Using Stepwise Feature Introduction in Practice: An Experience Report

Ralph-Johan Back, Johannes Eriksson, and Luka Milovanov

Turku Centre for Computer Science, Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland
{backrj, joheriks, lmilovan}@abo.fi

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.

1 Introduction

Stepwise Feature Introduction (SFI) [1] 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 [2][3] 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 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 [4]. 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 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

---

**Fig. 1.** Alternative extension orders

**Fig. 2.** Interacting components