Chapter 6

Conclusions

This chapter concludes the thesis. In section 6.1 we state again the contributions that have been made by this research. In section 6.2 we present a number of proposals for future work that would extend this research, and finally, in section 6.3, we make some concluding remarks.

6.1 Contributions

The principle contributions of this thesis were stated in chapter 1. Here we restate them:

- *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[1], and a prototype software tool has been built that can apply these seven design patterns to Java programs. The methodology has also been applied to the remaining patterns in the Gamma *et al* pat-

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

tern 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.
- *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].

Other contributions are:

- *The notion of Precursor.* We introduced the notion of a precursor for a design pattern, i.e., a design structure that expresses the intent of the design pattern in a simple way, but that would not be regarded as an example of poor design. We demonstrated the usefulness of this notion by developing precursors for the Gamma *et al* design patterns, and using them as starting points for our design pattern transformations. This set of precursors provides an insight into the type of program to which a given pattern can be applied.
- *A refactory for Java.* The lowest layer of transformations is a collection of refactorings that can be applied to a Java program, and this can

serve as a basis for other transformation work. An extensive set has been designed and implemented, and these are described in appendix B. Some are naturally similar to existing refactorings, while others are peculiar to the development of design pattern transformations.

- *A Precondition Categorisation.* In section 4.4.1 we described how each clause of the precondition to a design pattern transformation can be put into one of four categories. We also described how this categorisation can be used in practice to decide how to deal with the failure of a precondition clause.

6.2 Future Work

In the following subsections we consider possible future work in the area of this thesis.

Practical Tests of the Design Pattern Tool (DPT)

The software prototype we have built as part of this work, DPT, has been tested on several sample programs to establish a base-level confidence that it operates correctly. Naturally, extensive further testing and updating would be required to bring the quality of this prototype to production level.

A more interesting issue in this context relates to programmer acceptance of the type of transformation DPT performs. DPT makes sweeping changes to a program when it applies a pattern, and it is an open question whether a programmer would be content to allow a large system to be updated in this way. Indeed, a software tool can fail in practice for any number of reasons [83], and arguing abstractly that it is nevertheless useful is futile. The author's position is that a programmer will use a software tool only if they have a very clear mental model of what the tool does. Compilers, debuggers and profilers all fit into this category. As design patterns become more established, we can expect programmers to become more comfortable with the type of transformations DPT applies.

One way to aid the programmer's comprehension of the transformation DPT has applied is to present each of the program changes to them and ensure that they are satisfied with each one. If they are not, the whole program can be rolled back to its pre-transformation state. A more ambitious approach is to try to explain the pattern to the programmer (depending on their pattern expertise), and put the changes in this context. Note that existing work in the area of program comprehension has focused on comprehension as part of software maintenance (e.g., [86]). The problem described here, that of presenting the effects of a large refactoring in a comprehensible manner, is a future challenge for this field.

Further Construction of Pattern Transformations

Our refactorings and minitransformations provide a library of reusable components for design pattern transformation development. As with any such library, many iterations are required to fully comprehend the domain and to provide a stable set of components. With each new design pattern development, our understanding of the minitransformations was refined, and frequently this resulted in the refactoring of the library itself. We do not claim that this process is complete. As more design pattern transformations are developed using this approach we can expect more minitransformations to appear and the existing ones to require further work and refinement.

Automation

At present the construction of the behaviour preservation arguments is fragile, in that any change made to a low-level refactoring or analysis function requires that all proofs that use this refactoring or analysis function be rechecked. This dependency itself is unavoidable, but automated software support would be very useful to help manage it. A repository of refactorings, analysis functions and helper functions could be created and this used in performing syntax checking and typechecking of the behaviour preservation arguments. For example, if testing of DPT reveals that the precondition of a refactoring is not strong enough, the specification of this refactoring would then be updated in the repository. The automated assistance software could then highlight which minitransformations and design pattern transformations have to be revisited.

More ambitiously, an attempt could be made to automate the construction of the behaviour preservation argument. This is a challenging task, as we currently use semantic knowledge in building the behaviour preservation arguments. To completely formalise this would involve working with a formal semantics for Java (e.g., [47, 99]), and this would be likely to run into tractability problems. Partial automation is a more promising approach to take, and it would be interesting to see what contribution such an approach could make to the computation of pre- and postconditions for a design pattern transformation.

Pattern Maintenance

Applying a design pattern changes the program code, and some of these changes must be maintained in order for the pattern to remain intact. This means that certain constraints are put on the possible future evolutions of the program. For example, in a program where the Factory Method pattern has been applied, the addition of a new Product class means that a new method must be added to the Creator class as well.

Developing tool support to manage and check these constraints is a valuable extension to our work. The postcondition for a design pattern transformation provides a basis from which to develop the constraints associated with a design pattern. These constraints can be defined using our analysis functions. This enables a software tool to manage the constraints associated with patterns that have been applied to the program, and to notify the programmer if they are updating code that relates to a pattern. The programmer may be advised that their updates are violating a pattern-related constraint, and informed of what other changes are necessary in order to re-establish the pattern constraints.

Language Independence

In our work we focused on the application of design patterns to Java programs. This raises the question of the extent to which our approach is applicable to other programming languages. Some refactorings and minitransformations are applicable to any class-based, object-oriented language, while others are quite Java-specific, for example, those that deal with interfaces.

One approach would be to use the Template Method pattern to describe abstractly how the design pattern transformation operates, and provide the language specific details in subclasses. This is certainly possible; whether it is actually useful depends on the degree of commonality between a set of design pattern transformations that each apply the same pattern, but to programs written in different languages. All refactoring work to date has been language-specific, so this direction would present an interesting challenge.

Pre-transformation Refactorings

For each design pattern transformation we compute its pre- and postconditions, and add its precursor precondition where necessary. This precondition characterises the type of program to which the design pattern transformation can be applied. In section 4.4.1 we categorised the different types of precondition that a design pattern transformation can have. We stated that if a *refactoring precondition* fails, the program can be automatically refactored to correct the problem, and the transformation then applied.

We can view the design pattern transformation as describing a prototypical transformation. If a refactoring precondition fails, the program has to be massaged into a suitable state so that the prototypical transformation can be performed. This is an area for future investigation, and has the potential to make the transformations we have developed applicable to a much broader range of programs.

Pattern Applicability

Our current preconditions simply ensure that the design pattern transformation can be applied without changing program behaviour. It is left up to the programmer to decide if applying the pattern is a good idea or not. We argued strongly in section 2.2 that there are aspects of patterns that require human insight, and that automated attempts to locate suitable places to apply a pattern are of limited value.

However, a software tool could do more in terms of assessing whether the pattern is applicable or not, by asking the programmer certain questions about their intention. For example, in applying the Visitor pattern, the tool might ask the programmer "Do you expect the classes in the object structure to change often?" The answers from the programmer may cause the tool to suggest that the pattern is not a suitable solution, or to configure the exact manner in which the pattern is applied.

Pattern Removal

An over-zealous programmer might apply a pattern even though it is not required, thus obscuring the program rather than enhancing its clarity [84, p.23]. It might also be useful to optimise a program prior to compilation by removing any unnecessary patterns, as they typically have a detrimental effect on runtime performance. An interesting extension to our work is therefore to develop transformations that remove patterns, rather than apply them. In this case, the design pattern structure is the starting point for the transformation, and the corresponding precursor is the target. The informal statement of the starting point for this type of transformation would be "the design pattern structure is present, but its flexibility is not required."

This is not as simple as defining an inverse for each refactoring, and applying them in reverse order. Many refactorings require extra state to be maintained in order to define their inverse. For example, the inverse of a refactoring that deletes an unused class must have access to the deleted class in order to restore it. Even if this extra state is maintained, any changes to the program between the pattern being applied and it being removed might render the inverse refactorings unusable. This area may be interesting to look at, though it is obviously of less impact than the application of design patterns[2].

[2]Unless of course the current interest in design patterns turns to disdain, and the software industry starts "reengineering to depatternise."

6.3 To Conclude

We stated the fundamental thesis of this work in chapter 1 as follows:

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

The research presented in this dissertation has demonstrated the validity of our original thesis. In section 5.7 we found that an excellent transformation was constructed for close to half the patterns considered, and in only 26% of cases could no useful precursor or transformation be found. For seven of the design patterns considered, a rigorous argument of behaviour preservation was also developed. We achieved strong reuse of the minitransformations, as is depicted in figure 5.3 on page 155.

Design patterns have been gaining acceptance in the software engineering community, though the lack of formalisation or automated support has been a weakness of this field. Refactoring has also been gaining support, though again, most of the recent interest has been in non-automated approaches. We have contributed to the formalisation of the refactoring field, and used our contribution to develop a rigorous and practical approach to the automated application of design patterns.