Intensional changes avoid co-evolution!

Bart Meyers
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
bart.meyers@ua.ac.be

Peter Ebraert*
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
peter@ebraert.be

Dirk Janssens
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
dirk.janssens@ua.ac.be

ABSTRACT

Modularization is key to support the maintainability of software systems. In some cases, however, maintenance requires certain modules to evolve together. This phenomenon complicates software maintainability and is commonly referred to as co-evolution.

In this paper, we tackle co-evolution in the domain of change-based feature-oriented programming (ChOP). In ChOP, feature modules – each matching the implementation of one requirement – are specified as sets of first-class change objects. Our solution is based on *intensional* changes: descriptive changes that are automatically evaluated with respect to the other feature modules before they are applied. We present a maintenance scenario and use it to show how intensional changes avoid co-evolution.

1. CO-EVOLUTION

Software systems are often the subject to changing requirements [15]. As a consequence, these systems have to be adapted and maintained. Highly modular systems are more cost-effective when evolved, as in many cases evolution can be performed locally, only affecting a part of the system [13]. In such systems, the modules are usually loosely coupled, i.e., the inter-dependencies are kept to a minimum. As a consequence, the modules themselves are highly cohesive.

Consider the evolution scenario depicted in Figure 1 in which the modules are represented as rounded rectangles. The initial version of the system is shown on the left side of the figure. It consists of only one module: `Buffer`. Due to changing requirements, the buffer is extended by a logging functionality, implemented in the `Logging` module. The goal of the logging functionality is to log every action that is performed when using the system. The modular feature-oriented implementation of the `Buffer` ensures that every functional requirement is implemented in a separate module. Moreover, it envisions that the implementation of a new requirement (such as the need for a logging functionality) only affects one module (`Logging`) and leaves the other ones (`Buffer`) unaffected. As the logging functionality for a buffer can only be added to a system that implements the buffer functionality, a dependency arises between the `Logging` and `Buffer` modules. This dependency means that `Logging` can only be implemented if `Buffer` is implemented. The resulting system is shown in the center of Figure 1.

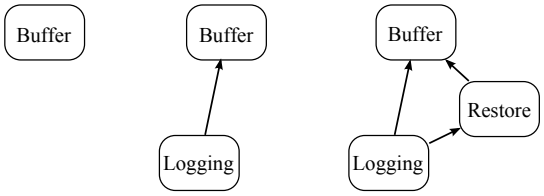

Figure 1: Modular system development: first version (left), second version (middle), third version (right)

After a period of time, yet another requirement arises. Suppose the system is extended by a roll-back functionality and that this functionality is implemented by the `Restore` module. As shown on the right side of Figure 1, the addition of the `Restore` module introduces a new dependency between the `Restore` and the `Buffer` modules. Again, the `Buffer` module is not affected by the introduction of the `Restore` module. A second dependency between the `Logging` and `Restore` modules, however, is also introduced. Moreover, when adding the `Restore` module, the `Logging` module must also be adapted as roll-back actions have to be logged as well.

Functionalities such as logging are named *crosscutting* as in practice, they are tightly coupled to other modules and thus can not be easily modularized [10]. Consequently, the code implementing crosscutting functionality is often scattered over and tangled across a system, leading to quality, productivity and maintenance problems [13]. When for example a new functionality is implemented, the crosscutting functionality often has to be adapted as well. This phenomenon is called *co-evolution* [5]. In this paper, we tackle this problem in the context of change-based feature oriented programming (introduced in Section 2). As several case reports claim that co-evolution hinders maintainability [1], we present a solution which decreases the need for co-evolution

*Dr. Ebraert is funded by the "Agentschap voor Innovatie door Wetenschap en Technologie" via the Optimma research project.

in Section 3. In Section 5 the approach is evaluated by applying it on a case study. We conclude in Section 6.

2. CHANGE-BASED FEATURE ORIENTED PROGRAMMING

In *Feature-Oriented Programming* (FOP), a software system is modularized based on the functionalities it provides. A module that adds a functionality to a software system is called a *feature*. In FOP, features are the main development entities [16, 17]. The idea of FOP is to produce software variations by composing the feature modules that provide the desired combination of functionalities. As a result of FOP, programmers find it easier to design and compose different variations of their systems. In other words, FOP enables software product lining. In previous work [9], we already pointed out that the state-of-the-art approaches to FOP (E.g. Mixin-layers [18], AHEAD [3], Lifting Functions [16], Composition Filters [4], FeatureC++ [2] and the aspect-oriented programming (AOP) approaches [13]) lack expressiveness and hinder the reusability of feature modules. In that same work, we propose change-based FOP, as an alternative FOP approach that addresses those problems.

Change-based FOP is based on change-oriented programming (ChOP) [8], which centralizes a change as the main development entity. Some examples of developing code in a change-oriented way can be found in most interactive development environments: the creation of a class through interactive dialogs or the modification of the code by means of an automated refactoring. This is different from manually modifying the source code in the sense that the change is applied in a single step, possibly driven by some parameters provided by the developer. ChOP goes even further, however, as it requires all software 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).

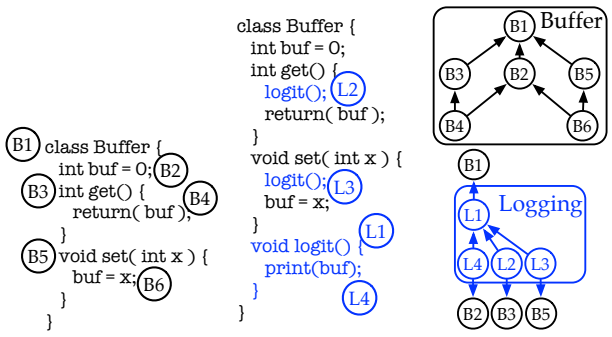

Figure 2: Buffer: source code (left), change objects (right)

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, change-based FOP allows to specify features as sets of changes that add, modify or delete software building blocks. This increases the control of how features can be specified in two ways. (a) Features can express changes down to the statement level, which is more fine-grained than the state-of-the-art (usually only allowing the expression down to the method level). (b) Features can include the deletion of certain building blocks, which is not supported by the state-of-the-art on FOP. Another advantage of specifying features by change objects is that it enables a method 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 development), change-based FOP allows the development of such an application in an incremental way (bottom-up development). Also top-down development or a combination of top-down and bottom-up approaches – in which the coarse-grained software structure is made in a top-down way and the detailed parts are constructed in a bottom-up way – is supported by change-based FOP. For a more excessive comparison between change-based FOP and the state-of-the-art to FOP, we refer to [9].

In the upper right part of Figure 2, the change-based specification of the `Buffer` module is shown. Within the `Buffer` module (the rounded rectangle), the changes that are instantiated for creating the `Buffer` are depicted (the labelled circles). By applying the changes, the source code (the left part of Figure 2) of the `Buffer` can be obtained. The change objects are identified by a unique number: $B1$ is a change that adds a class `Buffer`, $B4$ is a change that adds an access of the instance variable `buf`. In ChOP, the dependencies between change objects are also maintained: $B4$ depends on the change that adds the method to which `buf` is added ($B3$) and on the change that adds the instance variable that it accesses ($B2$). The set $\{B1, B2, B3, B4, B5, B6\}$ forms the `Buffer` feature. Note that the resulting source code is annotated. These annotations represent the causal link between the source code and the changes that relate to that code.

In change-based FOP, crosscutting functionality, such as logging, is represented by a feature that includes changes which affect (create, modify or remove) software building blocks scattered over the system. `Logging`, for instance contains changes which introduce code statements scattered around every method in the `Buffer` class. Figure 2 also presents the code (in the middle) and the change diagram (bottom right) of the logging functionality fort the buffer application. Note that, while `Logging` is a crosscutting feature, its change-based specification remains modularized.

A positive property of ChOP is that the implementation of crosscutting functionality is actually never scattered over the software application. All the changes of a feature are always grouped in one change set, no matter if it implements a crosscutting functionality or not. It is the application of the changes that actually produces a software system containing the scattered building blocks of the crosscutting functionality. This is similar to static weaving as exhibited by several AOP approaches [12, 11]. The difference is that change application operates on change objects, whereas static weaving typically operates on (byte) code.

The approach, however, still suffers from inconveniences with respect to co-evolution. Although modularized, the `Logging` feature still has to be adapted when adding, removing or changing features. In Figure 3 for example, the changes $L5$ and $L6$ have to be added to the `Logging` feature when implementing the `Restore` feature.

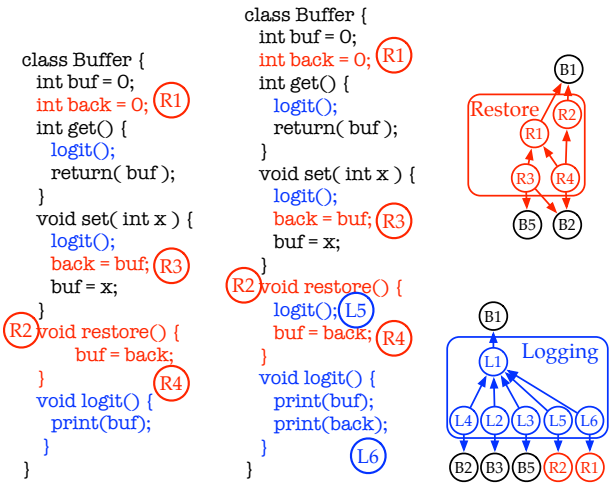

Figure 3: Source code of buffer with logging and restore functionalities (left), corrected version of the buffer (middle), change objects of `Restore` and `Logging` features (right)

3. INTENSIONAL CHANGES

In order to overcome the issue of co-evolution in the context of change-based FOP, we propose to use *intensional changes*. Intensional changes were first introduced in [7]. In this paper, we distinguish between changes (the ordinary ones) and the intensional changes of which we now briefly recall the basics. In mathematics, there are two ways of specifying a set: *extensionally* or *intensionally*. A set can be defined extensionally by explicitly enumerating all elements of the set. For example, we can define the set E of all even numbers extensionally as follows:

$$E = \{0, 2, 4, 6, 8, ...\}$$

The same set can also be specified intensionally by means of a description:

$$E = \{2x \mid x \in N\}$$

The same applies for sets of changes. The `Logging` feature can be specified by an extensional set of changes $F_{Logging}$ which can be applied on any variation of `Buffer` in order to add the logging functionality:

$$F_{Logging} = \{L1, L2, L3, L4, L5, L6\}$$

An intensional description of the same `Logging` feature that adds logging to a class C could be: $F_{Logging} =$

{add *logit* method to C}∪
{add statement print(v) in *logit* for every inst.var. v in C}∪
{add an invocation to *logit* in every method of C}

In order to provide tool support for intensional changes, there is a need for a programming language in which such changes can be specified. Additionally, it must be possible to evaluate these specifications so that the corresponding extension of changes is automatically generated for the desired product variation. After evaluation, the system is specified as a set of change objects, obfuscating the use of intensional changes. A language and evaluator for intensional changes are presented in [7]. We exemplify this language by specifying the `Logging` feature in an intensional way. First, the `logit` method is added to the `Buffer` class. This is done by means of one ordinary change and represented by the tuple:

$$\{(?id, AddMethod, (Buffer, false, logit()), \\ ?timestamp, ?user, \text{“}Logging\text{”})\}$$

Next, this method is provided with an implementation: for every attribute of the `Buffer` class, a print statement has to be added, effectively resulting in the `logit` method outputting the state of the current `Buffer` instance. This is achieved by looking for all *AddAttribute* changes for `Buffer` and inserting an *AddStatement* change that invokes the `logit` method immediately just after:

$$\{c : \{(?id, AddStatement, (Buffer, false, logit, \\ \text{“}print(c.parameterList.attribute)\text{”}), \\ ?timestamp, ?user, \text{“}Logging\text{”}) \\ \textbf{after } c\} \mid \\ \{\forall c : (AddAttribute(c) \wedge c.parameterList.class = Buffer)\}\}$$

This is an intensional change since it quantifies over *all* attributes. Note that the query part is found on the last line. The two parts mentioned so far basically construct the logging infrastructure. What is left to do is to make sure that the `logit` method is actually called from every method in the `Buffer` class. This is achieved by finding all *AddMethod* changes for `Buffer` and once more inserting an *AddStatement* change immediately after:

$$\{c : \{(?id, AddStatement, (Buffer, false, ?m, \text{“}logit()\text{”}), \\ ?timestamp, ?user, \text{“}Logging\text{”}) \\ \textbf{after } c\} \mid \\ \{\forall c : (AddMethod(c) \wedge c.method = ?m \wedge \\ c.parameterList.class = Buffer)\}\}$$

The intensional change construct is similar to the aspect construct from AOP, in which some code (the advice) is described to be woven in at a described set of places (the pointcut). The difference between both is that an AOP aspect is applied to source code, while an intensional change is applied to a transformation (i.e., the changes). This can be illustrated by the following example. An aspect looks like: “add some statement at all places where some method is called”, while an intensional change rather is: “add some statement at all places where some method call is added”.

3.1 Intensional change evaluation

An intensional change can be applied to a software program p in order to apply the changes described by that intensional change to p. This application consists of two phases. First, the intensional change needs to be evaluated with respect to the set of all changes that produce p whenever they are applied. The result of this evaluation is the set of tuples that specify the changes that correspond to the intensional change. This step produces an enumeration of changes which are all applied on p in the second step.

As an example, consider a software program implementing a buffer composed of the features `Buffer`, `Restore` and

Logging. The application of Logging as it is specified by means of intensional changes evaluates to the extension of the change set $\{L1, L2, L3, L4, L5, L6\}$ (bottom right of Figure 3), while the application on the same Logging feature in a composition the features Buffer and Logging evaluates to the extension of the change set $\{L1, L2, L3, L4\}$ (bottom right of Figure 2). The order in which the features are specified in a composition determines the outcome of the evaluation of intensional changes. This is due to the growing change set on which the intensional changes are evaluated.

The resulting enumeration of changes represents the outcome of the first step of the application of intensional changes. The second step consists of applying all the changes of the enumeration. The application of the AddMethod change, for instance, results in a Buffer containing an empty logit method. The application of all changes results in source code that implements the functionality specified by the intensional changes. In the following section, we show how we can use intensional changes in order to avoid co-evolution.

4. FOTEXT

We have implemented both the change cut language and the change cut model in ChEOPS – a proof-of-concept implementation of ChOP [8]. Some experiments were conducted in ChEOPS to validate that intensional changes decrease the need for change-based feature modules to co-evolve. The following section elaborates on those experiments and tries to demonstrate that change-based FOP supports the modularization of crosscutting functionality and that intensional changes make such modules even more robust against co-evolution.

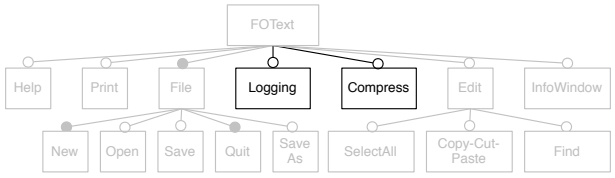

Figure 4: FODA diagram of FOText

Figure 4 presents the FODA diagram of the FOText application. In its original form [14], the FOText application contains all the grey features. Features such as: New, Quit, Open, Save, SaveAs, Print, Help, InfoWindow and all Edit features are self explanatory. File, New and Quit are mandatory features and must consequently be included in every product variation. All other features are optional and – if required – can be safely omitted from a variation.

We implement FOText in a standard object-oriented way by using the VisualWorks for Smalltalk IDE that was instrumented with the ChEOPS tool to capture our development operations as first-class change entities. At the beginning of the development of a new feature, we inform the IDE of its name. By doing that, ChEOPS is capable of *automatically classifying* changes in feature modules.

For the sake of simplicity, we introduce an artificial feature, Base, which is the basic feature that needs to be included in every FOText variation. It consists of the changes that should always be included in a composition: FOText, File, New and Quit. It provides the main functionality: a basic word processor that provides a window to type text and a menu with two choices: new and quit – which are respectively introduced by the New and Quit features.

We present an evolution scenario in which we extend FOText with the Compress and the Logging features. The Logging feature is a feature that adds logging behavior to the text editor. The Compress feature provides the ability to compress text files before they are saved, and decompresses them before they are opened. Both are crosscutting functionalities and involve changes that depend on many FOText features.

We first implement the Logging feature and specify it by means of intensional changes. Its specification consists of three parts, each corresponding to one intensional change. First, a logit method is added to every class of FOText. Afterwards, each of those methods is filled with statements that each print the value of one instance variable. Finally, an invocation of the logit method is added to every method of every class in FOText.

Second, we implement the Compress feature as an enumeration of changes that (1) removes the statements from the saveas method that invoke the functionality for saving a file to disk, (2) adds a dedicated method savenow for saving, (3) adds statements to savenow that compress the text contents, (4) adds statements to savenow that save the compressed text and (5) takes similar actions for decompressing a file when opening it.

Feature	# changes	Feature	# changes
Base	130	SaveAs	88
Save	65	Open	101
Copy_Cut_Paste	72	Find	86
SelectAll	89	Print	182
Help	137	InfoWindow	159
Logging	3	Compress	151

Table 1: FOText changes

Table 1 presents an overview of the change objects that are instantiated for expressing the FOText features. Note that, while Compress is added after Logging, its implementation does not require Logging to be adapted. In order to validate that the compress functionality is also logged, we compose some variations of the text editor.

We now present two compositions in which we include Logging. The first composition consists of a variation of the viewer version of FOText (which is composed of the Base and Open features) and which has logging capabilities. The composition contains 412 changes, of which 182 were changes generated by the application of the 3 intensional changes that specified the Logging feature.

The second composition contains all the features of FOText including Compress and Logging. This composition consists of 1672 change objects of which 541 changes are expressing Logging. The reason for the higher number of logging changes is that it requires more changes to add Logging to a composition that contains more instance variables and meth-

ods as `Logging` is supposed to log the values of *all* instance variables whenever *any* method is invoked. Note that this composition does log the compress behavior without requiring extra effort from the developer.

These two compositions show how features that contain intensional changes evaluate to different change sets depending on the context in which they are applied without having to adapt their implementation. This principle shows how intensional changes can be used to (1) avoid co-evolving feature modules and (2) modularize crosscutting functionality in reusable modules without giving up on flexibility when it comes to software composition.

5. EVALUATION

Intensional changes allow for developers to *describe* sets of changes instead of enumerating them. This is what one wants when implementing a crosscutting functionality. The description of an intensional change is evaluated with respect to a change set in order to produce the corresponding extension of the intensional change. Consequently, an intensional change can be *reused* in different compositions, as it will basically evaluate to the right extension anyway. This makes an intensional change *more flexible* than an ordinary change collection. Consequently, the intensional changes allow our approach to FOP to be more robust against changes in the feature composition

Drawbacks of the intensional changes are fourfold. First they require an additional change cut language and evaluation step, making them somewhat *more complex* than ordinary changes. A second drawback is that an intensional change cannot be obtained by logging a developer or differentiating between source code files, but must *always be specified by a developer*. Thirdly, intensional changes *complicate debugging*, as they evaluate differently in different compositions. Finally, while the *order* of the features within a composition was not important before the intensional changes were included in the model, it now has become important as it influences the way the intensional change is evaluated. In [6], we hint at how the second issue could be overcome.

6. CONCLUSION

We started this paper by explaining that the implementation of crosscutting functionalities is usually scattered over modular software and that this poses problems when the software evolves. When modules are added, modified or removed, the crosscutting functionalities must usually be adapted too. The phenomenon of multiple software modules that need to evolve together is called co-evolution. This phenomenon hinders software maintainability and should be avoided whenever possible.

We present our solution in the context of feature-oriented programming (FOP), in which software building blocks are encapsulated in features which better match the specification of requirements. Change-based FOP is a state-of-the-art approach to FOP which proposes to specify features as sets of first-class change objects which can add, modify or delete building blocks to or from a software system. Advantages of change-based FOP over standard FOP include an increased control of how features are specified and that it makes a bottom-up approach to FOP possible.

We introduce a buffer application that consists of three functionalities: base, restore and logging. We then show that even a feature that is notoriously difficult to modularize, such as logging, can easily be modularized in change-based FOP. Afterwards, we used the same application to demonstrate that in some cases, even a cleanly separated module of crosscutting functionality has to co-evolve when another one is added to the application. As co-evolution hinders maintainability, we present a solution that is based on *intensional* changes: descriptive changes that can evaluate to an extension of changes. We used the application scenario in order to exemplify intensional changes and showed how they assist in avoiding co-evolution.

We evaluate our solution by implementing a text editor in our proof-of-concept implementation of intensional changes. It consists of 14 features and is extended with two extra features. We show how implementing a feature by means of intensional changes avoids that feature to co-evolve when another feature is added. We present two variations of the text editor and explain that the those variations provide the desired functionality without requiring extra effort from the developer.

7. REFERENCES

[1] B. Adams. *Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems*. PhD thesis, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent (Belgium), May 2008. ISBN 978-90-8578-203-2.

[2] 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.

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

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

[5] T. D'Hondt, K. D. Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *SACT Symposium Proceedings*. Kluwer Academic Publishers, 2000.

[6] P. Ebraert. *A bottom-up approach to program variation*. PhD thesis, Vrije Universiteit Brussel, 2009.

[7] P. Ebraert, T. D'Hondt, T. Molderez, and D. Janssens. Intensional changes: Modularizing crosscutting features. In ACM, editor, *Proceedings of the 25th Annual ACM Symposium on Applied Computing*, 2010.

[8] 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.

[9] P. Ebraert, J. Vallejos, Y. Vandewoude, Y. Berbers, and T. D'Hondt. Flexible features: Making feature modules more reusable. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing*, 2009.

[10] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and

H. Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001.

[11] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectJ. In *Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science,*, volume 2072, pages 327 – 353. Springer Verlag, 2001. http://aspectj.org.

[13] 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.

[14] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented software designs. In S. Reiff-Marganiec and M. Ryan, editors, *FIW*, pages 178–197. IOS Press, 2005.

[15] T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.

[16] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241:419–434, 1997.

[17] Y. Smaragdakis and D. Batory. Implementing reusable object-oriented components. In *In the 5th Int. Conf. on Software Reuse (ICSR 98*, pages 36–45. Society Press, 1998.

[18] 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.