A Methodology for the Automated Introduction of Design Patterns

Mel Ó Cinnéide
Department of Computer Science
University College Dublin
Ireland.
Mel.OCinneide@ucd.ie

Paddy Nixon
Department of Computer Science
Trinity College Dublin
Ireland.
Paddy.Nixon@cs.tcd.ie

Abstract

In reengineering legacy code it is frequently useful to introduce a design pattern in order to add clarity to the system and thus facilitate further program evolution. We show that this type of transformation can be automated in a pragmatic manner and present a methodology for the development of design pattern transformations. We address the issues of the definition of a starting point for the transformation, the decomposition of a pattern into minipatterns and the development of corresponding minitransformations that can introduce these minipatterns to a program. We argue that behaviour preservation is a key issue and develop a rigorous argument of this for each minitransformation we discover. The architecture of an existing software prototype is also discussed and the results of applying this methodology to develop a transformation for the Factory Method pattern are presented.

Keywords: Software re-engineering and restructuring; legacy software; software evolution and modernisation; behaviour preservation; design patterns.

1. Introduction

Legacy systems frequently require restructuring in order to make them more amenable to future changes in requirements. This restructuring is performed either by hand or through the use of automated tools (e.g., [1]). In the latter case, the designer usually specifies certain operations to be carried out, e.g., 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.

We are interested in developing automated support for program transformations that use a more sophisticated target structure than that of the examples just mentioned. A designer usually has a architectural view of how they wish the program to evolve that is at a higher level than simply creating a new method or moving an existing method. One interesting category of higher-level transformation that a designer may wish to apply comprises those transformations that introduce *design patterns* [10]. Design patterns loosen the binding between program components thus enabling certain types of program evolution to occur with minimal change to the program itself. For example, the creation of a Product object within a Creator class could be replaced by an application of the Factory Method pattern (See appendix A for a brief description of design patterns and the Factory Method pattern in particular). This enables the Creator class to be extended to use another type of Product object without significant reworking of the existing code. Our aim is to develop a tool that automates the introduction of design patterns to an existing object-oriented program.

The scenario we assume is as follows: An existing (legacy) program is being extended with a new requirement. After studying the code and the desired requirement, it is concluded that the existing structure of the program makes the desired extension difficult to achieve, and that the application of some design pattern would introduce the desired flexibility to the program. It is at this point in the process that we aim to provide tool support to the designer. A key aspect of this approach is that it is the designer that decides what design pattern to apply and where it is to be applied. We are not attempting to automate quality in any way; our aim is purely to remove the burden of tedious and error-prone code reorganisation from the designer. In this paper we present a methodology and tool support for the development of these design pattern transformations.

In section 2 we outline the methodology we propose for the development of design pattern transformations. Sections 3, 4 and 5 discuss key aspects of this methodology in more detail, using the Factory Method pattern as an example. In section 6 we present the final transformation that introduces this design pattern. The architecture of our current prototype is described in

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

section 7 and some practical results are presented. In section 8 we compare our approach with other work in this area and finally in section 9 we present some conclusions.

2. Methodology

2.1. Motivation

There are several criteria we wish our methodology to fulfil:

- The design pattern transformations developed must preserve program behaviour.
- The transformations are to be applicable to real programs.
- Reuse of portions of existing transformations must be feasible.
- The possibility of automating the development of the transformation itself should be provided for.

We expand on these criteria in the following paragraphs.

Behaviour preservation: Software maintenance activities frequently involve updating existing software in order to address some new requirement. This problem is usually simplified by breaking it into two stages:

1. Refactoring: This involves changing the design of the program so as to make it more amenable to the proposed change, while not changing the behaviour of the program. This corresponds roughly to the *consolidation* phase described in [9].
2. Actual Updating: Here the program is changed to fulfil the new requirement. If the refactoring step has been successful, this step will be considerably simplified.

We are aiming to construct a tool that supports the refactoring step. For this to be successful in practice, the designer must have a strong degree of confidence that the transformations being applied do indeed preserve program behaviour. In our approach we therefore place a heavy emphasis on demonstrating that the design pattern transformations are behaviour preserving.

Applicability to real programs: The tool developed should be applicable to real programs and be able to cope with the complexities of source code representation of design structures. This conflicts to a certain extent with the previous point, in that formally proving properties of programs written in industrial-strength languages is unrealistic. We resolve this by working with an industrial language, Java, and taking a semi-formal approach to demonstrating behaviour preservation.

Reuse where possible: Design patterns have a lot in common so it is to be expected that design pattern transformations will have a lot in common as well. In our methodology we seek to decompose the transformations into reusable chunks and to make these chunks available to later developments of design pattern transformations.

Allow for future automation: There is a non-trivial amount of work involved in building a new transformation for a given design pattern. At present we automate the application of the design pattern transformation; we aim ultimately to automate the building of the transformation as well. This means that given a definition of the structure of a design pattern and a starting point for the transformation, the system can semi-automatically generate the appropriate transformation. At present we use transformations with semi-formal pre- and postconditions (à la Opdyke [16]) and use them to reason rigorously about program behaviour. This opens the possibility of taking a fully-formal approach in the future thus enabling the derivation of the transformation algorithm itself.

2.2. Outline of methodology

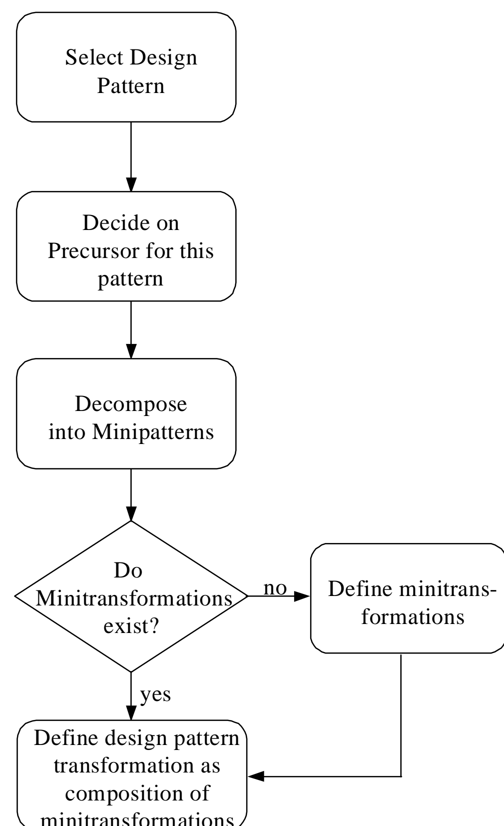

Figure 1. The design pattern methodology.

The methodology is depicted as an activity chart in figure 1. Initially a design pattern is chosen that will serve as a target for the design pattern transformation under development. We then consider what the starting point for this transformation will be, that is, what sort of design structures it may be applied to. This starting point is termed a *precursor*, which is described in more detail in section 3. It has now been determined where the transformation begins, (the precursor) and where it ends (the design pattern itself). This transformation is then decomposed into a sequence of *minipatterns*. A

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

minipattern is a design motif that occurs frequently; in this way it is similar to a design pattern but is a lower-level construct. Section 4 examines minipatterns in more detail.

For every minipattern discovered a corresponding *minitransformation* must also be developed. A minitransformation comprises a set of preconditions, a sequence of transformation steps, a set of postconditions and an argument demonstrating *behaviour preservation.* Minitransformations are our unit of reuse, so for any minipattern identified we first check if a minitransformation for it has already been built as part of the development of a previous design pattern transformation. If so, that minitransformation can be reused now, otherwise a new minitransformation must be developed. A key element in our approach is that the overall transformation must not change the behaviour of the program. We achieve this by demanding that all the component minitransformations be behaviour-preserving as well and this is demonstrated as part of the development of the minitransformation. Minitransformations and our approach to demonstrating behaviour preservation are the subject of section 5.

The final design pattern transformation can now be defined as a sequence of minitransformations. As each minitransformation is behaviour-preserving, so too is the composition of minitransformations so long as the preconditions for each one are shown to hold. An example of a complete design pattern transformation is presented in section 6.

In the following sections we describe this process in more detail, using the Factory Method transformation as an example throughout. In particular, all the terms italicised above are discussed in detail.

3. The starting point for the transformation: *precursors*

Much of the existing work on design pattern transformations ([2], [4], [5], [8]) assumes what can be termed a *green field situation.* By this we mean that when the design pattern transformation is applied to the program, the components that take part in the transformation do not already have any existing relationships pertaining to the pattern. Consequently these approaches do not support the breaking of existing relationships as part of the transformation process. From a software maintenance perspective this is inadequate because in a legacy system, the basic intent of the pattern may well exist in the code already, but in a way that is not amenable to further program evolution. For example, in the case of the Factory Method pattern, the Creator class may already create and use instances of a Product class, but not in the flexible manner that allows easy extension to other Product classes.

At the other extreme there is the *antipattern* ([12]) approach, which was investigated in our earlier work ([14], [15]). In this approach the assumption is made that the designer has failed to appreciate the need for the pattern in the first instance, and has used some inadequate design structure to deal with the situation. The philosophy behind this approach is that the code may have been developed by a designer who was not aware of patterns. For example, in the case of the Factory Method pattern, the client of the Creator class may have to configure it with a flag to tell it what type of Product class to create. We discovered several problems with this approach:

- For any pattern there are several variants and for each variant there can be several antipatterns. The volume of antipatterns rises sharply and each one has to be dealt with individually.
- The design knowledge encapsulated in design patterns has been developed over many decades of software development. A designer who is "not aware of patterns" and chooses an inappropriate solution to a design problem has really just made a mistake. The problem of transforming an antipattern to a design pattern then becomes that of transforming poor design to good design, which cannot of course be solved generically.

For these reasons we use a different starting point for our transformations. For a large class of design patterns, one may regard the effect of the pattern as being to make certain program evolutions easier. This suggests that in the simple case the design pattern is not needed, but as future changes in requirements demand greater flexibility from the software, it becomes necessary. For example, it is frequently the case that a class A creates an instance of a class B, but normally this relationship does not require the application of a design pattern. However a future change in the requirements may well require that class A have the flexibility to work with any one of a number of different product classes, and so the need for the Factory Method pattern arises. The designer of the original system did not make an error of judgement; software systems will always evolve in ways that the original designers simply cannot foresee. Indeed, applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system.

This leads us to our description of a precursor: a precursor is a design structure that expresses the intent of a design pattern in a simple way but that would not be regarded as an example of poor design. This is not a formal definition, but it serves to exclude both the green field situation where there is no trace of the intent of the pattern in the code, and the antipattern situation where the designer had tried to resolve the problem in an inadequate way.

For example, the precursor we use for the Factory Method pattern is simply this: the Creator class must create an instance of the Product class, or in our notation:

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

```
creator.creates(product)
```

This may appear a trivial condition, but it is a natural precursor to the Factory Method pattern. The Creator class creates and uses an instance of the Product class and while this is adequate for the moment, a new requirement may demand that the Creator class be able to work with other types of Product class and this will require the application of the Factory Method pattern.

4. Minipatterns and minitransformations

In developing a transformation for a particular design pattern we naturally wish to reuse our previous efforts as much as possible. To obtain maximum leverage, this reuse should be at the highest level possible. Examining the design pattern catalogues ([3], [10]) it is clear that certain motifs occur repeatedly across the catalogues. For example, a class may know of another one only via an interface, or the messages received by an object may be delegated to a component for detailed processing. These design motifs or *minipatterns* are combined in various ways to produce different design patterns. In this way a pattern is a composition of minipatterns. By focusing on developing transformations for minipatterns, we are able to develop a library of useful transformations that can be reused whenever that minipattern is identified again in a later development. The transformation that corresponds to a minipattern is naturally called a minitransformation.

In the case of the Factory Method pattern we can identify three component minipatterns:

1. ABSTRACTION: The Product class must have an interface that reflects how the Creator class uses the instances of Product that it creates.
2. ENCAPSULATE CONSTRUCTION: In the Creator class, the construction of Product objects must be encapsulated inside dedicated, overrideable methods.
3. ABSTRACT ACCESS: Apart from within the construction methods described in (2) the Creator class must have no knowledge of the Product class except via the interface described in (1).

This amounts to a declarative description of the structure of the Factory Method pattern. It is obvious that other patterns use some of these minipatterns as well. For example, Abstract Factory uses all of them. Many design patterns make use of the first one.

As the same minipatterns occur in many different patterns, we build up a library of them and try to reuse the existing ones in our current effort.

5. Minitransformations and behaviour preservation

For each minipattern identified, a minitransformation is developed. A minitransformation comprises a set of preconditions, a sequence of transformation steps, a set of postconditions and an argument demonstrating *behaviour preservation*. The preconditions for the initial minitransformation must simply be the program state described by the precursor. Later minitransformations will in general rely on the postconditions promised by the earlier minitransformations.

Each minitransformation is defined in terms of low-level *refactorings*. The preconditions are described using *predicates* over the source code. The steps of the transformation frequently use *helper functions* in order to extract useful information from the code. For space reasons we do not provide the full definitions of the refactorings, predicates and helper functions, but see Appendix B for a brief description of the ones we refer to in this paper.

Behaviour preservation is crucial in this approach. We take a view similar to Opdyke in his work on C++ refactorings ([16]) in that our arguments are semi-formal and are applied at the level of program code. We attempt to mimic the argument that a programmer would use were they to perform the design pattern transformation by hand. While our arguments lack the rigor of a fully-formal approach, they capture the logic of the type of behaviour-preservation arguments that programmers make to themselves in reorganising program code. By centralising the argument in one place, we absolve the programmer from having to think about behaviour preservation in applying these transformations. Also, if in regression testing it transpires that a transformation is not in fact behaviour-preserving, the error can be traced back to the original argument and corrected there.

A key difference in our work is that we use postconditions as well, as we want to be able to compose refactorings and minitransformations and argue behaviour preservation about the composition.

The structure of our behaviour preservation arguments matches the top three levels of the software architecture depicted in figure 2. We describe this in more detail in the following paragraphs. At the lowest level we have a layer of refactorings that perform minor transformations to the program code. These are naturally close to Opdyke's refactorings and it is in most cases straightforward to demonstrate that they are behaviour preserving. Some of these refactorings are more complicated and themselves make use of simpler refactorings. For each refactoring we define its pre- and postconditions and provide an argument for behaviour preservation. At this level we also have a layer of helper functions and predicates that are used to extract information from the program code. For example, one helper function, `usesInterface`, determines the actual interface that is accessed via a given object reference. A predicate `creates` determines if one class creates instances of another.

The next layer up is a layer of minitransformations each of which corresponds to a minipattern. Pre- and postconditions are defined for minitransformations, and also an algorithm is given that performs the

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

transformation. This algorithm is defined in terms of the refactorings and helper functions defined at the layer below. The argument for behaviour preservation then is simply a case of demonstrating that the preconditions of each of the component refactorings hold.

The top layer contains the actual design pattern transformations themselves and follows the same pattern of definition and behaviour-preservation argument as for the minitransformations. A design pattern transformation is defined as much as possible in terms of minitransformations, but may also need to use some refactorings from the next lower layer.

The minitransformations for the three minipatterns ABSTRACTION, ENCAPSULATE CONSTRUCTION and ABSTRACT ACCESS are presented below. All helper functions, refactorings and predicates are underlined and brief definitions of these are given in Appendix B.

Name: **ABSTRACTION**

Arguments:

`String className`: name of the class to be abstracted.
`String infName`: name of the new interface to be created.

Description: Builds a new interface, `infName`, to the class `className` and adds it to the program.

Preconditions:

No name clashes:

```
∀ i:Interface i.name != infName ∧
∀ c:Class c.name != infName                    (pre1)
```

The class must exist:

```
exists(className)                              (pre2)
```

Algorithm:

```
(1) Interface inf = abstractClass(className,
    infName)[1];
(2) addInterface(inf);
(3) addImplementsLink(className,infName);
```

Behaviour Preservation:

1: Invokes a helper function and therefore does not update the program. From `pre2` we know that the class must exist.
2: From `pre1` we know that `infName` does not already exist in the program, so this step adds an *unreferenced* interface to the program, which cannot affect compilation or behaviour.
3: `pre2` ensures that `className` exists while the postcondition of (2) ensures that `infName` exists. The postcondition of (1) ensures that `className` fully implements the interface defined in `infName`, so all preconditions for (3) are fulfilled.

Postconditions:

The new interface `infName` precisely reflects the public methods of `className`:

```
equalInterface(className,infName)
```

Name: **ENCAPSULATE CONSTRUCTION**

Arguments:

`String creator`: name of the class to be updated.
`String product`: name of the product class.
`String createP`: name of the new constructor method.

Description: For every constructor in the `product` class, a new method called `createP` that performs this construction is added to the `creator` class. All creations of `product` objects in the `creator` class are replaced with invocations of the appropriate `createP` method.

Preconditions:

The given classes must exist:

```
exists(creator) ∧
exists(product)                                (pre1)
```

The name for the new constructor method must not already exist in the creator class:

```
!exists(creator,createP)                       (pre2)
```

Algorithm:

```
(1) ForAll c:Constructor s.t. c ε product {
(2)   Method m = c.makeAbstract();
(3)   m.setReturnType(absP);
(4)   addMethod(creator, m);
    }
(5) ForAll e:ObjectCreationExpression ε creator
      s.t. e.classCreated() == product &&
      e.containingMethod().name() != createP
    {
(6)   replaceObjCreationWithMethInvocation(e,
      createP);
    }
```

Behaviour Preservation:

1: To demonstrate behaviour preservation over a set operation like this, we show that it holds in the general case for an element of the set:
2: Invokes helper function.
3: Sets the return type of a method that is not yet in the program.
4: Adds a method to the `creator` class. From `pre2` we know that it can't already exist in the class so behaviour is not affected.
5: See (1) above.
6: Steps (2) to (4) add methods `createP` to `creator` that given a list of arguments, return a `product` object constructed with the same argument list. This step replaces each `product` object creation expression with an invocation of the appropriate `createP` method.

Postconditions:

The only expressions in the `creator` class that create `product` objects are found in the `createP` methods:

```
∀ e:ObjectCreationExpression ε creator s.t.
e.classCreated() == product,
  e.containingMethod().name() == createP
```

[1]The new interface created here reflects the entire public interface of the concrete product class. To ensure correctness, all that is required are the parts of the public interface that are actually used in the creator class. However, if the creator class happens not to use an essential part of the concrete product class, this would result in the creation of an unintuitive interface. Therefore we take a conservative approach here and abstract the entire public interface of the product class.

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

Name: ABSTRACT ACCESS
Arguments:
`String creator`: name of the class to be updated.
`String product`: name of the product class.
`String infName`: name of the interface.
`String createP`: name of methods to be excluded.
Description: Replace in the `creator` class all uses of the `product` class name with `infName`. Any method in `creator` named `createP` is not affected.
Preconditions:
The given classes and interfaces must exist:
```
exists(creator) ^
exists(product) ^
exists(infName)                          (pre1)
```
The interface `infName` must precisely reflect the public methods of `product`:
```
equalInterface(product, infName)         (pre2)
```
The `product` class must not have public data:
```
!product.hasPublicField()                (pre3)
```
The `product` class must not have any static methods:
```
!product.hasStaticMethod()               (pre4)
```
Algorithm:
```
(1) ForAll o:ObjectRef ε creator s.t.
      o.type() == product  &&
      o.containingMethod().name() != createP
      {
(2)     replaceClassWithInterface(o, infName);
      }
```
Behaviour Preservation:

1: To demonstrate behaviour preservation over a set operation like this, we show that it holds in the general case for an element of the set:

2: `o` is of class `product` so from `pre3` and `pre4` we see that the only attributes of `o` accessible in `creator` are the public methods of `product`. From `pre2`, all public methods of `product` are part of the `infName` interface, so we can safely redefine the type of `o` to be `infName`.

Postconditions:
The `creator` class holds no references to the class `product` except in the `createP` methods:
```
∀ o:ObjectRef ε creator s.t.
o.type() == product,
  o.containingMethod().name() == createP
```

6. The factory method transformation

In this section we describe the final design pattern transformation that is the culmination of the previous sections. The complete preconditions, algorithm and behaviour-preservation argument is given together with a discussion of the issues involved.

6.1. Specification of the transformation

Name: APPLY FACTORY METHOD
Arguments:
`String creator`: name of the class that creates product objects.
`String product`: name of the product class.
`String absP`: name of the new interface to the `product` class.
`String createP`: name of the new factory method itself.
Description: Applies the Factory Method design pattern to the program.
Preconditions:
```
exists(creator)                          (pre1)
exists(product)                          (pre2)
!exists(absP)                            (pre3)
!exists(creator,createP)                 (pre4)
creator.creates(product)                 (pre5)
!product.hasPublicField()                (pre6)
!product.hasStaticMethod()               (pre7)
```
Algorithm:
```
1. ABSTRACTION(product, absP)
2. ENCAPSULATE_CONSTRUCTION(creator, product,
                             createP)
3. ABSTRACT_ACCESS(creator, product, absP,
                   createP)
```
Behaviour Preservation:
We need only show that the preconditions for each of the component minitransformations hold:

1: `pre2` and `pre3` guarantee the preconditions for this step.

2: `pre1`, `pre2` and `pre4` guarantee the preconditions for this step.

3: `pre1`, `pre2`, `pre3`, `pre6`, `pre7` coupled with the postcondition of (1) guarantee the preconditions for this step.

The algorithm is strikingly simple: it makes use of the three minitransformations we defined earlier so the behaviour preservation argument need only demonstrate that the preconditions for each of these minitransformations hold. This is promising in that it suggests that as we develop a more complete set of minitransformations, other design pattern transformations will be similarly straightforward to define.

6.2. Categorisation of the preconditions

The preconditions for the Factory Method transformation can be divided into four categories. The first four preconditions simply ensure that the classes referred to in the parameters to this transformation exist and that the names for the new program entities to be introduced by this transformation do not already exist. These preconditions are trivial but are necessary to ensure the safety of the transformation. If one of them fails the designer need only be requested to choose a different name to replace the offending choice.

The fifth precondition is the key *precursor precondition*. This describes the essence of the starting point for the transformation. It implies that there is a tight binding between the `creator` class and the `product` class and this is what applying this pattern is going to ameliorate. In general, if a precursor precondition fails, it is senseless to continue with the transformation. In the

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

Factory Method example, the transformation can continue, but it is effectively a *green field* beginning then, and some of the transformations performed will be needless.

The sixth precondition is an example of a *refactoring precondition*, which is a minor problem that prevents the transformation from being applied. The class has a public data field, which is a well-established example of poor class design. This prevents the transformation from being performed as public fields cannot be accessed through an interface. If the designer agrees, this class can be refactored automatically to make this data private and instead provide access to the offending fields via public accessor and mutator methods. This then removes this obstacle to the application of the transformation.

The final precondition is termed a *contraindication* and indicates that there is a more serious problem in applying the Factory Method pattern. The `product` class has a static method that is used by the `creator` class. This implies that the `creator` class depends on the *concrete class* of the product it uses and this cannot be replaced by access via an abstract interface. This is an *inherent* problem in the design of the program that prevents the application of the pattern transformation. In this case the design must be revisited by the programmer to determine if it is possible to resolve this issue.

7. Tool architecture and results

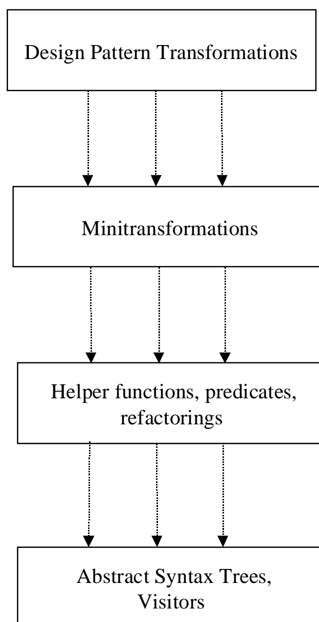

Figure 2. Architecture of the design pattern tool.

We have constructed a prototype software tool, called DPT (Design Pattern Tool) that applies design pattern transformations to Java programs in the manner described above. It has a 4-tier architecture (see figure 2) that matches the layers we defined in the structure of the behaviour preservation arguments:

- Design Pattern transformations.
- Minitransformations.
- Helper functions, predicates and refactorings.
- AST operations.

The top three layers implement the corresponding parts of the argument system described in section 5. The bottom layer implements the actual changes to the code by performing surgery directly on parse trees generated from the Java source files. Visitors ([10]) are frequently used at this level to perform operations that involve an entire parse tree. The parsing of the source files and the construction of the parse trees were implemented using JavaCC ([11]).

We provide an example of the application of the Factory Method transformation to a generic program:

```
class Creator {
  public void doIt() {
    Product p = new Product("some text");
    Product q = new Product(1234);
    p.foo();
    q.foo();
  }
}
class Product {
  public Product(int x){}
  public Product(String s){}
  public void foo() {}
}
```

The Factory Method transformation is now applied to the program above as follows:

```
  applyFactoryMethod("Creator","Product",
                     "absProduct","createProduct")
```

DPT applies the transformation and outputs the following code:

```
class Creator {
  public void doIt (){
    absProduct p = createProduct("some text");
    absProduct q = createProduct(1234);
    p.foo();
    q.foo();
  }
  public absProduct createProduct (int x) {
    return new Product(x);
  }
  public Product createProduct (String s) {
    return new Product(s);
  }
}
interface absProduct {
  public void foo ();
}
class Product implements absProduct {
  public Product (int x) {}
  public Product (String s) {}
  public void foo() {}
}
```

Note how in the `Creator` class all references to `Product` have been changed to `absProduct` and instantiations of the `Product` class only occur via invocations of the new construction methods, `createProduct`. The only change to the `Product` class is that it now implements the new interface `absProduct`,

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

which describes the complete interface to the `Product` class. The significance of these changes is that it is now easy to build a `Creator` class that works with a new type of `Product`. Simply add an `implements` link from the new `Product` class to the `absProduct` interface and subclass the `Creator` class, overriding the `createProduct` methods to instantiate the new type of `Product` class. No further changes are necessary.

We have also developed partial results for other creational patterns: Abstract Factory, Builder and Singleton.

8. Related work

Our ideas on refactoring and behaviour preservation build upon the work of William Opdyke on refactoring C++ programs ([16]). He developed a suite of low-level refactorings that can be applied to a C++ program and used these as a basis for several higher-level refactorings, e.g., conversion of an inheritance relationship to an aggregation relationship. This work was also used as the basis for the Smalltalk Refactory ([17]) developed several years later in the same research group. Smalltalk proved to be a more successful language to use as it does not suffer from the low-level complexities of C++. Our work extends this by using refactorings as a basis for developing a more sophisticated type of refactoring that can introduce a design pattern.

Gert Florijn and his group in Utrecht have developed a patterns tool that provides a broad range of support for a programmer working with patterns ([8], [13]). Their focus is on the representation of design patterns within the tool itself and the maintenance of the constraints associated with a design pattern, i.e., checking that changes to the program do not violate any of the design patterns present in the code. Their work also deals with pattern application, but the starting point of their transformations is the green field situation, so the issues of behaviour preservation and reorganisation of existing relationships as part of the transformation process do not arise.

Yehudai, Gil and Eden ([5]) have developed a prototype tool called the *patterns wizard* that can apply a given design pattern to an Eiffel program. This work is very similar to ours in that it takes a metaprogramming approach and organises the transformations into four levels: design pattern, micro-pattern (our minipatterns), idioms (our refactorings) and abstract syntax tree. The starting point they use is the green field situation rather than attempting to deal with a precursor as we do. This makes the patterns wizard unsuitable for reengineering certain types of legacy code that our approach can handle. If the designer has already partially introduced the intent of the pattern to the code, using the patterns wizard to apply this pattern will leave an amount of manual work for the programmer to do in order to bring the program to a consistent state. As a consequence of taking a green field approach, behaviour preservation is not so important and is more or less ignored in their work. The micropatterns developed in their work are valuable and are used in the specifications of many design pattern transformations. However, of the three minipatterns we used to define the Factory Method transformation, only one, ABSTRACTION, appears in Eden's catalogue ([7]). This is again a consequence of our taking a precursor as the starting point for our transformations: certain minipatterns are necessary in our approach that would not be needed otherwise. Yehudai, Gil and Eden have also developed a declarative language called LePUS for formally specifying the structural and behavioural aspects of design patterns ([6]). This has potential to be developed into a tool that applies design patterns, but practical results in this area are not evident yet in their published work.

The work of Benedict Schultz and Walter Zimmer is also close to what we have presented here ([18], [19]). They merge William Opdyke's refactoring work with so-called *design pattern operators* to produce behaviour-preserving transformations that introduce design patterns. Their published work to date presents only their initial ideas.

9. Conclusions

We have presented a methodology for the development of design pattern transformations in a behaviour-preserving fashion. The use of precursors as the starting point for these transformations is novel and makes this approach especially useful in the area of maintenance of legacy software. The layered architecture enables the transformations to be described in a language-independent way by delegating all language detail to the lower layers. While Java has been used as the vehicle language for this work, we expect that much of what we developed would apply to any similar statically-typed, class-based, object-oriented language.

This layered model also enables us to reuse large parts of existing transformations at a high level. This both speeds the development process and makes the transformations easier to understand. The approach taken to the demonstration of behaviour preservation is both compelling and straightforward to apply. In the future we aim to formalise this approach and examine the potential of automatically deriving a design pattern transformation given the static structure of the design pattern and a suitable precursor.

We have also provided a categorisation of the preconditions for design pattern transformations and described how the different categories can be handled in suitable ways by a transformation tool.

Our methodology has been applied successfully to structure-rich patterns such as Gamma *et al*'s ([10]) creational patterns: Abstract Factory, Factory Method,

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

Singleton and Builder. We also applied an earlier version of this methodology to a structural pattern, Decorator ([14]). The applicability of this approach to strongly behavioural patterns, where the structure of the pattern is less important than its dynamic aspects, has yet to be established.

Our current work involves applying this methodology to a broad variety of design patterns and evaluating our software tool further in a practical context.

References

[1] Bennet, K. H., *Do Program Transformations Help Reverse Engineering?*, Proceedings of the International Conference on Software Maintenance ICSM '98, IEEE Press, 1998.

[2] Budinsky, F. J. et al, *Automatic code generation from design patterns*, IBM Systems Journal, Vol. 35, No. 2, 1996.

[3] Buschmann, F. et al, *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons. 1996.

[4] Demeyer, S., Meijler, T.D. and Rieger, M., *Towards Design Pattern Transformations*, Proceedings of the Workshop on Object-Oriented Software Evolution and Re-Engineering, European Conference on Object-Oriented Programming, June 1997.

[5] Eden, A.H., Gil, J. and Yehudai, A., *Precise Specification and Automatic Application of Design Patterns*, Proceedings of the Twelfth IEEE International Automated Software Engineering Conference, 1997.

[6] Eden, A.H., Hirshfeld, Y. and Yehudai, A., *Towards a Mathematical Foundation for Design Patterns*, available from: http://www.math.tau.ac.il/~eden/ bibliography.html.

[7] Eden, A.H. and Yehudai, A., *Tricks Generate Patterns*, Technical Report 324/97, Department of Computer Science, Tel Aviv University, 1997.

[8] Florijn, G., Meijers, M. and van Winsen, P., *Tool Support in Design Patterns*, European Conference on Object-oriented Programming, June 1997.

[9] Foote, B. and Opdyke, W. F., *Lifecycle and Refactoring Patterns that Support Evolution and Reuse*, Pattern Languages of Programming, 1994.

[10] Gamma, E. et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] JavaCC™, The Java Parser Generator. Available from: http://suntest.com/JavaCC.

[12] Koenig, A., *Patterns and Antipatterns*, Journal of Object-Oriented Programming, April, 1995.

[13] Meijers, M. *Tool support for object-oriented design patterns*, Masters thesis, Department of Computer Science, Rijksuniversiteit Utrecht, August 1996.

[14] Ó Cinnéide, M., *Towards Automating the Introduction of the Decorator Pattern to Avoid Subclass Explosion*, Technical Report TR-97-7, Department of Computer Science, University College Dublin, Ireland (also accepted for the Object-Oriented Evolution and Re-engineering Workshop, OOPSLA, San José, October 1996).

[15] Ó Cinnéide, M. and Nixon, P., *Program Restructuring to Introduce Design Patterns*, Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering, European Conference on Object-Oriented Programming, Brussels, July 1998.

[16] Opdyke, W. F., *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois, 1992.

[17] Roberts, D., Brant, J. and Johnson, R., *A Refactoring Tool for Smalltalk*, Theory and Practice of Object systems, 3(4), 1997.

[18] Schulz, B., *Design Patterns as Operators Implemented with Refactorings*, Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering, European Conference on Object-Oriented Programming, Brussels, July 1998.

[19] Zimmer, W., *Frameworks und Entwurfsmuster*, PhD thesis, Forschungszentrum Informatik Karlsruhe, 1997.

Appendix A: The factory method pattern

This section provides a brief description of design patterns and the Factory Method pattern in particular. For more detail see [10], which is also the source of the example we use here.

Design Patterns encapsulate established solutions to commonly-occurring design problems in a flexible and reusable way. They are not clever, novel architectures but are rather solutions that have been used many times and so have proven their worth.

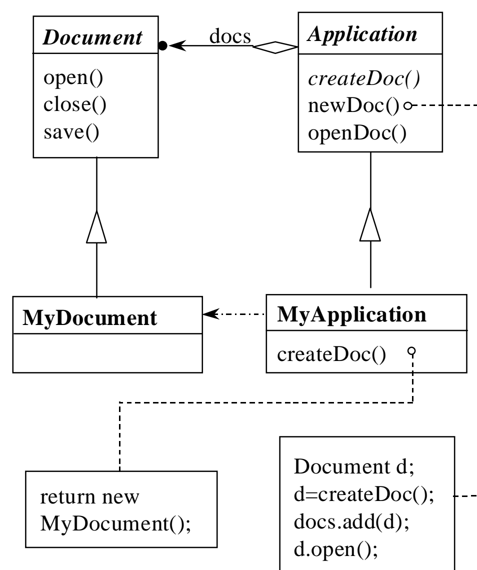

Figure 3. Factory method pattern structure.

The Factory Method pattern is used to loosen the binding between a class (creator) and another class that it instantiates (product). Specifically, it enables the creator class to defer instantiation to a subclass; in this way it is easy to extend the creator class to work with a new type of product class. For example, consider a framework that can present multiple documents to the user. Two key

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.

abstract classes in this domain are Application and Document. The designer has to subclass these classes in order to realise the required functionality. Consider for example using these classes to build a drawing application. The designer would create a subclass of Application, DrawingApplication and a subclass of Document, Drawing Document. The Application class is responsible for creating and managing Documents, but it only knows *when* it should create a Document; it does not know *what kind* of Document to create. This is the kernel of the problem: the framework must create instances of Document, but it knows nothing of the concrete Document classes it should instantiate.

The Factory Method pattern offers a solution to this problem (see figure 3). It encapsulates the knowledge of which Document to create and defers this to a subclass. The abstract Application class invokes an abstract method, createDoc, whenever it needs to create a Document object. Each concrete subclass of Application must now override the createDoc method to create and return an instance of the appropriate type of Document. In figure 3, the MyApplication class redefines the createDoc method to return an instance of MyDocument. The other methods in Application work with this instance through the Document interface.

Appendix B: Helper functions, predicates and refactorings

In this appendix we provide a brief description of the helper functions, predicates and refactorings used in sections 5 and 6.

`abstractClass`: Construct and return an interface that reflects all the public methods of the given class.

`addImplementsLink`: Add an `implements` link from the given class to the given interface.

`addInterface`: Add the given interface to the program.

`addMethod`: Add the given method to the given class.

`classCreated`: Return the class of the object created by a given object creation expression.

`containingMethod`: Return the method that contains the given program entity (expression, object reference etc.).

`equalInterface`: Return true only if the given interface precisely reflects the public methods of the given class.

`exists`: Determine if the given program entity exists.

`hasPublicField`: Return true only if the given class has a public field.

`hasStaticMethod`: Return true only if the given class has a static method.

`makeAbstract`: Return a method that creates the same object as the given constructor.

`replaceClassWithInterface`: Change the type of the given object reference to the given interface.

`replaceObjCreationWithMethInvocation`: Replace the given object creation expression with an invocation of the given method using the same parameter list.

`setReturnType`: Set the return type of the given method to the given class/interface.

`type`: Return the type of the given object reference.

Authorized licensed use limited to: University of Canberra. Downloaded on June 22,2020 at 03:24:09 UTC from IEEE Xplore. Restrictions apply.