Refining a Graphical User Interface

Janet Feigenspan
Otto-von-Guericke-University
Magdeburg, Germany
janet.feigenspan@st.ovgu.de

Abstract

Stepwise refinement has been proven useful in developing complex programs by incrementally adding features to a basic program.

The importance of graphical user interfaces is known since the development of personal computers. Users should be able to use a program without putting much effort in learning how to use it.

In this work, we want to answer the question if refining a graphical user interface using feature oriented programming offers a practical approach.

1 Introduction

In the past few years, *feature-oriented programming (FOP)* was suggested for the stepwise refinement of complex programs like *software product lines (SPLs)*. It overcomes problems of *object-oriented programming (OOP)*, especially the problem of crosscutting concerns, which makes it difficult to implement and extend software modularly [5], a crucial property for implementing SPLs [7].

FOP is used mainly for refining the functionality of a program, but to the best of our knowledge, there are only few case studies on refining a *graphical user interface (GUI)* [5].

In everyday life, we interact with many technical devices via their user interfaces. Technical devices can be as simple as a door (even they can be tricky to use, see [13] for an example), or as complex as a personal computer. In respect to computer programs, the user interface is commonly a GUI. For efficiency reasons, it is desirable that the user interface is well designed. According to Spolsky [15], a GUI is well designed, when a program behaves exactly how the user thought it would.

Taking this into account, it is preferable that GUIs should be able to be adapted to the user's needs. Just consider the graphical desktop of a computer: You can add shortcuts to folders, programs, or files. You can resize and move opened windows so that your desktop looks exactly like you want it to be. As another example, think of the Microsoft Windows Calculator. You can choose between a standard and a scientific view. The standard view offers basic arithmetic operations, whereas in the scientific view, you can work with different numeral bases and angle measures.

The following question arises: Is it practical to use stepwise refinement for building a GUI SPL? To answer that, we implemented a basic calculator, refined it, and evaluated the resulting source code, regarding three criteria: The *lines of code (LOC)* of the source code, the structure of the source code, and the obliviousness principle.

In the next Section, we describe the basic calculator, the use of two source versions and their refinements. In Section 3, we compare the final source code of the two versions containing all features regarding the three criteria mentioned above. In Section 4 we point out some other approaches for the refinement of a GUI that could solve the encountered problems. We conclude with a summary of our results in Section 5.

2 Calculator Case Study

In this Section, we describe our case study. We used a calculator as example for a GUI, because the interface is simple and one can easily think of features to add. We implemented two versions of the basic calculator and refined them separately. At the end of this Section, we evaluate the refinements for every version.

We begin with a description of the GUI of the basic calculator, regarding some basic principles of design. Then, we describe the two implementations of the basic calculator and the their refinements for both versions separately.

We generated two versions of source code for the basic calculator, both in Java. The GUI looks the same in both implementations. The first version was implemented by hand. A second version was build with an editor that allows generating GUIs using the *what you see is what you get principle (WYSIWYG)*, the Borland JBuilder 2.00 [2]. The reason for the handwritten source code is, that former

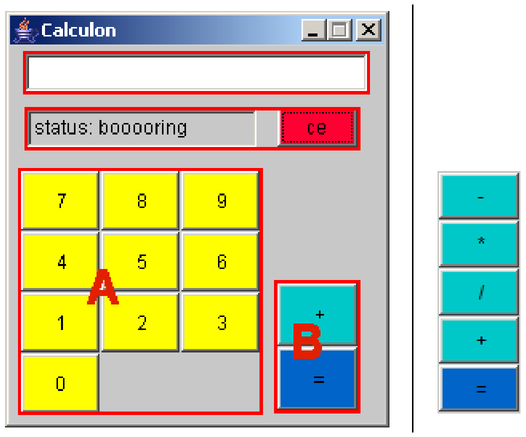

Figure 1. Left: The basic calculator. The red frames mark the panels, where A is the panel containing the number buttons (number panel) and B containing the calculate buttons (calculate panel). Right: Panel B containing five calculate buttons.

case studies on the refinement of complex programs were done with handwritten source code [5]. However, GUIs are often implemented using a WYSIWYG-Editor. With our approach, we can compare our results with those of former case studies and furthermore compare a typical source code for GUIs with a handwritten version. The two versions and their refinements are explained separately.

2.1 The Basic Calculator

In Figure 1 the basic calculator is shown. We applied some basic principles of interface design during layouting the GUI. As the principle of consistency [15] implies, the design is similar to that of a usual calculator: The numbers are ordered in the same way and the buttons for basic arithmetic operations are placed on the right hand side. Both groups of buttons are separated by space and color, which leads to a visual grouping of the different buttons, structuring the GUI [14]. The only operation available in the basic calculator is plus.

When a term is entered, it appears in the text field on the top, so that the user does not have to memorize it, aiding the user in long and complex calculations [15]. When the “=”-button is pressed, the status display changes to an intermediate message. Thus, the user has direct feedback that his input is being processed by the program [13].

We refined each source code version separately and then evaluated each source code in order to find some criteria on structuring a base source code for GUIs that should be refined (cf. Section 3). To maintain comparability, both

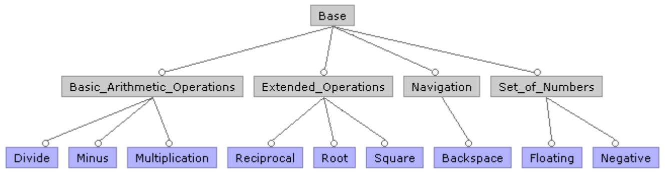

Figure 2. The Feature Model

version used Java AWT components.

Both versions were structured in the same way: The main class creates the GUI and adds the listeners, a calculator class instantiates the GUI object and is designed to perform the calculations. An observer pattern is used to react to events [9].

2.2 Refinements

For refining both versions, FeatureIDE [3] was used. The remaining basic arithmetic operations subtraction (*Minus*), division (*Divide*) and multiplication (*Multiplication*) were added, as well as some standard operations like square root (*Root*), reciprocal value (*Reciprocal*) or squaring (*Square*). Furthermore, negative (*Negative*) and floating point (*Floating*) numbers were introduced and a backspace button (*Backspace*), which deletes the last number or operator entered. Figure 2 depicts all implemented features in a feature model.

In the next two Sections, the refinement of the two base source code versions will be explained, the encountered problems of each implementation and how these problems were solved.

Handwritten Source Code

In the source code implemented by hand, all buttons are grouped into Java AWT panels semantically (cf. Figure 1). The grouping to panels has the advantage that shifting buttons to get space for new features can be done by simply shifting the panel containing the buttons. All panels and the according buttons are instantiated and added to the main frame in specialized methods.

For every feature added, some changes had to be made in the basic source code. First of all, since FeatureIDE uses mixins and thus refinements are transformed to class inheritance during composition, the access of most methods and fields had to be changed from `private` to `protected`. Of course, this could have been anticipated so that the fields and methods could have been declared `protected` in the first place. However, since Java coding conventions suggest private access [4] and the source code should not have been prepared for refinements, they were declared private.

Second, since many of the features needed new buttons to be added to an existing panel, panels that were initially declared as local variables in the according methods had to be made accessible for the refining classes. There are several ways to accomplish this, we preferred defining them as fields instead of local variables.

Aside from the above mentioned adaptations, adding the Negative and Floating feature as well as the Backspace feature led to no further changes in the basic source code.

Listing 1. Excerpt of the source code of the implementation of the Minus feature for the handwritten version

```
GridLayout gl = new GridLayout();
switch(calculateButtonPanel.getComponentCount()) {
  case 0:
    gl.setRows(3);
    gl.setColumns(1);
    calculateButtonPanel.setBounds(170, 130, 50, 120);
        break;
  case 1:
    gl.setRows(4);
    gl.setColumns(1);
    calculateButtonPanel.setBounds(170, 100, 50, 150);
    break;
  case 2:
    gl.setRows(5);
    gl.setColumns(1);
    calculateButtonPanel.setBounds(170, 100, 50, 152);
    break;
}
```

Implementing the remaining basic arithmetic operations subtraction, division and multiplication was rather complicated, because the panel containing those buttons has different properties depending on how many buttons it contains. This led to many lines of code and code replication in all of the three features. Listing 1 shows an example of the source code for the Minus feature. In Figure **??** the panel owning two buttons is compared to the same panel containing five buttons to make this problem clear.

For the features Root, Square, and Reciprocal a new panel was defined. Like the panel for basic arithmetic operations, this panel has different properties depending on the buttons it contains. This problem was solved in the same way (cf. Figure **??** and Listing 1).

Concluding the refinements, it was surprisingly comfortable to add new features to the basic calculator. However, the source code of the basic calculator had to be adjusted for adding new features. This indicates that the obliviousness principle does not apply for the refinement of GUIs using source code like our handwritten source code, i.e. grouping buttons to panels and creating both in specialized methods.

JBuilder Source Code

Like in the handwritten version of the source code, the Borland JBuilder generates a main frame to which the components are added. However, the buttons are not grouped into panels, but added one by one to the frame. The creating and adding of components was realized in one single method. All variables were initialized as private fields.

For the refinements, the access of the method that creates the components and the access of most of the fields had to be changed from `private` to `protected` as in the handwritten version of the source code.

Since the buttons are all added separately to the frame, adding the remaining basic arithmetic operations subtraction, division and multiplication led to some difficulties. The size and positions of those buttons depend on how many arithmetic operations are already available. In the first version, the number of buttons used for basic arithmetic operations was easily obtained by calling a method on the panel containing the buttons. However, in this version, there is no panel grouping similar buttons, so the buttons with the corresponding labels have to be searched in all components of the frame, which leads to a loop and more lines of code. In Listing 2 an excerpt of the source code for the Minus feature is depicted.

Listing 2. Excerpt of the source code of the implementation of the Minus feature for the JBuilder version

```
Component[] componentsInFrame = this.getComponents();
int calculateButtonCount = 0;
for (int i = 0; i < componentsInFrame.length; i++) {
  //count buttons for basic arithmetic operations
}

switch(calculateButtonCount) {
  case 0:
    this.add(minus, new XYConstraints(154, 86, 45, 45));
    break;
  case 1:
    this.add(minus, new XYConstraints(154, 59, 45, 40));
    //replace one of the other buttons (multiply/divide)
    //replace the calculate base buttons
    //...
    break;
  case 2:
    //replace all existing calculate buttons
    //...
    break;
}
```

For the Backspace feature, all buttons need to be moved (cf. Figure 1). Since the shifting has to be done for every single button, not just for a panel grouping the buttons as in the handwritten version of the source code, code replication occurs, because the new position has to be set explicitly for every single button.

To sum up, it was not difficult to refine the basic calculator, but it was tedious and time-consuming to deal with the components one by one. Furthermore, it led to more lines of code compared to the handwritten version as discussed below.

In the next Section, we compare the two source code versions regarding the three above mentioned criteria: LOC, structure of the source code containing all features and the obliviousness principle.

3 Discussion

The last two Sections showed that in both implementations problems occurred when we refined the base source codes. We will now evaluate and compare the two versions regarding the above mentioned criteria: LOC, structure of the source code and the obliviousness principle.

1. *lines of code (LOC)*. LOC is an easily obtainable measure to evaluate the source code [12]. Less LOC usually is better.

2. Structure of the source code. Well structured code is easier to maintain and to refine. Hence, the structure of the source code is an important indicator for how easily the source code can be refined.

3. The obliviousness principle. The obliviousness principle implies refining a source code without changing it first [8]. Or, as the preplanning problem describes, how to design source code so that it can be refined without adaptations. Hence, source code that does not need to be changed for refinement is better suited for adding features than source code that has to be adapted.

From the results we will deduce a guideline when to choose which kind of source code.

Comparing the LOC of the two source code versions over all features, the handwritten source code has 718 and the JBuilder version 993, indicating that the first source code is better suited for refinement, because it leads to less LOC.

Evaluating the structure of the source code, the handwritten source is better structured than the JBuilder source code. In the latter one, some workarounds had to be applied to generate the refined GUI. For example, for the Minus feature, the existing calculate buttons had to be searched in all components to determine the size and position of the minus button. (cf. Listing 2).

For the evaluation of the obliviousness principle, we compare the number of changes in the base feature. As Section 2.2 pointed out, both source code versions needed to be adapted for refinement. Thus, the obliviousness principle does not apply. However, there are differences in the number of changes in the two source code versions: The JBuilder source code only yields 22 LOC of change, whereas the handwritten source code has 40, almost twice as much. This indicates that it is easier to refine a GUI which was build using a source code in which the components are defined as fields and dealt with in a single method, as the JBuilder generated it.

In conclusion, both versions have their shortcomings: Either the refined source code is rather bad structured, unmaintainable and has many LOC (JBuilder version), or the base source code has to be restructured for the refinements (handwritten version).

From this result we can deduce guidelines for designing code that should be refined. We can provide different advise depending on whether the features are known upfront (proactive adaption) or developed later (reactive adaption) [6].

Our results imply that if you want to build a GUI that should be extended without knowledge of the features to be added, it is recommended to implement a source code similar to the one generated by the Borland JBuilder, i.e., regarding the components one by one without grouping and defining them as fields. This implementation allows refining the base source code without changing it much. However, implementing the refinements is tedious and time-consuming, because every component has to be managed individually. As another shortcoming, the resulting source code containing all features has many LOC and is harder to maintain, because some workarounds have to be applied. This leads to a rather bad structured source code, e.g., for the Minus feature finding all created buttons for basic arithmetic operations (cf. Listing 2).

On the other hand, if the resulting GUI can be planned in advance very well, it is more advisable to write the source code by hand, because it can be prepared for refinements. The resulting source code is shorter, better structured and maintainable. However, using Java and no framework or visual aid tool for generating a GUI is rather tedious, because the standard Java layout managers do not give enough support to implement a well designed GUI, however using no layout manager at all requires placing every component by hand without direct feedback of the placement (in contrast to the WYSIWYG principle).

All this evidence suggests that it is possible to build a well designed GUI SPL using stepwise refinement. However, despite different versions of a basic source code, some difficulties can occur: None of the versions could be refined without changes in the base source code, indicating that the obliviousness principle does not apply to the stepwise refinement of GUIs. For the implementation of a GUI that should be refined, the source code has to be planned very carefully. However, as the preplanning problem describes, this is a very complicated issue, in most cases even impossible.

In the next Section we point out some ideas for future work on the stepwise refinement of a GUI, giving some approaches that may solve the encountered problems as discussed in this Section.

4 Related Topics

In the last two Sections, we showed that using FOP for the stepwise refinement of the calculator led to some difficulties in the implementation of the features. It would have been easier if refinements were possible inside methods, not just before or after them (e.g., in the handwritten version, the panels grouping the buttons could have remained as local variables in the according methods).

Another approach for the stepwise refinement of a program is *Aspect Oriented Programming (AOP)*. It was introduced by Kiczales [11]. Aspects react to events occurring during the execution of a program, like the initializing of a variable, the instantiating of an object or the calling of a method. This suggests that it is possible to add source code to arbitrary points in a program.

AOP is useful, when the refinements are homogeneous, i.e., adding the same source code into multiple classes or methods. In the present implementation, code replication occurred, e.g., replacing all buttons for the Backspace feature. Implementing the Backspace feature as aspect, e.g., with AspectJ [1, 10], would have probably reduced the LOC.

Regarding the LOC of changes in the base source code, one particular problem was outlined in Section 2: The accessibility of methods and fields had to be changed from `private` to `protected`. This is a problem of the current implementation of FeatureIDE, since mixins are used for the combining of the features. Future implementations of FeatureIDE could take into account changing the access of methods and fields of classes, that are refined, from `private` to `protected` automatically or use jampack composition instead of mixins.

5 Conclusion

We implemented a calculator GUI using a feature oriented approach for stepwise refinement. We used two versions of a basic calculator and refined each separately using FeatureIDE to build a GUI SPL. The results indicate that it is practical to use stepwise refinement to build a GUI SPL. However, a feature-oriented approach does not seem to be the optimal way to refine a GUI, because the obliviousness principle does not apply and the resulting source code of all features contains some workarounds and code replication.

We developed a guideline for the use of the two source code versions: Knowing the GUI with all features in advance suggests a base source code in which the components are grouped into panels according to semantic issues and the creating of the grouped components is implemented in separate methods, because it can be better prepared for refinements. However, having no requirements to the refinements, it is more advisable to use a source code like the Borland JBuilder generates it, i.e., defining the components as fields and creating them in one method, because the generated source code is better suited for refinements.

References

[1] ApsectJ, 2008. `http://www.eclipse.org/aspectj/`.

[2] Borland JBuilder, 2008. `http://www.borland.com/de/products/jbuilder/`.

[3] FeatureIDE, 2008. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`.

[4] Java coding conventions, 2008. `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transaction on Software Engineering*, 30(6):355–371, 2004.

[6] P. Clements and C. Krüger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adaption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[7] P. Clements and L. Northrop. *Software Product Lines - Practice and Pattern*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2002.

[8] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. *In Workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2047:327–353, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *In Proc. Europ. Conf. on Object-Oriented Programming (ECOOP)*, 1241:220–242, 1997.

[12] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, Washington, 1993.

[13] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, 2001.

[14] B. Preim. *Entwicklung Interaktiver Systeme*. Springer-Verlag, Berlin, Heidelberg, 1999.

[15] J. Spolsky. *User Interface Design for Programmers*. Springer-Verlag, Heidelberg, 2001.