Software variation by means of first-class change objects

Peter Ebraert, Leonel Merino, Theo D'Hondt
Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium.

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 Software product lining

Customers are becoming more and more demanding and cost-conscious. They want specific products that exactly cope with their needs at the lowest cost possible. From the producers point of view, these two requirements are usually conflicting. The development of a specific product for every client takes a lot of time and will consequently be more expensive. The development of a more generic product is cheaper but usually does not exactly cope with the specific needs of the customer.

In order to find the good balance of both requirements, producers tend to use a business strategy called *product lining*: offering for sale several related products of various sizes, types, colors, qualities or prices. The more variations the product line offers, the more specific and expensive its products tend to get. The fewer variations the product line contains, the cheaper and less specific its products become. Adopting this business strategy, the producer's goal boils down to cover the entire scope of the product line, at the lowest possible production cost.

Software companies are the producers of either pure software products or products with an important software component (embedded systems). Driven by consumer's demand, they are also forced to increase variability of their products. Over the last decade, the management of this variability has become a major bottleneck in the development, maintenance and evolution of software products. Next to that, many companies do not even reach the desired level of variability or fail to do so in a cost efficient manner. An explanation of this can be found in the development approaches used by those companies.

A fundamental problem with many current development approaches is that they view systems from the perspective of producers, rather than consumers. Producers tend to specify their systems in terms of *software building blocks* while the consumers tend to specify requirements primarily in terms of *features*. This mismatch complicates variability, since there is no direct mapping between a composition of features and the software building blocks that implement that composition. Recent research in software construction increasingly reflects a common theme: the need to realign modules around features rather than software building blocks [1].

Feature-oriented programming (FOP) is the study of feature modularity, where features are raised to first-class entities [2]. 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. Many case studies show that FOP is an appropriate technique to cope with the problems stated above (e.g. [3], [4], [5], [6]).

The following section briefly explains FOP, the state-of-the-art approaches to FOP and their limitations. Section 3 proposes an alternative way to specify features and shows how this overcomes the limitations that are pointed out in Section 2. Section 4 shows the advantages of specifying features with first-class changes. Conclusions and future work are pointed out in Section 5.

2 Feature-oriented programming

Pioneering work on software modularity was made in the 70's by Parnas [7] and Dijkstra [8]. 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

• *E-mail: {pebraert},{lmerinod},{tjdhondt}@vub.ac.be*

Research funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (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.

can easily be reused, configured and extended. FOP is an implementation of that idea and modularises every concern as a separate feature.

Aspect-oriented programming (AOP) [9] 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” [10]. *Mixin Layers* [11], *AHEAD* [10], *FeatureC++* [12], *Composition Filters* [13] and *Delegation Layers* [14] 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 [10], 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 [10] 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 [1], 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).

3 CHANGE AS FIRST-CLASS OBJECTS

Together with us, other researchers pointed out the use of encapsulating change as first-class entities. In [15] 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 [16] 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 *feature-oriented programming*.

1. An anti-feature example can be found in the camera industry. While it is more difficult for producers to make a camera that outputs JPEG than a camera that outputs RAW, they charge more for a camera that can output RAW than an identical one that can output JPEG.

3.1 Model of changes

We use the FAMIX model [17] 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.

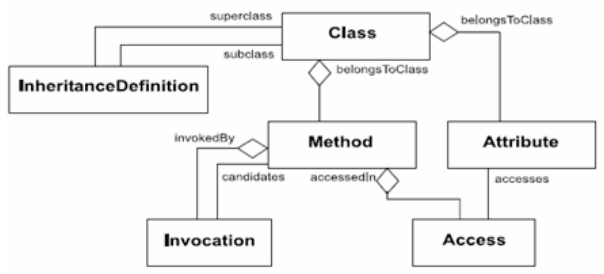

Fig. 1. Famix - Core Model

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.

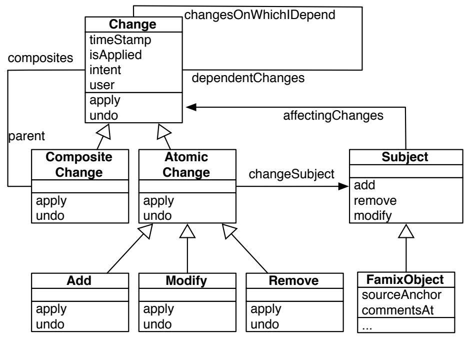

Fig. 2. ChEOPS - Core Model

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 [18].

3.2 Change-oriented programming

In [19] and [20] 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 interactive development environments (IDE): the creation of a class though 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 implements the model of changes that was described above and does not fall back on ChangeList [21] – a change management tool included in most Smalltalk IDEs. Reasons for this are elaborated on in [19].

ChEOPS fully supports change-oriented programming but also has the capability of logging developers producing code in the standard OO way. For that, ChEOPS instruments the IDE with hooks and uses them to produce fine-grained first-class change objects that represent the actions taken by the developer. Those objects can later be grouped into a change set that specifies a transition which might be applied to a base in order to extend the base with the feature expressed by that change set.

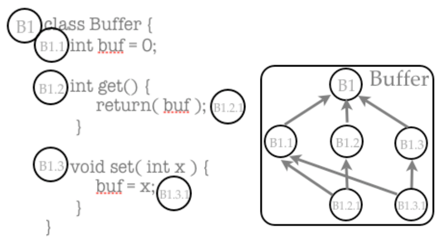
```
class Buffer {
  int buf = 0;
  int get() {
    return( buf );
  }
  void set( int x ) {
    buf = x;
  }
}
```

Fig. 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: *syntactic* dependencies – imposed by the meta-model of the used programming language (FAMIX) 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. Grouping changes in ChEOPS can be done in an ad-hoc way after making the changes, or up-front by letting the IDE know that all changes in the coming session will implement one feature. The latter seems better as it is less tedious, but requires a developer to cleanly split the development in separate sessions that each implement one feature.

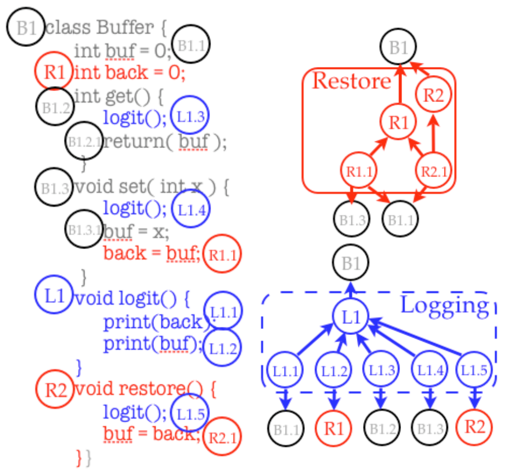
```
class Buffer {
  int buf = 0;
  int back = 0;
  int get() {
    logit();
    return( buf );
  }
  void set( int x ) {
    logit();
    buf = x;
    back = buf;
  }
  void logit() {
    print(back);
    print(buf);
  }
  void restore() {
    logit();
    buf = back;
  }}
```

Fig. 4. Buffer, Restore, Log features: (left) 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. ChEOPS uses this semantic information to verify whether a feature composition is valid and to support the developer in resolving composition conflicts. The next section elaborates on how ChEOPS does this.

4 Advantages for Software Variation

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 composition. Next to that, features can express changes below statement level, which is more fine-grained than the state-of-the-art. Finally, features can include the deletion of 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*. In this model, a feature composition is valid if the union of the change sets of the features in that composition does not contain a change that has a dependency to a change object that is not in the composition. In case we want to make a composition of `Buffer` and `Logging`, `L12` and `L15` would form a problem as they 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. In case the `Logging` feature would not be flexible, the dependencies could be used to tell the developer where the composition is failing (`L12` and `L15`) and what actions could be taken to resolve the conflict (add a feature that includes changes `R1` and `R2`).

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 [22], 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.

5 Conclusions

In this paper, we advocate feature-oriented programming (FOP) as the right development technique for software companies to provide variation in their software products for satisfying the demand of customers who are becoming more and more demanding and cost-conscious. 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 first-class 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 fine-grained information to validate feature compositions. Finally, this way of specifying features allows a bottom-up approach to do FOP.

Acknowledgment

We would like to express are gratitude to Jorge Vallejos, Tom Tourwé and Pascal Costanza for their valuable contributions to this research.

References

[1] M. Kratochvíl and C. Carson, *Growing Modular. Mass Customization of Complex Products, Services and Software*. Springer, March 2005, no. 3540239596.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197.

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

[4] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 4, pp. 355–398, 1992.

[5] 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*. New York, NY, USA: ACM, 1995, pp. 27–37.

[6] D. Batory and J. Thomas, "P2: A lightweight dbms generator," University of Texas at Austin, Austin, TX, USA, Tech. Rep., 1995.

[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM*, vol. 15, no. 12, pp. 1053–1058, dec 1972.

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

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th Europeen Conf. Object-Oriented Programming*, ser. LNCS, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer Verlag, 1997, pp. 220–242.

[10] D. S. Batory, "A tutorial on feature oriented programming and the ahead tool suite," in *GTTSE*, 2006, pp. 3–35.

[11] 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)*, vol. 11, no. 2, pp. 215–255, 2002.

[12] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming," in *GPCE*, ser. Lecture Notes in Computer Science, R. Glück and M. R. Lowry, Eds., vol. 3676. Springer, 2005, pp. 125–140.

[13] L. Bergmans and M. Akşit, "Composing crosscutting concerns using composition filters," *Comm. ACM*, vol. 44, no. 10, pp. 51–57, 2001.

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

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

[16] 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*. New York, NY, USA: ACM, 2007, pp. 25–49.

[17] S. Tichelaar, "Modeling object-oriented software for reverse engineering and refactoring," Ph.D. dissertation, University of Bern, 2001.

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

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

[20] P. Ebraert, E. V. Paesschen, and T. DíHondt, "Change-oriented round-trip engineering," Vrije Universiteit Brussel, Tech. Rep., 2007.

[21] University of Illinois, "Visualworks: Change list tool," http://wiki.cs.uiuc.edu/VisualWorks/Change+List+Tool, 2007.

[22] 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*. New York, NY, USA: ACM, 2006, pp. 112–121.