Change-Oriented Software Engineering

Peter Ebraert[1], Jorge Vallejos[1], Pascal Costanza[1],
Ellen Van Paesschen[2], Theo D'Hondt[1]

[1] Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2 – B-1050 Brussel – Belgium
{pebraert,jvallejo,pascal.costanza,tjdhondt} @vub.ac.be
[2] Laboratoire d'Informatique Fondamentale de Lille – University of Lille 1
59655 Villeneuve d'Ascq – France
ellen.vanpaesschen@lifl.fr

Abstract. We propose a first-class change model for Change-Oriented Software Engineering (COSE). Based on an evolution scenario, we identify a lack of support in current Interactive Development Environments (IDEs) to apply COSE. We introduce a set of five extensions to an existing model of first-class changes and describe the desired behaviour of change-oriented IDEs to support COSE. With the help of an evolution scenario, we show why those extensions are required. Finally we describe ChEOPS: a prototypical implementation of a change-oriented IDE on top of VisualWorks and illustrate how it supports the extended first-class change model. ChEOPS is finally used to validate COSE as a solution for the shortcomings of existing IDEs.

1 Introduction

The evolution history of a software system describes a series of changes made to a software program in response to changes of requirements. The changes are integrated with the base functionality of the program as new versions are released. To keep track of the evolution of the program, developers use file-comparison and versioning tools [1]. However, none of these tools store changes explicitly and the information about changes has to be recovered by reasoning over two subsequent versions of the system. This makes the program history difficult to understand and reason about [2].

In [3], Robbes and Lanza argue that to obtain more accurate information about the evolution of a program, changes should be considered first-class entities, i.e. entities that can be referenced, queried and passed along in a program. First-class change entities, modelled as objects in Robbes's and Lanza's approach, represent the behaviour of the different kinds of changes required for a program (for example to add, remove, and modify classes). The classes of those objects represent the means to create, apply and undo the change objects. However, those change objects only represent modifications to class and method signatures which may be insufficient to reveal the programmer's intention for the changes, for example if the programmer changes the body of a method or performs higher-level changes such as refactorings.

In this work, we propose to deepen the idea of treating changes as first-class entities in the program environments. We argue the need for Change-Oriented Software Engineering in which the program history is represented as a list of first-class changes rather than just a set of resulting versions. The changes should (1) have different levels of granularity to better describe the programmers' intentions, (2) provide an expressive way of representing a program's history and (3) provide better ways of exploring the evolution history of a program. We validate our proposal by implementing the model of first-class changes and by integrating it in the VisualWorks Smalltalk Interactive Development Environment as an extension to its ChangeList tool.

This paper is structured as follows. In Section 2 the use of a first-class change model is motivated based on the shortcomings of ChangeLists change management system. To identify the shortcomings, that section uses an evolution scenario of a simple chat application. Section 3 discusses the requirements to support Change-Oriented Software Engineering. It lists five extensions to the first-class change model of ChangeList and ways in which IDE's could exploit that model to support COSE. ChEOPS, a prototypical tooi for COSE is presented in Section 4. It is used to validate how the extended first-class change model overcomes the shortcomings of the ChangeList tool. A discussion on COSE and its related work can be found respectively in Section 5 and 6. Before drawing some conclusions in Section 8 some tracks of future work are enumerated in Section 7.

2 Motivation

Most Smalltalk interactive development environments (IDEs) include a tool called *ChangeList* [4, 5] to manage the changes applied to a software system. Using this tool, programmers can inspect, compare, edit and merge changes performed on classes or methods. Each Smalltalk image[3] holds one single change list which records all the performed changes on that image. Hence the user may recover from a crash by backtracking to the most recent non-erroneous state of the image and reapplying changes listed by the ChangeList tool.

Changes are modelled as normal Smalltalk objects in the ChangeList tool and as such they can be referenced, queried and passed along. Every time a system is modified, the Smalltalk IDE logs this modification by creating a change object for it, linking this object to the project, and adding the object to the list handled by ChangeList.

Change objects of the ChangeList tool suit properly the notion of first-class changes for software evolution defined by Robes and Lanza [3]. According to them, change objects provide more accurate information about the history of a program than *file*-based and *snapshot*-based versioning change management

[3] Most Smalltalk systems represent the application code (for example classes) together with the application state (objects) in a single memory region called *image*. Images can be saved and loaded by the Smalltalk environment as a snapshot of the current code and state of a program.

systems. Change objects also better represent the incremental and entity-based way in which the evolution of a program occurs (changes are expressed in terms of system entities and not over text). However, change objects still suffer from a number of shortcomings which we illustrate in the following scenario.

2.1 Evolution Scenario

Consider the scenario of software evolution for the chat application depicted in Figure 1. This application originally consists of two classes, `User` and `Chatroom`, which respectively maintain a reference `cr` and `users` to one another. A user can subscribe to a chatroom using the `register` method and exchange messages with the rest of users of the chatroom using the `send` and `receive` methods. Messages sent to the chatroom are propagated to all the registered users.

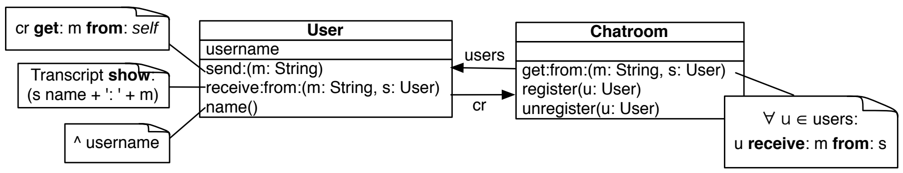


Fig. 1. Class diagram of the chat application

Assume that there are two developers working simultaneously on new features for this application. The first developer is responsible to introduce *types* of users so that it can be possible, for instance, to differentiate between registered and guest users. The second developer is responsible to ensure the privacy of the users, which in this case corresponds to encrypting and decrypting the messages when they are sent and received respectively.

For the first change the developer adds two subclasses of `User` to the application program, `RegisteredUser` and `Guest`. In this example, the only difference between these two types of users is that the registered users can be identified by their name in the chat room whereas the guests cannot: Accordingly, the `username` attribute of `User` class is moved to the `RegisteredUser` class. Figure 2 shows this first feature added to the application.

To ensure user privacy, the second developer adds two methods to the `User` class, `encrypt` and `decrypt`, which are called from the `send` and `receive` methods respectively. Figure 3 shows this second feature added to the application.

After implementing these two new features in the application, the developers merge the two changes resulting the class diagram showed below (Figure 4).

Figure 5 illustrates how these three steps are registered by the ChangeList tool in Smalltalk. Each line in this list corresponds to a change required for the implementation and merging of the two new features described above. Each of

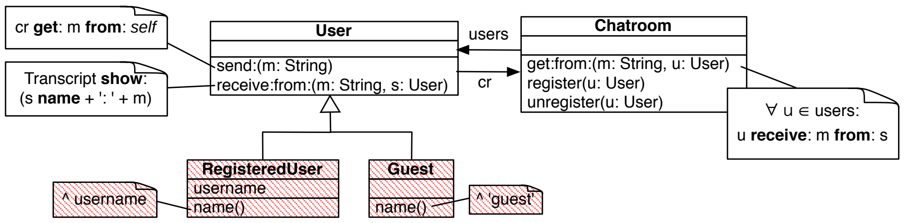


Fig. 2. First change: differentiating users

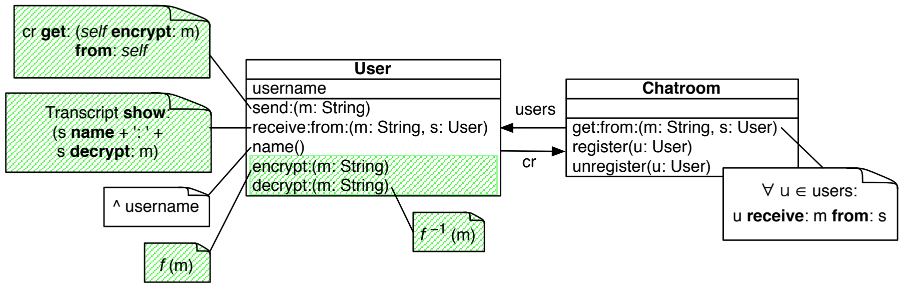

cr get: (self encrypt: m) from: self
Transcript show: (s name + ': ' + s decrypt: m)
^ username
f (m)
User
username
send:(m: String)
receive:from:(m: String, s: User)
name()
encrypt:(m: String)
decrypt:(m: String)
f^{-1} (m)
users
cr
Chatroom
get:from:(m: String, s: User)
register(u: User)
unregister(u: User)
∀ u ∈ users: u receive: m from: s

Fig. 3. Second change: ensuring user privacy

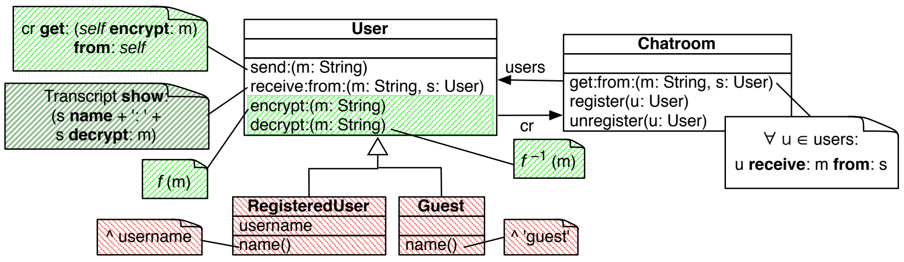


Fig. 4. Merging the changes

these changes is stored in a change object. We observe a series of problems in this representation of the evolution of the chat application:

```
 1    Created package ChatApp
 2     define User
 3     doIt User organisation addCategory:#messaging
 4    User receive:from: (change)
 5     define User
 6     define User
 7    User send: (change)
 8     doIt User organisation addCategory:#accessing
 9    User name (change)
10     define Chatroom
11     define Chatroom
12     doIt Chatroom organisation addCategory:#messaging
13    Chatroom get:from: (change)
14     doit Chatroom organisation addCataegory:#registering
15    Chatroom register: (change)
16    Chatroom unregister: (change)
17
18    define Guest
19    doIt Guest organisation addCategory:#messaging
20   Guest name (change)
21    define RegisteredUser
22   RegisteredUser name (change)
23   User name (remove)
24    define User
25    define RegisteredUser
26
27    doIt User organisation addCategory:#encryption
28   User encrypt: (change)
29   User decrypt: (change)
30   User send: (change)
31   User receive:from: (change)
```

Fig. 5. Evolution Scenario *user privacy*: Change list by ChangeList

Restricted level of granularity The entities contained in the change list are restricted to classes and methods. Additions or removals of attributes are lost and thus they can only be detected by differentiating different versions of the program. This is what happens, for instance, with the `send:` method which is first defined and then modified. As changes within methods are not logged, these two changes result in two occurrences of `User send: (change)` on lines 7 and 30.

Term overloading The definition of a change object is in some cases ad hoc and inconsistent. The same kind of Smalltalk change object can represent several kinds of modifications. For instance, a `ClassDefinitionChange` object is required to add a class to the program (for example to add the `User` class) but also to add or remove attributes (for example to add the `username` attribute in the `User` class). As a result of this term overloading, the change `define User` appears four times in the change list (lines 2, 5, 6 and 24). This hinders the understanding of the changes in the list.

Lack of high-level changes The ChangeList tool does not allow the explicit monitoring of high-level changes which better represent the intention of the developers. In this evolution scenario, the two intentions of the developers (introducing user kinds on lines 18-25 and ensuring user privacy on lines 27-31) are concatenated in the final change list. This may become a problem if the developers need to change the way in which the two features are merged, for instance, to only enable registered users benefit from encrypted communication (see Figure 6). The developers have to manually recompose their implementations from the change list.

No exploration facilities The three shortcomings described above illustrate how difficult the change management can become. After making several modifications to a program, the change list can contain large amounts of change objects. In such a case, the exploration of the change lists is cumbersome and error-prone.

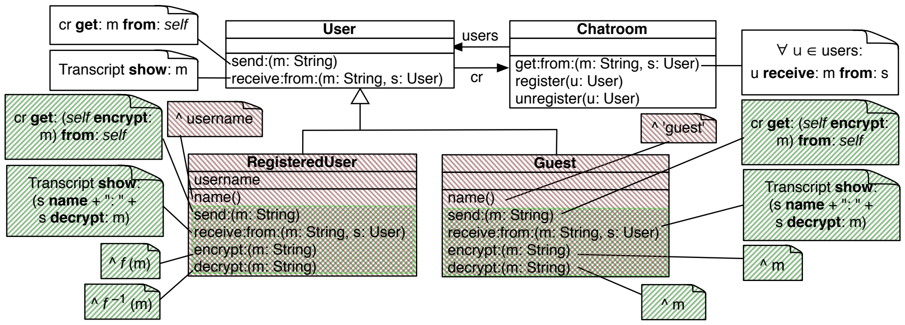


Fig. 6. New way of merging the changes

In summary, the first-class change objects featured by the ChangeList tool of Smalltalk provide information about the history of each entity of a program. However, this information has a restricted level of granularity, overloads terminology, lacks high-level changes and does not facilitate exploration. In the next section, we present a model that extends first-class changes and overcomes these issues.

3 A First-Class Change Model for Change-Oriented Software Engineering

To address the issues discussed in the previous section, we now introduce a first-class change model for software evolution. This model is an extension to the underlying change model of ChangeList.

3.1 Fine-grained First-Class Changes

In the ChangeList tool, the different kinds of changes are structured in a hierarchy of change classes. This hierarchy eases extending the set of changes, stimulates reuse and improves maintainability. Therefore we preserve the idea of structuring the types of changes in an inheritance hierarchy.

The first-class changes of the ChangeList tool express changes about packages, classes, attributes or methods. The statements of a method body are not explicitly considered as a subject of change. The dissection of a method body, however, always reveals more detailed information that can be used to study the evolution of the concerned program. The fact that a method statement includes an invocation of another method, implies that there is a relation between both methods, and that the former method could be affected when the latter is changed.

Another example of such a restriction of granularity can be found in the modification of attributes. Changes to method parameters are also not explicitly modeled by the the ChangeList tool. Assume a method call that has a complex expression as an argument. This expression can contain method invocations which reveal links between the caller and the callee.

In our model, we introduce dedicated change objects for all possible associations between program entities. An `AddInvocation` change for instance, describes the addition of an invocation of a specific method and maintains a reference to the method it invokes. Keeping this reference avoids the need to recover it later (which is even not always possible due to dynamic typing). Logging changes at this level of granularity enables the change list to contain the information which can be used to recover the invocations of a method.

3.2 Composable First-Class Changes

In order to introduce a new feature in a software system, a programmer usually needs to apply several changes. For example, ensuring privacy for users incorporates four changes (lines 28-31 in Figure 5). Every one of those four changes has the same *Raison d'Être*: ensuring user privacy. As such, they could be grouped together based on their commun Raison d'Être. We distinguish between two kinds of changes: atomic and composite changes, as depicted in Figure 7.

Atomic changes are operations that manipulate the abstract syntax tree of a particular program. These operations consist of adding or removing an entity (for instance a class) as well as changing the properties of an entity (for instance the name of a class). Each atomic change operation is applicable which makes it possible to reproduce each development stage that a program went through during its evolution. This can be done by iterating over the complete list of changes.

Composite changes consist of multiple changes (which can in turn be composite or atomic changes). They group some changes in a composition, which can be more powerful than the atomic changes on their own. A `ChangeMethod`,

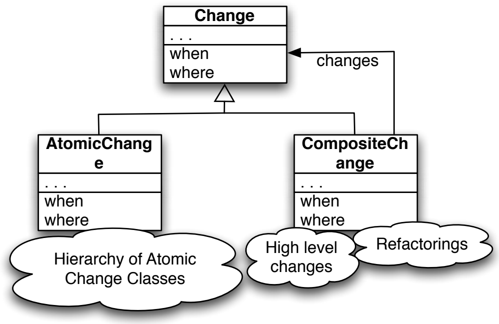


Fig. 7. Composable First-Class Changes Design

for instance, is a composition of a `RemoveMethod` and an `AddMethod`. First removing a method is not allowed, as it would violate the system invariant which states that a method can not be called if there is no definition for it. As such, all the invocations of that method would have to be removed before applying the `RemoveMethod` and re-added after applying the `AddMethod`. First adding the new method would also be prohibited as this would violate the system invariant that a class can only contain one method with a certain signature. When a `RemoveMethod` and an `AddMethod` are applied in as a `ChangeMethod` composition, the invocations of that method do not need to be altered.

Composing changes also improves the understandability of the change list. A system history consisting of only atomic operations leads to an enormous and poorly organized amount of information. Therefore atomic operations are composable: they can be grouped together into higher-level operations with a more abstract meaning.

A final application of composite changes can be found in domain-specific changes. Imagine that we envision the addition of lots of new kinds of users to our Chat application. In that case, it probably would be better to define an `AddUserClass` change, which will in turn add a subclass to the `User` hierarchy and add methods for instanciating that new class. As such, the level of abstraction is raised, improving the change list's understandability.

3.3 Dependent First-Class Changes

A change is always applied on a building block, which we refer to as the *subject* of change. *Creational changes* are changes which have as subject a new entity that they produce. In general, a change `c1` is said to *depend* on a change `c2` if that is the creational change of the subject of `c1`.

In our model, every change has a set of preconditions that should be satisfied before a change is applied. Such preconditions are related to system invariants imposed by the programming language (usually defined by the meta-model of the language), for example, methods can only be added to existing classes. Preconditions enable expressing dependency relationships between changes. In the scenario of Section 2, for instance, the change that adds the method `name` to the `RegisteredUser` class depends on the change that added the `RegisteredUser` class to the system, as the latter is the creational change for the subject of the former.

3.4 Intensional First-Class Changes

A set of changes can be specified extensionally – by listing them – or intensionally – by expressing them declaratively. Assume a change in which we rename the `name` method of the users of Figure 6 to `username`. This evolution step can be implemented by the following two algorithms 1 and 2.

Algorithm 1 Change method body: extension

```
Change the name of method name in class RegisteredUser to username
Change the name of method name in class Guest to username
Change the invocation s name in method receive:from: from User to s username
```

Algorithm 2 Change method body: intension

```
Change the name of method name in all subclasses of the class User to username
Change every invocation of method name to an invocation of method username
```

Both algorithms 1 and 2 represent the same modification to the system: a change name refactoring to method `name`. There is an important difference between both, however. Combining this change with the addition of another invocation of method `name` would result in an inconsistency when the extensional list is applied – the other invocation of `name` would not be changed). As the intensional change finds *every* invocation of `name` by definition (Algorithm 2), such conflicts are avoided. Such changes are called intensional changes. Note that we need a way for expressing qualifiers like *every* for specifying intensional changes. A logic-based declarative language is proven to be suited for such purpose [6].

3.5 Change-Oriented Interactive Development Environment

In order to support COSE, an Interactive Development Environment (IDE) is necessary which allows developers to develop (and evolve) their software by means of changes. In fact, from this point of view, developing is not different from evolving software. Concretely, a change-oriented IDE should:

– incorporate an extensible change hierarchy allowing new kinds of changes,

– support the verification of pre-conditions of changes,
– allow the specification of composed atomic changes and log their commun raison d'être,
– support the declaration of intensional changes,
– maintain the references between the program entities and their creational changes,
– include ways for developers to evolve their software by means of changes [4].

4 Validation: ChEOPS

As a proof-of concept, we have developed ChEOPS which stands for Change- & Evolution-Oriented Programming Support and is written as a plugin for the Smalltalk VisualWorks IDE. It supports COSE with a focus on object-oriented programs. As such, the subjects of the ChEOPS changes correspond to the building blocks of an object-oriented design. As a meta-model for those object-oriented designs, the *Famix* model [8] was chosen. It offers a language-independent expression of class-based object-oriented designs and splits up between arguments(for instance parameters), entities (for instance classes), associations (for instance invocations) and models (which are not taken into account by our implementation) [9]. Famix Objects have some properties which also depicted in Figure 8.

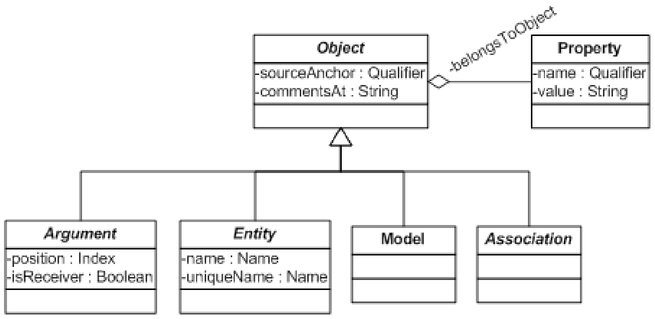


Fig. 8. Famix Objects

Change types for all Famix building blocks were defined and implemented in ChEOPS. Figure 9 highlights the part of the ChEOPS change class hierarchy which have entities as their subject. We refer to [10] for a more complete and detailed explanation of the change classes.

[4] In fact, some IDE's already provide some support to that regard. Eclipse [7] and VisualWorks [4] for instance, both provide an interactive way of adding a new class to a system. Both do this by means of graphical dialogs which request the desired

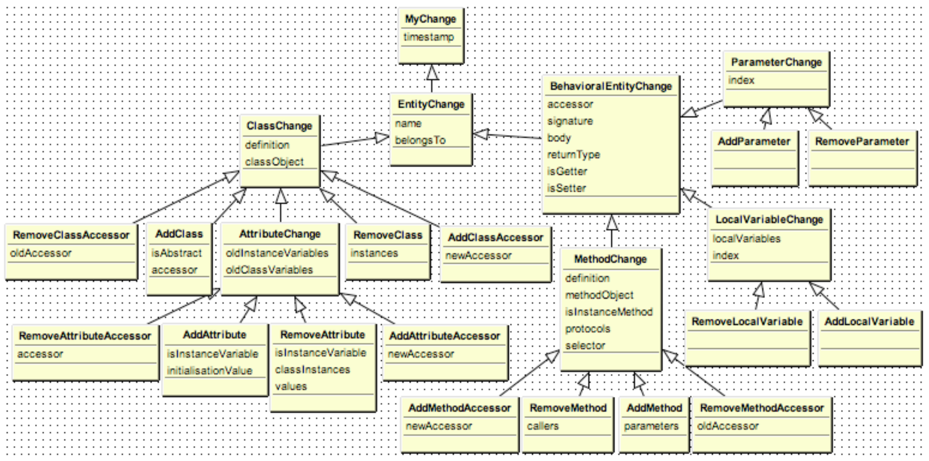


Fig. 9. Changes on the Famix metamodel

The following goes back to the evolution scenario from Section 2.1 and shows how the support provided by ChEOPS overcomes the identified four issues. Figure 10 shows the list of changes of the basic 2-class chat application, as they are logged by ChEOPS. Figures 11 and 12 respectively show the changes of adding different user kinds and ensuring user privacy in the way that ChEOPS logged them.

As opposed to Figure 5, the ChEOPS change list allows a clear separation between an addition of a new class (line 2 of Figure 10) and the addition or removal of instance variables to a class (lines 5 and 6 of Figure 10). This is a consequence of *not overloading* the `AddClass` change, but separating every change in a different kind of change class. This improves the understandability of the change list.

In the ChEOPS change list, every modification on the method level is not only represented by a statement like `User send: (change)` (lines 7, 30 of Figure 5). Instead, ChEOPS distinguishes between the addition (line 7 of Figure 10) and the removal of a method (line 10 of Figure 12). Next to that, ChEOPS does not only log changes on the method level, but also logs more *fine-grained changes* on the statement level of the method bodies. This overcomes the problem of *the restricted granularity* which was identified in Section 2.

When both extensions need to be merged in order to only allow registered users to send and receive encrypted messages, we actually want to obtain a change list like in Figure 13. This shows that just concatenating the changes of both programmers does not do the job. In fact, the four changes of the second programmer which were originally applied on the single class `User` now need

information from the developers. Support to apply other kinds of changes, like adding methods or more fine-grained changes, however, is not included in those IDE's.

```
Changes on ChatApp package:
  Add new class "User"
  Add new instance method "receive: m from: s" to class "User"
              -> Invocation tree
  Add new instance variable "cr" to class "User"
  Add new instance variable "username" to class "User"
  Add new instance method "send: m" to class "User"
             -> Invocation tree
  Add new instance method "name" to class "User"
             -> Add Read Access
  Add new class "ChatRoom"
  Add new instance variable "users" to class "ChatRoom"
  Add new instance method "get: m from: s" to class "ChatRoom"
             -> Invocation tree
  Add new instance method "register: u" to class "ChatRoom"
             -> Invocation tree
  Add new instance method "unregister: u" to class "ChatRoom"
       Add invocation "users remove: u"
```

Fig. 10. Chat Application: change list by ChEOPS

```
Changes on ChatApp package:
  Add new class "Guest"
  Add new instance method "name" to class "Guest"
            -> Invocation tree Added
  Add new class "RegisteredUser"
  Add new empty instance method "name" to class "RegisteredUser"
  Add read access to behavioral entity "name" >> return value of  variable "username"
  Remove instance method "name" from class "User"
            -> Invocation tree Removed
```

Fig. 11. Adding Different Users: change list by ChEOPS

```
Changes on ChatApp package:
  Add new instance method "encrypt: m" to class "User"
            -> Invocation tree Added
  Add new instance method "decrypt: m" to class "User"
            -> Invocation tree Added
  Remove instance method "receive: m from: s" from class "User"
            -> Invocation tree Removed
  Add new instance method "receive: m from: s" to class "User"
            -> Invocation tree Added
  Remove instance method "send: m" from class "User"
            -> Invocation tree Removed
  Add new instance method "send: m" to class "User"
            -> Invocation tree Added
```

Fig. 12. Adding user privacy: change list by ChEOPS

to be applied on both the `RegisteredUser` class and the `Guest` class. As such, we want to group these changes by their intention and parameterize them with the class on which they need to be applied. Additionally, the two classes differ in what kind of encryption and decryption functions are required. Consequently these functions have to be expressed as additional parameters to this composition of changes. The change list of Figure 14 shows the same ChEOPS change list as Figure 13, but expressed by the high-level `AddEncryptionChange`.

This does not only solve the merging problem of the evolution scenario, but also brings along improved support for reusing changes. This high-level change can now be applied on all kinds of classes that understand the `name` message and have access to an instance variable `cr` which behaves like a `Chatroom`. Next to that, it also improves the readability of the change list, as the extensive list of composite changes is abstracted away behind the high-level change.

```
Changes on ChatApp package:
  Add new instance method "encrypt: m" to class "RegisteredUser"
            -> Invocation tree Added
  Add new instance method "decrypt: m" to class "RegisteredUser"
            -> Invocation tree Added
  Remove instance method "send: m" from class "RegisteredUser"
            -> Invocation tree Removed
  Add new instance method "send: m" to class "RegisteredUser"
            -> Invocation tree Added
  Add new instance method "encrypt: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "decrypt: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "send: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "receive: m" to class "Guest"
            -> Invocation tree Added
```

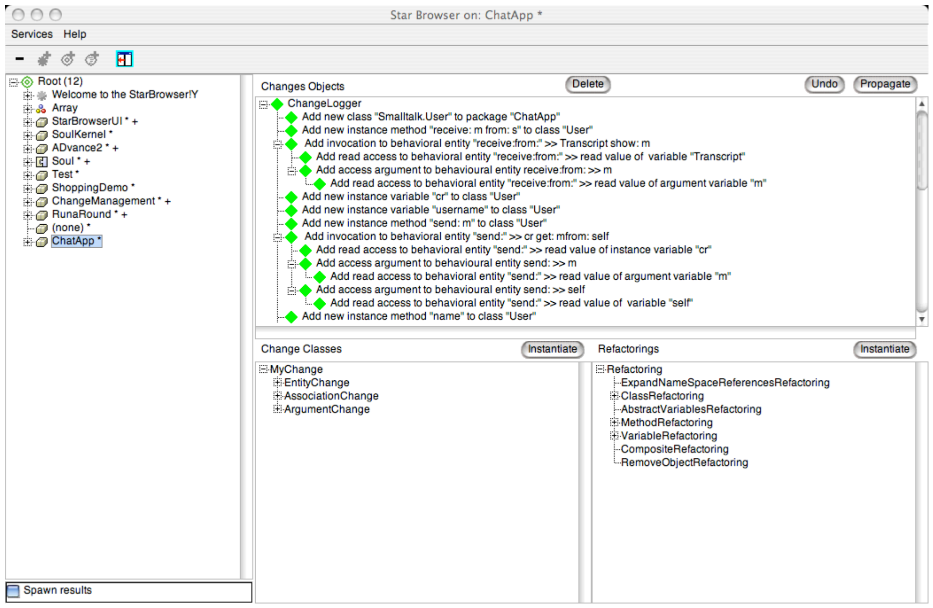
Fig. 13. Adding user privacy correctly: change list by ChEOPS

```
Changes on ChatApp package:
  Add encryption for class "RegisteredUser" with function "f(x)= encrypt(x)" and
   "f(x)= decrypt(x)"
            -> composite changes
  Add encryption for class "Guest" with function "f(x)=x" and "f(x)=x"
            -> composite changes
```

Fig. 14. Adding user privacy correctly: (compositional) change list by ChEOPS

Exploring the change list of ChEOPS is easier and more user-friendly than in the traditional ChangeList tool. This is a consequence of exploiting the *improved exploration support* brought by the first-class change model behind COSE. In ChEOPS, changes can be looked at from three perspectives: ordered on time, grouped by affected entity, grouped by intension.

– The first view (depicted in Figure 15) is similar to the traditional ChangeList approach, with the difference that ChEOPS organises the changes in a tree structure where the fine-grained changes beyond the method-level are hidden in the branches of the method-changes.



```
Root (12)
  Welcome to the StarBrowser!Y
  Array
  StarBrowserUI * +
  SoulKernel *
  ADvance2 * +
  Soul * +
  Test *
  ShoppingDemo *
  ChangeManagement * +
  RunaRound * +
  (none) *
  ChatApp *
```

```
ChangeLogger
  Add new class "Smalltalk.User" to package "ChatApp"
  Add new instance method "receive: m from: s" to class "User"
  Add invocation to behavioral entity "receive:from:" >> Transcript show: m
    Add read access to behavioral entity "receive:from:" >> read value of  variable "Transcript"
    Add access argument to behavioural entity receive:from: >> m
      Add read access to behavioral entity "receive:from:" >> read value of argument variable "m"
  Add new instance variable "cr" to class "User"
  Add new instance variable "username" to class "User"
  Add new instance method "send: m" to class "User"
  Add invocation to behavioral entity "send:" >> cr get: mfrom: self
    Add read access to behavioral entity "send:" >> read value of instance variable "cr"
    Add access argument to behavioural entity send: >> m
      Add read access to behavioral entity "send:" >> read value of argument variable "m"
    Add access argument to behavioural entity send: >> self
      Add read access to behavioral entity "send:" >> read value of  variable "self"
  Add new instance method "name" to class "User"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
```

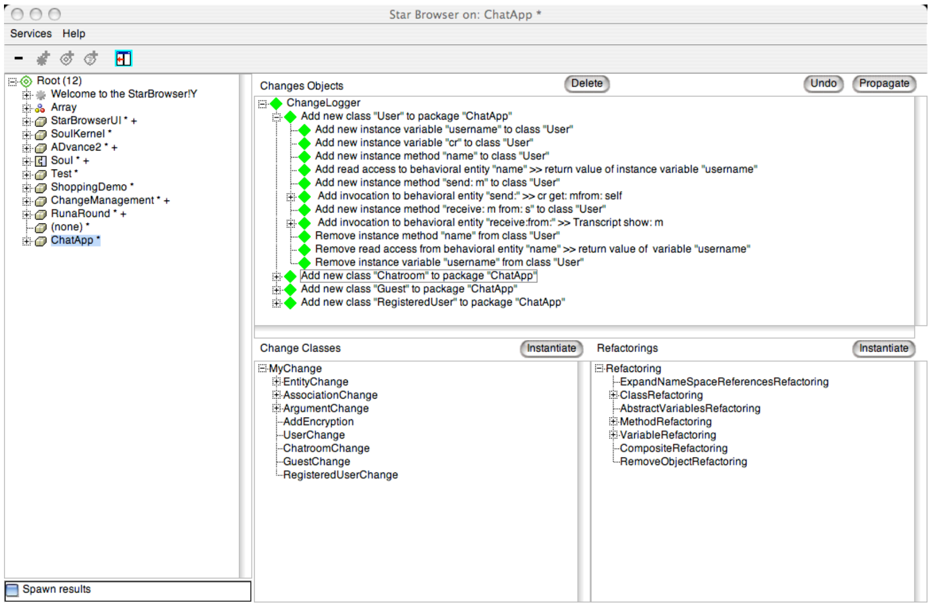
```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

Fig. 15. ChEOPS view on change list (ordered by time)

– Figure 16 shows how the second view groups the changes by the entity they affect. This entity can be any of the building blocks of the Famix meta-model: class, method, attribute, etc. This view also contains a tree of changes, where the dependent changes are hidden in the branches of the creational change of their subject. For example, the dependent changes of the creational change of `User`, are structured in a branch below that change.

– A third view which is included in ChEOPS groups the changes by their composition. Figure 17 shows the composite `AddAncryptionChange`, and the atomic changes it contains. This view dramatically decreases the number of changes to be displayed.

– Yet another view could group the changes by their commun intension. this is however, not implemented yet in ChEOPS. The nodes of this tree view should contain intensional changes, while the branches contain the corresponding extensions. Note that, when an intensional node is expanded, first it's intension is evaluated against the current application and second the corresponding extension is produced and listed in the view. As such, the



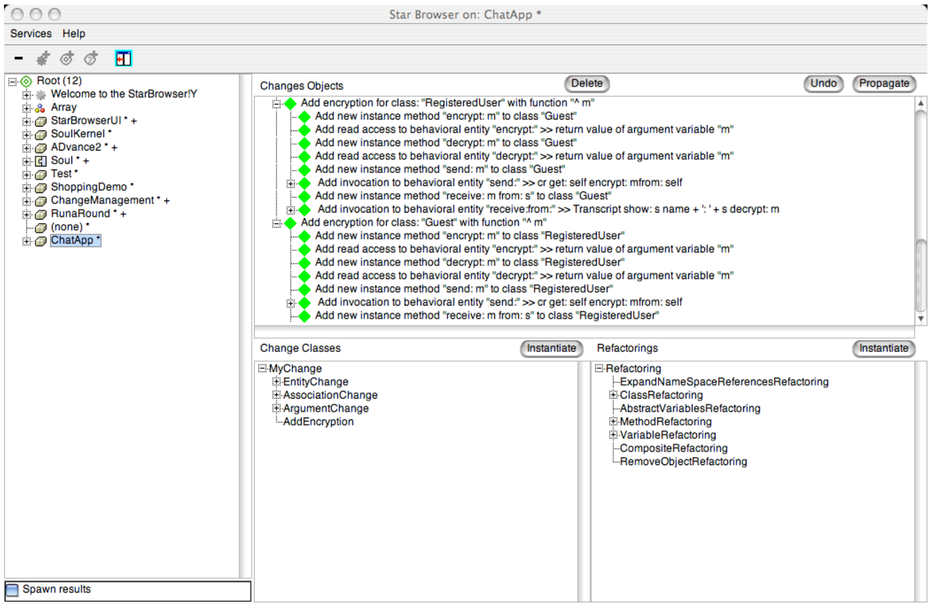
```
Root (12)
  Welcome to the StarBrowser!Y
  Array
  StarBrowserUI * +
  SoulKernel *
  ADvance2 * +
  Soul * +
  Test *
  ShoppingDemo *
  ChangeManagement * +
  RunaRound * +
  (none) *
  ChatApp *
```

```
ChangeLogger
  Add new class "User" to package "ChatApp"
    Add new instance variable "username" to class "User"
    Add new instance variable "cr" to class "User"
    Add new instance method "name" to class "User"
    Add read access to behavioral entity "name" >> return value of instance variable "username"
    Add new instance method "send: m" to class "User"
    Add invocation to behavioral entity "send:" >> cr get: mfrom: self
    Add new instance method "receive: m from: s" to class "User"
    Add invocation to behavioral entity "receive:from:" >> Transcript show: m
    Remove instance method "name" from class "User"
    Remove read access from behavioral entity "name" >> return value of  variable "username"
    Remove instance variable "username" from class "User"
  Add new class "Chatroom" to package "ChatApp"
  Add new class "Guest" to package "ChatApp"
  Add new class "RegisteredUser" to package "ChatApp"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
  AddEncryption
  UserChange
  ChatroomChange
  GuestChange
  RegisteredUserChange
```

```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

Fig. 16. ChEOPS view on change list (ordered by affected entity)



```
Root (12)
  Welcome to the StarBrowser!Y
  Array
  StarBrowserUI * +
  SoulKernel *
  ADvance2 * +
  Soul * +
  Test *
  ShoppingDemo *
  ChangeManagement * +
  RunaRound * +
  (none) *
  ChatApp *
```

```
  Add encryption for class: "RegisteredUser" with function "^ m"
    Add new instance method "encrypt: m" to class "Guest"
    Add read access to behavioral entity "encrypt:" >> return value of argument variable "m"
    Add new instance method "decrypt: m" to class "Guest"
    Add read access to behavioral entity "decrypt:" >> return value of argument variable "m"
    Add new instance method "send: m" to class "Guest"
    Add invocation to behavioral entity "send:" >> cr get: self encrypt: mfrom: self
    Add new instance method "receive: m from: s" to class "Guest"
    Add invocation to behavioral entity "receive:from:" >> Transcript show: s name + ': ' + s decrypt: m
  Add encryption for class: "Guest" with function "^ m"
    Add new instance method "encrypt: m" to class "RegisteredUser"
    Add read access to behavioral entity "encrypt:" >> return value of argument variable "m"
    Add new instance method "decrypt: m" to class "RegisteredUser"
    Add read access to behavioral entity "decrypt:" >> return value of argument variable "m"
    Add new instance method "send: m" to class "RegisteredUser"
    Add invocation to behavioral entity "send:" >> cr get: self encrypt: mfrom: self
    Add new instance method "receive: m from: s" to class "RegisteredUser"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
  AddEncryption
```

```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

Fig. 17. ChEOPS view on change list (ordered by composition)

extension corresponding to an intension depends on the other changes in the change history.

5 Discussion

Table 1 shows an overview of the four problems we identified in Section 2. The vertical axis shows the extensions that we propose to the first-class change model (Section 3). Cells containing an x denote that the extension of the cell's row helps to overcome the problem of the cell's column. The rest of this section discusses how this is achieved:

Extension	Restricted granularity	Term overloading	Lack of high-level changes	No exploration support
Fine-grained changes	x	x		
Composable changes	x		x	x
Dependent changes				x
Intensional changes			x	x
IDE support			x	x

Table 1. Problems handled by extensions to first-class change model

Restricted level of granularity The restricted level of granularity is noticeable by the lack of both very fine-grained and very coarse-grained changes. Our model includes not only coarse-grained changes (such as the addition of classes or methods), but also more *fine-grained changes* such as method invocations and accessors. The possibility to compose changes into *composite changes*, allows the definition of more coarse-grained changes, which can be used to abstract away from the fine-grained level. These extensions provide the granularity required to reason about and understand the evolution history of programs.

Term overloading Our model provides a different change for every building block of the Famix meta-model. As such, every change on such a building block is captured by a specific change. This avoids overloading change classes to capture different kinds of changes. We can conclude that in our model, the definition of a change is unique.

Lack of high-level changes Our model enables to define high-level changes that better represent the developers' intentions. The model is extensible so that developers can define their own domain-specific changes. This is achieved (1) by the use of an inheritance hierarchy in which all the change types reside and (2) by the composite design pattern (distinguishing between atomic changes and *composite changes*). High-level changes can also be defined as *intensional changes*, which describe a pattern of change. The application of high-level changes is conditioned by the fulfillment of their preconditions. This is supported by an *IDE*, which guides the programmers in defining new kinds of changes, applying their changes, undoing changes and verifying the preconditions to ensure the application consistency.

No exploration facilities All the notions of first-class change described in this work are implemented in the *ChEOPS IDE*, which supports program exploration by providing different views on the changes. The views structure the changes based on the *dependencies* between them, on their *composition*, on their *intension* or on the time on which the changes were made. Different views on changes can be used for different goals. While the dependency view, seems interesting to undo and redo changes, the intensional view seems more interesting for understanding the changes.

6 Related Work

Change-Oriented Software Engineering centralises changes. As such, managing changes is a crucial part of it. In this paper, we focussed on a change management system which is entity-based and works incrementally. This section broadens up this restriction to the entire scope of change management systems, and explains why ChangeList's model of first-class changes was chosen as a starting point of this paper. Afterwards, some change-oriented approaches are explained which are similar to the approach we identified.

6.1 Change Management Systems

Change management systems are used to capture, store and reuse changes of evolving software systems. Those changes are then managed at a central repository where each involved party has access to in order to retrieve/publish the desired changes. Hence, change management systems contain useful information about the evolution of the managed software systems (for example created classes during the lifetime of a system) and they are a valuable source to provide information about the evolution of software. Change management systems can be classified into two dimensions: time (when are changes stored) and structure (how are changes stored).

The first dimension differentiates between snapshot-based and incremental approaches. In a snapshot-based setting, developers explicitly store changes to software systems at the central repository of their change management system. Hence, each developer carries the responsibility of storing these changes on a regular basis, called commits. In an incremental setting, it is the Interactive Development Environment (IDE) which is responsible of logging the changes whenever they are performed. As such, in the latter approach, every separate change is automatically committed ensuring that changes are logged as fine-grained as possible and in the order in which they were applied.

The second dimension separates the file-based from the entity-based approaches. While the former approaches store and manage changes as differences on textual files, the latter treat changes on program entities. Program entities are the building blocks of programs. The exerted programming paradigm decides which kind of building blocks must be considered. The file-based approaches require parsing and comparing files to recover information about the changes. This makes it hard to extend the available spectrum of changes as the comparison algorithms would have to be adapted to incorporate new patterns of changes.

Table 2 shows the possible combinations of the two dimensions which were explained above, and presents at least one example per combination. Preference goes out to the incremental entity-based change management systems.

	File Based	Entity Based
Snapshot Based	CVS, SNV	CatchUp!, StORE
Incremental	AJC ActiveBackup FileHamster	ChangeList Spyware

Table 2. Change Management Systems

CVS or Concurrent Versions System is a version control system that enables the recording of the history of source-files and documents. CVS uses a client-server architecture and allows multiple connections from different locations. It is developed to organize and maintain a collection of source files which are stored at the explicit request of team members. Instead of storing each committed file separately, CVS uses an optimized technique. It stores all the committed versions of a file in one single file by only containing the textual differences between them (ΔV). For more information about CVS we refer the reader to [11].

SVN or Subversion is an advanced, open source version control system. Its main goal is to help you track the changes to directories of files under version control. Subversion also uses a client-server architecture allowing multiple connections from different locations. Developers can commit revisions anytime. Each revision has its own root which is used to access its contents. Subversion maintains for each file a reference to its most recent version. We refer interested readers to [12] for more details.

CatchUp! Henkel and Diwan propose a lightweight approach for recording and replaying refactorings [13]. Their proof-of-concept tool CatchUp! is implemented as an Eclipse plugin. After a change is committed by a user, a corresponding Java object is created and stored in an XML trace file. The captured refactorings can be replayed manually by the user or automatically by the provided CatchUp! tool that recreates the refactoring based on the trace file.

StORE is the version control system used by the VisualWorks for Smalltalk environment [4]. It is based on a client-server architecture: It uses a centralized server with a database acting as the central repository. Developers have the possibility to publish (commit) packages which will be versioned by StORE. Instead of versioning files, StORE works on a granularity-level of program entities (for instance a class or method) which facilitates for example the merging of source code of different developers.

AJC Active Backup is an automatic revision control system that continuously monitors changed files [14]. The user may configure which folders and files are monitored by specifying wildcards. Every time changes are saved to a monitored file, it is revised in a compact archive that acts as a local repository. The incremental approach implemented by this tool results in a complete record of what the user has been doing. AJC Active Backup also offers the possibility of comparing monitored files and showing the differences between them. Instead of storing each changed file separately, AJC Active Backup compresses the archived files by only storing the changes to files.

FileHamster is a version tracking application focused on meeting the needs of content creators [15]. It functions in a similar way as AJC Active Backup: It continuously monitors user-specified files and automatically creates incremental backups whenever those files are modified. FileHamster allows the annotation of changes with notes for a detailed overview or quickly locating specific revisions. It also provides the possibility of viewing the differences between two file revisions. The core of FileHamster stores each changed file separately but the tool supports a multitude of plugins for extra functionalities (for example compression).

ChangeList is tool which is included in the mainstream Smalltalk IDE's. It maintains changes applied to a Smalltalk program as first-class entities and stores change information geared towards specific Smalltalk program entities. The model of first-class changes behind ChangeList, however, has some shortcomings which show up in the tool's frontend. Section 2 elaborates on these shortcomings.

SpyWare is a change-based software repository which was proposed by Robbes and Lanza as the solution for the shortcomings introduced by using snapshot- or file-based change management systems for analyzing software evolution [4, 5]. SpyWare [3] is an IDE plug-in for the Squeak Smalltalk environment. In Spyware, a system history is viewed as the sequence of changes applied to that system. Each change is capable of reconstructing its successive state of source code, expressed by an abstract syntax tree (AST). A system is thus represented by an evolving abstract syntax tree. That abstract syntax tree is composed of program entities specific for the Smalltalk language. As such, SpyWare changes have only Smalltalk specific subjects, implying that SpyWare does not support language independent reasoning. Currently, Robbes and Lanza are working on porting SpyWare to the Java/Eclipse platform by isolating common concepts between Java and Smalltalk.

6.2 Other Related Change Tools and Approaches

There are already some approaches which partially support COSE. This section explains those approaches and relates them to COSE.

Refactoring Browser The refactoring browser [16] is a powerful Smalltalk browser which allows the programmer to perform various automated refactorings on Smalltalk source code such as renaming variables and methods. In this case the focus is on rapidly and automatically performing a set of standard refactorings without introducing errors. The changes which are applied by the Refactoring Browser, however, are not stored as composite changes, but in as an extensive list of atomic changes.

MolhadoRef This Eclipse plugin is described in [17]. It is a version control system that never loses the history of refactored elements, by tracking the evolution history at a fine level of granularity. MolhadoRef is aware of the program elements and treats refactorings as first-class changes. The difference between ChEOPS and MolhadoRef lies in the granularity of the preserved first-class changes. While MolhadoRef only treats refactorings as first-class, ChEOPS also treats the more fine-grained changes as first-class.

IDE support Many environments such as Eclipse [7] or VisualWorks [4] already initiate interactive dialogues to add a class. Some of them provide interactive support for refactorings [18, 16] To the best of our knowledge, there are no environments that provide dialogues to maintain all kinds of changes (for instance adding a method, defining a new kind of change, undoing a refactoring, etc).

Conditional transformation In [19] a new kind of refactoring tools is proposed, which allow users to create, edit and compose refactorings. The challenge in this work is the computation of the precondition of the composite refactoring from the preconditions of the composed refactorings. For this purpose, a formal model for automatic, program-independent composition of conditional program transformations, is introduced. These techniques can be used to derive the preconditions of composite changes.

7 Future work

Change-Oriented Software Engineering lifts up the level of abstraction of the development process. Instead of expressing the programs in a programming language, they are expressed in terms of changes to the building blocks of a chosen programming style. In this paper, we focussed on class-based object-oriented programming, and more specifically on the Famix model.

This work is situated in the domain of software evolution in which component-based programming is frequently asserted. We envision an implementation of COSE for a component-based meta-model, which consists of components, provided services, required services, ports to those services and connectors to connect the ports. Primary experimental results show that COSE is also applicable in a component-based context. More experiments need to be conducted in order to validate its applicability and to find out the opportunities that this brings along.

Another point of future work lies in the modification of the change semantics. Change lists contain a list of changes, which can be applied in order to produce a software system. To apply the changes of a change list, semantics need to be defined for every kind of change from that list. These semantics can be modified in order to tweak the generation of the software system. Possible applications of modification of semantics include platform-specific code-generation, automatic generation of uninstall programs and the propagation of changes to a running system.

8 Conclusion

In this paper, we envision a new programming paradigm: Change-Oriented Software Engineering (COSE). COSE is a programming paradigm which targets software evolution and which centralises changes as the main entity in the development process. The subject of the change refers to the building block of a programming language in which the program is being developed. As such, COSE builds on top of another programming paradigm in which those building blocks are written. In this paper, we build on the object-oriented programming and take the Famix meta-model as a model for describing the programs that are to be changed.

We show why first-class changes are desired to program in a change-oriented way. We then identify four problems with respect to the model of first-class changes, as it is presented in a related approach. The restricted level of granularity in the different types of changes, the overloading of change types, the lack of high-level changes and the lack of program exploration facilities hinder good software evolution support. This explains the need to extend the model of first-class changes in such a way that these problems are overcome.

Four extensions to the model of first-class changes are presented: fine-grained, composable, dependable and intensional changes. These extensions overcome the problems that are identified in the existing model of first-class changes. ChEOPS, a Smalltalk implementation of COSE is explained. It is based on the existing implementation of first-class changes, but extended with solutions for the four extra requirements that were identified. Experiments in ChEOPS made a validation of the extensions to the model possible. In ChEOPS, we have implemented an evolution scenario in which we show how COSE manages to overcome the four identified problems of the change model.

References

1. Estublier, J.: Software configuration management: a roadmap. In: ICSE - Future of Software Engineering Track. (2000) 279–289
2. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: Proceedings of Eighth International Workshop on Principles of Software Evolution, IEEE Computer Society (2005) 155–164
3. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (2007) 93–109
4. Howard, T., Goldberg, A.: VisualWorks - Application Developer's Guide. Cincom Systems (1993-2005)

5. University of Illinois at Urbana-Champaign: Visualworks: Change list tool. http://wiki.cs.uiuc.edu/VisualWorks/Change+List+Tool (2007)
6. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. In: Journal on Expert Systems with Applications. Volume 23., Elsevier Publications (2002) 405–413
7. The Eclipse Corporation: Eclipse. http://eclipse.org (2007)
8. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne (1999)
9. Demeyer, S., Ducasse, S., Tichelaar, S.: Why famix and not uml? uml shortcomings for coping with round-trip engineering. UML'99 Conference Proceedings (1999)
10. Ebraert, P., Mens, T., D'Hondt, T.: Enabling dynamic software evolution through automatic refactorings. In: Proceedings of the Workshop on Software Evolution Transformations (SET2004), Delft, Netherlands (2004)
11. Price, D.R.: Cvs - open source version control. http://www.nongnu.org/cvs/ (2006)
12. CollabNet: Subversion. http://subversion.tigris.org/ (2006)
13. Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support api evolution. In: ICSE '05: Proceedings of the 27th international conference on Software engineering. (2005) 274–283
14. AJC Software: Ajc active backup. http://www.ajcsoft.com/AJCActBk.php (2007)
15. Mogware: Filehamster - a personal revision control solution for content creators. http://www.mogware.com/FileHamster/ (2006) [Last accessed 30 May 2007].
16. Brant, J., Roberts, D.: Refactoring browser. Technical report, http://wiki.cs.uiuc.edu/RefactoringBrowser (1999)
17. Dig, D., Nguyen, T.N., Manzoor, K., Johnson, R.: Molhadoref: a refactoring-aware software configuration management tool. In: OOPSLA'06 Companion, Portland (2006)
18. Ekman, T., Asklund, U.: Refactoring-aware versioning in eclipse. Electr. Notes Theor. Comput. Sci. **107** (2004) 57–69
19. Kniesel, G., Koch, H.: Static composition of refactorings. Science of Computer Programming **52**(1-3) (2004) 9–51