Chapter 1

Introduction

Getting a design right first time is impossible. One of the major advances in software development thinking in the past decade has been the notion that the process of building a software system should be an evolutionary one [10, 81, 48, 3]. Rather than the classical waterfall model where analysis is fully completed before design, and design fully completed before implementation, evolutionary approaches are based on building a simple version of what is required and extending this iteratively to build a more complicated system. As John Gall put it:

> "A complex system that works is invariably found to have evolved from a simple system that worked." [40, p.50]

Or in Kent Beck's inimitable style:

> "Start stupid and evolve." (quoted in [96])

We are interested in developing a particular type of automated transformation to provide support for software evolution. In section 1.1 we explain more exactly what type of transformations we will focus on and describe this in the context of software evolution. In section 1.2 we show how our approach also addresses problems faced in the reengineering of legacy systems. In section 1.3 we state both the thesis and principle contributions of our work, and finally, in section 1.4, we provide a road map of this dissertation.

1.1 Evolutionary Approaches to Software Development

In an evolutionary approach to software development, a simple working system is built which subsequently undergoes many evolutions until the desired system is reached[1]. At each stage there is a working system which is to be extended with a new requirement or set of requirements. It is very unlikely that the design of the initial system will be flexible enough to elegantly support the later requirements to be added in. Consequently, it is to be expected that when the system is to be extended with a new requirement, its design will also have to be made more flexible in order to accommodate the new requirement elegantly. Current thinking recommends breaking this process of extending a system into two stages [5, 35, 45], [38, p.7]:

1. Program Restructuring: This involves changing the design of the program so as to make it more amenable to the new requirement, while not changing the behaviour of the program.

2. Actual Updating: Here the program is changed to fulfill the new requirement. If the restructuring step has been successful, this step will be considerably simplified.

[1]As remarked in [92], one can never speak of the “final” system. Useful systems tend to evolve continually during their lifetime.

This thesis will present a novel approach to providing sophisticated automated support for the restructuring step.

Let us consider now what type of restructurings a designer may want to perform in order to make a system more flexible and able to accommodate a new requirement. A designer usually has an architectural view of how they wish the program to evolve that is at a higher level than, for example, simply creating a new class or moving an existing method. Probably the most interesting and challenging category of higher-level transformation that a designer may wish to apply comprises those transformations that introduce a *design pattern*[2] [41]. Design patterns typically loosen the coupling between program components, thus enabling certain types of program evolution to occur with minimal change to the program itself. For example, the instantiation of a Product class within a Creator class could be replaced by an application of the Factory Method pattern[3]. This would enable the Creator class to be extended to instantiate a subclass of the Product class without significant reworking of the existing code.

The restructurings we develop in this thesis will be those that automate the introduction of design patterns to an existing object-oriented program. The scenario we consider is as follows: An existing program is being extended with a new requirement. After studying the code and the new requirement, the designer concludes that the existing program structure makes the desired extension difficult to achieve, and that the application of some particular design pattern would introduce the necessary flexibility to the program. It is at this point that we aim to provide automated tool support. The designer selects the design pattern to be applied and the program components that

[2]See section 2.2 for a more detailed description of design patterns.

[3]See appendix A for a description the Factory Method pattern

are to take part in the restructuring, and our tool applies that design pattern to the given program components in such a way that program behaviour is maintained.

A key aspect of this approach is that the intellectual decision of what design pattern to apply, and where to apply it, remains with the designer. We are not attempting to formalise or automate quality; our aim is to remove the burden of tedious and error-prone code reorganisation from the designer. In this thesis we will present and validate a methodology for the development of automated design pattern transformations.

1.2 Legacy Systems

Brodie and Stonebraker provide a widely-accepted definition of a legacy system:

> "[A legacy system is one] that significantly resists modification and evolution to meet new and constantly changing business requirements." [12, p.xv]

Legacy systems frequently require restructuring in order to make them more amenable to changes in requirements. This restructuring is performed either by hand, or through the use of automated tools, for example, [6]. In the latter case, the designer usually specifies certain operations to be carried out, for example, to extract a method from existing code or to move a method from one class to another, and the tool handles the mundane details of performing the transformation itself.

There are clear similarities between a designer restructuring a program that is still under development as described in the previous section, and the restructuring of a legacy system. In both cases the following conditions exist:

- A new requirement (or requirements) has arisen that the program must fulfill.
- The structure of the program is not flexible enough to accommodate the new requirement(s) easily and elegantly.
- The existing program exhibits useful behaviour that must be maintained by any reorganisation that takes place.

The similarity between the forward engineering scenario and the restructuring of a legacy system becomes even clearer when the following points are considered:

- The notion of a legacy system usually evokes an image of an aged system developed with now-defunct technology. However, in the above definition there is no mention of age; a week-old program developed using the latest technology can perfectly fit the definition of a legacy system.
- An evolutionary-centric development methodology such as Extreme Programming[4] can be viewed as actually encouraging the creation of a series of legacy systems. Little up-front design is performed, so with each new requirement that is added, the program is restructured just enough to elegantly accommodate the new requirement.

The conclusion is that evolutionary software engineering and legacy systems reengineering are not such different processes. The design pattern transformations described in this thesis are applicable in both cases.

[4]Extreme Programming is discussed further on page 60.

1.3 Thesis and Contributions

In the last two sections we described how introducing design patterns to a program is part both of forward software engineering and of reengineering of a legacy system. The fundamental thesis of this work can be stated as follows:

> *Automating the application of design patterns to an existing program in a behaviour preserving way is feasible.*

The following are the principle contributions of this thesis:

- *A methodology for developing design pattern transformations.* This is the essential contribution of this work. The methodology we have developed has been applied with full rigour to seven common design patterns[5], and a prototype software tool has been built that can apply these seven design patterns to Java programs[6]. The methodology has also been applied to the remaining patterns in the Gamma *et al* pattern catalogue [41], though these pattern transformations have not been prototyped. The essence of our methodology has been published in summary form in [74, 72], and more completely in [75].

- *A minitransformation library.* Design pattern transformations have a strong degree of commonality and this has been captured in a set of six minitransformations. These minitransformations have been implemented and demonstrated to be widely applicable in developing design pattern transformations.

[5]The seven design patterns to which the methodology has been fully applied are Abstract Factory, Factory Method, Singleton, Builder, Prototype, Bridge and Strategy [41].

[6]We have used Java as the vehicle language for this work. The possibility of language independent approaches is discussed on page 165 in section 6.2.

- *A model for behaviour-preservation proofs.* The transformations we develop must be invariant with respect to program behaviour. In order to prove this rigorously for the sophisticated program transformations that we develop, we have extended existing refactoring work by allowing the transformation definition to contain not only simple sequences, but also iteration and conditional statements. This model has been applied in full rigour to several examples, and has been published in [76].

1.4 Thesis Outline

This thesis is structured as follows:

Chapter 1 (this chapter) introduces the topic of automated design pattern transformations and places it in the context of evolutionary approaches to software engineering and legacy system reengineering.

Chapter 2 describes in detail the background to this work, namely program restructuring and design patterns. Note that research that is very directly related to our work is discussed in the relevant later chapter.

Chapter 3 presents our approach to demonstrating that a program transformation preserves the behaviour of the program and applies it in full rigour to a realistic example.

Chapter 4 describes our methodology for the development of automated design pattern transformations by applying it in detail to a single flagship example.

Chapter 5 applies the methodology to the entire Gamma *et al* design pattern catalogue [41] and analyses the results.

Chapter 6 contains our overall conclusions and presents future work in the area of automated design pattern transformations.

Appendix A contains a description of the Factory Method design pattern, which is the subject of chapter 4.

Appendix B contains the complete specification of all analysis functions, helper functions and primitive refactorings that are used in this work.

Appendix C describes briefly the minitransformations that we developed, and provides a reference to the more detailed description in the main text.

Appendix D describes the architecture of the software prototype developed in this work and presents an example of its application.