChEOPS - Core Model

semantic dependencies, they are maintained by a reference from `Change` to `Feature`. `Feature` is a collection class that has a unique name, a type (Flexible or Monolithic) and a set of changes of which it consists. ChEOPS maintains the syntactic dependencies in an automatic way and supports semantic dependencies by allowing the grouping of change objects in a set that implements one feature.

The `Subject` of the change is a building block of the programming language used to develop the software system. The different building blocks of a programming language are specified by the meta-model of that programming language. As a meta-model, we choose the *FAMOOS Information Exchange* (FAMIX) model because (a) it allows the expression of building blocks till the level of statements and (b) it provides a generic model to wich most class-based programming languages (e.g. Java, C++, Ada, Smalltalk) adhere.

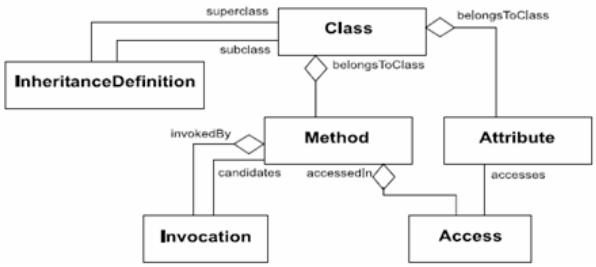

Figure 5. Famix - Core Model

FAMIX was created to support information exchange between interacting software analysis tools by capturing the common features of different object-oriented programming languages needed for software re-engineering activities [8, 18]. Figure 5 shows that the core of the FAMIX model consists of Classes, Methods, Attributes and relations between them. Thanks to the `Invocation` and `Access` relations, our model allows the specification of changes on the level of granularity of statements.

4 Advantages

The major advantage of specifying features by change objects is that it enables a methodology for *bottom-up FOP*. Instead of having to design a complete feature-oriented application up-front (top-down), our approach allows the development of such an application in a standard OO way after which functionalities are decomposed in feature modules (bottom-up). In [13], Liu shows that Ahead can also be used to do bottom-up FOP, but that it requires manual annotation of all building blocks with information that denotes the feature that building block belongs to. That is a tedious task in comparison to our approach.

A second advantage of our approach is its *expressiveness*. While in the state-of-the-art approaches to FOP, a feature can only be specified as a set of program building blocks that might extend or modify existing building blocks, our approach also allows the specification of deletion of building blocks. Finally, our approach is *applicable to any programming language* that has a meta-model specifying its building blocks and the syntactic dependencies amongst them. The decomposition of programs in feature modules that contain changes and the recomposition algorithms to produce valid software variations are unaffected by the asserted programming language.

5 Conclusions

We present feature-oriented programming (FOP) as a good development technique to modularize software systems. We find the state-of-the-art approaches to FOP not satisfactory because they only provide top-down approaches to FOP. We present a model of first-class changes which can add, modify or delete building blocks to or from a software system. We propose to specify features in terms of those changes. The dependencies between the change objects provide the necessary information to validate feature compositions. We present ChEOPS, a proof-of-concept implementation of our approach and show how it can be used to do *bottom-up FOP* and conclude that other advantages of our approach include its *expressiveness* and *applicability to any programming language*.

References

[1] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[2] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating reference architectures: an example from avionics. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 27–37, New York, NY, USA, 1995. ACM.

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] D. S. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *GTTSE*, pages 3–35, 2006.

[6] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, 2001.

[7] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[8] S. Ducasse and S. Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, 1999.

[9] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 3–24, New York, NY, USA, 2007. ACM.

[10] P. Ebraert, E. Van Paesschen, and T. D'Hondt. Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel, 2007.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM.

[14] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 89–110, London, UK, 2002. Springer-Verlag.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, dec 1972.

[16] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, pages 93–109, 2007.

[17] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[18] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[19] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering*, 2005.