International Symposium on Applied Corporate Computing, Monterrey, Mexico, October 1995

Automated Software Evolution via Design Pattern Transformations

Lance Tokuda and Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
{unicron, batory}@cs.utexas.edu

Abstract

Software evolution is often driven by the need to extend existing software. "Design patterns" express preferred ways to extend object-oriented software and provide desirable target states for software designs. This paper demonstrates that some design patterns can be expressed as a series of parameterized program transformations applied to a plausible initial software state. A software tool is proposed that uses primitive transformations to allow users to evolve object-oriented applications by visually altering design diagrams.

Keywords: object-oriented design patterns, program transformations, software evolution.

1 Introduction

In 1979, Parnas argued that software developers should design each program as the first in a family of programs [Par76]. Initial programs spawned versions as a consequence of evolution and maintenance. By making program families explicit in software design, the cost of evolution and maintenance can be substantially reduced.

During the 1970s, evolution and maintenance accounted for 35 to 40 percent of the software budget for an information systems organization. This number jumped to 60 percent in the 1980s. It was predicted that without a major change in approach, many companies will spend close to 80 percent of their software budget on maintenance by the mid-1990s [Pre92].

Object-oriented design methodologies offer important opportunities for reducing maintenance costs. For example, object-oriented software is organized into classes and frameworks that provide modularity and enhance reusability. This, in turn, can substantially reduce the cost of adding new functionality to a product. Language features such as inheritance also contribute to reuse and maintenance by allowing specializations of a class to be built without altering the original class.

Another kind of reuse is the reuse of designs. Commercial object-oriented languages such as C++ provide features to declare classes and relationships between classes (e.g., public inheritance, private inheritance, object pointers as instance variables of a class) facilitating the construction and reuse of larger software artifacts. Recent work has recognized an important kind of design reuse: *object-oriented design patterns*, i.e., recurring patterns of relationships between classes, objects, methods, etc. that define preferred solutions to common object-oriented design problems [Gam92, Coa92, Joh92, Gam94].

While design patterns are useful when included in an initial software design, they are often applied in the maintenance phase of the software lifecycle [Gam93]. For example, the original designer may have been unaware of a pattern or additional system enhancements may arise that require unanticipated flexibility. Alternatively, patterns may lead to extra levels of indirection and complexity inappropriate for the first software release.

We have discovered that some design patterns can be expressed as compositions of primitive program transformations. Moreover, we have noted that many of the program transformations can be automated. This raises the intriguing possibility that a software tool could perform these transformations on application code automatically by allowing users to graphically alter class diagrams that capture the application design. Automating transformations would reduce the cost of certain kinds of program evolution, and eliminate programming errors and debugging that would otherwise have been introduced. We explore these ideas in this paper.

We begin by reviewing the concept of design patterns and explain their relationship to program transformations and program evolution.

2 Design Patterns

Design patterns capture expert solutions to many common object-oriented design problems: support for multiple implementations of a method, creation of compatible components, adapting a class to a different interface, subclassing versus subtyping, isolating third party interfaces, etc. Patterns have been discovered in a wide variety of applications and toolkits including Smalltalk Collections [Gol84], ET++ [Wei88], MacApp [App89], InterViews [Lin92], etc.

In this section, we present a design pattern and explain how it can be viewed as a program transformation. The notation we use for displaying class diagrams is adopted from Rumbaugh [Rum91] with extensions for representing code fragments [Gam92] and general class dependencies. A summary of the notation is given in Appendix A.

2.1 A Design Pattern Example

Subclasses are often designed for use specifically with other subclasses. The *Abstract Factory* design pattern employed by InterViews [Lin92] and ET++ [Wei88] and documented in [Gam93] ensures that compatible objects are created.

For example, consider the classes in Figure 2.1. The superclass **ScrollBar** has two subclasses: **MotifScrollBar** and **OpenLookScrollBar**. **Window** also has two subclasses: **MotifWindow** and **OpenLookWindow**. Motif scrollbars are intended to work with Motif windows. While it may be possible for a program to create a Motif scrollbar and an OpenLook window, it is unlikely that this combination of objects will function properly.

A preferred solution to this problem is given in Figure 2.2. The *abstract factory* **WindowFactory** is created as an abstract class. It provides creation methods (known as *factory methods*) for returning new window system objects, i.e. scrollbars and windows. Two *concrete factories*, **MotifFactory** and **OpenLookFactory**, are created as subclasses of **WindowFactory**. **MotifFactory** only returns Motif objects while **OpenLookFactory** only returns OpenLook objects. At run time, the appropriate concrete factory object is instantiated. Requests to create window system objects are passed to this factory object. The factory ensures that any window system objects that are created will work together.

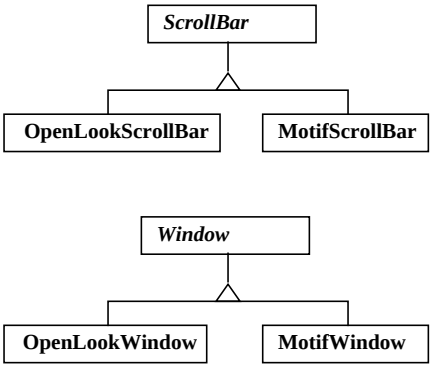


Figure 2.1: Scrollbar and window class hierarchies

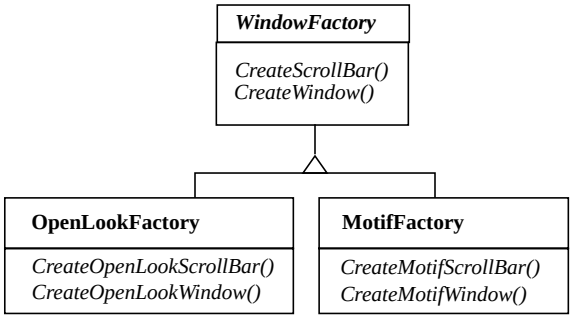


Figure 2.2: Abstract factory

2.2 Design Patterns as Transformations

Design patterns define target states for program transformations. Suppose an application **A** relied on the **MotifScrollBar** and **MotifWindow** classes. Now suppose **A** needs to be generalized to **Aprime** to use OpenLook widgets. The Abstract Factory design pattern just described defines the needed generalization. A useful tool would automatically transform the application code of **A** to that of **Aprime** by applying the Abstract Factory pattern to the **MotifScrollBar** and **MotifWindow** classes of **A**.

We believe that such tools can be created and that many pattern transformations can be automated. In fact, we show in Section 4 how an application comparable to **A** could be generalized by applying a sequence of primitive pattern transformations. Tools that modify application code automatically would not only relieve users of tedious and error-prone tasks, but also would drastically reduce application evolution and maintenance costs.

We have analyzed a variety of recognized design patterns, and those that we believe could be automated are given in Appendix B. This list is expected to grow as additional pattern transformations are identified. Furthermore, we have observed that these patterns are actually compositions of a small number of primitive object-oriented transformations. In the next section, we present some of these transformations.

3 Object-Oriented Transformations

Object-oriented transformations are program transformations which alter the classes or frameworks in an object-oriented software system. Transformations have the following properties:

- Description
- Arguments
- Initial state
- Target state
- Preconditions for application

In the following sections, we define three parameterized object-oriented transformations. We use C++ to represent the initial and target states of source code that is being transformed. Arguments such as iv^* and $m^*()$ are used to denote zero or more instance variables or methods provided as arguments to the transformation. Other instance variables and methods not specified are assumed to be unaffected.

3.1 Inherit[C1, C2, vm2*(), m2*(), iv2*]

The **Inherit[ ]** pattern transformation establishes a superclass-subclass relationship between two existing classes, **C1** and **C2**. It also supports the promotion of subclass methods **m2*()** and variables **iv2*** to the designated superclass, as well as adding virtual methods **vm2*()** to the superclass. The arguments of **Inherit[ ]** are:

C1 - superclass name
C2 - subclass name
vm2*() - list of superclass virtual methods to be provided by subclass
m2*() - list of methods and implementations moved from subclass to superclass
iv2* - list of instance variables moved from subclass to superclass

Figure 3.1 depicts the initial and target states of this transformation. The preconditions for **Inherit[ ]** are:

- Other subclasses of **C1** (if any) must support virtual methods **vm2*().**
- Subclass **C2** must support all virtual methods of superclass **C1.**
- Methods **m2*()** moved from subclass **C2** must not reference any subclass specific methods or instance variables.

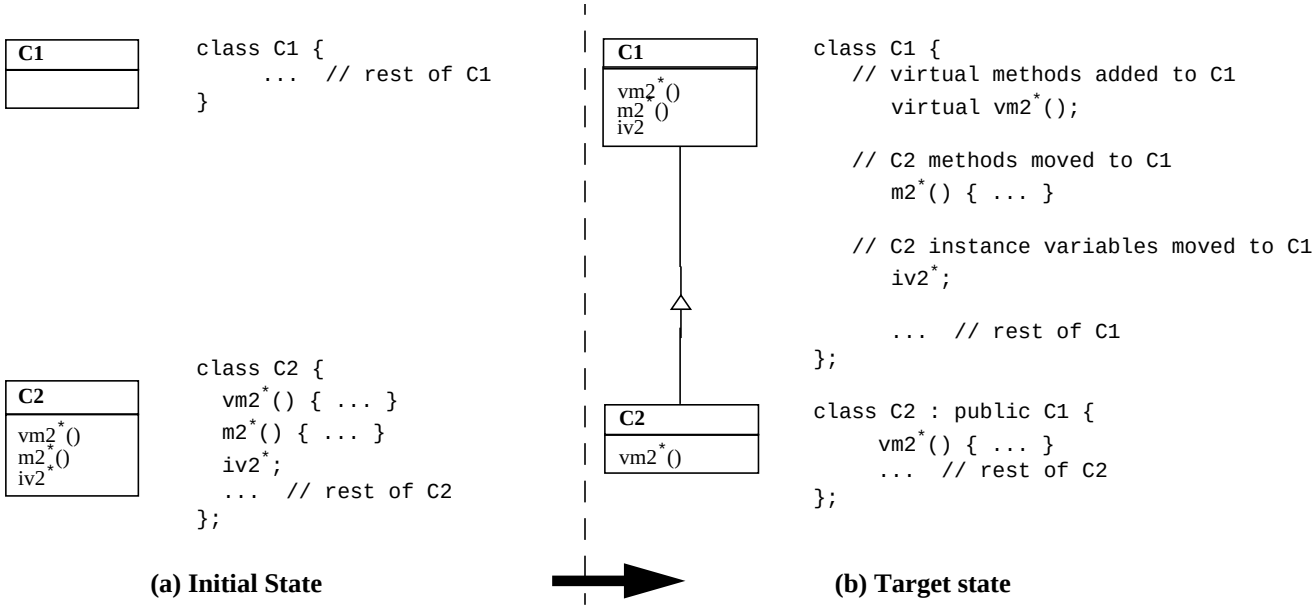


Figure 3.1: Inherit transformation

3.2 FactoryMethod[C1, C2, m(), C3, v]

The **FactoryMethod[ ]** transformation adds a method **m()** to class **C1** which creates new objects of class **C2**. A call to this method replaces occurences of `new C2` in the program. Expression **v** must return a **C1** object so that `v->m()` returns an object of class **C2**. The arguments of **FactoryMethod[ ]** are:

C1 - factory class name
C2 - product class name
m() - name of new factory method to create a new product
C3 - class of object returned by method m()
v - an expression which returns an object supporting method m()

Figure 3.2 depicts the initial and target states of this transformation. The preconditions for **FactoryMethod[ ]** are:

- **m()** must not already exist in **C1**
- **C3** must be **C2** or a superclass of **C2**
- **v** must be a valid expression for all occurrences of `new C2`. Automated checking of this condition is equivalent to performing the substitution, recompiling, and checking for 'invalid expression' or 'type mismatch' errors.

3.3 Substitute[C1, C2, C3]

The **Substitute[ ]** transformation substitutes class **C1**'s references to class **C2** with references to class **C3**. All pointers in class **C1** including instance variables, method return types, method arguments, method local variables, etc. are converted. Class **C3** must support class **C2**'s interface to class **C1**.

If class **C3** is concrete, then occurences of `new C2` are replaced by `new C3`. If class **C3** is an abstract superclass of class **C2**, then no substitution is made. Otherwise, an error is flagged.

The arguments of **Substitute[ ]** are:

C1 - class whose association is being changed
C2 - class originally associated with **C1**
C3 - class to be associated with **C1**

Figure 3.3 depicts the initial and target states of this transformation. The preconditions for **Substitute[ ]** are:

- **C3** must support **C2**'s interface to **C1**. Automated checking of this condition is equivalent to performing the substitution, recompiling, and checking for 'undefined variable' or 'undefined method' errors.
- Calls to **C1** methods returning a pointer to a **C2** object must accept objects of type **C3**. Automated

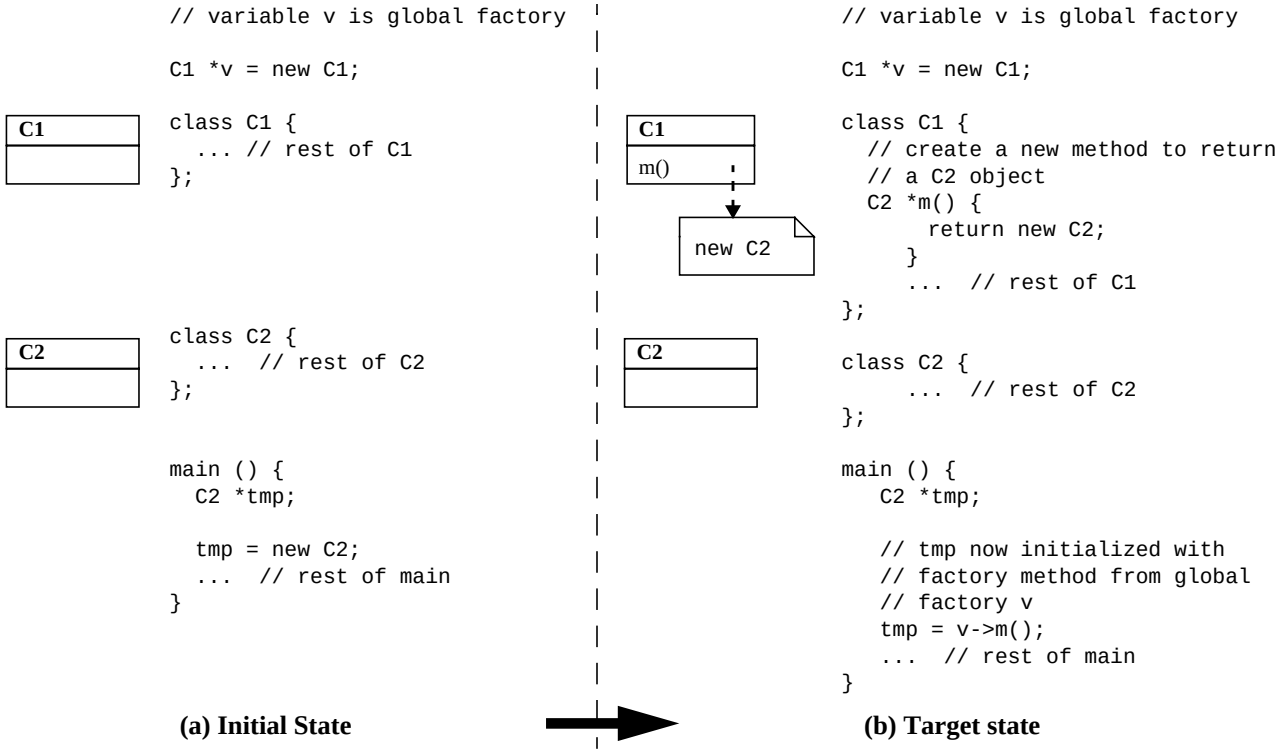


Figure 3.2: FactoryMethod transformation

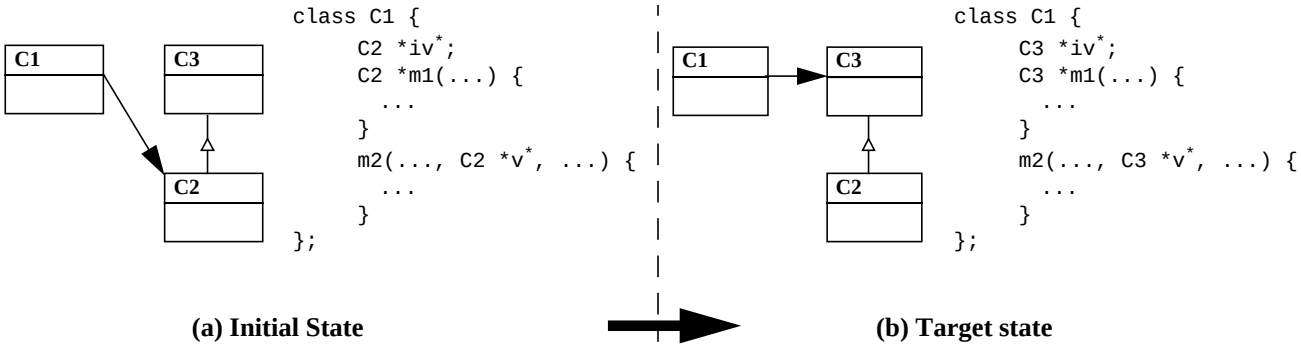


Figure 3.3: Substitute transformation

checking of this condition is equivalent to performing the substitution, recompiling, and checking for 'type mismatch' errors.

- If **C1** contains the statement `new C2`, then **C3** must be a superclass of **C2** (in which case no substitution is made) or **C3** must be a concrete class (in which case the expression `new C3` is substituted).

3.4 Other Transformations

Two other transformations will be used in our example of Section 4. **CreateClass[C1]** creates a new class. It takes the name of the new class as an argument.

CreateInstanceVariable[C1, type, iv, init] adds a new instance variable to a class. It takes the class, the type of the new variable, the name of the new variable, and an initializer as arguments.

4 Program Transformation Example

We now show how pattern transformations can be used to evolve a program. Our example is a simple program that creates a Honda Prelude with a **VTEC2_2** engine and **GY184_HR14** tires. The car is driven for one million miles. During the drive, tires are rotated and changed, and the engine is replaced. The program and its class diagram are displayed in Figure 4.1a and 4.2a.

The program only works for a Prelude. At some point, we want other cars to create and drive. There are several problems that must be addressed to generalize this program. In this example, we use object-oriented transformations to create and install an abstract factory and demonstrate that the resulting program is easier to extend and reuse.

The first step creates superclasses for **GY184_HR14** and **VTEC2_2**. **CreateClass[ ]** creates the **Tire** and **Engine** classes. **Tire** and **Engine** are then declared to be superclasses of **GY184_HR14** and **VTEC2_2** respectively using the **Inherit[ ]** transformation. The resulting code changes and class diagram is displayed in Figure 4.1b and 4.2b.

```
Step 1:Superclass Tire and Engine

   CreateClass[Tire]

   CreateClass[Engine]

   Inherit[
      Tire,              - superclass,
      GY184_HR14,        - subclass
      nil,               - virtual methods
                           from subclass
      (Drive),           - methods from sub-
                           class
      (miles, max_miles)]
                         - instance vari-
                           ables from sub-
                           class

   Inherit[Engine, VTEC2_2, nil, (Drive),
      (miles, max_miles)]
```

Next, we take advantage of the new Tire and Engine superclasses by substituting them for **GY184_HR14** and **VTEC2_2** respectively in the **Car** class. The **Car** class can then operate with any **Tire** and **Engine** class. Note that this transformation can only be applied if the enabling conditions are satisfied. In this example, the class in which substitution occurs must not reference any subclass specific instance variables or methods of **GY184_HR14** or **VTEC2_2**. The transformed program now employs the Abstract Factory design pattern. The final class diagram is displayed in Figure 4.1c and 4.2c.

```
Step 2:Generalize Car to use Tire and Engine

   Substitute[
      Car,               - class in which
                           substitution
                           occurs
      GY184_HR14,        - old association
      Tire]              - new association

   Substitute[Car, VTEC2_2, Engine]
```

We can now construct a concrete factory for creating tires and engines. An instance variable is added to class **App** which stores the concrete factory to be used by the program. Factory methods can then be added to **PreludeFactory** to create the appropriate tires and engines. The resulting code changes and class diagram is displayed in Figure 4.1c and 4.2c

```
Step 2:Create a concrete factory

   CreateClass[CarFactory]

   CreateInstanceVariable[
      App               - class to receive
                          new instance
                          variable
      PreludeFactory*   - type of new vari-
                          able
      car_factory       - name of new vari-
                          able
      new PreludeFactory]
                        - initializer

   FactoryMethod[
      PreludeFactory,   - concrete factory
      GY184_HR14,       - product produced
                          by method
      MakeTire,         - name of factory
                          method
      Tire,             - return type of
                          method
      app->car_factory] - expression
                          returning a fac-
                          tory object. The
                          factory method
                          of the object is
                          used to replace
                          occurences of
                          "new GY184_HR14"

   FactoryMethod[PreludeFactory, VTEC2_2,
      MakeEngine, Engine, app->car_factory]
```

Next we create the abstract factory **CarFactory** as a superclass of the concrete factory **PreludeFactory**. The resulting class diagram is displayed in Figure 4.1e and 4.2e.

```
Step 4:Superclass PreludeFactory

   Inherit[
      CarFactory,       - superclass
      PreludeFactory,   - subclass
      (MakeTire, MakeEngine),
                        - virtual methods
                          from subclass
      nil,              - methods from sub-
                          class
      nil]              - instance vari-
                          ables from sub-
                          class
```

Using **Substitute[ ]**, we generalize the **App** class to work with any **CarFactory**. The resulting class diagram is displayed in Figure 4.1e and 4.2e.

```
Step 5:Generalize App to use CarFactory

   Substitute[
      App,              - class in which
                          substitution
                          occurs
      PreludeFactory,   - old association
      CarFactory]       - new association
```

The program now contains the abstract factory **CarFactory** with concrete factory **PreludeFactory**. **CarFactory** objects produce objects of the **Tire** and **Engine** classes. **PreludeFactory** produces the **GY184_HR14** and **VTEC2_2** objects required by a Prelude.

5 Benefits

The transformed program now employs the Abstract Factory design pattern which guarantees that only coordinated car components will be produced. A comparison of the initial (Figure 4.1a) and final (Figure 4.1f) class diagrams reveals that the transformed program is more complex. In exchange for increased complexity, the new program is more general and offers a number of advantages over the original:

- other tire and engine subclasses can be added which inherit state and behavior from the original tire and engine
- switching the engine or tires for a Prelude requires modification of only one factory method in **PreludeFactory**
- other factories can be added to create other cars
- once another concrete factory has been implemented, it is easy to reuse the program to drive this new car.

The following sections illustrates each benefit.

5.1 Adding Other Classes

In this example, the new tire and engine inherits state and behavior from **Tire** and **Engine** respectively (Figure 5.1). This inherited state and behavior originally belonged to the **GY184_HR14** and **VTEC2_2** classes.

```
// adding engine used in Honda Accord which
// inherits from new Engine superclass
class AccordEngine : public Engine {
  AccordEngine {
    max_miles = 120000;
    miles = 0;
  }
}

// adding tire used in Honda Accord which
// inherits from new Tire superclass.
class Bridge184_SR14 : public Tire {
  Bridge184SR_13 {
    max_miles = 30000;
    miles = 0;
```

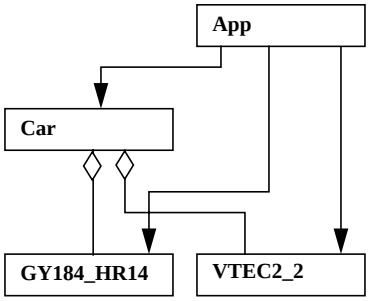


Figure 4.1a: Initial class diagram

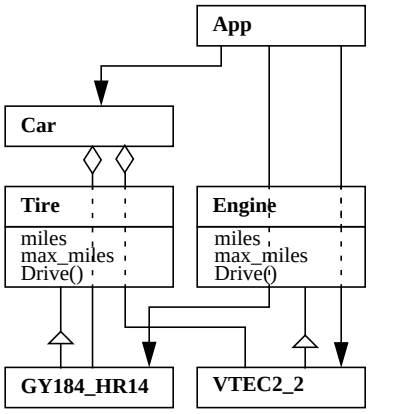


Figure 4.1b: Tire and Engine superclasses created

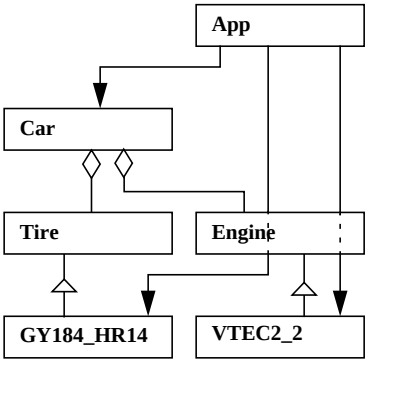


Figure 4.1c: Car dependency on GY184_HR14 and VTEC2_2 is removed

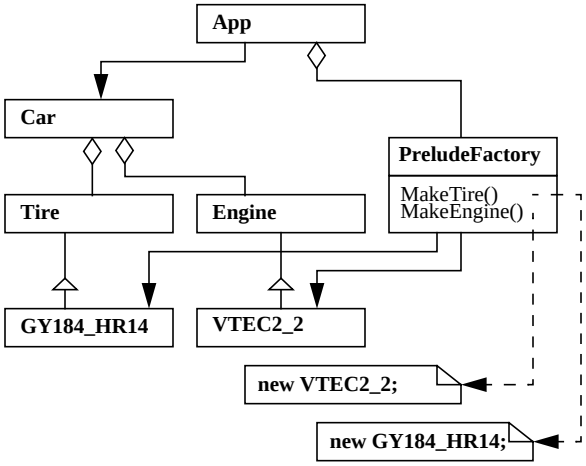


Figure 4.1d: Concrete factory added

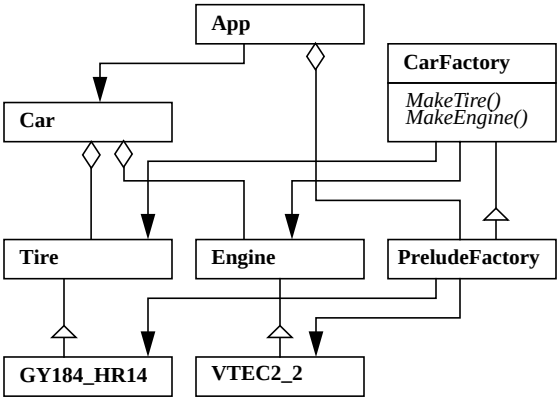


Figure 4.1e: Abstract factory created

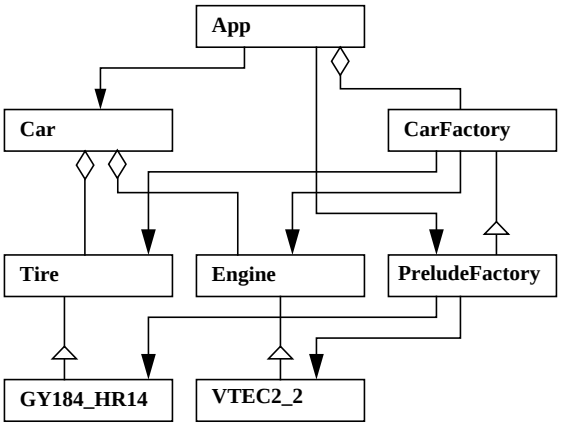


Figure 4.1f: App reference to PreludeFactory interface is removed. App still creates a PreludeFactory object

```
#include <iostream.h>

class GY184_HR14 {
public:
  int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
  GY184_HR14 () {
    max_miles = 40000;
    miles = 0;
  }
};
class VTEC2_2 {
public:
  int miles, max_miles;
  void Drive(int m) {
    miles += m;
  }
  VTEC2_2 () {
    max_miles = 100000;
    miles = 0;
  }
};
class Car {
private:
  GY184_HR14 *lf_tire, *lb_tire,
    *rf_tire,*rb_tire;
  VTEC2_2 *engine;

public:
  int miles;
  Car();
  void RotateTires() {
    GY184_HR14 *tire;

    tire = lf_tire;
    lf_tire = lb_tire;
    lb_tire = lf_tire;
    tire = rf_tire;
    rf_tire = rb_tire;
    rb_tire = rf_tire;
    cout << "Rotating tires\n";
  }
  void ReplaceTires(GY184_HR14 *lf,
      GY184_HR14 *rf, GY184_HR14 *lb,
      GY184_HR14 *rb) {
    delete lf_tire;
    delete lb_tire;
    delete rf_tire;
    delete rb_tire;
    lf_tire = lf;
    lb_tire = lb;
    rf_tire = rf;
    rb_tire = rb;
    cout << "Replacing tires\n";
  }
  void ReplaceEngine(VTEC2_2 *e) {
    delete engine;
    engine = e;
    cout << "Replacing engine\n";
  }
  void Drive(int m) {
    miles += m;
    engine.Drive(m);
    lf_tire.Drive(m);
       //     lb_tire.Drive(m);
    rf_tire.Drive(m);
    rb_tire.Drive(m);
    cout << "Driving " << m << " miles\n";
  }
  int TireMiles() {
    return lf_tire->miles;
  }
  int MaxTireMiles() {
    return lf_tire->max_miles;
  }
  int EngineMiles() {
    return engine->miles;
  }
  int MaxEngineMiles() {
    return engine->max_miles;
  }
};

class App {
public:
  App() {
  }
  void run();
};

App *app;

Car::Car(GY184_HR14 *lf, GY184_HR14 *lb,
    GY184_HR14 *rf, GY184_HR14 *rb, VTEC2_2 *e)
{
  engine = new VTEC2_2;
  lf_tire = new GY184_HR14;
  lb_tire = new GY184_HR14;
  rf_tire = new GY184_HR14;
  rb_tire = new GY184_HR14;
}
void App::run() {
  Car *car = new Car(new GY184_HR14,
      new GY184_HR14, new GY184_HR14,
      new GY184_HR14, new VTEC2_2);
  while (car->miles < 1000000) {
    car->Drive(1000);
    if (car->TireMiles() >=
        car->MaxTireMiles())
      car->ReplaceTires(new GY184_HR14,
        new GY184_HR14, new GY184_HR14,
        new GY184_HR14);
    else if (car->TireMiles() %5000 == 0)
      car->RotateTires();
    if (car->EngineMiles() >=
        car->MaxEngineMiles())
      car->ReplaceEngine(new VTEC2_2);
  }
  cout << "Total miles driven: " <<
    car->miles << "\n";
}

int main () {
  app = new App;
  app->run();
}
```

Figure 4.2a: Example Application

```
class Tire {
public:
  int miles, max_miles;  // moved from subclass
  void Drive(int m) {    // moved from subclass
    miles += m;
  }
};

class Engine {
public:
  int miles, max_miles;  // moved from subclass
  void Drive(int m) {    // moved from subclass
    miles += m;
  }
};

// subclass of Tire
class GY184_HR14 : public Tire {
  GY184_HR14 () { ... }
};

// subclass of Engine
class VTEC2_2 : public Engine {
  VTEC2_2 () { ... }
};
```

Figure 4.2b: Code changes for creating Tire and Engine superclasses

```
class Car {
public:
  // variables changed from GY184_HR14 to Tire
  Tire *lf_tire, *lb_tire, *rf_tire,
    *rb_tire;

  // variable changed from VTEC2_2 to Engine
  Engine *engine;

  void RotateTires() {
    // variable type changed from GY184_HR14
    // to Tire
    Tire *tire;
    ...
  }

  // variables changed from GY184_HR14 to Tire
  void ReplaceTires(Tire *lf, Tire *rf,
                    Tire *lb, Tire *rb) {
    ...
  }

  // variable type changed from VTEC2_2
  // to Engine
  void ReplaceEngine(Engine *e) {
    ...
  }

  ...
};
```

Figure 4.2c: Code changes to generalize Car class

```
class PreludeFactory {
public:
  // new method to create Tires
  Tire *MakeTire() {
    return new GY184_HR14;
  }


  // new method to create Engine
  Engine *MakeEngine() {
    return new VTEC2_2;
  }
};

class Car {
public:
  ...
  // arguments to constructor generalized
  Car (Tire *, Tire *,
    Tire *, Tire *, Engine *);
  ...
}

class App {
public:
  Car *car;

  // new instance variable created
  PreludeFactory *car_factory;

  App() {
    // new instance variable initialized
    car_factory = new PreludeFactory;

    car = new Car;
  }
  void run();
};

Car::Car(Tire *, Tire *,
  Tire *, Tire *, Engine *) {
  miles = 0;

  // use factory method to create the
  // engine
  engine = app->car_factory->MakeEngine();

  // use factory method to create the tires
  lf_tire = app->car_factory->MakeTire();
  lb_tire = app->car_factory->MakeTire();
  rf_tire = app->car_factory->MakeTire();
  rb_tire = app->car_factory->MakeTire();
}

void App::run() {
  // use factory methods to create the tires
  // and engine
  Car *car = new Car(
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeTire(),
    app->car_factory->MakeEngine());
  while (car->miles < 1000000) {
    car->Drive(1000);

    // use factory method to create the
    // tires
    if (car->TireMiles() >=
        car->MaxTireMiles())
      car->ReplaceTires(
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire(),
        app->car_factory->MakeTire());
    else if (car->TireMiles() %5000 == 0)
      car->RotateTires();

    // use factory method to create the
    // engine
    if (car->EngineMiles >=
        car->MaxEngineMiles)
      car->ReplaceEngine(
        car_factory->MakeEngine());
  }
  cout << "Total miles driven: " <<
    car->miles << "\n";
}
```

Figure 4.2d: Code changes to add concrete factory

```
class CarFactory {
public:
  // virtual method added
  virtual Tire *MakeTire();

  // virtual method added
  virtual Engine *MakeEngine();
};

// PreludeFactory is now a subclass of
// CarFactory
class PreludeFactory : public CarFactory {
  ...
};
```

Figure 4.2e: Code changes to create abstract factory

```
class App {
public:
  ...
  // instance variable changed from
  // PreludeFactory to CarFactory
  CarFactory *car_factory;
  ...
}
```

Figure 4.2f: Code changes to generalize App class

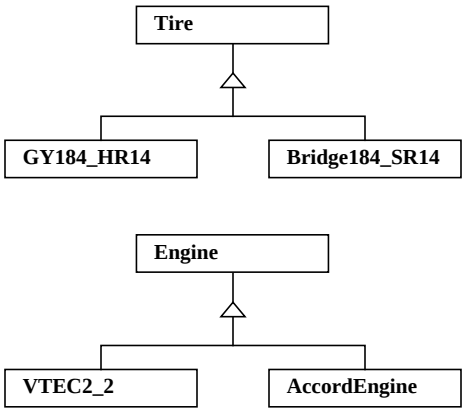


Figure 5.1: Engine and Tire subclasses added

```
  }
}
```

Note that code templates for new subclasses could be generated automatically.

5.2 Switching Classes

Switching the tires for a Prelude requires that only one factory method be modified in **PreludeFactory** (Figure 5.2). The original program would have required more than one dozen changes.

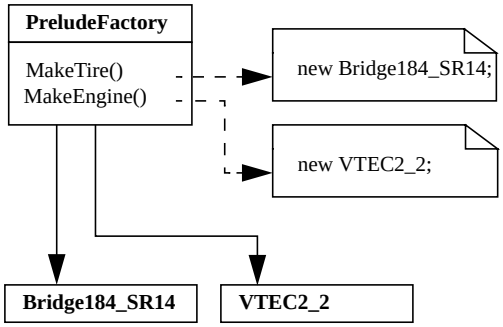


Figure 5.2: PreludeFactory altered to produce Bridge184_SR14 tires

```
class PreludeFactory : public CarFactory {
  Tire *MakeTire() {
    // tire produced by factory method changed
    // from GY184_HR14 to Bridge184_SR14
     return new Bridge184_SR14;
  }
  Engine *MakeEngine() {
    return new VTEC2_2;
  }
}
```

Note that this change could be automated by using a **FactoryMethod[ ]** transformation to redefine the **MakeTire()** method:[1]

```
FactoryMethod[PreludeFactory,
   Bridge184_SR14,MakeTire, Tire,
   app->car_factory]
```

5.3 Adding Factories

Other concrete factories can be defined to create new cars (Figure 5.3). This example creates a factory for producing Honda Accords.

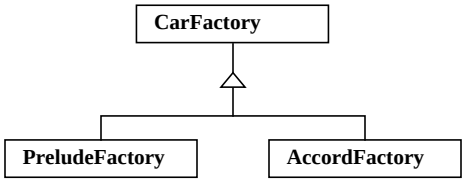


Figure 5.3: Concrete factory added

```
class AccordFactory : public CarFactory {
  Tire *MakeTire() {
    return new Bridge184SR_14;
  }
  Engine *MakeEngine() {
    return new AccordEngine;
  }
}
```

5.4 Application Reuse

Once other concrete factories have been implemented, it is easy to reuse the application for driving other cars (Figure 5.4)

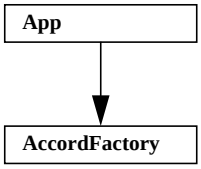


Figure 5.4: App class changed to use AccordFactory

Modifying the transformed program to create and drive an Accord (instead of a Prelude) involves a change

1. This is possible provided that pattern transformations have “overwrite” semantics: i.e., the ability to overwrite existing code with generated code.

to a single statement in the **App** class initializer:

```
car_factory = new AccordFactory;
```

The original program would have required approximately twenty error-prone changes to achieve the same result. Note that the change could also have been automated with the transformation:

```
Reassociate[App, PreludeFactory,
   AccordFactory]
```

6 An O-O Program Evolution Tool

The example presented illustrates the potential of pattern transformations applied to evolving software systems. The resulting program is more general, extensible, and reusable than the original.

A key observation is that both the checks for satisfaction of pattern transformation preconditions and the actual code transformations performed in the example appear to be automatable. Automating transformations reduces the risk of introducing new errors when upgrading software. It also allows the user to perform selected changes quickly which can increase the speed at which evolution can take place.

A second observation is that it is often easier to view the effects of transformations at the class diagram level rather than by inspecting a list of source code differences. Class diagrams document classes and frameworks. Since most transformations evolve classes and frameworks, they are a useful method for documenting the effects of transformations. Class diagrams also provide a language independent means for representing changes that occur.

Our approach to reducing software maintenance costs is to automate as much of the evolutionary process as possible. This allows users to concentrate on essential design decisions and leaves the burden of low-level source code modifications to tools. We believe that object-oriented transformations are a promising method that can be exploited. In this section, we consider a tool which evolves class diagrams and the underlying software via object-oriented transformations.

6.1 Description

The purpose of a tool would be to reduce evolutionary maintenance costs by automating the implementation of common extension mechanisms employed in object-oriented software systems.

The tool would parse source code and produce a class diagram for the system. A user could then request that transformations be applied to the class diagram to evolve the classes and frameworks. The tool would verify the enabling conditions of each transformation before applying the transformation. The underlying source code would also transformed. Source code could be automatically checkpointed after each transformation so that operations could be rolled back.

Choice of a language will affect the list of transformations provided. For example, some languages support factory methods directly. C++ was chosen as the first target language for this tool because of its widespread acceptance in industry.

A primary focus of research will be to develop a complementary set of primitives which can be composed to form design patterns occuring in the target language. More advanced parameterized transformations could also be provided which implement design patterns in a single step. For example, a transformation might directly implement the Abstract Factory design pattern given the classes involved in the pattern. The design patterns in Appendix B would be candidates for these advanced transformations.

6.2 Expected Benefits

“Designing frameworks requires a great deal of experience and experimentation” [Joh88]. Design patterns capture object-oriented design experience. Object-oriented transformations provide an evolutionary means for employing design patterns in current systems. A tool which automates object-oriented transformations would give users the freedom to experiment with different designs, something that is very difficult and costly to do today.

7 Conclusions

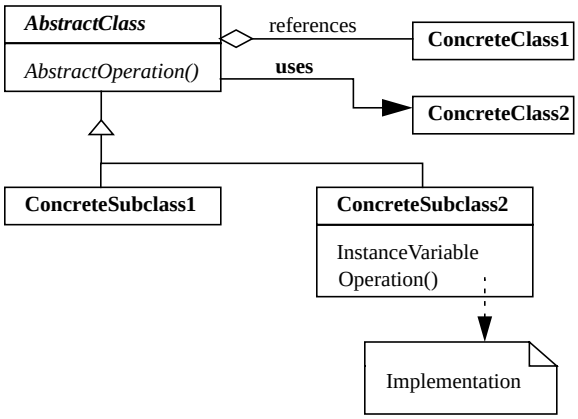
The evolutionary phase of the software lifecycle can be expected to account for up to half of a software engineering organization’s budget. One method for dealing with this cost is reuse.

Design patterns involve the reuse of designs. They are often employed in evolving systems to achieve additional degrees of flexibility or extensibility. Many design patterns can be decomposed into a set of parameterized object-oriented transformations. An example is presented to demonstrate that a program can be evolved to employ a design pattern using object-oriented transformations. It is argued that the check for enabling conditions and the actual source code transformations are automatable.

We believe it is possible to build a tool that allows users to employ design patterns in evolving systems by transforming class diagrams. Required code changes are carried out automatically which reduces the risk of introducing new errors and facilitates rapid generalization and evolution of classes, frameworks, and

programs. The result is an automated tool for producing more extensible and reusable software architectures.

8 Appendix A - O-O Notation



AbstractClass references **ConcreteClass1** if one of its class or instance variables is of type **ConcreteClass1. AbstractClass** uses **ConcreteClass2** if any **AbstractClass** method mentions **ConcreteClass2** in its type signature or uses **ConcreteClass2** local variables. **ConcreteSubclass1** and **ConcreteSubclass2** are concrete subclasses of **AbstractClass. ConcreteSubclass2** has one instance variables and one method.

9 Appendix B - Design Patterns

Object-oriented transformations could provide support for the following patterns from [Gam94]:

- Abstract Factory: Abstract Factory supports the creation of coordinated components.
- Adapter - Object: A class adapter allows an adapter object to exhibit state and behavior of another object by storing a pointer to an instance of the object.
- Adapter - Class: A class adapter allows an adapter object to exhibit state and behavior of another object by storing a pointer to an instance of the object.
- Bridge: Bridge is a fancier version of object adapter. In this pattern, the adapter and adaptee both have superclasses. The adapter superclass points to the adaptee superclass.
- Builder: Builder is analogous to Abstract Factory. A concrete factory is used to create a coordinated set of components. A builder object is used to provide some coordinated behavior for another object. A pointer to a builder is kept and the builder state and behavior is accessed through the pointer. Different builders can be swapped in to provide different behaviors.
- Command: Command objectifies methods and provides a uniform interface to them. Command objects store pointers to objects needed to implement the command (not the other way around).
- Composite: Composite creates a sibling class which stores a pointer to one or more objects of the same superclass. Some of the superclass methods may be implemented for the composite object looping through all the objects being pointed to and calling the same method.
- Exemplar: Exemplar objects have a common superclass which supports the Clone method -- a method which returns a copy of itself. An Exemplar Manager maintains a list of known Exemplar objects. New object requests are handled by the manager. Requests are parameterized by a key which specifies some exemplar known to the manager. The exemplar is then cloned and the clone is returned by the manager.
- Factory Method: In Factory Method, one class has a method which returns a new object of another class.
- Glue: Glue is an Adapter for multiple classes.
- Mediator: When many different objects need to talk to many other different object objects, it may be useful to introduce a Mediator. The mediator maintains pointers to objects on both sides. It also interprets requests and routes them to the appropriate objects.
- Solitaire: Solitaire provides a method for returning a single instance of a class. Solitaires are useful for storing global variables or one-of-a-kind objects such as Concrete Factories. Solitaires are slightly better than globals because they reduce name space pollution.
- Strategy: Strategy objectifies an algorithm. Commands store pointers to the objects on which an algorithm is implemented. In contrast, objects store pointers to strategies which implement an algorithm.
- Template Method: Template Method uses inheritance to implement some part of an algorithm.
- Walker: Walker provides an extra level of abstraction on aggregate structures. It is used to implement methods on aggregates eliminating the need to define the same methods for all aggregates.

References

[App89] Apple Computer Inc. *Macintosh Programmers*

Workshop Pascal 3.0 Reference. Cupertino, California, 1992.

[Coa92] P. Coad. Object-Oriented Patterns. In *Communications of the ACM*, V35 N9, pages 152-159, September 1992.

[Gam92] E. Gamma et. al. A Catalog of Object-Oriented Design Patterns. Technical Report in preparation, IBM Research Division, 1992.

[Gam93] E. Gamma et. al. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings, ECOOP '93*, pages 406-421, Springer-Verlag, 1993.

[Gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gol84] Adele J. Goldberg, *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.

[Joh88] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, pages 22-35, June/July 1988.

[Joh92] R. Johnson. Documenting Frameworks with Patterns. In *OOPSLA '92 Proceedings*, SIGPLAN Notices, 27(10), pages 63-76, Vancouver BC, October 1992.

[Kra88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. In *Journal of Object-Oriented Programming*, pages 26-49, August 1988.

[LaL91] W. LaLonde and J. Pugh. Subclassing != Subtyping != Is-a. In *Journal of Object-Oriented Programming*, pages 57-62, January 1991.

[Lin92] M. Linton. Encapsulating a C++ Library. In *Proceedings of the 1992 USENIX C++ Conference*, pages 57-66, Portland, Oregon, August 1992.

[Mey88] Ware Meyers. Interview with Wilma Osborne. In IEEE Software 5 (3), pages 104-105, 1988.

[Rum91] J. Rumbaugh et. al. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Wei88] A. Weinand, E. Gamma, and R. Marty. ET++ -- An Object-Oriented Application Framework in C++. In Object-Oriented Programming Systems, Languages, and Applications Conference, pages 46-57, San Diego, California, September 1988.