Ralph-Johan Back | Johannes Eriksson | Luka Milovanov

Experience on Using Stepwise Feature Introduction in Software Construction

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 705, August 2005

Experience on Using Stepwise Feature Introduction in Software Construction

Ralph-Johan Back
Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
backrj@abo.fi
Johannes Eriksson
joheriks@abo.fi
Luka Milovanov
lmilovan@abo.fi

TUCS Technical Report
No 705, August 2005

Abstract

Stepwise Feature Introduction is an incremental method and software architecture for building object-oriented system in thin layers of functionality, and is based on the Refinement Calculus logical framework. We have evaluated this method in a series of real software projects. The method works quite well on small to medium sized software projects, and provides a nice fit with agile software processes like Extreme Programming. The evaluations also allowed us to identify a number of places where the method could be improved, most of these related to the way inheritance is used in Stepwise Feature Introduction. Three of these issues are analyzed in more detail here: diamond inheritance, complexity of layering and unit testing of layered software.

Keywords: Agile Methods, Software Architecture, Stepwise Feature Introduction, Empirical Software Engineering, Gaudi Factory

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Stepwise Feature Introduction (SFI) [2] is a bottom-up software development methodology based on incremental extension of the object-oriented system with a single new feature at a time. It proposes a layered software architecture and uses Refinement Calculus [3, 10] as the logical framework.

Software is constructed in SFI in thin layers, where each layer implements a specific feature or a set of closely related features. The bottom layer provides the most basic functionality, with each subsequent layer adding more and more functionality to the system. The layers are implemented as class hierarchies, where a new layer inherits all functionality of previous layers by sub-classing existing classes, and adds new features by overriding methods and implementing new methods. Each layer, together with its ancestors, constitutes a fully executable software system.

Layers are added as new features are needed. However, in practice we cannot build the system in this purely *incremental* way, by just adding layer after layer. Features may interact in unforeseen ways, and a new feature may not fit into the current design of the software. In such cases, one must *refactor* [15] the software so that the new feature fits better into the overall design. Large refactorings may also modify the layer structure, e.g. by changing the order of layers, splitting layers or removing layers altogether.

An important design principle of SFI is that each extension should preserve the functionality of all previous layers. This is known as *superposition refinement* [9]. A superposition refinement can add new operations and attributes to a class, and may override old operations. However, when overriding an old operation, the effect of the old operation on the old attributes has to be preserved (but new attributes can be updated freely). No operations or attributes can be removed or renamed.

Consider as an example a class that provides a simple text widget in a graphical user interface. The widget works only with simple ASCII text. A new feature that could be added as an extension to this widget could be, e.g., formatted text (boldface, italics, underlined, etc). Another possible extension could be a clipboard to support cut and paste. We could carry out both these extensions in parallel and then construct a new class that inherits from both the clipboard text widget and the styles text widget using multiple inheritance (this is called a *feature combination*), possibly overriding some of the operations to avoid undesirable feature interaction. Or, we could first implement the clipboard functionality as an extension of the simple text widget, being careful to preserve all the old features, and then introduce styles as a new layer on top of this. Alternatively, we could first add styles and then implement a clipboard on top of the styles layer. The three approaches are illustrated in Figure 1.

A component is divided into layers in SFI. Layers will often cut across components, so that the same layering structure is imposed on a number of related

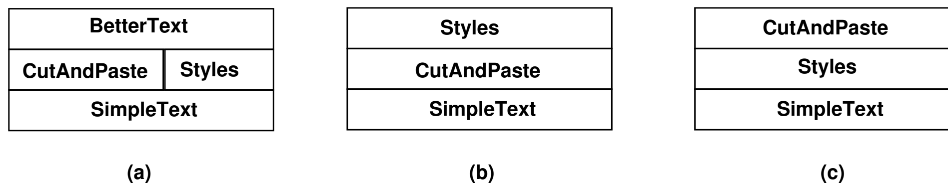


Figure 1: Alternative extension orders

components. As an example, consider building an editor that displays the text widget. In the first layer we have a simple editor that only displays the simple ASCII text widget. Because of the superposition property of extension this simple editor can in fact also use the `CutAndPaste`, `Styles` or `BetterText` widgets, but it cannot make use of the new features. We need to add some features to the simple editor so that the functionality of the extended widget can be accessed (menu items for cut and paste, or for formatting, or toolbar buttons for the same purpose). We do this by constructing a new an extension of the simple editor (a better editor), which uses the `BetterText` widget and gives the user access to the new functionality. The situation is illustrated in Figure 2.

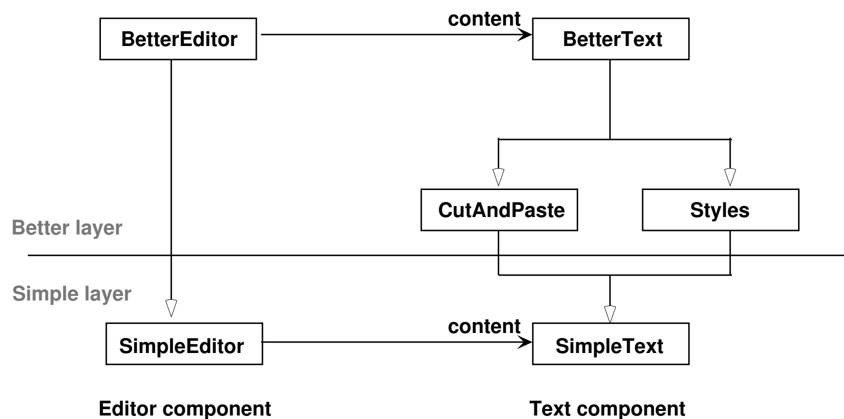


Figure 2: Interacting components

The new editor is, however, restricted to only work on the better text widget, because the features it assumes are only available on this level. Hence, there are two layers in the design, the Simple layer and the Better layer.

Stepwise Feature Introduction has been tried out in a number of real software projects. This allows us now to evaluate the merits of this approach and to spot possible drawbacks as well as opportunities for improvement. Our purpose in this paper is to report on these case studies, and to provide a first evaluation of the approach, together with some suggestions on how to improve the method.

The paper is structured as following: Section 2 present the software projects

where SFI was applied. We summarize our experience with the methodology in Section 3. In Sections 4-6 we then consider in more detail three interesting issues that arose from our experiments with Stepwise Feature Introduction. The problems with implementing feature combinations using multiple inheritance is discussed in Section 4. The problem of class proliferation is discussed in Section 5, where metaprogramming is considered as one possible way of avoiding unneccessary classes. In Section 6, we show how to adapt unit testing to also test for correct superposition refinement. We end with a short summary and some discussion on on-going and future work.

2 SFI Projects in Gaudi

The software projects where SFI was evauated were all carried out in the Gaudi Software Factory at Åbo Akademi University. The Gaudi Software Factory is an academic environment for building software for the research needs and for carrying out practical experiments in Software Engineering [6]. Our research group defines the setting, goals and methods to be used in the Factory, but actual construction of the software is done in the factory, following a well-defined software process. The work is closely monitored, and provides a lot of data and measures by wich the software process and its results can be evaluated. The software process used in Gaudi is based on agile methods, primarily *Extreme Programming* [11], together with our own extensions.

We will here describe four software projects where Stepwise Feature Introduction was used throughout. The settings for all these projects were similar: the software had to be built with a tight schedule, and the Gaudi software process had to be followed. The programmers employed for these projects (4-6 persons) were third-fifth year students majoring in Computer Science or related areas. Each project had a customer who had final saying on the functionality to be implemented. The projects were also supervised by a coach (a Ph.D. student specializing in Software Engineering), whose main task was to guide the use of the software process and to control that the process was being followed. There has also been one industrial software project [1] with SFI, but this is outside the scope of this paper, as it was not carried out in the Gaudi Factory, and the software process used was not monitored in a sufficiently systematic manner.

All of the projects used SFI, but the ways in which the method was applied differed from project to project. We describe the projects in chronological order below. For each project, we present the goals: both for the software product that was to be built, and for the way in which SFI was to be evaluated in this project. We give a general overview of the software architecture, show how SFI was implemented, what went right and what went wrong, and discuss the lessons learned from the project.

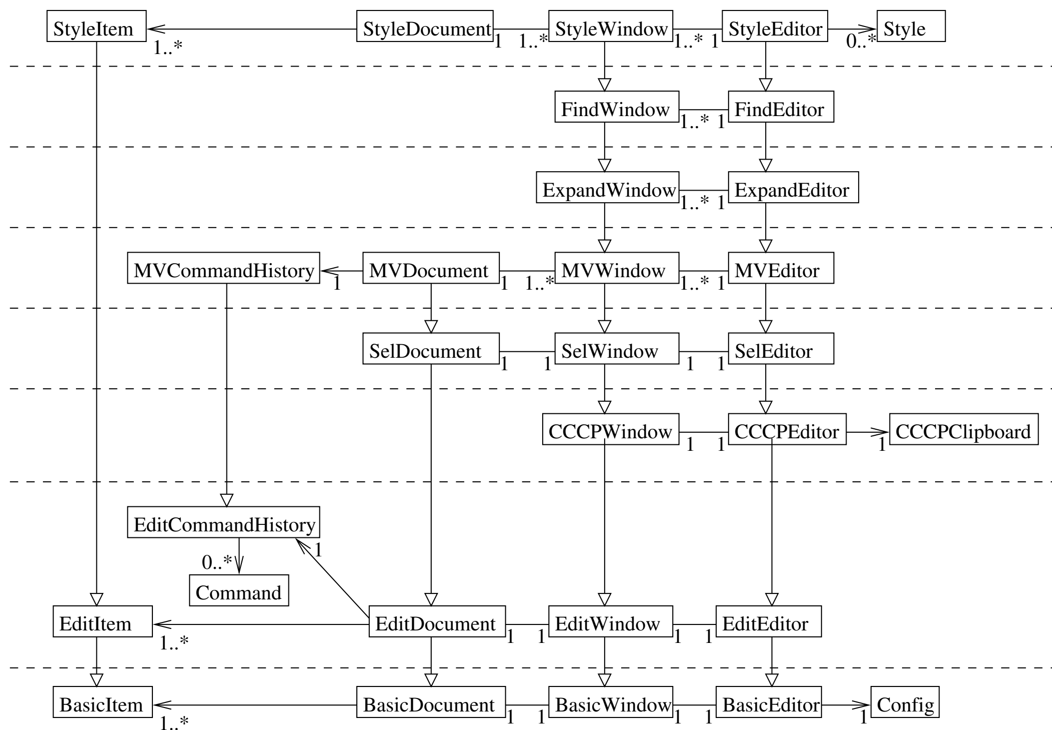


Figure 3: The layers of the editor

2.1 Extreme Editor

The Extreme Editor project [7] was the first application of SFI in practice. It ran for three months during the summer 2001 and involved six programmers. The programming language of the project was Python [18]. The software product to be built was an outlining editor which became a predecessor for the Derivation Editor described in Section 2.2. The goal for the project was to obtain the first experience from practical application of SFI with a dynamically typed programming language. There were no technical guidelines for the application of SFI except that the extension mechanism for classes (the feature introduction—Section 1) should be inheritance.

Figure 3 shows the layered architecture of the Extreme Editor. There were eight layers in the system:

Layer 1. (Basic): Only one text file (document) can be opened per session. Documents are interpreted as outlined texts. Implemented functionality for *open, save, save-as* operations. The opened document, however, cannot be edited yet. The editor can also create a configuration file, write a set of configuration options in it and read them later.

Layer 2. (Edit): The basic editing operations such as type and delete characters, insert and remove items, change the indentation of items are implemented.

Command history which allows *undo* and *redo* operations is also implemented. New document can now be created.

Layer 3. (Clipboard): Implemented *copy, cut* and *paste* operations. However they can be applied only to the whole document because the selection is not implemented yet. Some utility functions are bound to hot keys.

Layer 4. (Selection): Implemented the smart selection operation. The user can select characters of a string or several items. The operations from the previous layer as well as changing the indentation now work under selection.

Layer 5. (Multiple Windows): Multiple window interface based on the Model-View-Controller [16] pattern is implemented. The user can have many different documents opened in one session. The user can also have several views of the same document. The document can be edited in any of its views. The content of all views is synchronized.

Layer 6. (Expand and Collapse): Composite items of outlined texts can now be collapsed and expanded. The expansion operation has three models: ”expand all”, ”expand one level” and ”remember expansion”.

Layer 7. (Find and Replace): The standard operations for finding and replacing regular expressions in documents are implemented.

Layer 8. (Styles): The concept of styles is introduced. Two styles can now be applied to the items of a document: *plane* and *title*.

Each layer introduced new functionality into the system, without breaking the old features. The software was structured into these layers in an ad hoc way. A new layer extended its predecessor by inheriting its corresponding classes and possibly introducing one or more new classes. There were no physical division of the software into the layers on the level of the file system: each class name had a prefix—the name of the layer where the class belongs to. More on the architecture of the software and detailed description of the layers can be found in the technical report [7].

The feedback on SFI from this project was rather positive. The method supported building software with a layered architecture quite well. The developers found it rather easy to add new features as new layers. The fact that functionality of the system was divided into layers made the overall structure of the system clearer to all the programmers. Another advantage was ”bug identification”: the layered structure made it easy to find the layer in which the bug occured and its location in the source code.

On the other hand, even in the three-month project, as more features were implementer and the more classes were introduced into the system, it was getting harder to navigate among them without any tool support, any automatic documentation describing the layer structure and without a systematic naming convention

for classes, layers and methods. We also found that SFI requires a special way of unit testing (the testing classes should be extended in the same way as the ordinary classes of the system). More on unit testing will be discussed in Section 6.

2.2 Editor for Mathematical Derivations

MathEdit [14] was an effort to implement tool support for structured derivations and is currently the largest project developed using the SFI methodology. MathEdit was developed in the Gaudi Software Factory as two successive summer projects, in 2002 and 2003. One of the objectives of the first summer project was to try out feature combination by multiple inheritance. The second objective of the project was to assess how well a new team could embrace an existing SFI codebase and continue its development. The continuation project in 2003 shared only one out of four developers with the 2002 project. Still, it turned out that the new programmers were able to start working productively with the existing code base within the first three weeks.

MathEdit consists of totally 16 layers. Each layer is named after the key feature implemented. All layers, each accompanied by a short description of its major features, are ordered from the most basic (lowest) to the most advanced (highest) in the following list.

Base: Basic GUI: main window, menus and toolbars.

Frame: Multiple empty MDI (Multiple Document Interface) child windows.

Text: Plain-text editing using a custom editor widget, Open, Save, Save As, Save All. Document model.

MV: Multiple views of the same document.

Edit: Clipboard interaction, find and replace, and spell checking.

CmdList: Support for undo/redo of editing commands. Extends the document model with a command list.

Outline1: Extends the document model and editing capabilities of the text widget to support an outlining structure.

Outline2: Support for folding, i.e., collapsing and expanding an item. Collapsing hides all lines indented deeper than the current line, expanding shows them again.

Format: Rich text formatting of text, and hyperlinks.

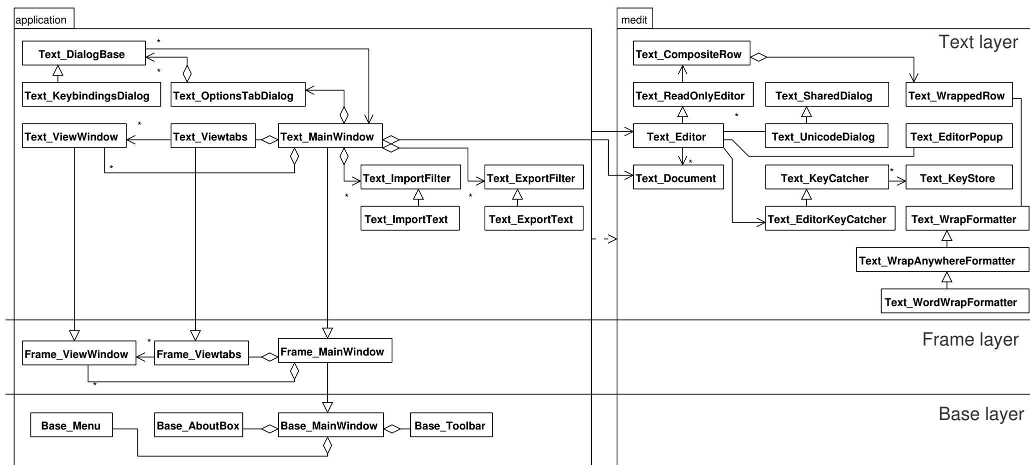


Figure 4: The first three layers of MathEdit, unit tests excluded

Derivation: Formula syntax. Possibility to define mathematical expressions, which the editor parses according to the grammar of the (default) mathematical profile. Structured derivations can be constructed by using ruiles to rewrite expressions.

Profile: Support for switching profiles, and adding custom profiles.

Filter: Export documents to HTML files with Javascript support for folding. Export to LaTeX.

Highlight: Syntax highlighting for programming languages.

Program: Program syntax, programming with nested states.

Picture: Possibility to add pictures to documents.

Hook: Scripting interface, the user can connect custom functions to hooks that are executed for specific events in MathEdit.

We chose to associate classes with layers using naming conventions instead of by grouping them into Python modules, mainly because we wanted layering and modularization to be orthogonal partitionings of the software. Classes are named according to the template `Layer_ClassName`, where `Layer` is the name of the layer `ClassName` the name of the class. A diagram showing the classes of the first three layers (unit tests excluded) can be seen in figure 4. Due to space considerations, we cannot display a complete class diagram for MathEdit.

On the highest level, MathEdit is separated into three major components: a document-view component (`medit`), the application-level UI (`application`) and

a mathematical profile plug-in. The `medit` and `application` components are layered as described above. The layering cross-cuts these top-level components, so the layering is *global*. The profile component was designed as a plug-in, so it was not layered—users should be able to write custom profiles without having to care about the internal layer structure of MathEdit.

Combining two layers using multiple inheritance was attempted but ultimately abandoned in the MathEdit project. The main reason were practical problems arising from the use of multiple inheritance; e.g., classes in the graphical toolkit used (Qt) did not support multiple inheritance well. Also, the development team was quite small, so it did not seem fruitful to work on two features in parallel as if they were independent, when it was already known that the features would be combined later on. Instead, a feature was implemented with extensibility in mind, so that it was easy to add the next feature in a new layer. The gains of parallel development would probably be much higher in a larger project, but this still remains to be evaluated.

The MathEdit application is executable with any layer as the top layer, providing the functionality of this layer and all layers below. The gives us the possibilities to fall back to an earlier working version in case of malfunction, and to locate bugs that were introduced in a some unknown layer. We simply implement a test that exposes the bug and run it with different top layers. The lowest layer that exhibits the erroneous behavior is either harboring the bug, or triggering a bug in a previous layer.

2.3 Software Construction Workbench

The Software Construction Workbench (SCW) project (summer – fall 2002) was an effort to build a prototype for IDE supporting Stepwise Feature Introduction and Python. This application was built in Python, as an extension of the System Modeling Workbench [4]. The main feature are modeling software systems in a SFI fashion with UML, automatic Python code generation and execution of the constructed system and support for unit tests in the environment. SCW also included basic support for Design by Contract [20] and reverse engineering.

As far as SFI is concerned, the goals in this project were to get further feedback on the practical application of the methods, to try out the layered approach to unit testing (discussed in Section 6) and to try out new naming conventions (described below in this section). A special feature of this project was that it used its own medicine: the software needed to support software development with SFI was built itself using this method. Since the architecture of the SCW is rather complex, we do not present it here, for the readers' convenience. Instead, we illustrate how the SFI method was applied in this project on a small and simplified fragment of the software.

Figure 5 shows an example of the layer structure implementation. SFI layers were implemented using Python's packaging mechanism: each SFI layer corre-

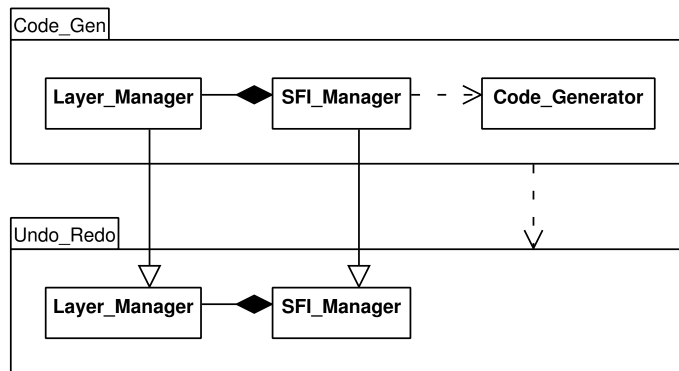


Figure 5: Layer structure, Python implementation

sponds to a Python package. To show that a layer is a successor of another layer we draw a dependency from the successor to its ancestor; in practice this dependency was the Python `import` statement. The mechanism for extension of classes was inheritance, as shown in Figure 5. Every class, once introduced, keeps its original name through all of its extensions. This was simple to implement: Python packages provide a namespace so we had no conflicts with the names of the classes. The diagram in Figure 5 will correspond to the following implementation:

```
Undo_Redo/__init__.py
Code_Gen/__init__.py
```

Files `Undo_Redo/__init__.py` and `Code_Gen/__init__.py` will have the definitions of classes `Layer_Manager` and `SFI_Manager`, or they will import their definitions from somewhere else, i.e. from the corresponding files in the following directories:

```
Undo_Redo/   Layer_Manager.py   SFI_Manager.py
Code_Gen/   Layer_Manager.py   SFI_Manager.py   Code_Generator.py
```

When a successor layer is created, all elements should be imported into it for their usage with the statement **`from`** `Undo_Redo` **`import`** `*`. For the extensions of the elements, the ancestor layer should be imported into its successor with the statement **`import`** `Undo_Redo`.

In our example the directories `Undo_Redo` and `Code_Gen` should be located in the directory `SCW` which should also contain the init-file with the following content:

```
# system SCW
Layers = ['Undo_Redo', 'Code_Gen']   #all system's layers
TopLayer = Layers[-1]                #top layer: the latest
                                     #system's extension
```

```
exec('from %s import *' % TopLayer) #import all elements
                                    #from the top layer
```

All init-file in the layer implementation should start with two import statements. The first import imports classes from ancestor for further use and the second imports the ancestor itself for making the extensions of the classes. In our example (figure 5), file `Code_Gen/__init__.py` will starts as:

```
from Undo_Redo import *
import Undo_Redo
```

A class can be extended with new methods and/or some inherited methods can be overwritten. When an old method is overwritten in the next class extension, it is a good practice to have a call to the original method inside the body of the extended method. Suppose class `Undo_Redo.Layer_Manager` has a method `f()`, and we want to improve it in its extension class `Code_Gen.Layer_Manager`, and we also want to add a new method `g()`. This should be done in the following way:

```
class Layer_Manager(Undo_Redo.Layer_Manager):
  def f(self, x):
    ... # extension code
    r = Undo_Redo.Layer_Manager.f(self)+1 #old code reuse
    ... # more extension code
    return r
  def g(self):
    .. # do something
```

Note that the redefined method *f* contain the call to its ancestor, this is the way to reused old code in SFI, and it should be always when possible used when extending a method. The extended method `Code_Gen.Layer_Manager.f(x)` has one parameter and return a tuple unlike its ancestor `Undo_Redo.Layer_Manager.f()`. Due to the highly dynamic nature of Python it is allowed to change number and types of methods parameter and return values.

According to the developers, the implemented layered structure of the software together with the name conventions really clarified the software. The Software Construction Workbench project was carried out as two subprojects, such that half of the developers were new in the second subproject, and in the beginning had no understanding of the software at all. Nevertheless, the new programmers were able to take over the code easily because of the division of features into layers. The new programmers also commented that the layering made it much easier to navigate in the code, modify code and search for bugs.

The SCW project showed that in order to use the SFI methods properly and efficiently, tool support is really needed. Because of the way Python packaging was used to implement the SFI layers, it took a lot of time to divide the code into directories corresponding to the layers. Building software according to SFI also promotes refactoring (a practice enforced in our Software Factory). For example, when changing something in the lower layer, it can often affect the successive

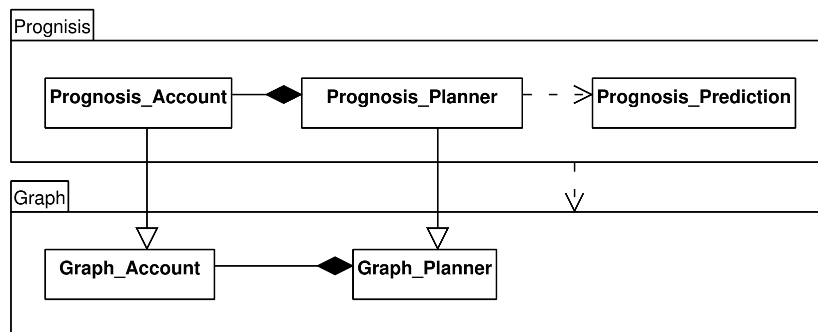


Figure 6: Layer structure, Eiffel implementation

layers, so they should be slightly changed. According to the developers a simple tool helping with the navigation among the layer structure, i.e. from ancestor to successor and the other way around, would here save a lot of time.

2.4 Personal Financial Planner

The Personal Financial Planner project [5] (FiPla) was the first application of SFI with a statically typed language—Eiffel [19]. The software goal for the project was to build a personal financial planner. The features required of this product type include tracking of actual events (manually or automatically), planning (such as budgeting and creating scenarios), and showing future scenarios.

SFI was evaluated in this project to see how the method would work with a statically typed language. Another goal was to see how well the SFI layers layers correspond to the short release cycles used in the Gaudi Software Factory [6], so that each short iteration starts a new layer. Finally we also wanted to see how well SFI and Design by Contract [20] fit together.

SFI layers in Eiffel are implemented using Eiffel clusters. However, unlike Python packages, a cluster in Eiffel does not provide a namespace for the classes. It means that all names of the classes in the Eiffel software system should have unique name, so it was impossible to have the same naming convention as in the SCW project. For this purpose we used another naming convention, where each class name should have the name of the layer that the class belongs to as a prefix. Figure 6 shows a simplified fragment of the software architecture which corresponds to the following implementation:

```
system "fipla"
root   application: make
cluster
 fipla:          "$"
 library base:    "$ISE_EIFFEL/library/base"
```

```
 graph     (fipla): "$/graph"
 prognosis (fipla): "$/prognosis"
end
```

—the LACE (Language for the Assembly of Classes in Eiffel) file `fipla.ace`. The cluster directories `graph` and `prognosis` correspond to the SFI layers, and files:

```
graph_account.e graph_planner.e
prognosis_account.e prognosis_planner.e prognosis_prediction.e
```

contain the definitions of the corresponding classes shown on the diagram. There is no need for the import statement in Eiffel since all classes from the system's clusters are already accessible. However, unlike in Python, the names of classes change in each layer, see figure 6.

The example with the extension of method f() and addition of new method g() from the previous Section 2.3 in Eiffel will be implemented as the following:

```
class PROGNOSIS_ACCOUNT inherit
  GRAPH_ACCOUNT
    redefine f end
create  make
feature -- Extended routines
  f is
    do
      ... -- extension code
      result := precursor + 1 -- old code reuse
      ... -- more extension code
    end
feature -- New routines
  g is
    do
      ...
    end
end -- class PROGNOSIS_ACCOUNT
```

Class `PROGNOSIS_PLANNER` is the extension of the class `GRAPH_PLANNER`, and the extended class is assosiated with class `PROGNOSIS_ACCOUNT` (extension of `GRAPH_ACCOUNT`). This is implemented in the following way:

```
class GRAPH_PLANNER
feature -- suppliers
        account : GRAPH_ACCOUNT
...
class PROGNOSIS_PLANNER inherit
        GRAPH_PLANNER
        redefine account end
feature -- redefined suppliers
        account : PROGNOSIS_ACCOUNT
```

	EE	MathEdit	SCW	FiPla
LOC	3300	44372	16334	8572
Test LOC	1360	4198	14565	2548
Total LOC	4660	48570	30899	11120
Classes	52	427	66	59
Test classes	23	53	42	25
Methods	344	3790	610	331
Test methods	85	279	355	177
LOC/class	63	104	247	145
LOC/test class	59	79	347	102
Methods/class	6.6	8.9	9.2	6.0
Test methods/class	3.7	5.3	8.5	7.0

Table 1: Basic product metrics for SFI projects

SFI worked well with Eiffel when we applied the methods in the same way as in our Python projects. Structuring the software system into layers according to the small releases defined by the customer turned out to be a good idea. However, a few important technical issues that needed improvement were found. These issues only came up when using SFI with a statically typed language.

The extension of classes using pure inheritance did not work well with Eiffel. The types of the parameters and return value of redefined routines should be at least of conforming types. Eiffel does not support parametrized polymorphism, hence, the number of parameters should be constant in all extensions of a routine. It is possible to overcome these limitations using routine renaming or rewriting a routine completely previously undefining it with Eiffel's *undefine* statement. The last case is, however, not recommended since it will not be a real extension of the routine.

To avoid these problems one must pay more attention to the overall system architecture and plan a bit further ahead then just for the next iteration. Extensive refactoring was needed in some cases, when the planning had not been done carefully enough. To refactor a SFI system efficiently, tool support was again deemed necessary.

3 Experience of using SFI

In this section we summarize our experience from using SFI in the software projects mentioned above, based on the quantitative and qualitative data that was collected during these projects. Table 1 shows some basic code metrics for the presented SFI projects. It is easy to see that even in small projects like Extreme Editor and FiPla, the number of classes is growing fast. On the one hand this helps

with debugging: as the developers were often commenting, it is easy to find the source of a bug in the code because of the separation of functionality into the layers. On the other hand, managing a large number of classes manually eventually becomes quite complicated, suggesting that tool support for navigation among successive layers, classes and methods is needed.

SFI is a bottom-up approach for constructing software systems and is therefore not that well suited for developing graphical user interfaces. Constructing good GUIs is a complicated task in itself and needs to combine different approaches such as bottom-up, top-down, use of state charts and designer tools. Our experience showed that when using SFI, it is better to separate GUI development from the construction of the core of the system.

Stepwise Feature Introduction fits well in our software process and in general in the Extreme Programming philosophy of introducing small changes one at a time in a software project [6, 8]. The division of the system into layers can be driven by the XP iteration planning process. When the development team negotiates with the customer about what new features should be implemented for the next small release, it is then easy to see what should be included in the next layer. Every layer in SFI system together with its ancestor layers represent a functional, working system. Hence, each layer corresponds to a small release, making it easy to package a specific release so that the customer can do the acceptance testing.

We obtained good results regarding the practical usability of SFI from our projects. SFI has a formal basis and provides a sound way of structuring software, and SFI designs often capture the core concerns of the software (the features) more explicitly than many traditional OO designs. We have also identified some shortcomings in the method that we need to work on. The use of inheritance as an extension mechanism can be cumbersome and does introduce some complexity of its own into the system. SFI occasionally makes it difficult to add a feature that does not fit well into the layer hierarchy. In order to make SFI practically usable, it will be necessary to devise another extension mechanism or introduce SFI-aware development tools (such a tool is currently being worked on, as explained in Section 7).

In the remaining sections, we will discuss in more depth three specific issues that came up during our experiments, and which we think merit a much closer analyzis.

4 Feature Combination and Diamond Inheritance

SFI suggests combining two or more independently developed layers into a feature combination layer (see Section 1). As each layer may contain an extension of the same class, the feature combination layer combines the extensions of a class from each layer into a new subclass using multiple inheritance. It is then the responsibility of the new layer to synchronize the two independent features in a

meaningful way.

Multiple inheritance is significantly more complex than single inheritance for both language implementors and programmers. What constitutes correct use of multiple inheritance in object-oriented software is a subject of some controversy. Bir Singh [23] lists the four main uses of multiple inheritance, none of which correspond to the way it is used in SFI (combination of two implementations with a potentially large number of common methods):

- combination of multiple independent protocols;
- mix and match, where two or more classes are designed specially for combination;
- submodularity, to factor out similar subparts for improved system design;
- separation of interface and implementation;

This may suggest that multiple inheritance as implemented in most programming languages might not be ideal for feature combination as originally proposed in SFI. We will here focus on one serious design problem encountered numerous times in our experiments, namely *diamond inheritance*.

Diamond inheritance occurs when two or more ancestors of a class share the same base class. This situation arises fairly often in large systems, especially if the class hierarchy has a common root. In SFI diamond inheritance is likely to occur because of the way inheritance is used as a layer extension mechanism, especially if one uses the suggested feature combination.

An example of a situation in which the diamond pattern typically occurs in an SFI design can be seen in Figure 7. The Basic layer contains two classes, `BasicAccount` and a derived class `BasicCheckingAccount`, the latter supposedly having some extended behavior such as allowing withdrawals greater than the balance. In this case the programmer used inheritance to be able to handle objects of the two account types uniformly, i.e. to achieve polymorphism. Let us now assume that in the next layer (the Better layer), support for multiple currencies is added, and that this feature requires both `BasicAccount` and `BasicCheckingAccount` to be extended into `BetterAccount` and `BetterCheckingAccount` respectively. However, to preserve polymorphism, `BetterAccount` must also be extended to `BetterCheckingAccount`. We notice that mixing the two usage patterns of inheritance results in a diamond structure in the design.

Diamond inheritance causes difficulty when the same method is implemented in more than one base class. In this example the `withdraw` method of `BetterCheckingAccount` depends on functionality implemented in both `BasicCheckingAccount` and `BetterAccount`, and calls both (in some order). However, each of these calls trigger a call to `BasicAccount.withdraw`, resulting in two calls to this method. The code in `BasicAccount.withdraw` is thus

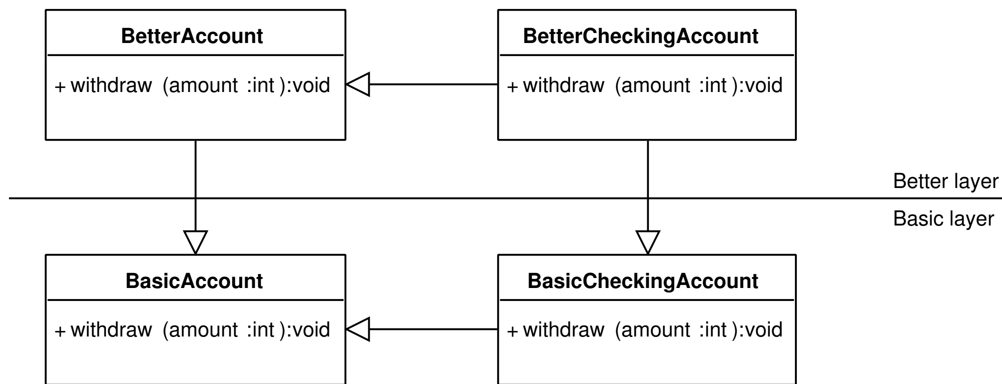


Figure 7: Diamond inheritance

executed twice, which is not the intended behavior (the sum is withdrawn twice from the account). The same situation occurs commonly with constructors—the constructor of the common base class in the diamond is called twice. This results in data structures and resources being initialized twice, potentially leading to resource leaks.

When implementing `BetterCheckingAccount.withdraw` in the example case, we want to call the `withdraw` method of both base classes: `BetterAccount` and `BasicCheckingAccount`. However, `BasicAccount.withdraw` must be called only once. In Python 2.3, which was actually designed with inheritance diamonds in mind [22], this is possible using the built-in `super` function which creates a linearization of the class hierarchy and returns for a given class the previous class in the linearization. By replacing direct calls to `__init__` with `super` the desired call order can be achieved. The drawback is that since we are no longer explicitly calling the base class method, we might not be sure which method actually gets called without considering the linearization of the whole class hierarchy. Because this would make the class design more complex we have not used `super` in any of our Python projects.

Most SFI projects developed in the Gaudi Software Factory have avoided diamond inheritance by not using multiple inheritance for feature combination or for mixing polymorphic extension with stepwise extension. Consequently we have not been able to add features to a base class in a polymorphic inheritance hierarchy using SFI layers. However, many times features are better implemented using *delegation*, where an object uses another object (the delegatee) to perform an operation. In this case we can simply replace the delegatee with a more advanced version in a higher layer. E.g., MathEdit heavily uses the *Bridge* and *Decorator* design patterns [16] to avoid inheritance diamonds.

An alternative design to Figure 7 using delegation can be seen in Figure 8. In this scenario there is no `BetterAccount` class, instead `BasicAccount` objects are parameterized in their implementation; calls to `BasicAccount.withdraw` are

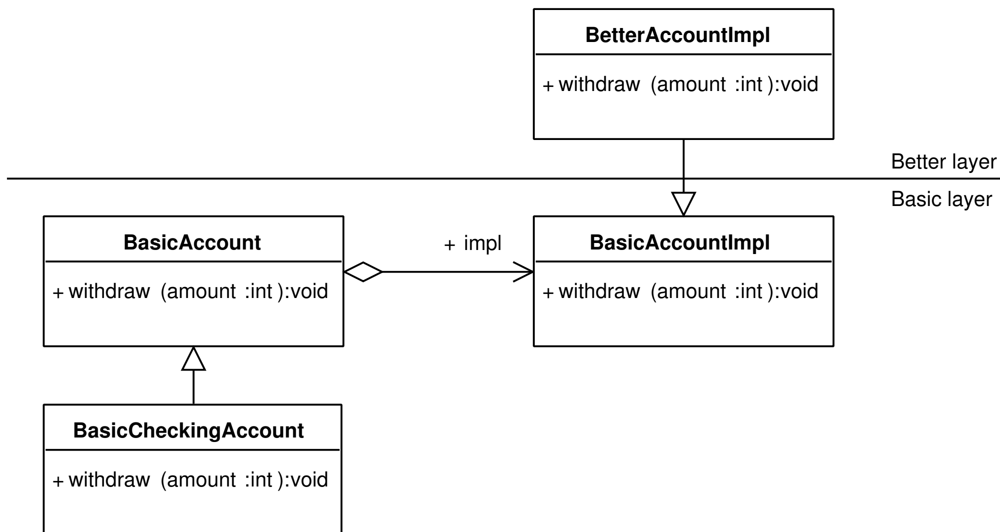


Figure 8: Adding features using delegation

delegated to `impl.withdraw`. The code instantiating `BasicAccount` objects is assumed to make sure `impl` is a `BasicAccountImpl` instance the Basic layer and a `BetterAccountImpl` instance in the Better layer.

5 Avoiding Trivial Classes with Metaprogramming

One problem discovered early on was that some SFI-supporting metaprogramming framework had to be implemented to reduce complexity that was primarily caused by *proliferating factory methods*. This problem occurs whenever one class (the factory) is responsible for creating instances of another class (the product), and subclassing the product to add a new feature results in having to subclass the factory only to override the factory method so that it creates an instance of the new product class. An example from MathEdit is illustrated in Figure 9.

The class `Text_ViewWindow` creates an object of type `Text_Editor` in the Text layer. In the Edit layer the `Editor` class is extended with a feature that does not affect the behavior of `ViewWindow` in any other way than that it now has to instantiate objects of type `Edit_Editor` instead of `Text_Editor`. A class `Edit_ViewWindow` (grayed out) has to be introduced only to override the factory method that creates the `Editor` object.

Since frequent subclassing and deep inheritance hierarchies are commonplace in SFI designs, this situation will occur whenever a product class is subclassed and results in a large number of trivial factory subclasses, cluttering the design and increasing the code size. To avoid introducing these subclasses the metaprogramming framework of MathEdit implements a routine that given a class name returns the correct Python class for the running layer (Python classes are first-

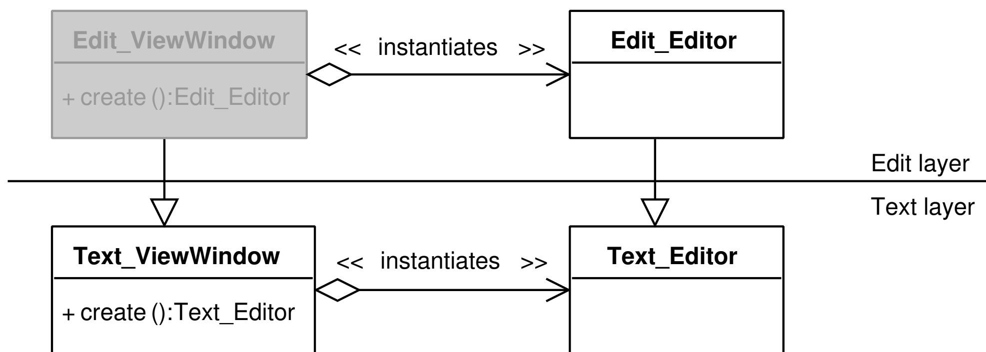


Figure 9: Subclassing to override factory method

class objects which can easily be passed around; in more static languages one might need to do this in compile time using e.g. macro substitutions). If a certain class is not extended in the current layer, the routine searches backwards in the layer stack until it finds the most recent class definition. For example, calling the routine to get the `ViewWindow` class when running layer Edit would return `Text_ViewWindow`.

A substantial problem we encountered with deep inheritance hierarchies is that the control flow through the call chains of overridden methods becomes difficult to overview. The programmers generally thought it was hard to grasp the order of method calls and how the object state changes in response to the calls, especially with many nested method calls. Also, finding a method declaration in the code could require searching through several classes in the inheritance chain, unless the programmer knew exactly in which layer the method was implemented. One programmer commented that "a drawback with having so many hierarchical levels is that you start to forget methods and variables that you defined on the lowest levels."

Our experience indicates that the refactoring stage of SFI is of very high importance for keeping the design clean. Especially when working with unstable requirements, features can not easily be added on top of each other. The programmers found some of the refactorings to be difficult and error-prone, partly because inheritance creates a rather tight coupling between classes. However, a good test harness will substantially aid in the detection of such errors.

6 Unit Testing of Superposition

Unit testing is testing of individual hardware or software units or groups of related units [17]. Extensive, automated unit testing has been proposed as an efficient way of detecting errors introduced by changes in the software [11, 12]. A *unit test* exercises some subset of the software's behavior and validates it against its

specification. Unit testing frameworks frequently group *test methods* into *test case classes*, which can further be aggregated into *test suites*. The complete *test harness*, consisting of all test suites, can then be executed with a single command.

SFI architectures should maintain the superposition refinement relationship between extensions and their bases—class invariants established in previous layers should not be violated in subsequent layers. A layered unit testing architecture allows us to easily create and maintain a test suite that aids in the detection of such violations, typically caused by programmer error or design mistakes. By writing tests for only new functionality and inheriting existing testing functionality, we introduce the requirement that a test introduced in one layer should also pass in all subsequent layers.

Most of our projects have utilized a unit testing architecture based on inheriting test cases. Our experience has shown it to be useful in practice; especially with many layers programmers easily forget assumptions and requirements introduced in a lower layer—if these are reflected in unit tests for the lower layer, possible violations are detected also when running tests for higher layers.

We assume that for testing we use a unit testing framework that provides us at least with a test case class. When constructing test cases in bottom layers, all test cases inherit the class from the testing framework. Test cases of the extended classes in successive layers should be extensions of the test cases from previous layers using the same extension mechanism as the application classes—inheritance. If an inherited method of an application class is overridden and extended with new functionality, the corresponding test method should be extended accordingly. If the body of the extended method contains a call to its ancestor method, the same technique should be applied in the body of the corresponding test method. This allows us to test both new and old functionality by writing tests just for the new functionality.

An example of a basic testing scenario with two layers can be seen in Figure 10. The Simple layer contains one application class (`SimpleText`) and its associated test class (`SimpleTextTest`), which tests the `insert` method of `SimpleText`. The Better layer extends `SimpleText` into `BetterText` by overriding `insert` and adding the `paste` method; correspondingly the `BetterTextTest` test case extends `SimpleTextTest` to override `testInsert` and adds a new test method for the paste method (`testPaste`). The new `testInsert` method should test directly only the new functionality introduced in `BetterText`, it should call the `testInsert` method from the Simple layer to test that the old functionality of insertion is preserved in `BetterText`. In this way, we test that `BetterText` is in fact a superposition refinement of `SimpleText`.

6.1 Python

We have used the PyUnit [21] unit testing framework, which is essentially a Python version of the JUnit testing framework for Java [12]. The programmers

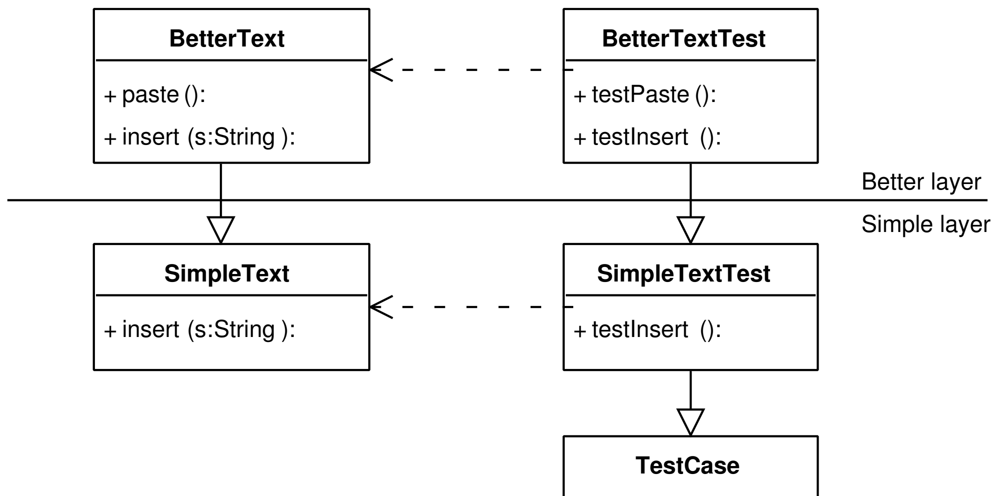


Figure 10: Layered test cases

found it easy to write tests in Python using PyUnit. No special compilation cycle for tests is required, and grouping all tests into a single suite makes it easy to run the tests often; programmers were encouraged to run the tests before committing any new code to the main source.

Test cases are created by subclassing `TestCase`. By default each method with a name starting in `test` is called when the test case is executed. The special methods `setUp()` and `tearDown()` should be implemented to respectively initialize and finalize the class to be tested; typically this is done by instantiating one or more objects and storing them as member variables. `setUp()` is called before running each test method, and `tearDown()` is called after the method has finished. This ensures that each test method has freshly initialized test objects.

Code stubs for the two test cases for the `SimpleText` and `BetterText` classes will look as follows:

```
class SimpleTextTest( TestCase ):
    def setUp( self ):
        self.text = SimpleText()
        # other initialization...
    def tearDown( self ):
        self.text = None
        # other finalization...
    def testInsert( self ):
        # test the insert method...

class BetterTextTest( SimpleTextTest ):
    def setUp( self ):
        self.text = BetterText()
```

```
        # other initializations...
    def tearDown( self ):
        self.text = None
        # other finalization...
    def testInsert( self ):
        SimpleTextTest.testInsert(self)
        # test extended behavior...
    def testPaste( self ):
        # test the paste method...
```

6.2 Eiffel

A number of open source testing frameworks for Eiffel are available. In our project the Gobo [13] tool was used for unit testing. Gobo provides a testing framework similar to that of PyUnit; the programmer subclasses `TS_TEST_CASE` and implements `set_up`, `tear_down` and test methods:

```
deferred class TEST_SIMPLE_TEXT
inherit TS_TEST_CASE
    redefine
        set_up, tear_down
    end
feature {NONE}
    text: SIMPLE_TEXT
feature -- Initialization
    set_up is
        do
            create text
            --Other initialization
        end
    tear_down is
        do
            text:=Void
            --Other finalization
        end
feature -- Testing
    test_insert is
        do
            --Test the insert method
        end

deferred class TEST_BETTER_TEXT
inherit TEST_SIMPLE_TEXT
    redefine
        set_up, tear_down, text, test_insert
    end
```

```
feature {NONE}
    text: BETTER_TEXT
feature -- Initialization
    set_up is
        do
            create text
            --Other initialization
        end
    tear_down is
        do
            text:=Void
            --Other finalization
        end
feature -- Testing
    test_insert is
        do
            precursor
            --Test extended behavior
        end
    test_paste is
        do
            --Test the paste method
        end
```

However, because the Gobo test framework is not integrated into the EiffelStudio environment, it was necessary to set up two different projects, one for compiling the system and one for compiling the tests and the system. The programmers found this arrangement inconvenient.

7 Conclusion and Future Work

We have above described our results from using Stepwise Feature Introduction, a formally defined method, in practical software engineering projects. The central idea in SFI is that layers are built stepwise as superposition refinements on top of each other; using class inheritance as the extension mechanism. Also, each layer together with lower layers should constitute a fully executable application.

We have carried out several case studies in Stepwise Feature Introduction. Our experience from these studies has shown us that SFI works well for structuring, debugging and testing the software under development. Combining SFI with an agile process like Extreme Programming provides architectural structure and guidance to an otherwise quite ad hoc software process, and has allowed us to deliver good working software in a timely manner. It is easy for developers to learn how to apply SFI, and the layer structure helps developers to understand the software architecture.

The main difficulties in applying the method have been caused by lack of automation and, to some extent, conflicting use of class inheritance. These observations point to a need for a generic SFI-supporting programming environment. Many of the programming taks involved in applying SFI can require considerable amount of time. However, most of them can be automated, which would provide a great help for the programmers. The Software Construction Workbench (Section 2.3) was the first attempt to build tool support for SFI, but it was Python-specific. A number of smaller case studies also showed that SCW somewhat too rigidly restricted the software architecture.

SFI-style extensions adds a new dimension to software diagrams, which can become quite large and difficult to overview. We are currently building and experimenting with SOCOS, a prototype tool for constructing and reasoning about software systems, that is intended to support SFI. SOCOS is essentially an editor for *refinement diagrams* [3]. Refinement diagrams have exact semantics and a mathematical base in lattice theory and refinement calculus. A software system is presented to the user as a three-dimensional diagram containing *software parts* and *dependencies* between parts.

The SOCOS system is currently in early stages and the framework is still being worked on. Our current focus is on developing an environment for constructing layered software systems and reasoning about their correctness on both architectural and behavioral levels. Stepwise Feature Introduction, using either inheritance or another layer extension mechanism, is intended to be the main method by which features are added to the system. The goal is to create a tool for correctness-preserving, incremental construction of SFI-layered software systems.

References

[1] Heikki Anttila, Ralph-Johan Back, Pekka Ketola, Katja Konkka, Jyrki Leskela, and Erkki Rysä. Coping with increasing SW complexity - combining stepwise feature introduction with user-centric design. In *Human Computer Interaction, International Conference (HCII2003)*, Crete, Greece, 2003.

[2] Ralph-Johan Back. Software construction by stepwise feature introduction. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 162–183. Springer-Verlag, 2002.

[3] Ralph-Johan Back. Incremental software construction with refinement diagrams. Technical Report 660, TUCS - Turku Centre for Computer Science, Turku, Finland, Jan 2005.

[4] Ralph-Johan Back, Dag Björklund, Johan Lilius, Luka Milovanov, and Ivan Porres. A workbench to experiment on new model engineering applications.

In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 96–100. Springer, 2003.

[5] Ralph-Johan Back, Piia Hirkman, and Luka Milovanov. Evaluating the XP customer model and design by contract. In *Proceedings of the 30th EUROMICRO Conference*. IEEE Computer Society, 2004.

[6] Ralph-Johan Back, Luka Milovanov, and Ivan Porres. Software development and experimentation in an academic environment: The Gaudi experience. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement - PROFES 2005, Oulu, Finland*, June 2005.

[7] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. An experiment on extreme programming and stepwise feature introduction. Technical Report 451, TUCS - Turku Centre for Computer Science, Turku, Finland, Dec 2002.

[8] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. XP as a framework for practical software engineering experiments. In *Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, May 2002.

[9] Ralph-Johan Back and Kaisa Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[10] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[11] K. Beck. *Extreme Programming Explained: Embrace Change*. The XP Series. Addison-Wesley, 1999.

[12] Kent Beck and Erich Gamma. Test-Infected: Programmers Love Writing Tests. *Java Report*, pages 37–50, July 1998.

[13] Eric Bezault. Gobo Eiffel Test. http://www.gobosoft.com/eiffel/gobo/getest/, 2001.

[14] Johannes Eriksson. Development of a mathematical derivation editor. Master's thesis, Åbo Akademi University, Department of Computer Science, 2004.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, 1990.

[18] Mark Lutz. *Programming Python*. O'Reily, 1996.

[19] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[20] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

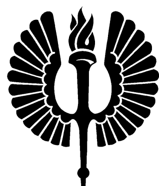
[21] Steve Purcell. PyUnit. http://pyunit.sourceforge.net/, 2004.

[22] Michele Simionato. The Python 2.3 method resolution order. http://www.python.org/2.3/mro.html, 2003.

[23] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 6(1):30–39, 1995.

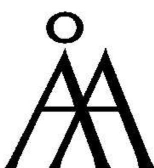
TURKU
CENTRE *for*
COMPUTER
SCIENCE

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1590-9
ISSN 1239-1891