First-class change objects for feature oriented programming

Peter Ebraert
Programming Technology Lab
Vrije Universiteit Brussel Pleinlaan 2, B-1050 Brussel, Belgium
pebraert@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. As a result, programmers find it easier to design and compose different variations of their systems. Feature-oriented programming (FOP) is the research domain that targets this trend. We argue that the state-of-the-art techniques for FOP have shortcomings because they specify a feature as a set of building blocks rather than a transition that has to be applied on a software system in order to add that feature's functionality to the system. We propose to specify features as sets of first-class change objects which can add, modify or delete 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 our approach contributes to FOP on three levels: expressiveness, composition verification and bottom-up development.

1 Feature oriented programming

Pioneering work on software modularity was made in the 70's by Parnas [17] and Dijkstra [10]. Both have proposed the principle of separation of concerns that suggests to separate each concern of a software system in a separate modular unit. According to these papers, this leads to maintainable, comprehensible software that can easily be reused, configured and extended. Feature Oriented Programming (FOP) is an implementation of that idea that modularises every concern as a separate feature. In FOP features are the basic building blocks that are raised to first-class entities [4]. They satisfy intuitive user-formulated requirements on the software system. A software product is built by composing features. Many case studies show that FOP is an appropriate technique to compose many variations of the same software product (e.g. [8, 3, 2, 5]).

Aspect-oriented programming (AOP) [14] is another implementation of that idea. Aspects focus on the quantification – by specifying predicates that identify join points at which to insert code, feature implementations are actually 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" [6]. *Mixin Layers* [19], *AHEAD* [6], *FeatureC++* [1], *Composition Filters* [7] and *Delegation Layers* [16] are all state-of-the-art approaches to FOP that implement features by cross-cuts that are *modifications* or *extensions* to multiple software building blocks.

The problem that we see with all approaches to FOP, is that they all specify a feature by a set of building blocks, rather than by a program transition that modifies a program is such a way that the functionality – that that feature implements – is added. In [6], Batory already pointed out that a feature can be looked at as a function that is applicable on a base (a set of program building blocks). The application of a feature on a base yields that the base is extended or modified with the building blocks specified by that feature. From that point of view, a software composition is a sequence of applied features to one base. AHEAD [6] is an algebra that formalises how features can be composed as functions.

We strongly agree with that vision, but find that features should not be limited to extend or modify existing programs. In some situations, a feature should also be able to remove building blocks from a program. Examples of such cases include anti-features (a functionality that a developer will charge users to not include, the creation of a demo-application which consists of all features but only to a certain extent, or the customisation of certain features so that the software system copes with specific hardware requirements (e.g. limited memory or computation power).

2 Change as first-class objects

Together with us, other researchers pointed out the use of encapsulating change as first-class entities. In [18] 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 [9], 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 a model of first-class changes and then show how these changes can be used to do FOP.

2.1 Model of changes

We use the FAMIX model [20] to express the building blocks of a software system. We chose FAMIX since it provides a generic model to wich most class-based programming languages (e.g. Java, C++, Ada, Smalltalk) adhere. Figure 1 shows that the core of the FAMIX model consists of Classes, Methods, Attributes and relations between them.

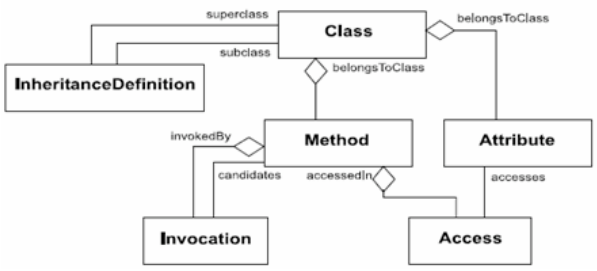

Figure 1. Famix - Core Model

The model of changes expresses the different kinds of change operations that can be applied on those building blocks. The UML class diagram of the model's core is presented in Figure 2. The building blocks that are specified by the FAMIX model (`FamixObject`) form the `Subject` of an `Atomic Change`. We identify three possible commands on those subjects: the addition, the removal and the modification of the building block. We model those commands with the classes `Add`, `Remove` and `Modify` respectively. A `Composite Change` is composed of `Changes` (which can in their turn be of any change kind). An elaborated discussion about atomic and composite changes is is omitted because it does not resides in the scope of this paper.

The figure shows a dependency relation between the change objects, that is explained deeper in the following section. Note that, thanks to the granularity of the FAMIX model, our model allows the specification of changes on the level of granularity of invocations and accesses (below method level). For more information about the model of changes, we refer to [11].

2.2 Change-oriented programming

In [12] and [13] we propose change-oriented programming (ChOP): an approach that centralises change as the

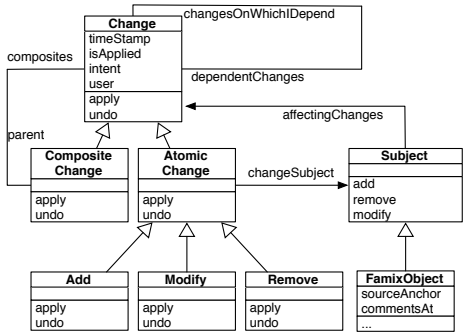

Figure 2. ChEOPS - Core Model

main development entity. Some examples of developing code in a change-oriented way can be found in most interactive development environments (IDE): 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 and deleted in a change-oriented way (e.g. adding a method to a class, removing a statement from a method, etc).

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.

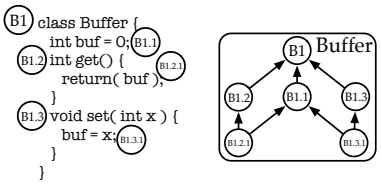

Figure 3. Buffer: (left) source code (right) change objects

Figure 3 shows the source code (on the left) and the changes (on the right) of a `Buffer`. The change objects are identified by a unique number: `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 by ChEOPS: `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: *syn-*

tactic dependencies – imposed by the meta-model of the used programming language and exemplified above – and *semantic* dependencies – that depend on domain knowledge. ChEOPS supports the former in an automatic way and the latter by allowing the grouping of change objects in sets that represent features – denoted by the rounded squares surrounding change objects.

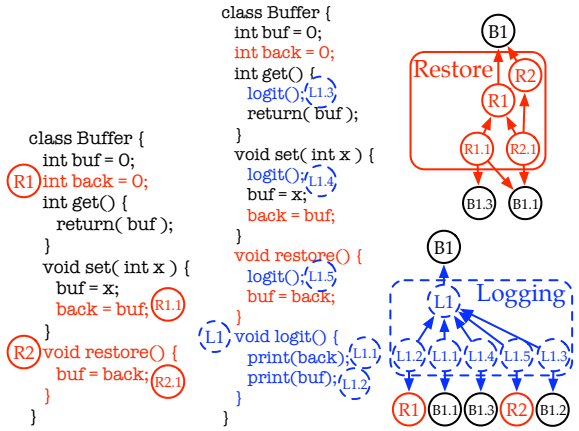

Figure 4. (left) Restore source code, (middle) Logging source code, (right) change objects

Figure 4 shows two extra features: `Restore` allows the buffer to restore its previous value, `Logging` makes sure that all methods of the buffer are logged when executed. Notice the dashed line surrounding `Logging`'s changes: It not only denotes that these changes implement the `Logging` feature, but also that `Logging` is a *flexible* feature. The difference between *flexible* (dashed lines) and *monolithic* features (full lines) is that the latter can only be applied as a whole, while the former can be applied partially. This information is used to verify whether a feature composition is valid, as elaborated on in the next section.

2.3 Software composition

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 graph*. The left side of Figure 5 shows the graph of the `Buffer` with the `Restore` and `Logging` features. The colors of the change objects denote the semantic dependencies that exist between them. 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`.

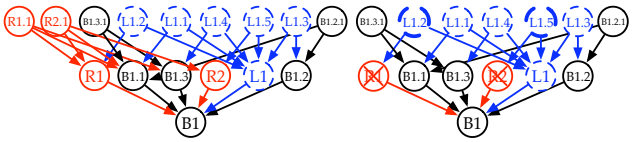

Figure 5. Composition based on first-class changes

The right part of Figure 5 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 the Logging feature is flexible, allows the exlusion of `L12` and `L15` from the composition. This results in a valid composition {`B1`, `B11`, `B12`, `B13`, `B121`, `B131`, `L1`, `L11`, `L13`, `L14`} that specifies a buffer with a logging feature. The information about what changes are not going to be applied (`L12` and `L15`) and the information about changes that should be added (`R1` and `R2`) for the composition to be valid can be presented to the developer in order to assist in producing a valid composition.

3 Advantages

We see three advantages in the specification of features in terms of fine-grained first-class change objects: *increased expressiveness*, *improved composability* and a novel *bottom-up approach to FOP*.

In comparison with state-of-the-art approaches to FOP, which allow the specification of features as a set of program building blocks that might extend or modify existing building blocks, our approach allows a *more expressive feature specification*. Features do not only express the building blocks that implement a feature, but also how that feature can be added to a software composition. Next to that, features can express changes up to statement level, which is more fine-grained than the state-of-the-art (only allowing the expression of addition or modification on class or method level). Finally, features can include the deletion of certain building blocks, which is not supported by the state-of-the-art.

The dependencies between change objects provide the fine-grained information that is required to *verify whether a certain feature composition is valid*. The notions of monolithic and flexible features allow distinguishing between

change sets that must be applied as a whole (the former) or that can be applied partially (the latter). This semantic information allows for more flexible compositions than the state-of-the-art approaches to FOP.

The final advantage of specifying features by change objects is that it enables a methodology for a *bottom-up approach to FOP*. Instead of having to specify a complete design of a feature oriented application before implementing it (top-down), our approach allows the development of such an application in an incremental way. Some state-of-the-art approaches also provide an implementation of this bottom-up approach. In [15], Liu shows that the ATS can be used to do so by manually annotating all building blocks with information that denotes the feature that building block belongs to. That is a tedious task in comparison to our approach.

4 Conclusions

In this paper, we advocate feature oriented programming (FOP) as the right development technique to modularize software systems. We find the state-of-the-art approaches to FOP not satisfactory and present an alternative approach based on the specification of features by sets of change objects rather than program building blocks. Features are functions that can be applied to add the functionality they implement.

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. This increases the expressiveness of features as they can specify adaptations to fine-grained building blocks (classes, methods, attributes and statements). The dependencies between the change objects provide the necessary information to validate feature compositions. Specifying features this way allows a bottom-up approach to do FOP.

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 stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] D. Batory and J. Thomas. P2: A lightweight dbms generator. Technical report, University of Texas at Austin, Austin, TX, USA, 1995.

[6] D. S. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *Generative and Transformational Techniques in Software Engineering*, pages 3–35, 2006.

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

[8] A. Brown, R. Cardone, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 76–85. ACM Press, 2002.

[9] 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.

[10] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[11] P. Ebraert, B. Depoortere, and T. D'Hondt. A meta-model for expressing first-class changes. In T. Mens, K. Mens, E. Van Paesschen, and M. D'Hondt, editors, *Proceedings of the Third International ERCIM Symposium on Software Evolution*, October 2007.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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*, 11(2):215–255, 2002.

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