Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2008

SOFTWARE VARIATION BY MEANS OF A CHANGE-BASED MODEL FOR FEATURE-ORIENTED PROGRAMMING

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Leonel Merino

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Advisors: Peter Ebraert (Vrije Universiteit Brussel)
Jorge Vallejos (Vrije Universiteit Brussel)

Abstract

Languages that implements *Object-Oriented Programming* (OOP) do not provide enough means to cope with software variations. Adding a functionality to a family of software variations by means of an OOP approach can be accomplished by implementing the solution directly into the code of each variation. That solution suffers from a combinatorial explosion and lack reusability [29].

A better approach is to use *Feature-Oriented Programming* (FOP). It allows to produce software variations by composing features. In FOP a feature is a modular building block that adds a functionality to a system. The state-of-the-art approaches to FOP lack expressiveness. All approaches specify features only by the *addition* or *modification* of software building blocks. Moreover, the granularity they provide rarely reaches the statement level. Finally, most of them do not provide means to manage features that implement crosscutting functionalities. In these approaches, a crosscutting feature has to be implemented in a different way depending on the software variation hindering the reusability of features. We believe that selecting appropriates building blocks to specify features would allow us to overcome these issues of expressiveness.

Several approaches successfully used first-class change objects in the research domain of software evolution. We propose a change-based approach to FOP based on the *Change-Oriented Software Engineering* model [14]. In our model, features are specified by a set of *changes* that have to be applied in order to implement the functionality that feature offers. *Changes* model all the operations (*addition*, *modification* and *deletion*) of software entities that can be applied to software building blocks. *Changes* are instrumented with explicit dependencies which provide information about the validity of feature compositions.

We introduce *flexible* features as an appropriate concept for modeling crosscutting features. A *flexible* feature is specified by a set of changes that does not have to be applied as a whole in order to add the feature to a composition. We present a composition verification algorithm that is capable of automatically customizing *flexible* features in such a way that they never make a composition invalid. This improves the reusability of features as they can be added to any composition without having to adapt them.

We provide an implementation based on ChEOPS [14], that captures the changes as

first-class entities and that allows the programmer to compose features. We also provide a graphical tool based on Mondrian [28] that advises the programmer when a composition is invalid, providing means to debug the composition by inspecting the changes that yield that situation.

We evaluate our approach by implementing FOText a simple word processor. We modularize FOText in 11 feature modules such as `Base`, `Open`, `SaveAs`, `Print`, `Compress`, `Status_Title` etc. We capture the changes that we produced when implementing such features and specify that `Compress` and `Status_Title` are *flexible* features. We produce several compositions and evaluate their validity, showing how our tools assist the developer when a composition is found to be invalid. Finally, we show how *flexible* features increase feature reusability by composing the `Compress` feature with different variations.

This evaluation shows that the usability of our graphical tools decreases when the number of changes grows. This insinuates that our model does not scale up. In order to tackle this issue, one of the tracks of feature work we propose, is to introduce filters that provide a customized view on the change objects, hiding away unwanted level of detail.

Acknowledgements

I would like to thank to Professor Theo D'Hondt for promoting this thesis and also for giving me the opportunity to work in the PROG laboratory.

To Eric Tanter who involved me in the EMOOSE program and made me believe that it was possible. It was!. Thank you!.

To Peter Ebraert who allowed me to participate in his research, who helped me to find a motivating thesis topic, who correct my English many times and who advised me when conducting this thesis. Also, for motivating me to present this work in the Software Variability Symposium 2008, which was a great experience. Thank you!.

To Jorge Vallejos for having a critic point of view that helped me to improve my work. Moreover, for helping me to produce the SVPP presentation. I had a great time working with you. Thank you!.

To my classmates: Alexis, Jerome, Leonardo, Mayleen, Roberto, and Walther that made of this year a great experience. Thank you!.

A mis padres Leonel y Bernardita, por darme la vida y motivarme a ser un hombre de bien. Gracias!.

To my wife Maria Antonieta for being my constant source of inspiration. Together we have no limits! Thank you for exist!.

Thanks to all.

Contents

Figures

Listings

Tables

Chapter 1

Introduction

1.1 Context

In the evolution of the automobile industry, the introduction of mass production provided a reliable and cost-efficient way to produce cars. Mass production itself, however, did not provide a solution for customers that required variation in the products. The automobile industry addressed this by splitting up their product in different modules: a standard core of a car and a set of optional parts, where the parts add different functionalities to the core of the car. By composing a core with some parts, a new variation of a car could be produced. This process is known as *car product-lining* and provided a solution for the variation problem in a cost-efficient way. The different kinds of cars produced by a car product-line is called a *car product-family*.

Just like cars in the automobile industry, software products can be seen as the composition of a core element which provides the base functionality and a set of elements which incrementally add functionalities to that base. This kind of software development is called *Software Product-Lining* (SPL). It addresses software variation by keeping common assets inside a core element and combining it with a set of extra functionalities that might be selected to provide variation. A selection is obtained by keywords, attributes, characteristics, behaviors or any other criterium which allows to describe the functionality. Product-line instances are the result of combining the core element and a set of functionalities. Not every combination, however, is possible. Interactions between the elements that are being composed can lead to an invalid composition. Reasoning about the validity of product-line instances still remains a challenge [4].

Developing variations in a product-family can be addressed in an ad-hoc way using *Object-Oriented Programming* (OOP) implementing functionalities directly into the code of the base program. The resulting code, however, contains an *IF-THEN* control statement at every place where the program chooses which variant to produce. This kind of implementation lacks *modularity* and *reusability* [30]. Another solution to implement the

variation is to use *polymorphism*. Then, *IF-THEN* control statements are replaced by instantiating different subclasses of a class which are modeling the specification of each variation that the *IF-THEN* wanted to represent. This approach requires a significant amount of manual labor [15]. Moreover, these approaches are not able to describe every kind of functionality, since they are restricted to the expressiveness provided by the OOP language.

A better alternative is to modularize the software based on the functionalities it provides. Modules which add a functionality to the system are known as *features*. *Feature-Oriented Programming* (FOP) is the discipline that addresses software development by using features as building blocks. The very basic idea of FOP is to produce software variations by composing features. The feature composition requires to manage feature interactions to ensure the validity of a composition. The interactions between features requires to understand the building blocks features consists of.

1.2 Motivation

The level of granularity of a FOP approach defines the level of detail of the feature building blocks. A fine-grained approach allows to specify features with very much detail but makes the specification more complex. A coarse-grained approach has simpler feature specifications that contain less specific contents. In a coarse-grained FOP approach there are some software constructs that cannot be expressed, since they are beyond the granularity provided. Consequently, we aim for an approach that allows a fine-grained of granularity.

In order to implement a functionality program building blocks (such as classes, methods, statements, etc.) needed to be added, modified or removed. *Anti-features* [16, 17] are features which remove a functionality from a program. There are several known cases such as DRM feature to protect music, region-coded protected DVDs and the RAW format in Canon cameras. In these examples, customers buy features that remove the restrictions in programs. These restrictions are not always encapsulated in a feature but they can be scattered over the code. Moreover, adding a functionality to a legacy software might require to remove some entities. Hence, we believe that an approach of FOP should allow the *addition*, *modification* and *deletion* of entities.

The building blocks of a feature might have dependencies on the building blocks of other features. This yields dependencies between features. When composing features, unsatisfied dependencies yield invalid software composition which do not compile. A FOP approach should provide an error-free methodology to produce valid by composing features. We believe that, in order to do so, the information about the dependencies should be provided in an *explicit* way.

A feature that implements a crosscutting concern always contains building blocks that

depend on building blocks of many features. Consider *logging* or *security*. In order to implement one of these functionalities, the produced code has to be inserted scattered along the program. It cannot be encapsulated into an uncoupled software entity, like a class or method. Such feature can only be included in a composition if all features on which it depends reside in that same the composition. In case one of the depending features is not included in the composition, it would be impossible to include the crosscutting feature without tweaking it in order to remove the unsatisfied dependencies. One way to do that is by creating a specific version of the crosscutting feature for each combination of features. This would lead to an exponential growth of the number of versions of the same feature, making it much harder to maintain and hindering its reuse. Consequently, we believe that a FOP approach should provide means to write a feature once and deploy it in a customized way depending on the features that are participating in the composition. This would increase the reusability and maintainability of features.

1.3 Goals

In this research we aim for a Feature-oriented model on which the feature building blocks have a fine-grained *granularity*, a model which provides an expressive set of *allowed operations* (such as addition, modification and deletion), which provides a *customized feature deployment* mechanism and *tools to support* development.

The kinds of entities on which an operation is applied are established by the *granularity* building blocks. We aim a fine-grained granularity level to provide a proper expressivity to specify features.

We aim to enhance feature composition by allowing the description of features not only by additions and modifications but also by deletions. This would allow a programmer to describe features with the same set of *allowed operations* by the language environment.

We pursue a model that provides a *customized feature deployment* to manage features which implement crosscutting functionalities. Allowing to create a feature once and composing it with several combinations of other features without having to rewrite or to adapt the features. We believe that features contain enough information to provide this kind of behavior.

We want to provide *tool support* for FOP which supports the production of software instances by composing features, that advises about invalid compositions and provides information to make valid composition invalid.

1.4 Thesis

> *We believe that a change-based model for Feature-Oriented Programming that adheres to the above established criteria is appropriate to increase the reusability of features and to validate feature compositions.*

In a change-based model features are specified as set of changes which are instrumented with explicit dependencies. This allows us to identify the specific changes that make a composition invalid. Exploiting this property our model can advise the programmer about the changes or even features that once added to the composition produce a valid one. It also is allowed to customize the deployment of features allowing to reuse them.

1.5 Overview

In chapter 2 we explore the concepts of *Feature-oriented programming*, we present five criteria and analyze the related work with respect to those criteria. In chapter 3 are presented *Changes as first-class entities*. We propose changes as a proper way to describe features. We analyze related work that successfully applied first-class changes in other research domains and choose an appropriate model for first-class changes. Chapter 4 presents our model of *Software composition*, the conceptual model of FOP and the tool support that we provide. In chapter 5 we show an *evaluation* where we tested our model by implementing a word processor `FOText`. We show different compositions and show how our tool advises when a composition is invalid. Finally, chapter 6 presents the *conclusions*, our contributions at conceptual and technical level, the limitations and future work.

Chapter 2

Feature-Oriented Programming

Feature-Oriented programming (FOP) is a discipline that proposes to produce software instances by composing features. *Features* are well-delimited building blocks that encapsulate the code needed to add a functionality to a base program. Since a software instance provides functionalities to its users, it can be specified by the list of functionalities that it provides. The required software instance can then be constructed by composing the features that map to the specified functionalities.

FOP takes modularization as a central concern, hence several software qualities are improved. FOP increases software *maintainability* [1] by modularizing software. *Reusability* is defined as the ease with which software can be reused in developing other software, which in fact is the way how FOP proposes to develop software[11]. FOP proposes to create software variants by adding features to a composition and *reusing* the features that already exist. However, composing features is not absent of issues, since there are interactions between features. These interactions make a feature to depend on other feature. These dependencies must be satisfied otherwise the composition turns invalid.

2.1 Concepts

Feature-Oriented Domain Analysis [22] (FODA) provides three mechanics to describe applications in a Feature-oriented way. It introduces *feature diagrams* to describe all possible combinations of features; *composition rules* to express constraints that can occur among features; and *rationale* to provide annotations for reasoning on the convenience of selecting a feature. We illustrate FODA with an example of a *Buffer* application as shown in Figure 2.1. It shows that a `Buffer` *must* have a `Base` functionality which *optionally* can be enhanced with `Logging` capabilities. The `Buffer` can *optionally* include a `Restore` functionality which can be a `Basic Restore` or `Multiple Restore`. The description is completed adding a *rationale* denoting that `Logging` is useful for debugging purposes, and a *composition rule* which states that `Logging` requires `Restore` and `Base`.

The composition rules of FODA allows to establish the interaction that can happen be-

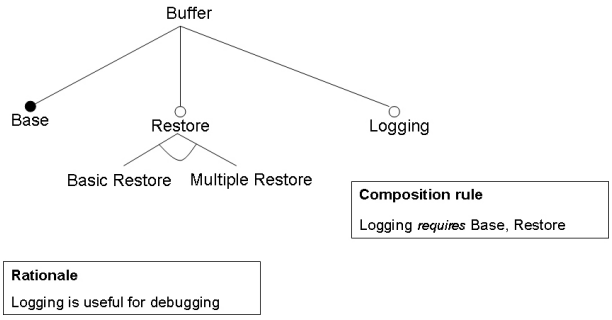


Figure 2.1: FODA of Buffer

tween features. Figure 2.1 shows that when producing a software by composing the feature `Logging` are required the features `Base` and `Restore`. This means that feature `Logging` depends on feature `Base` and `Restore`.

Although the number of combinations from a FODA diagram can lead to a combinatorial explosion, the number of the instances that, in fact, can be instantiated are a minor proportion of them as shown in [13] by Van Deursen *et al.*. In a FODA diagram there are declared the constraints that occurs between features. These constraints restrict some combinations.

2.2 Related Work

This section presents some state-of-the-art approaches related to FOP. While of them adhere to FOP explicitly, others provide helpful means that can lead to FOP. We first present a set of criteria to analyze these approaches. It would provide information about the relation between our work and the state-of-the-art approaches. These criteria are the qualitative properties that we aim to find in a FOP approach. They are motivated by the goals of our work (chapter 1). We use them to evaluate the state-of-the-art approaches.

1. *Granularity*: The granularity provided to express features establishes the smallest/biggest kind of information that compose a feature. Some approaches use native OOP entities such as: classes, instance variables, and method, as the building blocks of a feature. In that case, many of them establish the granularity at the level of methods. This implies that a feature would be not enabled to introduce a modification *within* a method, but it would be able to replace the entire method. The granularity establishes the kind of the specifications that a feature can declare. For instance, an approach that sets its granularity at the level of method is not able to declare a modification to the body of the method, but it has to declare the entire body again.

2. *Allowed operations*: The purpose of this criterium is to evaluate if the approach allows to express features by additions, modifications and deletions. Usually, the building

blocks that compose a feature are described in term of additions. We realized that deletions is a proper way for describing certain features such as anti-features [16, 17]. Anti-features are features that remove a functionality of the system. Most of the time, this functionality is scattered among several features, this makes that to remove a feature from the composition is not always an option.

3. *Dependency Management*: Approaches decide whether or not to provide a dependency management depending on the purpose and the strategy that they use to address FOP. However, when features are composed the model could manage feature dependencies and advise the developer how to debug and resolve invalid compositions.

4. *Customized feature deployment*: In a composition sometimes a feature cannot be deployed since some of its own building blocks have at least one dependency not satisfied. Approaches can provide means to tackle this situation. Doing so, they can allow compositions that otherwise would be invalid.

5. *Feature-specific language support*: This criterium checks whether the approach adds new constructs to a programming language which might be pre-compiled or whether it uses the capabilities of a main-stream programming language. This allows to analyze if the approach can be implemented using any programming language or requires a specific one.

2.2.1 AHEAD

Feature interactions are a key issue in Feature-oriented designs. A feature interaction occurs when one or more features modify or influence other features. Liu, Batory and Nedunuri [25] presented a model which improves FOP by proposing an algebraic theory of structural feature interactions that models feature interactions as derivatives.

Algebraic Hierarchical Equations for Application Design (AHEAD) is a unified formulation for FOP that integrates step-wise development, generative programming, and algebras [6, 26, 36]. This is based on step-wise development which proposes that a complex program can be built from a simple program (called a base program) by incrementally adding features. AHEAD models product-lines with a simple algebraic structure. A *base module* contains the definition of classes, members, and methods while a *derivative module* extends programs adding fragments of methods, classes, and packages. Thus, a feature is modeled by a composition of one *base module* and a set of *derivative modules*. Figure 2.2 shows two features `B` and `H`. `B` consist of a *base module* `b` and feature `H` consist of a *base module* `h` and a *derivative module* $\partial b/\partial H$ which extends the *base module* `b`. `GenVoca` is a methodology for creating application families and architecturally extensible software by expressing programs as sets of equations. It provides the same information that can be described by a *feature diagram* but adding *cross-tree relations* for denoting relations between features that are not expressed by the diagram. Since it provides a grammar, it can be used by

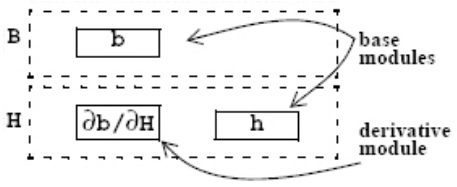


Figure 2.2: Application described by features B and H

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

Listing 2.1: AHEAD Base feature

applications in an automatic way.

The function • *weaves* a derivative on a base module. An *introduction sum* + is a binary operation that aggregates base modules by a disjoint set union. Thus, an application composed by features B and H is expressed by:

$$[H(B)] = h + \partial b/\partial H \bullet b \tag{2.1}$$

A derivative module which *refines* software artifacts—classes, members, and methods—is able to change the behavior of a method defining a new body for the method. This new body can call the execution of the original method at any point of the new method execution. A derivative modules which extends fragments introduced by a derivative module—affecting many features—is called a *second derivative module*. If a feature J has a second derivative module which extends the module $\partial b/\partial H$ from H, it is denoted by $\partial^2 b/\partial J \partial H$. Listings 2.1, 2.2 and 2.3 show a product-line example with three features: `Base`, `Restore`, and `Logging`. `Base` implements the base functionalities that include a the definition of a `Buffer` class with a `buf` instance variable, a `get` and `set` method for getting and setting the value of `buf`; `Restore` adds an instance variable `back` to store the old value of `buf` anytime it is set and adds a method to do the restoration. `Logging` adds a method which prints the value stored in `buf` and `back` and adds invocations in the methods `get` and `set`. The expression $[L(R(B))] = \mathit{Logging} \bullet \mathit{Restore} \bullet \mathit{Base}$ represents an application with the three functionalities. It could be desirable to compose an application with `Base` and `Logging` features. However, this not possible since feature `Logging` would be making changes to the method `restore()` which would not exist since feature `Restore` would be

```
refines class Buffer{
    int back = 0;
    void set(int x){
        back = buf;
        buf = x;
    }
    void restore(){
        buf = back;
    }
}
```

Listing 2.2: AHEAD Restore feature

```
refines class Buffer{
    void logit(){
        System.out.println(buf);
        System.out.println(back);
    }
    int set(int x){
        logit();
        super.set(x);
    }
    void get(){
        logit();
        super.get();
    }
    void restore(){
        logit();
        super.restore();
    }
}
```

Listing 2.3: AHEAD Logging feature

not in the composition. This issue is called the `Feature Optionality Problem` [25]. It can be addressed by restructuring the features in such a way that the feature involved in the interaction is put in a separate feature. This solution allows to produce an application with *Base* and `Logging` functionalities, but has some limitations: Firstly decoupling of interactions requires a deep understanding of the feature implementation which in realistic cases could be difficult to address, secondly a feature can have interactions with many features at a time.

1. *Granularity*: Features can modify previous software entities by *refinements*. The smallest modification that they can introduce is statement level. However, is not possible to modify the execution of a method by introducing a statement in the middle. The unique way available is creating a new version of the method and invoking the old method at some point of the new one.

2. *Allowed operations*: Features can be described by *base modules* and *derivative modules*. The former allows to introduce new software entities, the latter allows to modify a previous entity.

3. *Dependency Management*: A feature consists of a base module and a set of derivative modules. The latter can be empty in case that the feature only introduces new entities and does not modify any previous entity. Dependencies are made explicit by derivative modules. A derivative module knows which other module it is modifying and by that introduces a dependency.

4. *Customized feature deployment*: Although AHEAD can denote the part of a feature which makes a composition invalid and by that restructuring the feature and encapsulating that part into a new one the adaptation must be addressed with human intervention. This makes it less useful for using in software product-lines.

5. *Feature specific language support*: AHEAD is a tool for implementing features which extends Java with some extra language constructs (e.g. refinements).

2.2.2 Generic Feature-Based Composition

A feature diagram [22] is a specification by comprehension of a software product-line. It allows the description of the relations between features and by that it can be used to derive product instances. However, dependencies between the artifacts that compose each feature are not covered by the feature diagram.

Artifact dependency graph are directed acyclic graphs, where nodes represents software artifacts while edges represent dependencies between them. Dependencies can be derived by the abstract syntax tree of the language or introduced explicitly.

In a Feature-oriented way, the *problem space* of a software product-line is represented by a feature diagram which maintains all possible combinations that can be produced by a

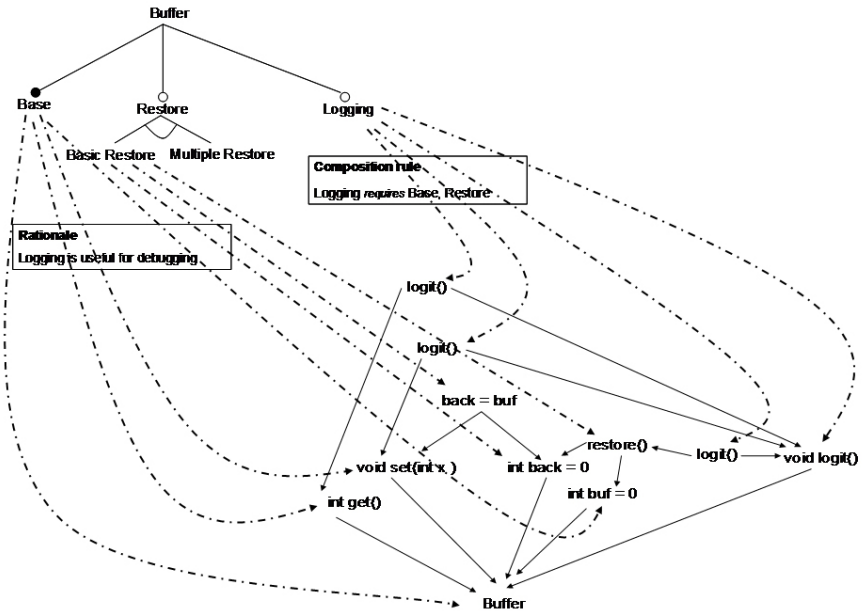


Figure 2.3: Mapping between Feature Diagram and Dependency Graph

set of features, and the *solution space* is described by the dependency graph of the software artifacts that compose each feature.

Van Der Storm [37] proposes a model where product instances are derived from a set of features within a software product-line. A feature captures elements of the problem domain while a product configuration consists in choosing features for instantiating a product. A product configuration maps features from the problem domain to the software artifacts that compose the features in the solution domain. The model proposed is aware that a feature composition can be invalid, if it includes a feature incompatible with another feature that is in the same composition. The proposed model is independent of the programming language used, as well as the software development methodology and the architecture. Filling the gap between problem and solution space model can be achieved by mapping the *problem space* modeled by feature diagrams and the *solution space* modeled by dependency graphs. The composition of those two representations is proposed as a solution for software variability. By mapping every node of the feature diagram with one or several nodes of the dependency graph a general description of all possible variations is obtained. That mapping can introduce ambiguities since the dependency graph lacks expressivity—artifacts, for instance, can be abstract classes and the mapping cannot assure which implementation must be applied. Figure 2.3 depicts a mapping between the features in a feature diagram—at the top of the picture—and the artifacts that compose features in the dependency graph—at the bottom of the picture.

1. *Granularity*: This is a general purpose model which addresses the gap between the problem space and solution space. Specifying the artifacts that describe a feature is

not a concern to this model. Thus, this criterium is not applicable.

2. *Allowed operations*: This model describes features by the addition of software artifacts. This addition is addressed by mapping a set of software artifacts to the feature.

3. *Dependency Management*: This model advise for invalid composition. Dependencies between software artifacts are described by the dependency graph. Linking those artifacts with features the dependency between features is provided.

4. *Customized feature deployment*: In this model features cannot be customized to deploy in a different way depending on the context.

5. *Feature specific language support*: This model does not require any specific language support. It uses feature diagrams and dependency graphs.

2.2.3 Lifting functions

In [31], Prehofer introduces a model for flexible object composition from a set of features. He proposes a modular architecture for composing features with the required interaction handling, yielding a full object. Inspired on inheritance, it resolves feature interactions by *lifting functions* of one feature to the context of the other using *method overriding*. Features are described as a language extension of Java, maintaining the same expressivity as classes. Feature composition is achieved by composing two features at a time, using its previously defined lifter which knows how each feature must be adapted for composing with another one. The process can be executed many times for composing several features. This model is presented as an extension to Java and gives two translations to Java, one via inheritance and other via aggregation.
Figure 2.4 shows fourth features `F_1`, `F_2`, `F_3` and `F_4`; three composition of these features: (`F_1 - F_3`), (`F_1 - F_2 - F_3`) and (`F_1 - F_3 - F_4`). To accomplish these compositions are provided five lifting functions that handle the interactions between two features. In this example, they are: (`F_1,F_2`), (`F_1,F_3`), (`F_2,F_3`), (`F_2,F_4`) and (`F_1,F_4`).

This approach requires intensive human intervention since lifters need to be written for every pair of features which might be composed. Although, it increases the accuracy of the resulting composition it would introduce an extra work since it require the definition of lifter functions to address feature interaction.

1. *Granularity*: In this model the smallest entity that a feature can express is a statement. However, the ways in that a feature can add a statement to a method is very restricted. It is addressed by overriding the method with a new version and calling the old version of the method at a certain point. In general this model modifies entities by adding classes, methods and instance variables.

2. *Allowed operations*: This model expresses features using the same building blocks as the OOP languages. Moreover, by overriding methods, it allows the introduction of

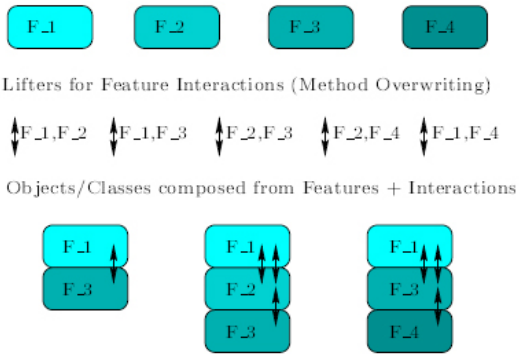


Figure 2.4: Composing Objects by means of lifting functions

new behavior. It does not allow modification of a specific statement or deletion of any entity.

3. *Dependency Management*: Lifters are defined for all pairs of features in a composition. In those lifters, dependencies are covered explicitly since they know which features are adapting the context of the others.

4. *Customized feature deployment*: This approach does not allow to customize the deployment of features. However, it offers to adapt the deployment of a feature manually by means of lifters.

5. *Feature specific language support*: This model extends Java by adding the notion of a feature. A feature introduces a lifting function which addresses the interaction of features.

2.2.4 Extra-functional features

The number of software variations in Software Product-Lines grows exponentially with the number of different features. Benavides *et. al* [8] propose a model for SPL using constraint programming that tackles that issue.

The nature of a SPL is to create instance products by reusing features. Feature models are used to model SPL in terms of features and relations amongst them. The graphical description of a feature model is a feature diagram. A SPL with a large number of features leads to a large number of potential products. Most models only address functional features and lack modeling artifacts that deal with extra-functional features. The model of Benavides *et al.* addresses this issue by providing an extra-functional feature management.

A feature attribute is any characteristic of a feature that can be measured. An extra-functional feature is a relation between one or more feature attributes. This model proposes

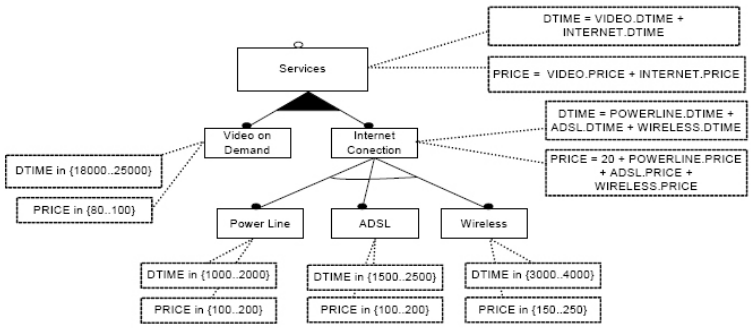


Figure 2.5: Extended feature diagram including extra-functional features

an extension for feature models allowing that each feature can have attached several extra-functional features. We illustrate the extra-functional features with the example presented in the Figure 2.5. It shows a feature diagram that establishes that `Services` can be a composed by `Video on Demand` *or* an `Internet Connection`, in the case of the latter, it can be a `Power Line` *xor* `ADSL` *xor* `Wireless`. Although the *or* operand is true whether at least one of the operands is true, a *xor* operator is true whether at most one of the operands is true.

Resolving a *Constraint Satisfaction Problem* (CSP)is about finding the correct value for a set of variables which satisfies a set of constraints. This approach proposes an algorithm to provide this set of values which resolves the CSP. Thus, the extended feature model is transformed into a CSP. Using constraining programming to reason on extended feature models this model provides information about: Firstly the number of potential products by interpreting the extended feature model, secondly the products produced by a specific set of characteristics, thirdly the production of product instances from the extended feature model, fourthly information about whether the extended feature model can produce at least one product—which means that is a valid model and finally the selection of a product based on a criteria for producing *optimum products*.

1. *Granularity*: This is a general purpose model. It does not address the granularity of feature since it is not its main concern.

2. *Allowed operations*: This model enhances the expressivity of features by adding the notion of *extra-functional features* as the interaction between the attributes that compose a feature. It does not address the building blocks that compose a feature, but enhances the graphical description that a feature diagram provides.

3. *Dependency Management*: This model introduces a notion of dependency between features and their attributes. It provides *extra-functional features* as the relation between attributes. However, feature dependency between features themselves is not addressed.

4. *Customized feature deployment*: It does not allow to customize the deployment of features. Features are fixed and can only be applied or omitted as a whole.

5. *Feature specific language support*: It extends feature diagrams by introducing the concept of *extra-functional features*.

2.2.5 Mixin-layers

The approach proposed by Smaragdakis *et. al* [35] uses a collaboration-based technique for examining large-scale refinements. A *refinement* adds units of functionality to a software system. It can affect many implementation entities such as classes, functions, etc.

Software components are described by *fragments* of multiple classes which encapsulate *fragments* of multiple functions. It pursues reusability by using these kinds of components as building blocks of large-scale applications. Large-scale refinements exhibit a *crosscutting* behavior, since they normally impact on many classes at the same time.

A *collaboration* is a set of objects and protocol—which specifies allowed behavior—that defines how objects must interact. The object's *role* in a collaboration is the part of it enforcing the protocol. Usually, objects of an application can participate in many collaborations. A role can be seen as the part of an object that takes place within a collaboration. Thus, collaboration-based design describes applications by composing collaborations.

Furthermore, a refinement of an Object-oriented class is encapsulated by a subclass, hence it can add new methods and attributes, as well as override methods of the superclass. Class inheritance is not enough, however, to cope with large-scale refinements of a collaboration-based design. A solution is proposed by presenting *mixins*. A mixin is an Object-oriented construction, similar to a class but with the ability of being defined without specifying a superclass. Thus, it is flexible since it is able to refine a class by instantiating different superclasses dynamically. Because a mixin only copes with one class while a collaboration contains many classes at a time, *mixin-layers* were introduced. These scale-up mixins so they can handle multiple smaller mixins that are collaborating.

Mixin-layers is a way for expressing large-scale refinements in a collaboration-based design. Since mixin-layers like collaborations encapsulate refinements which applied in certain way produce a software, it is a fertile field for software product-lines. Applying a Mixing to different layers assist in producing all instances of a software product-line. Figure 2.6 shows an example taken from [38]. It presents a collaboration diagram which models the graph traversal application that defines three different algorithms on an undirected graph: `Vertex Numbering`, `Cycle Checking` and `Connected Regions`. The application has three classes: `Graph`, `Vertex` and `Workspace`. The application is decomposed in five collaborations.

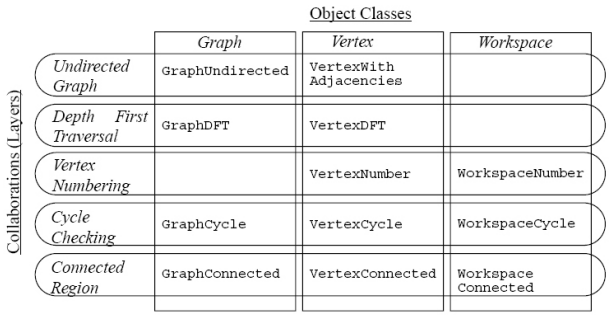


Figure 2.6: Collaboration diagram of the graph traversal application

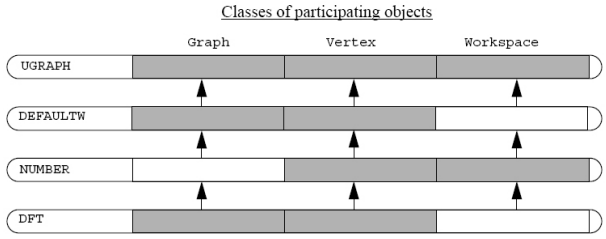


Figure 2.7: Composition by Mixin-layers

Thus, Figure 2.7 shows a composition which expresses the development of a vertex numbering application as a series of refinements. Mixin-layers are represented as ovals, they contain several mixins refines classes.

1. *Granularity*: A Mixin-layer is able to introduce modifications until the level of statements. However, is not possible to introduce a statement in the middle of the body of a method. This model addresses method modification by overriding the method with a new definition. Even though this definition is able to invoke the old one (allowing to maintain the old behavior with a encapsulated change), it does not cover all the possible statement insertions that can be desired to do to a method.

2. *Allowed operations*: This model provides *Mixin-layers* as building blocks for building a software. Since *Mixin-layers* are based on refinements, they only allow the addition and a specific kind of modification—for instance, to introduce a statement in the middle of the body of a method. It does not allow deletion.

3. *Dependency Management*: Dependencies between Mixin-layers are addressed explicitly.

4. *Customized feature deployment*: This model does not allow to customize the deployment of features.

5. *Feature specific language support*: They introduce *Mixin-layers* as a way to encapsulate features. Mixin-layers extend an Object-oriented language with a construct that models *Mixins* and *Mixin-layers*.

2.2.6 FeatureC++

Feature C++ is a language extension proposed by Apel *et. al*[2] to support *Feature-Oriented Programming* (FOP) and *Aspect-Oriented Programming* (AOP). FeatureC++ implements features by means of *Mixin-layers*. A Mixin-layer is a set of Mixins that implement a crosscutting concern. Mixins consist of *constants* or *refinements*. Constants are new software entities and refinements are increments on existing ones. In FeatureC++, Mixin-layers are represented as file system directories. Mixins found inside a directory collaborate in the Mixin-layer. FeatureC++ improves FOP in two ways: Firstly it enhances refinements with *multiple inheritance* to introduce new behavior to a class by making the class inherit from multiple classes; Secondly it improves refinements to produce generic transformations into classes. Using class and method templates, refinements can be parameterized improving variability. FeatureC++ addresses FOP issues related with crosscutting modularity by means of AOP. The following three concepts were introduced to that extend: *Multi Mixins*, *Aspect Mixin Layers* and *Aspectual Mixins*.

Multi Mixins: Traditionally, a mixin is able to modify only one mixin. This latter mixin is called the *parent* mixin. A Multi Mixin is a mixin able to refine a whole set of parent mixins. This is accomplished by introducing a wildcard % in the class or method name whether the mixin is specified.

Aspect Mixin-layers. The basic idea is boil down to embed aspects into a Mixin-layer. A Mixin-layer contains a set of mixins and a set of aspects, allowing mixins to implement static/dynamic, homogeneus/heterogeneus and hierarchy/non-hierarchy-conform crosscutting behavior.

Aspectual Mixins provide Mixin with the power of AOP. Mixins can contain pointcuts and advices, as well as common refinements and constants. The analysis of FeatureC++ with our criteria has produced the following results:

1. *Granularity*: The level of granularity provided by Mixin-layers allows manipulations at the level of statement. Although, a single statement cannot be modified, statements can be modified by refining a whole method. It means to add the new statement and invoke the old method after that.

2. *Allowed operations*: Features are represented as Mixin-layers in the form of Multi Mixins, Aspect Mixin Layers or Aspectual Mixins. Thus, a feature is able to introduce new attributes and methods and to modify methods in a limited way. This approach is not able to describe features by deletions because it uses aspects to implement features.

3. *Dependency Management*: Features dependency is addressed by the interaction between Mixin-layers which is explicit. However, dependency between the building blocks—Mixins—that compose a feature—Mixin-layer—is not addressed by the model.

4. *Customized feature deployment*: Aspectual Mixin-layers and Aspectual Mixins are variants of Mixin-layers which introduce AOP. How a feature is deployed, can be addressed by AOP. It allows to deploy a feature in a different manner depending on

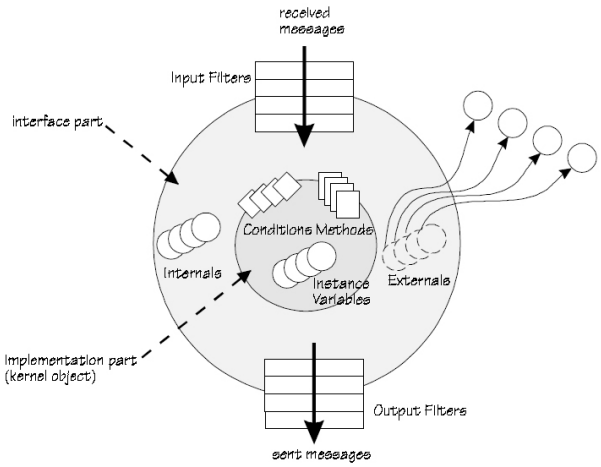


Figure 2.8: The components of the composition-filters model taken from [9]

the context. For instance, a Mixin-layer that adds an invocation to all classes where its name starts with `set` would add a different number of invocations depending on the features that are being composed.

5. *Feature specific language support*: This model introduces the concept of a feature as a language extension for C++.

2.2.7 Composition-Filters

The composition-filters model—proposed by Bergmans [9]—aims at providing techniques to support large-scale software. It is a modular extension to the conventional object model. Figure 2.8 depicts the model which consists of a *kernel* and an *interface*. The kernel is no more than an regular object which has instance variables, methods and conditions. A condition is a boolean expression which provides information about the current state of the object. It is implemented as a method that takes no parameters and returns a boolean object as a result. The interface is an addition to the object model which consists of input filters that are able to determine if an incoming message should be redirected to a method or rejected—this is done using the conditions and their properties—and a set of output filters which deal with messages that are sent by the object. Instance variables in the kernel are isolated, only methods and conditions can be accessed directly. Methods are accessed by messages—whether input filters allow them—and conditions by input filters. A special construct in the kernel is the *initial method* which encapsulates instance variable initialization, but can also initiate other activities in the system.

Since instance variables cannot be accessed directly, all activities in an Object-oriented computation model are performed through message invocations which makes `filters` a central artifact in this model. The mechanism used for `input filters` is the same that used for `output filters`, the only difference is that `input filters` process received mes-

sages and `output filters` do the same with sent messages. A filter is composed by a `filter pattern` which addresses the recognition of messages and a filter type which indicates the action that will be taken when a message is accepted by the filter. A filter is able to modify the content of the message and redirect it to a method, to another filter or discard it.

Although composition-filters is a general purpose model to separate crosscutting concerns, we think it is appropriate for Feature-oriented programming. It improves the expressivity of Object-oriented languages by introducing filters. Filters can be composed in many orders producing different behavior, and also reused in many composition-filters amongst an application.

1. *Granularity*: Composition-filters allow the modification of the entire behavior of an object. The granularity that this model proposes is set at the level of objects. It is accomplished by using the set of filters to process the incoming and outgoing messages of an object.

2. *Allowed operations*: A feature can be described by one or many composition-filters. They can add new functionality to a program and modify the behavior of previous software entities by using filters. Composition-filters do not allow the deletion of software entities.

3. *Dependency Management*: Composition-filters are able to represent features by manipulating the behavior of objects through the set of filters. Thus, dependency does not need to be tackled by the model, but is tackled by the compiler of the language where the composition-filters is implemented. Consequently, composition-filters do not provide feature dependency management.

4. *Customized feature deployment*: This model does not allow to customize the deployment of features.

5. *Feature specific language support*: Composition-filters is an enhancement for the object model.

2.2.8 AOP

In the construction of a program there are crosscutting concerns which implementation is scattered all over the code of the program making it more difficult to understand and maintain. This scattering issue is addressed by Kiczales *et. al* [24] in *Aspect-Oriented Programming* (AOP). Examples of crosscutting concerns are: tracing, transactions, logging, etc.

AOP introduces the concept of a *pointcut* as way to match the places of a code program where the concern occurs. It also introduces *advices* as an artifact that specifies code to

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

Listing 2.4: AOP Base feature

```
public aspect Restore{
    int back = 0;
    void restore(){
        buf = back;
    }
    public pointcut set(int x): target(x)
        && execution(void Buffer.set(*));
    void before(int x); set(x){
        back = buf;
    }
}
```

Listing 2.5: AOP Restore feature

introduce behavior at the places matched by the pointcut. Advices are *woven* to the program at the point of the *pointcut*. Doing so, a crosscutting concern can be stored in a modular way making the program's code easier to understand and to maintain.

Listings 2.4 and 2.5 show the `Buffer` example implemented with AspectJ [23] (the mainstream AOP approach). It introduces a `Buffer` class that specifies a `buf` instance variable and methods `set` and `get`; and an aspect that adds a `back` instance variable, a method `restore` to restore the value of `buf` and that introduce an statement *before* the execution of method `set`. Although AOP is a general purpose approach, we believe it is an appropriate technique to FOP. Aspects can implement features and the weaving phase implements the composition of the features to the base program.

When applying AOP in a Feature-oriented programming way means to develop the features by Object-oriented programming artifacts such as classes and by aspects. Doing so, a set of classes and aspects can specify a feature. FOP requires to compose features to produce a software instance. Although AOP provides means to weave aspects with classes and by that produce a valid application, it does not provide means to specify the set of features

that is desired to compose. In a FOP point of view, we realize that AOP always compose all features that are specified in the environment, not allowing to select a subset from them.

1. *Granularity*: In general AOP is able to introduce modifications at any level of program entities. However, there are AOP implementations, such as AspectJ [23], that provide a granularity down till the level of methods.

2. *Allowed operations*: Features implemented by means of aspects can express addition and restricted modifications to a base program. Although modifications can introduce behavior *before*, *after* and *around* a method, they cannot introduce a statement in the middle of a method.

3. *Dependency Management*: The feature dependency in terms of dependency between aspects and the base program is addressed by the pre-compiler which weaves aspects in the base program. Consequently, all approaches do not focus on dependency management at all.

4. *Customized feature deployment*: Pointcuts allow to match places of a program by pattern matching. Doing so, the places where an aspect may introduce behavior depend on the elements present in the program at the moment that the aspect is woven. Thus, features by means of aspects provide a customized deployment.

5. *Feature specific language support*: AOP enhances OOP by introducing new concepts. Many implementations provide AOP capabilities to existing OOP languages.

2.3 Discussion

Table 2.1 summarizes the analysis of each presented approach based on the provided criteria. There is no approach that fulfilled all criteria. Although most approaches allow operations such as additions and a specific kind of modifications, none of them allow deletions. In the addition of a feature to a legacy system, since the legacy system is already implemented, may require the deletion of certain parts of the system. Thus, new approaches to FOP should be established that does address that kind of operation. Next to that, most of approaches set the granularity at statement level, but all of them allow it with restrictions. For instance, in AHEAD a programmer is not allowed to introduce a statement between two statements in an existing method.

2.4 Conclusions

After inspecting the state-of-the-art in Feature-oriented programming we acknowledge that all approaches intend to encapsulate features in separated modules, but that most of the

	Granularity	Allowed Operations	Depend. Mgmt.	Customized deployment	Feature specific language support
AHEAD	Statement with restrictions	Addition and modification	Yes	No	Yes
Generic composition	Not applicable	Addition	Yes	No	No
Lifting functions	Statement with restrictions	Addition and modification	Yes	No	Yes
Extra-functional features	Not applicable	Not applicable	No	No	Yes
Mixin-layers	Statement with restrictions	Addition and modification	Yes	No	Yes
FeatureC++	Statement with restrictions	Addition and modification	Yes	Yes	Yes
Composition-filters	Object	Addition and modification	No	No	Yes
AOP	Conceptually statement, in practice method	Addition and limited modification	No	Yes	Yes

Table 2.1: Analysis of the FOP approaches based on our criteria

approaches do it in a different way. There is an awareness that features are not sufficiently described by Object-oriented languages constructs—classes, methods, etc. That is why all approaches make language extensions. They need new means for expressing the concepts they introduce. It is difficult to imagine writing a program in a Feature-oriented way whether each feature is perfectly enclosed in a class, we realized a feature normally impact in many software entities at a time. A popular technique that addresses this issue is a *refinement* which is able to introduce modifications to previous language entities. However, refinements lack expressivity, since they cannot describe specific modifications on a software entity. For instance, refinements do not allow the deletion of a class, method or statement. They allow a specific kind of modification of methods that is based on calling the original method *before*, *after* certain statements of the new one.

A field that has been improved by *Feature-oriented programming* (FOP) is *Software Product-Lines* (SPL). When managing large-scale programs arise the necessity of providing an automatic reasoning for feature composition. Prehofer's *liftings* are not applicable for large-scale programs due feature interaction must be handled explicitly for every feature in the composition. Bergman's *Composition-filters* seems extremely fine-grained to be

applied in large-scale programs, as well. In a large-scale SPL the number of features can make a composition not feasible.

A common trend in Feature-oriented programming is step-wise development. That means to create a program by adding features in a incremental way. It has the benefit that each feature has all its dependencies satisfied, since it will always be applied after the entities on which it depends.

Nowadays, software is created by many developers at a time, all of them work in parallel developing software artifacts that can depend on entities which will be implemented in the feature or by others co-workers in parallel. Thus, dependency information requires to be handled by features themselves. A feature must know on which other features it depends.

Chapter 3

Changes as first-class entities

The goal of this thesis is to use first-class change objects as the building blocks of the features. For that, we need a model of first-class changes. This chapter analyzes the state-of-the-art approaches that successfully used first-class change objects in other domains. As the goal of this chapter is to find an appropriate model of that first-class changes, we first establish a set of criteria to which we compare the model of all approaches.

3.1 Concepts

A *change* is an operation that captures the information about an adaptation of a software system. A change can be produced by monitoring the developer when producing a software system. An upcoming trend is modeling changes as first-class entities. *First-class entities* are entities which can be used in programs without restriction. Some characteristics of a first-class object are: being storable in variables, having an intrinsic identity, being passable as a parameter, being returnable as the result, being created at runtime, being printable, being readable and being transmissible among distributed processes [10]. Approaches modeling changes as first-class entities exploit those properties to ease the manipulation of changes and the storage of information related to them. A field where this kind of model of changes would be useful is *Software Generators* [7, 20, 3, 5]. Software generators are programs that build other programs.

In this chapter, we present the state-of-the-art approaches we found to model changes as first-class objects. We developed a set of criteria with the aim of analyzing and placing these approaches. We aim an approach that provides the same kind of operations as the IDE provides to the developer, a proper granularity that allows us to capture the modifications that are produced, without losing information and an explicit management of dependencies that allows to manipulate change objects with freedom.

1. *Granularity.* A change has a subject which is the proper entity that is affected by this change. This criterium analyzes which is the level of the granularity provided by the approach.

2. *Allowed operations*. A change could consist in additions, modifications or deletions of entities. The aim of this criterium is to establish which kind of operations can be captured by changes.

3. *Dependency Management*. Changes are modeled as first-class entities, thus they can encapsulate information. This criterium verifies if the approach stores information about the dependencies.

3.2 Related work

The following sections show relevant related work that models changes as first-class entities. We analyze these approaches based on our criteria to check how other approaches solved the concerns that our criteria states. In order to find an appropriate model to represent design.

3.2.1 SpyWare

The goal of software evolution research is to use the history of the software for analyzing its present state and be able to predict its future development. The most common source of information that software evolution researchers have for analyzing the history of a software is the revision control system. It copes with the management of multiple versions of the same unit of information—e.g: a software entity as a class, method, statement. Most implementations are based on taking snapshots of software. Comparing different versions of a same software entity provides information about how a software is produced. However, this technique does not allow the recognition of which modifications were introduced to the old version to produce the new one. The information about the modifications between commits to the revision control repository is missing. Consequently, it cannot reconstruct how a certain version was produced, in others words which actions were taken to produce a software entity.

Robbes and Lanza [32] propose a change-based approach to software evolution. In their approach, the *Integrated Development Environment* (IDE) is instrumented with hooks that allow an IDE plug-in when modifying the system. The plug-in creates first-class entities that represent those actions.

First-class change entities are objects that capture the history of a system in an incremental way. They are able to reproduce the software of which they represent the history. When executed, they yield the a representation of the program at a certain point in time. They contain additional information interesting for evolution researchers, such as when and who performed the change operation.

As a proof-of-concept they developed *SpyWare*. Spyware is a Squeak implementation which monitors developer activity by using event handlers located at IDE hooks and generates change operations—as first-class entities—from them. Doing so, it is able to provide graphical information for analyzing the evolution of a program.

1. *Granularity*: Spyware is a tool that allows to capture the changes that a developer produces until the level of statements.

2. *Allowed operations*: Changes can represent additions, modifications and deletions of software entities.

3. *Dependency Management*: This approach does not take into consideration change dependencies. It stores the entire evolution history of a program—while it is written—in an incremental way. Thus, the application of a change requires the application of all the previous changes in time. Changes are not intended to be used as modular units that can be applied.

3.2.2 CatchUp!

Library developers are constantly evolving the libraries they produce. When a library evolves, developers are careful about changing the *Application Programming Interface* (API). In the evolution of a library many changes can affect client code, such as renaming of methods or changing the scope of an instance variable. Any of these changes may introduce errors to client code which reference old entities.

A popular technique for handling library evolution is deprecation. When a method has been renamed, the developer produces a new method with the new name and marks the old method as deprecated. This mark warns developers to not use this method anymore even though it allows backward compatibility.

In the evolution of a library, a new version could decide to remove many methods and to provide new improved ones—maybe with new method names. This could produce that client applications—which intensively use this old methods—may choose to rely on the old version of the library for an extended period.

Library management can turn more complex in time. Consider an application that needs to use libraries $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ depends on the version 1.0 of a library $\mathcal{C}$ and $\mathcal{B}$ depends on the version 2.0 of the same library $\mathcal{C}$.

Henkel and Diwan [19] propose a model for capturing the changes that a developer produces on a library. These changes can be replayed in the client's code reacting accordingly to the corresponding API evolution. In [27] the authors claim that most changes that a developer produces in the evolution of a library are refactorings. Moreover, they state that any change to a software program that preserves behavior can be understood

as a refactoring. Using IDE hooks to catch the refactorings introduced to a library, they store that information within changes modeled as first-class entities.

The process starts with the recording phase which occurs when the developer is evolving the library. He introduces refactorings that produce changes which are logged in an incremental way. Then the developer can annotate the changes with semantic information. For instance, a change specifying it has been produced for `Renaming to avoid name clashes`. The outcome of this phase is a new version of the library and a file containing a list with all executed changes. In case a refactoring does not affect client code, the change which represents that refactoring can be removed from the list of changes. In order to integrate the new library version into client code, the list of changes needs to be replayed on the client code.

CatchUp! is presented as a proof-of-concept. CatchUp! is an Eclipse plug-in written in Java. It provides means for recording library refactorings which are stored as a list of first-class change objects. It allows to replay the list of changes to a client code updating it for using a new library version.

1. *Granularity*: CatchUp! maintains a list with the kinds of refactorings that it is capable of capturing. It is able to record changes at the refactoring level which depending on the implementation can be set at the method level.

2. *Allowed operations*: This model records changes that produce additions and several kinds of modifications, such as: renaming classes, moving Java elements, etc. It also is capable of capturing deletions such as `remove parameters or remove exception types`.

3. *Dependency Management*: This approach does not take change dependency into consideration since library evolution is produced in a step-wise way. The list of changes is always replayed in the same way that it constructed.

3.2.3 VisualWorks: Change List

Smalltalk VisualWorks records and maintains all changes which are made to the system in a *Change List* [34]. The *Change List* tool provides a wide variety of operations for reading change files, comparing the contents of a change file, to reorder, remove and replay the changes to the system. This process could end in an error since one operation might require that another operation is applied before. Some uses are: recovering from a crash, reverting to a prior version and merging several developments into a unique environment. *Change List* provides a conflicts view to merge changes made in different collections, and to construct a single file containing only the desired changes. It can also be an aid in crash recovery, by filtering older changes from a changes file. The Change List tool makes it easy to see the evolution of—and to examine the details of—the code at any stage of its development history. It is particularly useful when we need to see a prior version in order

to change the code back.

In this approach changes are modeled as first-class entities as instances of the `Change` class. It has many subclasses for specific kinds of changes.

1. *Granularity*: Changes in the *Change List* can represent modifications at the level of methods. Although, it does not provide facilities for detecting changes at the statement level, that kind of changes can be detected by comparing the method before and after the change is applied.

2. *Allowed operations*: Changes can represent additions, modifications and deletion depending on which entity is affected. For instance, if a statement is removed, the change produced would describe that a method has been changed.

3. *Dependency Management*: This approach does not provide dependency management. The developer is responsible to establish the dependencies by setting the right order of the changes in the list.

3.2.4 Change-impact Analysis

Applying changes to an Object-oriented program could introduce undesired effects, due to the use of subtyping and dynamic dispatch. Software maintenance phase consumes the longest time in its life-cycle [21]. Because of that, maintenance programmers, who need to fix bugs and add enhancements to Object-oriented systems are often hesitant to make invasive changes due to possible side effects that these changes might introduce.

Ryder and Tip [33] elaborate on a collection of techniques for determining the impact that a set of source code changes has on the software system. Source edits are transformed into a set of changes. Table 3.1 presents the kinds of changes that are defined by this approach. For instance, an LC change is produced by any kind of source code change that affects dynamic dispatch behavior. A source change may trigger many changes, e.g., the addition of an empty method may imply several changes, of types AM and LC. They formalized the method dispatch process defining a *Lookup* function. The impact of edit actions on lookup can be monitored.

In this model, dependency is defined as the interaction between changes that make a change to need another one to ensure the compilation success. The introduction of categories for changes establishes a partial ordering between them. This computation is produced automatically.

A set of test drivers $\mathcal{T}$ is associated with a program $\mathcal{P}$. For each test driver t_i in the set $\mathcal{T}$, a *call graph* is produced. Each *node* is a method of the program $\mathcal{P}$ called from the test. Each *edge* corresponds to the calling relationship between the methods of $\mathcal{P}$. The same description applies for an edited program $\mathcal{P}'$. The model is completed with the definition

Type	*Description*
AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of method
LC	Change Virtual method lookup
AF	Add a field
DF	Delete a field

Table 3.1: Categories of changes

of *AffectedTests* which is a function for computing the test drivers that are affected by a set of changes, and *AffectingChanges* for computing the changes that affect a specific test driver. These functions detect the impact of the changes by traversing call graphs $\mathcal{P}$ and $\mathcal{P}'$.

Although this approach does not state that it models changes as first-class entities, we believe it can be appropriate for maintaining all the information related with changes and to manipulate changes.

1. *Granularity*: This approach capture changes related with classes, methods, fields and any modification that affects method lookup.

2. *Allowed operations*: The operations that the changes in this model allow include addition, modification and deletion of entities.

3. *Dependency Management*: This approach takes into consideration the dependency between changes that ensure a correct compilation of the resulting program.

3.2.5 Changeboxes

Most software applications must change continuously to meet new demands. Programming languages and development environments, however, invest more effort into providing mechanisms that limit change than into those that enable or exploit change. To address these phenomena, Denker *et. al* propose Changeboxes [12]; a general-purpose mechanism that encapsulates change as first-class entities in a running software system. Figure 3.1 shows how Changeboxes are modeled. A `ChangeBox` may specify three kinds of changes: definition, renaming or deletion. `Elements` are the target of a `ChangeSpecification` and model classes, methods or fields. A `ChangeSpecification` of a `ChangeBox` defines how one version of an `Element` may be transformed into another version of that `Element`.

Changeboxes capture the incremental evolution of a software system. They can be linked to `ancestors ChangeBoxes`, that represent prior `ChangeBoxes` to the same system. A `MergeStrategy` defines how two `ChangeBoxes` can be merged. A `ChangeBox` is defined

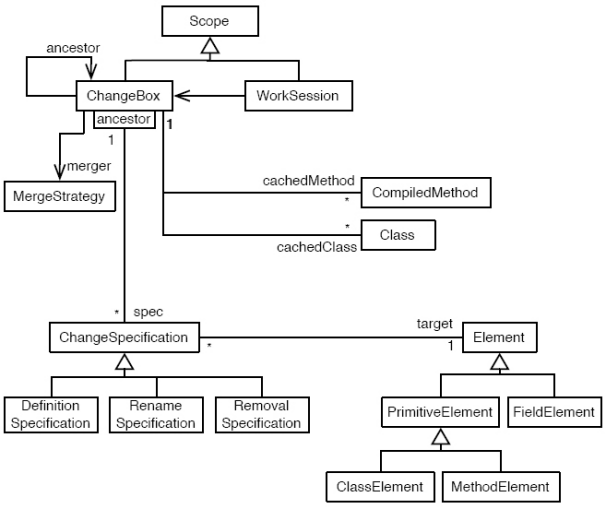


Figure 3.1: Changebox implementation class diagram

within a `Scope` which allows that multiple versions of a same entity execute at a time. The system is changed by introducing new Changeboxes which encapsulate a change specification. `CompiledMethod` and `Class` represent specific versions of classes and methods, that later can be used by the virtual machine to instantiate new objects.

A Changebox is an immutable entity that defines a snapshot of a system by encapsulating a set of change specifications, specifying a set of ancestor Changeboxes to which these changes apply and by providing a scope for dynamic execution.

A Smalltalk implementation of Changeboxes was successfully evaluated. It illustrates that bug fixes, new features and refactorings can be safely interpretable into a running system without impacting active sessions.

1. *Granularity*: The granularity of this model goes down to the level of fields, methods and classes, as we can see in the model diagram of Figure 3.1.

2. *Allowed operations*: This model allows defining, renaming and deleting software elements.

3. *Dependency Management*: Dependency is not modeled at the level of the elements that configure a Changebox, but at the level of Changeboxes itself. Changeboxes can be related with other Changeboxes by the *ancestor* relationship.

3.2.6 ChEOPS

Most interface development environments *IDE* lack support for system evolution. Most Smalltalk IDEs include a *Change List* tool which maintain a list of changes encapsulated

in first-class entities that can be referenced, queried and passed along. In [14], Ebraert *et. al* report on four shortcomings of the *Change List* tool.

Firstly, a *Restricted level of granularity*: Changes in *Change List* are restricted to classes and methods. Secondly, *Term overloading*: The same change object can represent several kinds of modifications. Thirdly, *Lack of high-level changes*: *Change List* does not provide facilities for monitoring the intention of a developer. Thus, a method refactory produces many changes which are captured by the *Change List* but the information that relates those changes with the original intention is missing. Finally, *No exploration facilities*: After several modifications to a program, the number of changes in the *Change List* can make the exploration of the list cumbersome and error-prone.

The authors propose a model that introduces five enhancements for overcoming *Change List* limitations. Change object should be fine-grained, composable, dependent, intensional and the IDE should be change-oriented.

Fine-grained first-class changes: Extending the class hierarchy of changes in the *Change List* tool, this model introduces dedicated change objects for all possible association between program entities. *Composable first-class changes*: When adding a new feature to a program many atomic changes are made. By grouping changes their intention—why those changes where introduced to the program—can be preserved. Doing so, changes can be classified into *atomic* changes and *composite* changes. This increases the ability to understand the change list. *Dependent first-class changes*: *Creational changes* are changes which introduce new entities to a program. Thus, a change which modifies an entity always *depend* on its creational change. *Intensional first-class changes*: Changes can be specified by an extension—by listing them—or an intention—by expressing them declaratively. A logic-based declarative language is proven to be suited for declaring changes intensionally. *Change-oriented IDE*: Interface Development Environments represent an ideal place for capturing the changes that a developer makes on a system. Most IDEs provide graphical means for commanding changes such as: creating a new class, a new method or applying a refactory. This semantic information can be captured using the hooks that the IDE provides.

The model introduces *preconditions* to manage the dependencies between changes. A precondition must be satisfied before a change is applied. Preconditions are imposed by the programming language, for example, methods can only be added to existing classes. The model is validated by the proof-of-concept implementation of *ChEOPS*. The Change- & Evolution-Oriented Programming Support *ChEOPS* is a proof-of-concept tool which enhances the *Change List* tool by adding five new features. It is implemented as a plug-in for the Smalltalk VisualWorks IDE.

1. *Granularity*: This model introduces dedicated change objects for all program entities and possible associations between them, such as: classes, methods, statements, invocations, accesses, etc.

2. *Allowed operations*: Changes can capture the addition or deletion of program entities. To capture a modification is provided by composing a deletion with a addition change object.

3. *Dependency Management*: In this model changes are aware of its dependencies. Dependencies are modeled as *preconditions* which must be satisfied before a changes is applied. Preconditions are imposed by the programming language, for example, methods can only be added to existing classes.

3.3 Discussion

Table 3.2 summarizes the analysis of each approach presented based on the provided criteria. In general, most approaches provide a fine granularity at least at level of statement. They also provides means to characterize changes as additions, modifications and deletions. However, only *Change-impact analysis*, *Changeboxes* and *ChEOPS* provide an explicit dependency manage model. Although these three approaches are appropriate for our model, we select *ChEOPS* as our change-based model. Our think that ChEOPS provides a simpler model which can be extended and adapted as needed.

	Granularity	Operations	Dependency Management.
SpyWare	Statement	Addition, modification and deletion	No
CatchUp!	Statement	Addition and specific modification	No
Change List	Method	Addition, modification and deletion	No
Change-impact Analysis	Class, method, field and lookup modifications	Addition, modification and deletion	Yes
Changeboxes	Class, method, field	Addition, modification and deletion	Yes
ChEOPS	Statement	Addition, modification and deletion	Yes

Table 3.2: Analysis of the approaches based on our criteria

3.4 Conclusions

The list of related work shows many approaches which successfully have modeled changes as first-class entities. All acknowledge that there is a necessity for capturing the operations

that are made when a program is written. Doing so, the construction of programs can be manipulated automatically by *software generators* [7, 20, 3, 5].

A first-class change must store information such as: when it was created, who created it, what dependencies it has and specific data related with its nature. It seems very useful to store that information within the change itself. Moreover, the first-class change object can be manipulated, assigned to variables, passed as argument to methods, returned as a result of operations, and so on. A convenient way for doing so is modeling changes as first-class entities.

Chapter 4

A change-base approach to FOP

This chapter presents how we elaborated on solutions that address the goals of this thesis. Chapter 2 explained how modularizing software with FOP opens opportunities to use program modules as building blocks to create new software variations. Chapter 2 presents many approaches to FOP and concluded with the shortcomings of these approaches.

We believe that modeling features by a set of changes provide many properties that help us to reach our goals. In chapter 3 we analyzed how changes have been proved successfully in other domains. In this chapter we first propose a model of first-class change objects. We present an explicit dependency management. Hence, we model features as set of changes and introduce *flexible* features a new concept to enhance feature reusability. We introduce tools that reify the concepts of our model.

4.1 Change-based software development

Change-Oriented Software Engineering (COSE) was first introduced by [14]. It centralizes *change* as the main development entity. In COSE, all operations a programmer performs while making a software system are captured in change objects.

We illustrate COSE by an example: a `Buffer` base program that follows the *value object* pattern [18]. It introduces a `buf` field and methods to `get` and `set` its value, then we incrementally add several features in a feature-oriented way.
Listing 4.1 shows the pseudo Smalltalk code for the `Buffer` class. It is annotated with the ID's of the corresponding change objects (on the left), which are depicted in the Figure 4.1. The figure does not really show the incremental development that was done in order to produce this code. Actually, first the `Buffer` class was added (B.1). Afterwards, an instance variable `buf` was added (B.1.1), then the methods `get` (B.1.2) with its body (B.1.2.1) and `set` (B.1.3) with its body (B.1.3.1) were added.
Afterwards, we implement the `Restore` feature as shown in Listing 4.2. It adds a field to store the old value of `buf` and a method to restore its value. We annotated the entities affected by the changes with a gray background. Figure 4.2 shows the changes that capture

```
B.1       class Buffer
B.1.1       buf := 0.
B.1.2       get
B.1.2.1           ^ buf
B.1.3       set: aValue
B.1.3.1           buf := aValue
```

Listing 4.1: Change-based Base feature

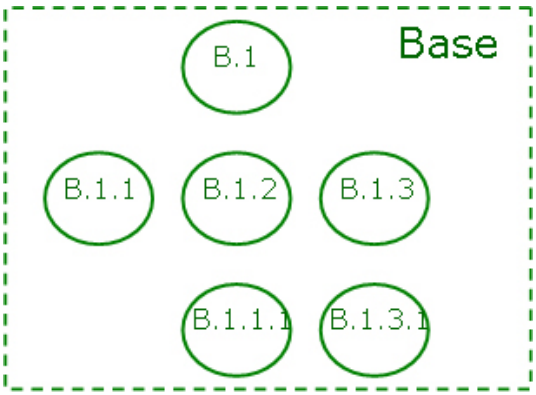


Figure 4.1: Changes representing the creation of the `Base` feature.

the specification of the restore feature.

A third module contains the functionality to print the values of the `buf` and `back` fields, by adding the `logit` method and placing invocations to this method in every previous one, as shown in Listing 4.3. Figure 4.3 represents the corresponding change objects that were applied. Finally, we add the `Multiple Restore` feature. It improves the `Buffer` with a functionality that allows to restore not only the previous object stored in `buf` but also its previous objects, as shown in Listing 4.4. Figure 4.4 represents the changes needed to produce the `Multiple Restore` feature.

4.2 Explicit dependency management

Some features depend on others in order to be able to do what they are supposed to do. A feature diagram, as shown in 2.1, keeps information about the relations between features. Moreover, using the composition rules and rationale, they provide information about feature interactions. The dependency relation is a binary relation that is finite, symmetric and reflexive. COSE's change objects are also related by such a dependency relationship. All changes on which a change `C` depends on are called the *parents* of `C`. We identify two types of change dependencies: *syntactical* and *semantical* [33]. Syntactic dependencies are enforced by the meta-model of the programming language that is used to develop the

```
          class Buffer
R.1       back := 0.
          buf  := 0.
          get
               ^ buf
          set: aValue
R.1.1          back   := buf
               buf    := aValue
R.2       restore
R.2.1          buf := back
```

Listing 4.2: Change-based Restore feature

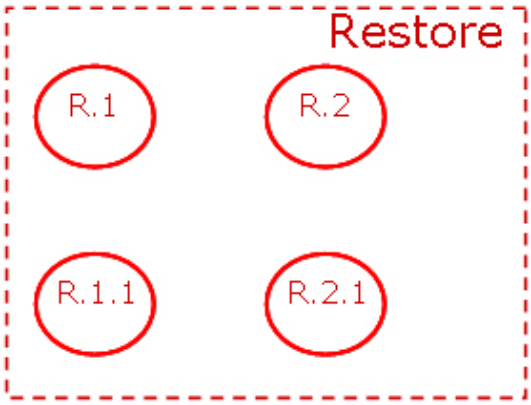


Figure 4.2: Changes representing the creation of the `Restore` feature.

```
          class Buffer
               back := 0.
               buf  := 0.
          get
L.1.3               self logit.
                    ^ buf
          set: aValue
L.1.4               self logit.
                    back   := buf.
                    buf    := aValue
          restore
L.1.5               self logit.
                    buf := back
L.1       logit
L.1.1               Transcript show: buf;cr.
L.1.2               Transcript show: back;cr.
```

Listing 4.3: Change-based Logging feature

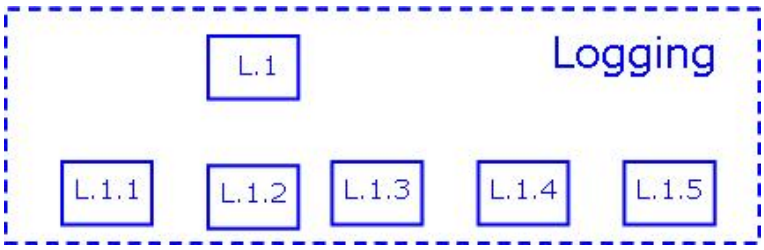


Figure 4.3: Changes representing the creation of the `Logging` feature.

```
      class Buffer
M.1       back := List new.
          buf  := 0.
          get
                self logit.
                ^ buf
          set: aValue
                self logit.
M.2           back addFirst: buf.
                buf  := aValue
          restore
                self logit.
M.3             buf := back removeFirst
          logit
                Transcript show: buf;cr.
M.4             Transcript show: back first;cr.
```

Listing 4.4: Change-based Multiple Restore feature

Figure 4.4: Changes representing the creation of the `Multiple Restore` feature.

software program. We find that language entities are related by the abstract syntax tree, we call these dependencies *hierarchical* dependencies. However, we realize that there are other kinds like *invocative* dependencies which are the dependencies between the change that adds a statement that is invoking a method and the change that adds the method that is being invoked. *Accessive* dependencies are found between changes that add instance variables and accesses to them. Finally, a *creational* dependency exists between a change that removes or modifies an entity and the change that added the entity, since an entity can be removed only whether it exists. Semantic dependencies come from the domain knowledge. Hence, the developer is required to establish these dependencies between software entities. One possible semantic dependency is the common intention of changes.

Consequently, a syntactic dependency is the one that is needed to ensure that a program compiles. Examples of a syntactic dependency are: a change that adds a method *depends on* the change that created the class where the method is added, or the change that adds an invocation to a method *depends on* the change that added the method that is invoked. A semantic dependency is a relation where the dependent entity does not exhibit the desired behavior whenever the entity on which it depends is not present. An example of a semantic dependence is where adding an invocation to method `m` only exhibits correct behavior in the presence of a modified version of the method `m`.

4.3 Changes as feature building blocks

We propose a model where features consist of a set of changes that are modeled as first-class entities. Doing so, we are able to exploit the first-class object properties described in chapter 3. We instrument changes with the information that models their dependencies. Changes that are aware of their dependencies can be transformed into a graph which the nodes are changes and the edges are dependencies. Figure 4.5 shows a view of the complete `Buffer` application of section 4.1. Features are encapsulated in dashed boxes containing circles which represent the changes produced when developing them. The arrows depict the syntactical dependencies between changes. However, all changes of a *monolithic* feature are said to be semantically dependent of each other.

Dependencies not only are confined to the feature, but they can reach changes within other features yielding feature dependencies. For example, in Figure 4.5, changes within the `Restore` feature depend on changes inside the `Base` feature, which yields in that the `Restore` feature depends on the `Base` . This means that feature dependency can be modeled by means of dependency between changes. Hence, a well-modularized program would contain less cross-feature dependencies. The number of cross-feature dependencies might be an interesting way to measure coupling between the features of a program.

The `Buffer` example shows several features that are added in an incremental way. The implementation of each feature consists of modifications to the building blocks of the base

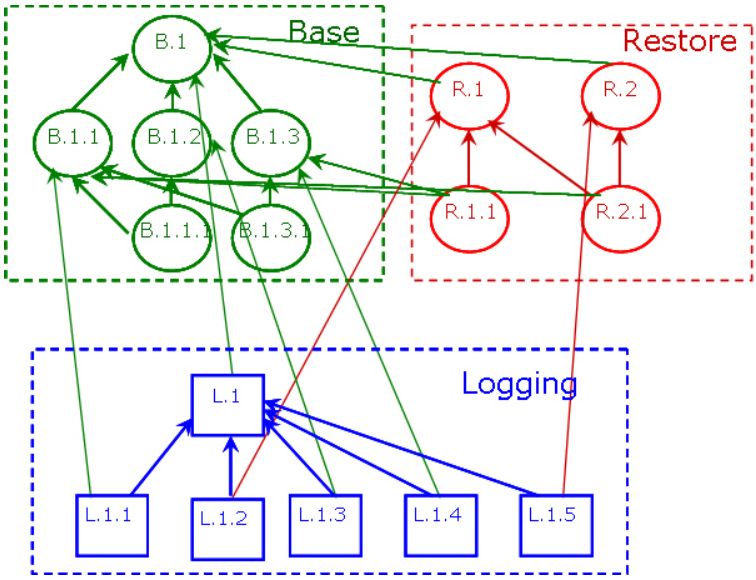


Figure 4.5: Change-based view of the `Buffer` example showing dependencies.

program and/or previously implemented features. These increments are modeled as a set of changes. The `Base` feature is an special case that modifies an empty program. Because these features are incrementally implemented, some changes do not only consist of additions but also modifications and deletions of software building blocks that can be: statements, instance variable accesses, methods or any other programming language construct. For instance, the `Multiple Restore` feature was created by modifying a statement—to initialize `back` with an empty list—and deleting the statement that assigned `buf` to `back` within the `set` method and adding in the same location a statement that calls the method `addFirst` of `back` which now is a list.

4.4 Feature Composition

The composition of features is the mechanism that allows the creation of a software variation based on the corresponding functionalities. The strategy to address the composition of features depends on how features are specified. We specify a feature by a set of changes that are aware of their dependencies.

A composition is valid if all parent changes of the change objects of the composition are part of the composition. Hence, an invalid composition is the result of composing features that contain a change of which at least one parent change does not reside int the composition. Listing 4.5 shows the implementation of an algorithm that verifies the validity of a composition. It receives as input a list containing the features of the required composition and the changes objects that specify the implementation of all features. It

```
FeatureComposition >> validateComposition: features
features do:
[:feature|
feature getChanges do:
    [:change|
    (change isIndependent
        || change allDependenciesSatisfied)
        ifTrue: [successful add: change]
        ifFalse:
        [
            self depthFirstStrategy: change
        ]
    ]
]
^List with: successful with: error
```

Listing 4.5: Algorithm to compose features

returns a list that consist of two lists. The `successful` list contains the changes in the order in which they must be applied to produce the required composition. The `error` list consists of the changes that could not be applied since they have at least one unsatisfied dependency. Thus, a feature composition from which the `error` list is not empty is an invalid composition.
The process starts by traversing the list of features. For each feature, it traverses its changes. If a change is independent—the change does not depend on any other change—or if it depends on changes that already were added to the `successful` list, it can be added to the `successful` list. Otherwise, a `depth-first traverse strategy` is applied starting from that change. The `depth-first traverse strategy` starts traversing the changes on which the former change depends, presented in Listing 4.6. All changes that are already in the `successful` list are excluded from this analysis. If from that process a parent change is found in the same feature that the change, it is asked if it does not depend on any other change. If so, it is added to the `successful` list, otherwise the same process is called recursively starting from the parent. However, if the parent change and the former change are in different features means that there is at least one dependency involving a change that is not in the composition at this point. Note that the output of this algorithm depends on the order of the features that are given as input. This improves the performance as we explain below.

In the best case, the order of the time complexity of this algorithm is $O(n)$, where n is the number of changes in the composition. It occurs when all changes do not have dependencies and can be added to the list directly without calling the recursive method. In the worst case the order is $O(n * (n + e))$ which is the result of applying n times a depth-first search in a graph with n nodes and e edges. It happens when the changes and

```
FeatureComposition >> depthFirstStrategy: aChange
aChange changesOnWhichIDepend do:[:parent|
(successful includes: parent)
    ifFalse:[
        (parent feature = aChange feature)
            ifTrue:[
                parent isIndependent
                    ifTrue:[
                        successful add: parent]
                    ifFalse:[
                        depthFirstStrategy: parent]
            ]
            ifFalse:[
                 error       add: aChange]
            ]
    ]
]
successful includesAll:
    (aChange changesOnWhichIDepend)
            ifTrue: [successful add: aChange].
```

Listing 4.6: depthFirstStrategy method for feature composition

their dependencies describe a list: all changes are chained, and the algorithm starts by the deepest change. For each change, the algorithm has to traverse the rest, adding one change for each cycle. In average, the order is $O(n^2)$ since the first part of the algorithm process each change—it means n times—and for each one it can apply a *depth-first* strategy which has an order $O(n)$ of time complexity when the number of edges is proportional to the number of nodes. Note that for each change in our dependency graph, the number of dependencies that it has would be at most the number of changes.

The main idea of our algorithm is to produce a totally ordered set from a number of partially ordered set. It yields a list with all changes from the features in the composition. Changes are ordered, in such a way, that their parents are in a previous position in the list. Doing so, the application of the changes in the same order of the `successful` list produces the required software composition. However, the algorithms presented introduce a constraint, the list of features to be composed require to be ordered in such a way that the dependencies of each feature are satisfied by its predecessors.

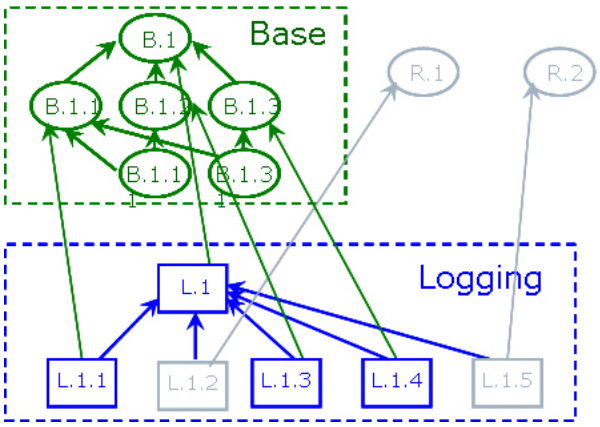


Figure 4.6: A valid composition of features `Base` and `Logging`.

4.5 Flexible features and their Composition

A *crosscutting* feature introduces changes that depend on changes that were introduced by more than one feature. This brings along an issue when one of those features is not in the composition, since that composition directly turns invalid. A quick and dirty solution would be to provide a feature for each combination of the features on which it depends. This kind of solution would increase the coupling between features and decrease reusability, which is the main purpose of modularizing features.

We advocate another solution which allows features to be partially deployed in a composition. A feature that shows crosscutting behavior can be implemented as a set of changes that does not have to be applied as a whole for a composition to be valid. When such a set is deployed in a composition, the composition algorithm should decide which parts should be included to and which omitted from the composition. Consequently, *flexible* features allow to apply a subset of the changes that they contain.

An example of such feature is the `Logging` feature in the `Buffer` example. The `Logging` feature consists of the addition of a method which implements its main functionality and several invocations that are placed in the methods that were introduced by other features. We argue that is such a case, although the changes that add such invocations depend on the changes that added the methods to which the former change add their invocations, we should be able to omit the former change from the composition to make it valid. In contrast to a feature that has to be applied as a whole (*monolithic*) we call features that can be partially applied *flexible* features. Figure 4.6 shows an example of a composition of `Base` and `Logging`. Since in this composition, the `Restore` feature is not present, the changes `L.1.2` and `L.1.5` from the `Logging` feature that depend on changes `R.1` and `R.2` are omitted, allowing to produce a valid composition.

Specifying a feature as *flexible* has to do with the semantics of the feature and must be

done manually by the developer. If a developer classifies a feature as *flexible*, this feature will be able to be included with all compositions. Composing a feature that was erroneously specified as *flexible* would yield a useless program. Consequently, programmers should understand the responsibility that comes with this flexibility.

Flexible features are not only confined to describe crosscutting functionality. For instance, a feature that implements a *facade* pattern [18], would implement a class with many methods each one invoking a complex service. It can be conveniently described by a *flexible* feature allowing to be composed with a set of features which not necessarily include all the services that the *facade* class references. In a composition, the *facade* class will provide only the methods that can be called in the composition. Once a *flexible* features is implemented, however, it cannot affect the features that are implemented afterwards.

We can ensure that the resulting program is able to compile, because all changes that will be deployed in the composition have their syntactical dependencies satisfied. The effect that produces *flexible* features in a composition is just to avoid some changes and not to avoid the syntactic dependencies.

In order to incorporate *flexible* features in our composition model, the algorithm presented in Listings 4.5 and 4.6 should be adapted. Doing so, the `validateComposition` method is provided with two more lists for storing an list with the changes that caused errors and a list of changes that were omitted when deploying a *flexible* feature as is presented in Listing 4.7. Moreover, we adapted the `depthFirstStrategy` method as shown in Listing 4.8. Whenever a change has a dependency with another change but the latter is in another feature which is not in the composition, we added a check to detect whether the change belongs to a *flexible* feature. If so, the change is omitted from the `successful` list and added to the `warning` list, otherwise it is added to the `error` list producing an invalid composition. Their order of time complexity is the same as in the previous versions.

4.6 Tool Support

We developed a tool to implement the concepts described in this chapter. It is based on the ChEOPS tool [14] for the change management and Mondrian [28] for the graphical part of depicting feature compositions.

The ChEOPS tool is a VisualWorks for Smalltalk enhancement that allows to capture the operations that a developer performs while developing a system into change objects. Change objects are modeled as first-class entities that are aware of their dependencies. It also provides means for specifying the *intention* of the modification that produced a change. We used this intention to specify which feature the change belongs to. A feature is specified by a *name* that allows its unique identification and a *type* for specifying whether

```
FeatureComposition >> validateComposition: features
features do:
[:feature|
feature getChanges do:
    [:change|
    (change isIndependent
        || change allDependenciesSatisfied)
        ifTrue: [successful add: change]
        ifFalse:
        [
            self depthFirstStrategy: change
        ]
    ]
]
^List with: successful with: error with: warning
```

Listing 4.7: Improved Algorithm to compose features

the feature is *flexible* or *monolithic*.

Before the developer starts writing the code of a feature, he switches on the recording tool of ChEOPS and specifies the name and type of the feature he is about to implement. Doing so, the changes produced during that session will all belong to that feature. After implementing a set of features with this technique, the developer is able to define a software composition that consist of a subset of those features. By invoking our algorithm to that composition the software variant that represents that composition can be created. The result of the algorithm can be a *valid* composition as shown in Figure 4.7, where the changes of each feature are in different colors. In that case, all changes that belong to the feature composition is returned in the order in which they need to be applied. The algorithm can also return a *valid* composition in which some changes highlighted in gray color (depicted by Figure 4.8). In that case, some changes are omitted from the `successful` list since some of them belong to a *flexible* feature and their dependencies were not completely satisfied. An *invalid* composition is shown in Figure 4.9. Here, the changes that have unsatisfied dependencies are colored in black.

Our tool also provides a graphical view which depicts the composition of features, showing changes of each feature in a different color. Note that in Figures 4.7 and 4.8, changes that belong to the *flexible* feature `Logging` are displayed as squares while the changes that belong to monolithic features are displayed as circles.

In the case of an invalid composition or a valid one that omits some changes by means of *flexible* features, the algorithm provides the developer with the exact set of changes that were omitted and what changes lead to the errors. With this information, the developer

```
FeatureComposition >> depthFirstStrategy: aChange
aChange changesOnWhichIDepend do:[:parent|
(successful includes: parent)
    ifFalse:[
        (parent feature = aChange feature)
            ifTrue:[
                parent isIndependent
                    ifTrue:[
                        successful add: parent]
                    ifFalse:[
                        depthFirstStrategy: parent]
            ]
            ifFalse:[
                aChange feature = #Flexible
                    ifTrue:
                        [warning    add: aChange]
                    ifFalse:
                        [error      add: aChange]
            ]
    ]
]
successful includesAll:
    (aChange changesOnWhichIDepend)
            ifTrue: [successful add: aChange].
error size > 0
    ifTrue: [warning removeAll]
```

Listing 4.8: Improved depthFirstStrategy method to manage flexible features

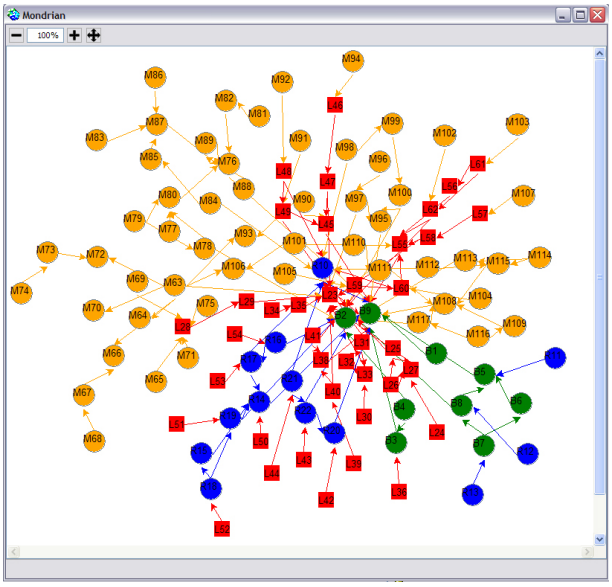


Figure 4.7: Valid composition of `Base`, `Restore`, `Logging` and `Multiple Restore`

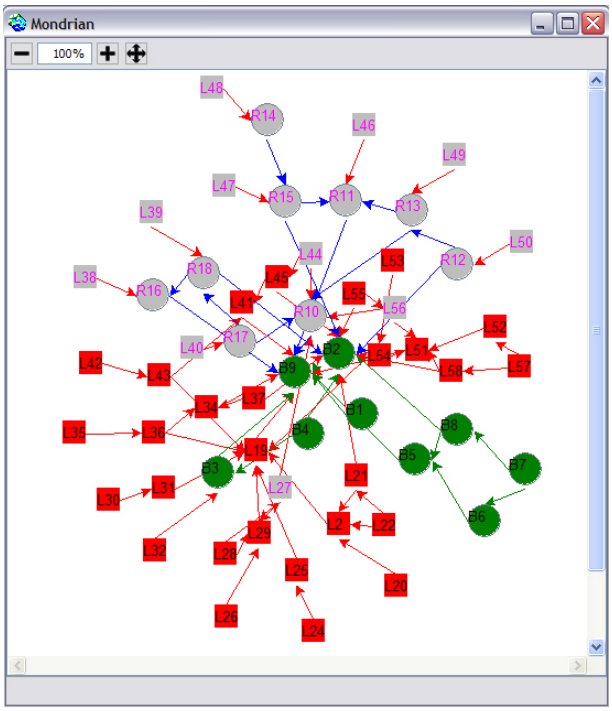


Figure 4.8: Valid composition of features `Base`, `Logging`

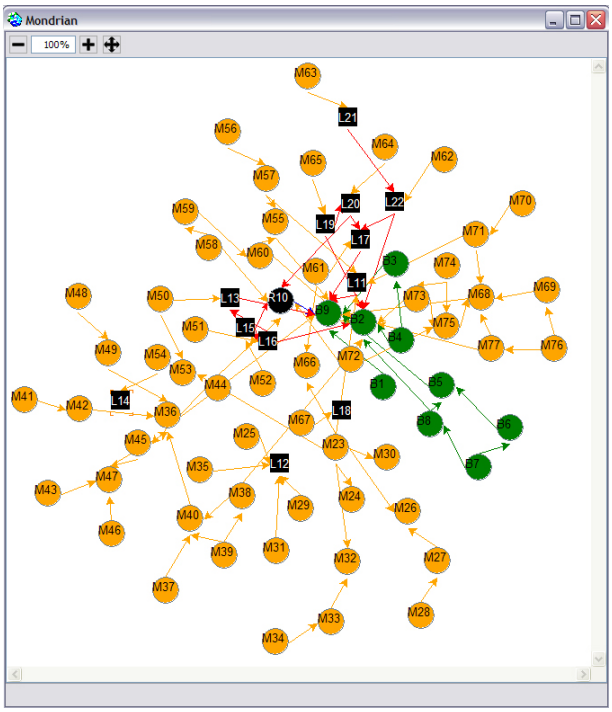


Figure 4.9: Invalid composition of features `Base` and `Multiple Restore`

can take the appropriate action to fix the problem (e.g. modify the composition, modify a feature implementation, etc.).

4.7 Requirements revisited

Table 4.1 presents the requirements that motivate our model and shows how our model satisfies all of them. Our approach provides mechanics to add, modify and delete software entities such as classes, methods, invocations, etc. By doing so, we tackle the issue of fine-grained granularity and allow all change operations: add, modify and delete. Moreover, we model features with changes that store their dependencies in an explicit way. By doing so, we have all the information needed to do dependency management. We provide a tool that allows the construction of features and their composition, which in case of invalid compositions warns the developer and presents information that can help the programmer to fix the problem. Finally, we introduce *flexible* features to model features that implement crosscutting functionalities. *Flexible* features can be reused in many compositions without any intervention. Next to that, our tool also supports:

- The encapsulating of operations performed in the IDE when writing source code as first-class change objects (via ChEOPS).
- The inspection of the change objects of a composition in the graphical view (via Mondrian). That might help to debug an invalid composition.

Granularity	Class, method, field, statement (access, invocation)
Allowed operations	Addition, modification and deletion
Dependency management	Changes store explicit dependencies
Customized deployment	Yes. Flexible features.
Feature-specific language support	Yes. ChEOPS and feature composition graphs.

Table 4.1: Requirements revisited

4.8 Conclusions

In this chapter, we presented a new model to address Feature-oriented programming. It uses changes as first-class entities for describing features. Exploiting the properties of first-class entities, our model is able to manipulate changes, passing them as arguments to methods and storing information about their dependencies into them.

We introduced a new concept that we call *flexible* features. In contrast to a *monolithic* feature, not all change objects of a *flexible* feature have to be applied when the feature is added to a composition. Using this concept, many compositions turn out valid that could not be composed before. The specification that a feature is *flexible* depends on the domain and has to be done by the developer.

Even though *flexible* features provide means to exploit a feature property by increasing its reuse, they could introduce unexpected behavior. If some of the features on which it depend is in the composition, the result will be a valid composition but the resulting application will actually not provide the functionality that is implemented in the *flexible* feature. It is up to the developer to decide whether a feature will behave correctly as a *flexible* feature.

As a technical contribution we provided a means to do FOP using VisualWorks for Smalltalk. However, the main goal of this implementation is to be a proof-of-concept tool with which we can validate our ideas: the algorithm to traverse the dependency graph of a composition, the graphical view that visualizes the composition and the ability to debug a composition by inspecting the changes.

Chapter 5

Evaluation

In this chapter, we evaluate our model by implementing `FOText`: a Feature-oriented implementation of a *word processor*. We expect that our model and tools fulfill:

1. Granularity: We expect that our model is capable of holding a granularity at the statement level. By that, it should be able to capture accesses to instance variables and method invocations.

2. Allowed operations: Our model must provide not only additions and modifications but also deletions of program entities.

3. Dependency management: Our model should manage the dependencies between changes and hence between features in an explicit way.

4. Customized feature deployment: Our model should increase the reuse of features by means of flexible features.

5. Feature-specific language support: Our tools should be useful when developing software in a Feature-oriented way.

5.1 FOText design

The FOText application provides a graphical interface in which the user types and edits texts. It also provides a menu—launched by the right button of the mouse—that allows users to invoke editing functions that are provided by FOText. FOText adheres to the *Model-View-Controller* design pattern [18].

We first define the list of all different features that FOText will provide. The result of this process is presented in the FODA diagram of Figure 5.1. Features such as: `New`, `Open`, `Save`, `SaveAs`, `Print`, `Copy-Cut-Paste`, `Find`, `SelectAll` are self explaining. The `Help` feature provides a pop-up window that presents information about the application. The

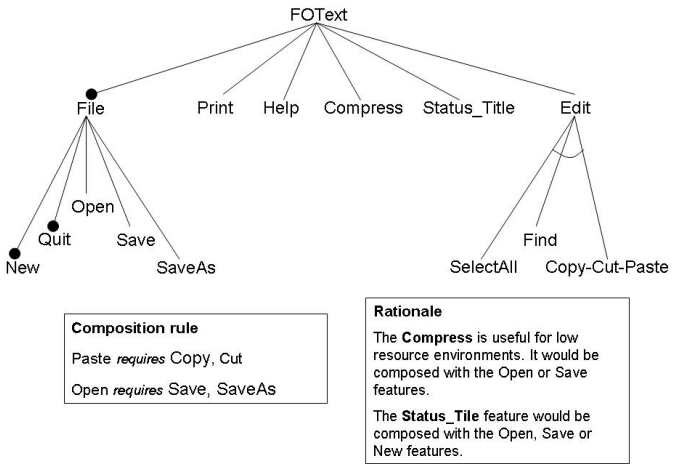


Figure 5.1: FODA of FOText

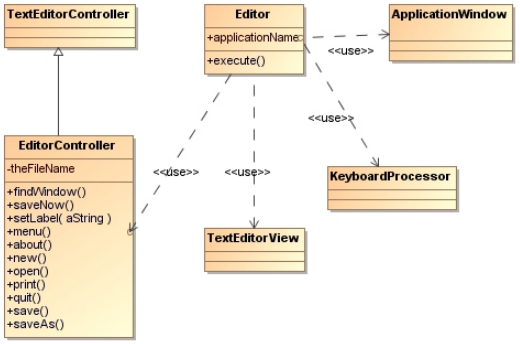


Figure 5.2: Class Diagram of FOText

`Compress` feature provides capabilities to use FOText under limited disk space availability. It adds the ability to compress the text files before they are saved, and decompresses them before they are opened. The `Status Title` feature displays in the title bar of the FOText window, the name of the file that is opened (each time a file is opened), and the name of the file that is being saved when that is the case. It also clears the window title bar when the user starts a `new` file.

The UML model in Figure 5.2 shows a class diagram with the classes that implement the functionalities provided by FOText. The diagram presents a main class called `Editor`. It has a method `execute` that creates an instance of the `ApplicationWindow` class. It provides the window to display and manipulate the text. The `execute` method also creates an instance of the class `TextEditorView` which is linked with an instance of the class `EditorController`. The instance of `TextEditorView` is linked to a `KeyboardProcessor`, to capture the events produced by the keyboard and is linked to the instance of the `ApplicationWindow` to embed the text area into the window. Based on this model,

our implementation provides a window with the capabilities to react to mouse and keyboard events. The `EditorController` class inherits from `TextEditorController`. It adds several functionalities such as: to provide a method `menu` to create the menu, which is how the user will lunch the FOText features, and to provide methods that implement the features provided by FOText. Note that `ApplicationWindow`, `TextEditorView`, `TextEditorController` and `KeyboardProcessor` are native Smalltalk classes, while `Editor` and `EditorController` are the classes that we introduce.

We use the ChEOPS tool [14] to capture the changes as first-class entities. Before implementing each feature (or the base program) we call the method: `toggleLoggingOn: 'leonel' for:'feature_name' type:#feature_type` on the `ChangeLogger`. In this invocation `feature_name` is a string with the name of the feature and `#feature_type` is a symbol that can be *flexible* or *monolithic*. Our tool logs changes as first-class entities that are instrumented with dependencies and tagged with a reference to the feature they implement. Doing so, the tool manage dependencies in an explicit way. This property allows us to manipulate changes with freedom and verify whether or not a change can be applied, just taking into account its dependencies.

5.2 FOText implementation

FOText is implemented in an incremental way. First the base program `Base` is implemented. It provides the window and some resources that will be used by the other features. Figure 5.4 shows the result of this implementation. A basic word processor that provides a window to type text and a menu with two features: `New` and `Quit`. To this base program we add the implementation of features one by one: `SaveAs`, `Save`, `Open`, `Copy-Cut-Paste`, `Find`, `SelectAll`, `Print`, `Help`, `Status in Title` and `Compress`. Most of those features affect the menu of the FOText application by adding a new functionality that the user can execute. Figures 5.5 and 5.6 show the increment of functionality by displaying pictures of the FOText's menu after each feature is implemented. While the `Base` feature consists only of *additions*, the rest of the features also introduce *modifications* (e.g. the `Open` feature modifies the menu introduced by the `Base` feature) and *deletions* (e.g. the `Compress` feature deletes several *statements* and introduce new ones). We observe that our tool allows operations such as: *additions*, *modifications* and *deletions*, and set the granularity level at classes, methods, instance variable and statements as depicted in Figure 5.3.

We do not provide pictures of the menu after the addition of features `Compress` and `Status Title` because they do not impact the menu, but rather add a behavior to the application that is only perceived while it is used.

Note that we only implement `FOText` in the VisualWorks for Smalltalk IDE as a way to capture the changes. Once the changes are represented by first-class objects, we are able to use them to compose features and produce a family of program variations. Table

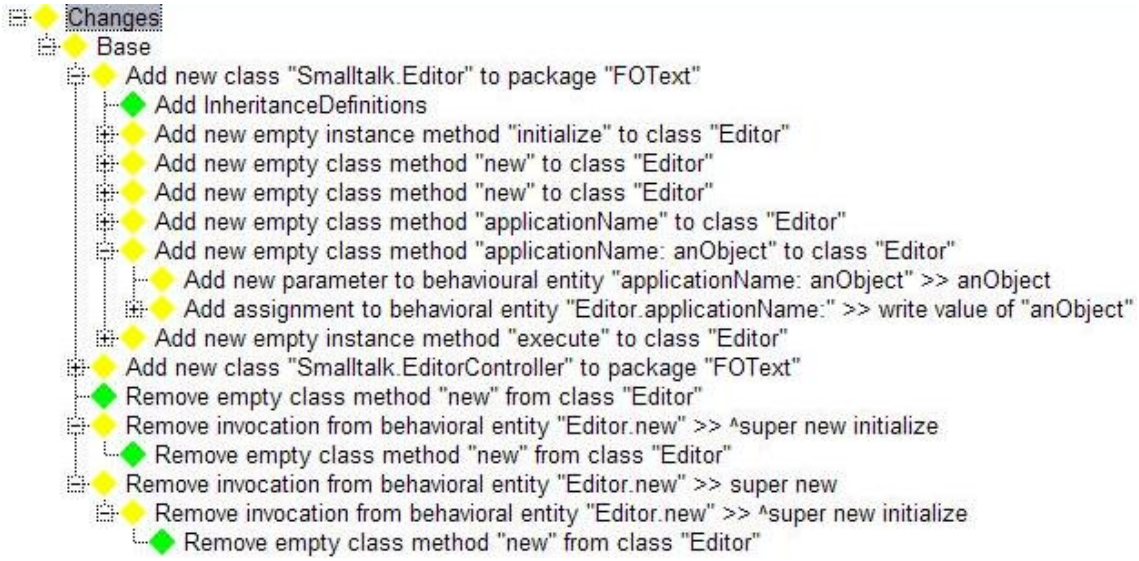


Figure 5.3: FOText: List of changes produced

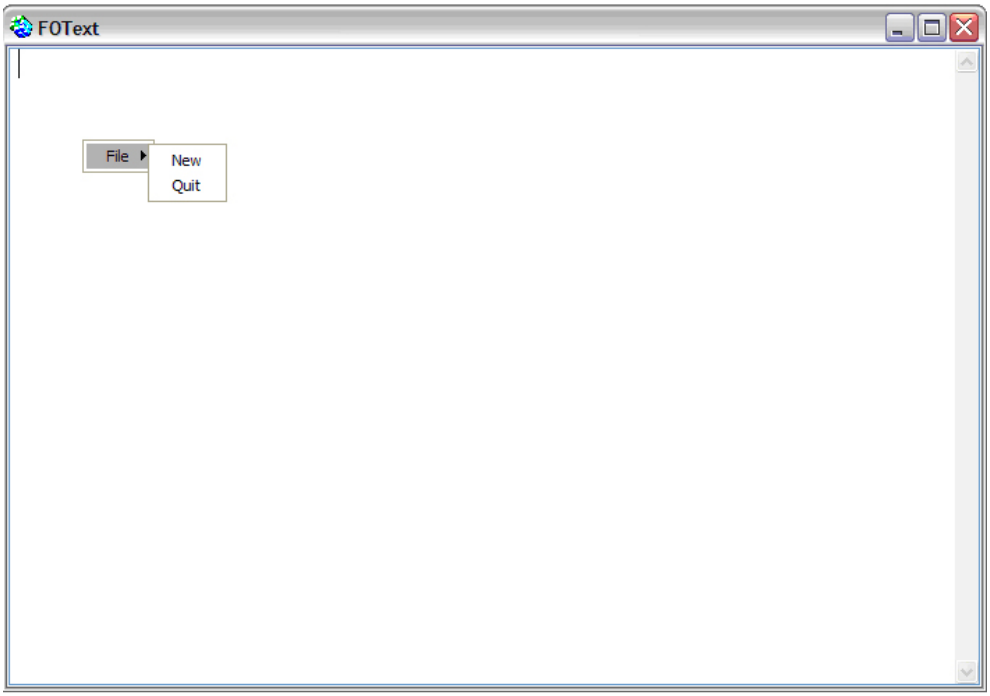


Figure 5.4: FOText: base program

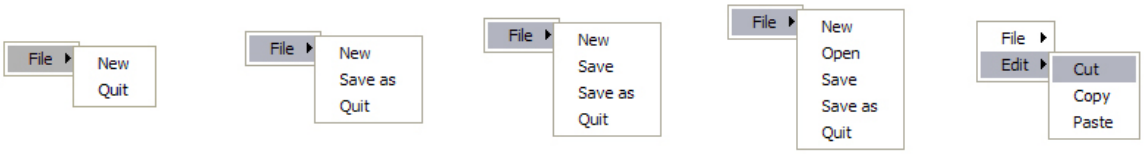


Figure 5.5: Menu evolution: `Base`, `SaveAs`, `Save`, `Open` and `Copy-Cut-Paste`

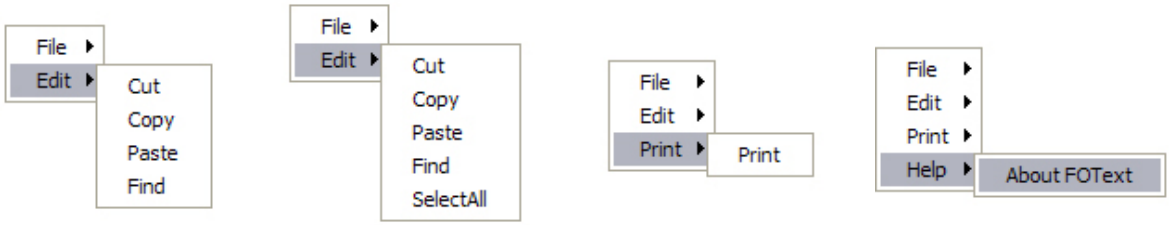


Figure 5.6: Menu evolution: `Find`, `SelectAll`, `Print` and `Help`

5.1 show some statistics about the number of changes and dependencies that are captured. The number of changes has the same order than the number of dependencies which yields that the order of time complexity of the algorithm that we propose to compose features goes down at $O(n)$ to n changes.

Feature	Number of changes	Number of dependencies
Base	130	158
SaveAs	88	106
Save	65	74
Open	101	121
Copy_Cut_Paste	72	82
Find	86	98
SelectAll	89	102
Print	182	226
Help	137	154
Status_Title	159	193
Compress	151	147
Total	1260	1362

Table 5.1: Changes captured after the implementation

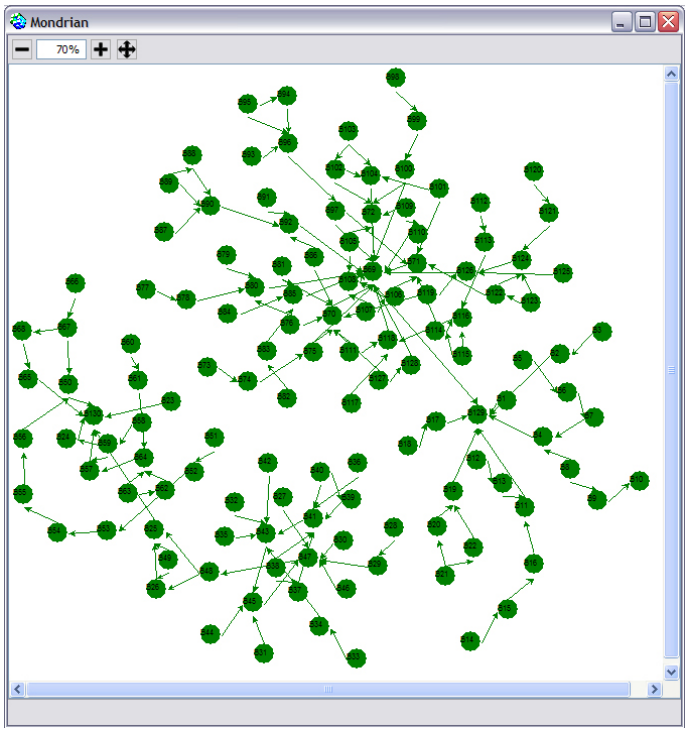


Figure 5.7: Dependency graph of the `Base` feature

5.3 Feature Dependency

Figure 5.7 shows the dependency graph of the `base` feature. The nodes represent the changes the `base` feature consists of and the edges represent the syntactic dependencies between the changes. Observe that our model provides an explicit dependency management between changes (and features).

5.4 Feature Composition

Once the changes are captured into features, our model allows to compose them and advises us whether the composition is valid or not. Although the natural step is to apply the changes producing the application that the composition specifies, our tool is not able to do so, since it is based on ChEOPS which presents some issues to perform that action. Nevertheless, our tool is able to provide information about the validity of composition. We show three compositions that characterize the possible situations.

5.4.1 A valid composition

When we compose the features: `Base`, `SaveAs`, `Save`, `Open`, `Copy_Cut_Paste`, `Find`, `SelectAll`, `Print`, `Help`, `Status_Title` and `Compress`, our tool inform that this composition is valid

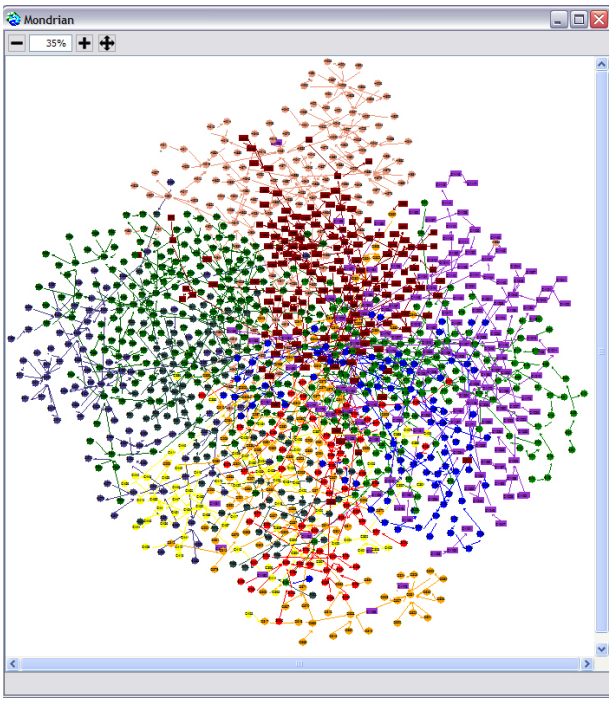


Figure 5.8: A valid composition of all features implemented

and depicts the dependency graph that the *changes* yield. A screenshot of this view is presented in Figure 5.8. This composition involves 11 features which are specified by 1260 changes. The time required to display the graphic of Figure 5.8 was 281873 milliseconds (4.63 mins. aprox.) in a computer with 2GB of RAM and a processor Intel Centrino Duo at 1.66GHZ. However, the most of that time was spent by the Mondrian tool in the layout which in fact took 281690 milliseconds. Our algorithm that validate the composition only took 183 milliseconds.

5.4.2 An invalid composition

The following composition involves `Base` and the `Save` feature. Figure 5.9 depicts the result of this composition. It shows the changes that belong to feature `Base` in red and the changes of the `Save` feature in green. The black nodes represent changes that are not in the composition but they are required. Every composition that shows black nodes is an invalid one.

By inspecting the black nodes of this composition, we found that there are changes from feature `Save` that depend on changes of feature `SavesAs` which is not in this composition. Our approach provides this graphical view as a tool support to advise the programmer about the changes that make a composition invalid. Hence, we can conclude that our tool

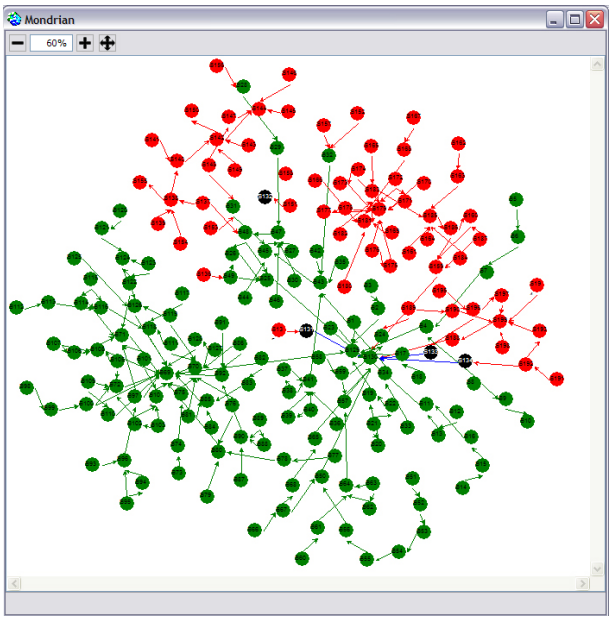


Figure 5.9: An invalid composition of features `Base` and `Save`

provides valuable support for developing in a Feature-oriented way.

5.4.3 Valid compositions by means of flexible features

Our approach provides *flexible* features which are deployed in a specific way depending on the context. It means the *flexible* feature will provide a customized functionality depending on the features that are present in a composition.

We compose an application with the `Base`, `SaveAs` and `Compress` features of which the last one is *flexible*. It means that the changes of feature `Compress` may be omitted if they would have unsatisfied dependencies. Figure 5.10 shows this composition, where changes belonging to `Base`, `SaveAs` and `Compress` are respectively depicted as green circles, blue circles and yellow boxes. Note that Figure 5.10 contains a gray node that belongs to the `Compress` feature, but that will not be applied due to its dependency to a change that does not reside in the composition (the gray circle).

In the following example we add the `Compress` feature to a viewer version of FOText which is composed by the `Base` and `Open` features. The result of this composition is depicted by Figure 5.11. changes of `Base`, `Open` and `Compress` are respectively depicted as green circles, blue circles and yellow boxes. In this composition, some changes of the flexible `Compress` feature are grayed out, meaning they will not be included in the composition.

A closer inspection of the gray entities of both figures learns us that different change objects are concerned: C259 in Figure 5.10 and C261, C258, C229 in Figure 5.11. This shows

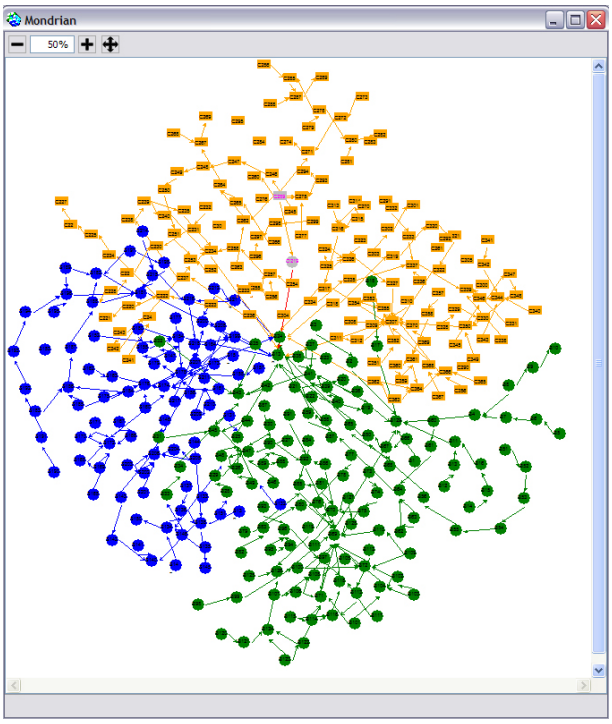


Figure 5.10: Valid composition of features `Base`, `SaveAs` and `Compress`

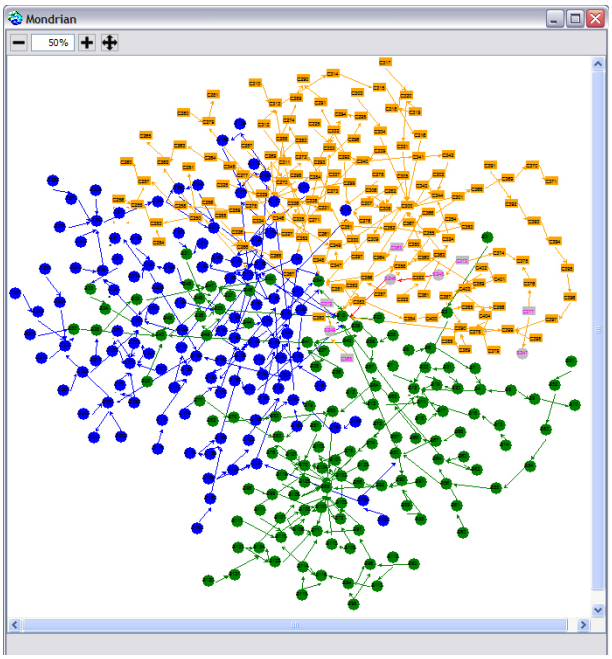


Figure 5.11: Valid composition of features `Base`, `Open` and `Compress`

how our approach and tools automatically customize *flexible* features to make compositions valid. It shows how this technique allows compositions that would not be permitted by other FOP approaches, but which do make sense. Consequently, this shows how our model allows a customized feature deployment, improving the reusability of features.

5.5 Conclusions

We evaluate our approach by declaring a set of requirements to which our approach adheres. We showed our approach allows operations such as: *additions*, *modifications* and *deletions*. We showed examples where such operations were captured on the level of classes, methods, instance variables and statements. We use the ChEOPS tool, that set the granularity level below the statement level. For instance, it is able to capture the *addition* of an invocation—which happens inside a statement. In our approach changes are instrumented with their dependencies in a explicit way. We use this property to allow to *flexible* features to decide which changes can be omitted. Doing so, a feature can provide customized functionality taking into account the feature present in the composition.

When composing the 11 features we realized that the graphic tool does not provide too much information, since the number of changes that ChEOPS produce is huge. However, our algorithm provides information to know whether the composition is valid, invalid or can be valid avoiding some changes. We believe that setting the granularity of changes at the level of statement would be better to handle changes with a graphical tool as we provided.

We present two examples which demonstrate the power of deployment by means of *flexible* features. We realized that to recognize when a feature can be specified as *flexible* is not simple. We believed that *flexible* features are not an invention but a discovery. It is a natural characteristic of some features, that now we recognize and exploit.

Although our tools provide a effective functionality to validate a composition and to debug an invalid composition, we found that the time that requires the Mondrian tool to display the graph with the composition would make it not scalable. Hence, we believe that improving the efficiency of Mondrian can be improved the usability of our tools.

The work presented in this chapter shows that our model can be applied successfully in a Feature-oriented development. Once changes are captured, our model is able to provide information about the feasibility of compositions and advise in case a composition turns out invalid.

Chapter 6

Conclusions

6.1 Conclusions

Several models have been proposed to address software variations. In order to implement an extra functionality to the variations of a program, we can implement the solution directly into the code which requires to reproduce an adaptation for all variations of the program. This solution hinders reusability and suffers from a combinatorial explosion [29]. An approach that tackles these problems is Feature-oriented programming (FOP). It addresses software development by modularizing a system in a base program which can be extended with any number of feature modules to form new software variations. However, sometimes a feature requires other features to produce a valid composition. In order to increase the reusability of features we identify five criteria to which FOP approaches should adhere:

1. *Granularity.* The kind of entities the FOP approach can handle should be as fine-grained as possible. This allows a very detailed specification of features.

2. *Allowed operations.* We want a FOP approach to allow the specification of the addition, modification and deletion of building blocks. We believe that providing these operations increases the expressivity of the FOP approach.

3. *Dependency management.* The approach should provide features with information about the dependencies they have. Doing so, a composition can be validated and support can be provided to fix an invalid compositions.

4. *Customized feature deployment.* The approach should allow to express crosscutting functionality as one feature module, that can be added to any composition, even if some of the features it depends on do not reside in the composition. This would increase the reusability of features, as a feature only needs to be created once and can than be composed with any variation of the program without having to adapt it.

5. *Feature-Specific language support.* The approach should provide tools to support Feature-oriented programming. Consequently, we expect support in validating fea-

ture composition and advise on what actions to take to make an invalid composition valid.

FOP has been addressed by many approaches. We study these approaches to check whether they satisfy the above-mentioned criteria. We acknowledge that all approaches struggle producing expressive means to describe the changes required to produce features, but any of them accomplished that task completely. Although, some approaches are aware about the dependencies between features, they do not provide means to make these dependencies explicit. Although, they allow operations such as addition and modification, they do not provide means to express deletion. Finally they do not provide means to reuse features when some feature dependencies are unsatisfied.

We propose changes modeled as first-class entities as a proper way to describe features. By modeling changes as first-class entities, we can manipulate them freely, passing them as method's arguments or storing into them information such as their dependencies. There are several approaches that used *changes* in other research domains. We evaluated their models to three criteria (*allowed operations*, *granularity* and *dependency management*), in order to choose an appropriate one. We found that there are approaches that successfully applied changes as first-class entities in the research domain of software evolution, providing them with explicit dependencies, allowing operations such as additions, modifications and deletions. This analysis reinforces our assumption that first-class changes are an appropriate building block of features. We choose the model of COSE [14] as a base for our change-based model for FOP.

We present a model that uses changes as first-class entities as the building blocks of features. Changes are instrumented with explicit dependencies which allow us to validate a composition and advise when a composition is invalid. We also present a new concept: *flexible* features that serve to express crosscutting functionalities. When composing a *flexible* feature with other features its unsatisfied dependencies are avoided which increases its reusability.

We provide a proof-of-concept implementation based on ChEOPS [14] to capture the operations that the programmer does in the IDE when developing a program as first-class changes. We implement a graphical tool based on Mondrian [28] to support the composition of features and advise the developer what action to take when a composition is not valid. The tool also allows the developer to inspect which changes are turning the composition invalid.

We evaluate our model by developing a simple word processor called `FOText`. By this implementation we check that our model and tools fulfill the requirements we listed above. We show the operations that our model allow are additions, modifications and deletions. We present some examples where features require these operations to describe their functionality. We also show that our tools allow to express building blocks at the level

of granularity of statements. We can even capture operations as method invocations and variable accesses. In our model changes and features—since features are set of changes—are provided with their dependencies. Those are used when composing features and allow to advise the developer about what actions to take to make an invalid composition valid. We show two examples where we used *flexible* features as an appropriate way to describe crosscutting functionalities. This allows to features to provide a customized functionality which increases the reusability of features. Our model provides tools do FOP working with *flexible* features and to establish valid compositions.

6.2 Contributions

At the conceptual level, we introduce a model to do Feature-oriented programming by means of first-class change objects. We provide an algorithm that traverses the graph described by the changes and their dependencies, advising when a composition is valid, invalid or it would be valid whether some changes are ignored. Besides, we introduce a new concept: *flexible* features. We believe that *flexible* features are a proper way to implement features that add a crosscutting functionality to a base program. By means of *flexible* features a feature can be implemented once and deployed in any composition without having to adapt it. This increases the reusability of features.

At the technical level, we provide tools that allow FOP in VisualWorks Smalltalk. The tools are based on ChEOPS [14] which allows to capture the modifications that a programmer performs when developing a software. Finally, we provide means to compose features as sets of changes and advise when a composition is invalid. We provide a graphical tool that uses Mondrian [28] to depict the graph of the changes and their dependencies, coloring the changes depending on the feature on which they belong. It also allows to inspect the changes which in the case of errors provides useful information to debug the composition.

6.3 Limitations

We observe that at the conceptual level a *flexible* feature is an extensional implementation of a functionality. It declares the intension of the programmer by specifying a concrete set of building blocks. A *flexible* feature that specifies a crosscutting concern once implemented, however, cannot introduce behavior to features implemented afterwards. Allowing to specify features by intensions more than extensions would be a proper way to address that issue.

The implementation that we provide to handle *flexible* features introduces a constraint. It requires to provide the list of features to be composed in a incremental way. It means that the dependencies of each feature in the list must be fulfilled by its predecessors.

We made an evaluation that shows some drawbacks at the technical level of our tools. When creating large applications, the amount of changes grows very high. That decreases the usability of the graphical tool. Although the amount of changes growth in large applications, it can be manageable by providing filters to visualize the composition at different levels of granularity.

The output of our composition tool is a list that specifies the order in which the changes should be applied in order to produce the software system specified. Our tool, however, is not able to apply the changes and produce the software code, since our tool are based on ChEOPS which keep dependencies within changes to the objects that they refer. Consequently, the changes turn obsolete in the absence of the objects.

The ChEOPS tool works as a recording machine that monitors the actions performed by the developer. When a mistake is made there are two alternatives: to keep the mistake and fix it carrying with the production of erroneous changes, or to stop ChEOPS of recording, purge the changes produced by the mistake and make ChEOPS to log again. However, any of these solutions seems to be appropriate.

6.4 Future Work

Flexible features are meant to increase the reusability of features. We found that they are a proper concept to specify crosscutting concerns. Once a *flexible* feature is implemented, however, the features that are implemented afterwards cannot be reached by the *flexible* feature. We can address this situation by allowing to specify *flexible* features by intensions more than extension.

Implementing large applications the number of changes grows high making our graphical tool less manageable. Providing the graphical view with filters that would allow to hide some *changes* based on a criteria—such us "filter all changes created after 01/01/1970"—would increase the usability of the tool for large-scale applications. Another filtering mechanism could display the composition at different levels of granularity. In such a way that the tool would show the macro changes at class level or at statement level. Filters would provide a customized view depending on the level of information that a developer wants.

The ChEOPS tool allows to capture the changes as first-class entities that are produced when a programmer is writing software. This allows us to manipulate these changes, to produce the compositions and to advise about invalid compositions. Even though the natural step in this process is to deploy this resulting list of changes that represent a software into a program code, we are not able to do that since our tool is based on ChEOPS on which *changes* maintain references with the software entity that was produced when capturing the change. For instance, whether a class is being added the change that captures

its creation has a reference to the created class. Hence, when the class is not present the change as a broken reference and cannot be applied. Although it is a technical issue that would be overcome without much effort and it is proposed as future work.

In our implementation the list of features to be composed require to be specified in such a way that the dependencies of a feature are satisfied by its predecessors features. A track of future work would be to release that constraint of our implementation improving our algorithm that traverses the dependency graph of changes, allowing to specify the features in an set without any specific order.

Since the recognition of a *flexible* feature must be done by the programmer, a characterization would make this task more reliable. We believe that a full characterization of *flexible* features is required to exploit it more. However, we realized that *flexible* features is a proper concept to manage features that implement crosscutting functionalities.

Chapter 7

Appendix A

Vrije Universiteit Brussel

Software variations by means of first-class change objects

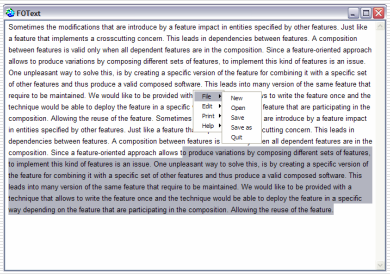
Peter Ebraert	pebraert@vub.ac.be
Leonel Merino	**leonelmerino@gmail.com**
Theo D'Hondt	tjdhondt@vub.ac.be

Agenda

- Software Variability
- Requirements
- Related work
- Discussion
- Our FOP model
- Demo
- Future work
- Conclusions

Software Variability

- FOText is a word processor that requires to be improved with a functionality to compress the files it produce.
- FOText has two variations:
  - FOText viewer
  - FOText full



Approach

- Ad-hoc: add the code needed directly into the application.
  - Solution tightly coupled

- Feature-oriented programming: create a feature that adds this functionality to the FOText base program.
  - Features can be reused

FOP requirements

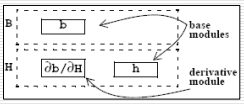
We identify the following requirements:

- The compress feature requires ***adding*** new statements and ***deleting*** existing ones in the *open* and *save* functionalities.
- Although we have two variations of FOText we would like to create the *compress feature* just once and **reuse** it.

We pursue an approach that fulfills these criteria

Related work — AHEAD *

- AHEAD addresses FOP by providing: step-wise development, generative programming and algebras.




```
class EditorController {
  Menu menu(){
    return new Array( 'new', 'quit');
  }
  void new(){window.title = "";}
}
class Editor{
  String applicationName = "";
  void execute(){
    window = new Window( new
EditorController);
  }
}
```

(a) Base

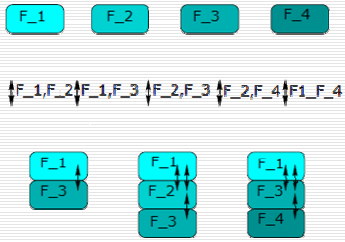
```
refines class EditorController{
String theFileName  = "";
  Menu menu(){
    return new Array( 'new', 'open', 'quit');
  }
  void new(){
    theFileName = null;
    super.new();
  }
  void open(){
    theFileName = new Dialog("Enter the
filename");
    window.title = theFileName;
    window.body = open( theFileName);
  }
}
```

(c)Open

* D. Batory, J. Sarvela, and A. Rauschmayer. Scaling stepwise refinement, 2003.

Related work — Lifting functions *

- This model allows flexible composition of objects from a set of features.
- Lifting functions resolve the interactions between features.



* Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241:419{434, 1997.

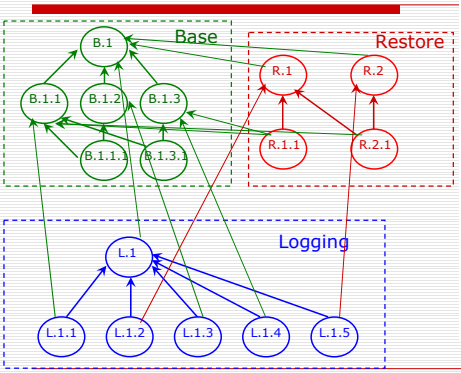
Discussion

	AHEAD	Lifting Functions	FeatureC++	Mixin-layers
Operations allowed	Addition and modification *	Addition	Additions and modifications	Additions and modifications
Granularity	Method, field and statement *	Statement	Method, field and statement *	Statement
Feature reuse	No	No	No	No
Dependency management	Implicit	Implicit	Implicit	Implicit

A change based solution for FOP

- ***Change:*** any operation produced by the programmer in the code.
- Feature as a set of first-class change objects.
- Changes are captured from the IDE using the ChEOPS tool.
- Explicitly stored dependencies between changes
- We aim at feature reuse

FOP model




```
B.1     class Buffer{
B.1.1     int buf = 0;
R.1       int back = 0;
B.1.2     void get(){
L.1.3           logit();
B.1.1.1         return buf;
          }
B.1.3     int set( int x ){
L.1.4           logit();
R.1.1           back = buf;
B.1.3.1         buf = x;
          }
R.2       void restore(){
L.1.5           logit();
R.1.2           buf = back;
          }
L.1       void logit(){
L.1.1           print( buf );
L.1.2           print( back );
          }
        }
```

Demo

Future work

- Develop a characterization for flexible features
- Not only allowing addition and deletion but modifications would decrease the number of changes.
- In ChEOPS the changes cannot be exported or applied*

Conclusion

- Programming languages do not provide enough tools to do FOP.
- We propose:
  - A conceptual model where features are described by changes and dependencies are explicit. Moreover, we introduce flexible features.
  - Tool support to compose features.

Thanks !

Software variations by means of first-class change objects

leonelmerino @ gmail.com

Chapter 8

Appendix B

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.

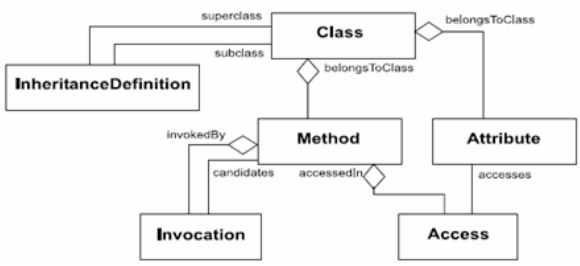


Fig. 1. Famix - Core Model

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.

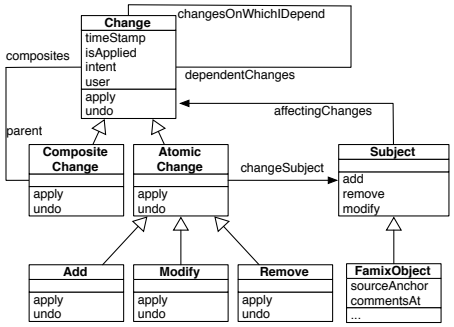


Fig. 2. ChEOPS - Core Model

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.

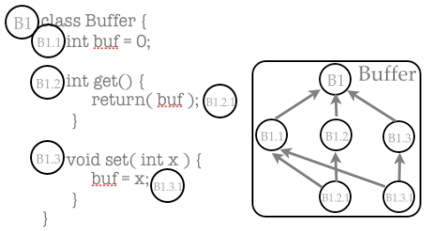


Fig. 3. Buffer: (left) source code (right) change objects

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.

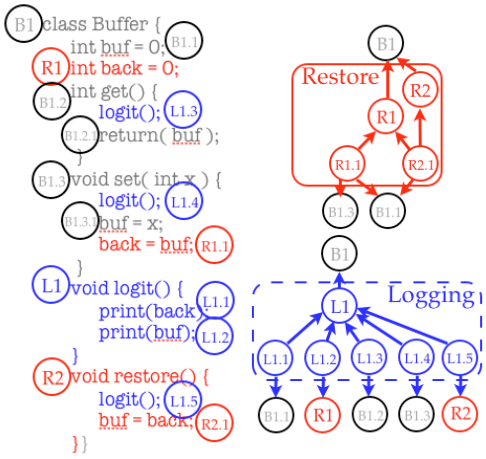


Fig. 4. Buffer, Restore, Log features: (left) source code (right) change objects

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 stepwise 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.

Bibliography

[1] Gentzane Aldekoa, Salvador Trujillo, Goiuria Sagardui Mendieta, and Oscar Díaz. Quantifying maintainability in feature oriented product lines. In *CSMR*, pages 243–247, 2008.

[2] Sven Apel, Thomas Leich, Marko Rosenmller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of Fourth International Conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia*, 2005.

[3] Don Batory. Intelligent components and software generators. Technical report, Austin, TX, USA, 1997.

[4] Don Batory, David Benavides, and Antonio Ruiz-Cortés. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[5] Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In *ICSR '96: Proceedings of the 4th International Conference on Software Reuse*, page 72, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling stepwise refinement, 2003.

[7] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Softw.*, 11(5):89–94, 1994.

[8] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.

[9] Lodewijk Bergmans. The composition filters object model. Technical report, Dept. of Computer Science, University of Twente, 1994.

[10] Rod Burstall. Christopher strachey—understanding programming languages. *Higher Order Symbol. Comput.*, 13(1-2):51–55, 2000.

[11] Krzysztof Czarnecki, Daimler benz Ag, Ulrich W. Eisenecker, Fachbereich Informatik, and Patrick Steyaert. Beyond objects: Generative programming, 1997.

[12] Marcus Denker, Tudor Grba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with changeboxes. In Serge Demeyer and Jean-Franois Perrot, editors, *ICDL*, volume 286 of *ACM International Conference Proceeding Series*, pages 25–49. ACM, 2007.

[13] Arie Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:2002, 2002.

[14] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo 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.

[15] Don Batory Ed Jung, Chetan Kapoor. Automatic code generation for actuator interfacing from a declarative specification, 2005.

[16] Franois Fleuret, Franois Fleuret, and Projet Imedia. images, donnes, connaissances, 2003.

[17] Franois Fleuret and Isabelle Guyon. Fast binary feature selection with conditional mutual information. *Journal of Machine Learning Research*, 5:1531–1555, 2004.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[19] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[20] Guillermo Jiménez-Pérez and Don Batory. Memory simulators and software generators. In *SSR '97: Proceedings of the 1997 symposium on Software reusability*, pages 136–145, New York, NY, USA, 1997. ACM.

[21] Fred P. Brooks Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[24] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[25] Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling interactions in feature oriented software designs. In Stephan Reiff-Marganiec and Mark Ryan, editors, *FIW*, pages 178–197. IOS Press, 2005.

[26] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM.

[27] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[28] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: an agile information visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144, New York, NY, USA, 2006. ACM.

[29] Tomi Mnnist, Timo Soininen, and Reijo Sulonen. Modeling configurable products and software product families. In *in Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-2001) - Workshop on Configuration*, 2001.

[30] Sathit Nakkrasae and Peraphon Sophatsathit. A formal approach for specification and classification of software components. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 773–780, New York, NY, USA, 2002. ACM.

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

[32] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electron. Notes Theor. Comput. Sci.*, 166:93–109, 2007.

[33] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.

[34] Cincom's Smalltalk. *Tool Guide*. Cincom Systems Inc, 2007.

[35] Yannis Smaragdakis and Don 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.

[36] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[37] Tijs van der Storm. Generic feature-based composition. In Markus Lumpe and Wim Vanderperren, editors, *Proceedings of the Workshop on Software Composition (SC'07)*, volume 4829 of *LNCS*. Springer, 2007.

[38] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. pages 359–369.