Some Patterns of Novice Programs

Eugene Wallingford
Dan Steinberg
Robert Duvall
Ralph Johnson

Version 2.1
August 18, 2004

Contact: Eugene Wallingford, wallingf@cs.uni.edu

Introduction

This paper describes some of the patterns we would like for novice programmers to learn when they first learn to write Java programs in a test-driven manner. We are writing these patterns in parallel with A Student Registration System, one section of a CS1 textbook we are writing, with a working title of The Essentials of Programming with Java. The goal of this project is to produce a pattern-directed textbook for the first course in computer science and a pattern language for the systems students read and write in the course. Our ultimate goal is to write a pattern language for our students.

Patterns

The patterns described here consist of several layers. First, getting started:

- Good start
- Classes from nouns

Second, writing tests:

- Test before code
- Test
- Leading test
- Specification before code

Third, identifying classes:

- Class
- One major responsibility
- Class Captures An Abstraction
- Class Must Have Behavior

Fourth, writing methods:

- Public methods first
- Method
- Delegation
- Helper method

And, finally, identifying instance variables:

- Instance variable

[*Refer to thumbnails at back...*]

Good start

You are reading a problem statement, a story that you want to implement in a program.

Starting a program is difficult.

Sometimes, a problem looks much too difficult to solve in a program. It deals with some new domain that you don't know much about, and so you feel like you have to learn about the problem area before you start writing the program. But that can be dangerous. You can spend a lot of time learning about some new area and not make progress toward your program. You probably don't need to know *everything* about the problem area, but how much is enough?

Other times, a problem looks really easy, and you think that writing the program will be a piece of cake. But when you sit down to write code, you find that your initial feeling of understanding melts away into a mess of questions about files and classes and methods and `if` statements. You are sure that you "get it", but writing the program is tough.

A program consists of a set of objects interacting to solve a problem. So you probably want to find the objects you need and then write classes to represent them. [Add a pointer to the discussion in the chapter.]

You could try to identify all the objects and classes that you need right away. That turns out to be hard. When you are working on a challenging problem, you probably won't understand it well enough at the beginning to find all the classes you'll need. And if you try to guess too far ahead of your understanding, you'll almost certainly make your job harder.

A single program can be written in an infinite number of ways. Of course, some ways are better than others. You'd like to write your program in a way that allows you to learn about the problem as you go and then feed that learning back into your program.

Classes are somewhat arbitrary. When you learn how to program well enough, you'll learn that you can move code from one class to another, you can rename classes, and you can merge and split classes. So getting all the classes "just right" right away isn't as important as it might seem.

Therefore, get a good start, implement a simple test, and you'll discover the rest.

Identify a class or two that you think will be in your program. Perhaps you can find classes from nouns in the story.

Then choose one class. Write a test before code to design some of the class's interface. Then you are ready to begin writing code, by focusing on public methods first.

Examples to write:

- start with the stories that lead to test before code's Toaster tests

- start with the stories that lead to Test's VirtualPest tests and code

---

Classes from nouns

You are getting a good start on your program by addressing a small part of the problem.

*OK, so how **do** I start?*

Usually, programs are easier to understand if the main classes in them represent concepts that are important to the users. In other words, the "objects" in the program should be the same kind of objects that users think are important, and talk about all the time. It is tricky to make program objects act like real-world objects, and it is not always easy to learn what kind of objects are really important to the user. Nevertheless, it is worthwhile to make the main objects in your system be objects that are important to the users.

A problem description usually talks about the most important concepts in the problem. The nouns in the description name things: concrete things, like people and toasters; processes, like the computation of a wage; and even abstract things, like categories. These names tell you about some of the concepts your program will need to model.

However, not all the nouns will map directly onto objects in a program. Some of the nouns will be synonyms of one another. Others name objects that are outside the scope of the system you need to model. [Describe a case of going too far...]

Therefore, start with the nouns used in the problem description as candidate names for classes.

Use this list of nouns only as candidate classes in your system. Remove duplication by eliminating synonyms and specific names. Add names of things that the description assumes must be present. Eliminate candidates that seem to be beyond the purpose of your program. Be sure that each class captures an abstraction. Also, each class must have behavior associated with it.

Test before code

You have identified a new class that needs to be created.

Starting to write a class is difficult, too. You need to know what messages the object responds to, seemingly before you know anything about the class.

You should design the interface to a class the users of the class. But when you are creating a class you don't yet have users for it. How can they use a class that doesn't exist yet?

The interface must provide all the power needed by its users, but it should also be simple and easy to learn. You can only really know this if you write the user code, too.

Therefore, write a test for the class before you write any code for the class. The test is a user of the class you are designing. This means that you can use the test to specify some intended behavior of the class, which involves sending an object a method. This message will tell you about at least one method that the class needs.

The need for user code to guide the design of an interface is captured in the classic Yogi Berra quote, "You've got to be very careful if you don't know where you're going, because you might not get there."

For example, suppose you are writing a program to model a toaster making toast. So you decide to create a new class named `Toaster`. A `Toaster` toasts certain objects of type `Bread`. If you start by sketching out everyting that a `Toaster` might do, you may come up with a class that has too many methods in its interface. Even still, you may miss something and end up with a class that does not perform some important task.

Instead, begin with a simple test like the following.

```
public void testCanSetDoneness()
{
    Toaster toaster = new Toaster();
    toaster.setDoneness(Toast.MEDIUM);
    assertEquals( toaster.getDoneness(), Toast.MEDIUM );
}
```

This small test method reflects quite a few design decisions about `Toaster`s. You have said quite a bit about the public interface with these three lines. Do not concentrate on whether thes design decisions are the "right" ones; instead, focus on what the body of `testCanSetDoneness()` implies about the `Toast` and `Toaster` classes.

- There is a `Toaster` class.
- There is a `Toast` class.
- `Toast` contains a static constant named `MEDIUM`.
- `Toaster` contains a method named `setDoneness()` that takes a parameter of type compatible with `Toast.MEDIUM`. (Although not specified, convention implies that `setDoneness()` doesn't return anything.)
- `Toaster` contains a method named `getDoneness()` that takes no arguments but returns a value of the same type as `Toast.MEDIUM`
- `Toaster` has some way of remembering the degree of doneness so that the value set by `setDoneness()` can later be retrieved with `getDoneness()`.

Once you know that you can set the doneness on a toaster, you probably want to make sure that the bread reflects that level of toasting when the `Toaster` is done. So you start to write the test.

```
public void testToastIsToastedToCorrectDegreeOfDoneness()
{
    Toaster toaster = new Toaster();
    Bread bread = new RyeBread();
    toaster.setDoneness( Toast.MEDIUM );
    assertEquals( bread.getDoneness(), Toast.NOT_TOASTED );
    toaster.toast( bread );
    assertEquals( bread.getDoneness(), Toast.MEDIUM );
}
```

This test shows that you probably have not thought through the interface clearly yet. It contains both `Bread` objects and `Toast` objects. What is the relationship between them? Maybe you should take smaller steps. Just toast a piece of `Bread`.

```
public void testToastBread()
{
    Toaster toaster = new Toaster();
    Bread bread = new RyeBread();
    assertFalse( bread.isToasted() );
    toaster.toast( bread );
    assertTrue( bread.isToasted() );
```

```
}
```

At this point you need to add...

- a `Bread` class
- an `isToasted()` method to `Bread`
- a `toast()` method to the `Toaster` class that takes a `Bread` object as an argument

Hmmm. The only thing we've used the `Toast` class for is to contain the `static final` values that indicate doneness. Maybe we can move those values to the `Toaster` class? That means changing the first test to:

```
public void testCanSetDoneness()
{
    Toaster toaster = new Toaster();
    toaster.setDoneness( Toaster.MEDIUM );
    assertEquals( toaster.getDoneness(), Toaster.MEDIUM );
}
```

We can now eliminate the `Toast` class. All of our tests should continue to run and pass.

Think back a moment at what you have done. This design of the `Toaster` and `Bread` classes should feel a lot like design in real life. You think you want something and so you describe it. Then you see the implementation of what you thought you were describing and say, "No, now that I see it, I'd like you to move that thing over there." We are doing the same thing here -- with code.

Test

You are writing test before code.

Once you have a test that tells you something about a class, you still have to write the code.

It is often difficult to start writing a class when you have no code that represents the other classes it interacts with. You have to begin somewhere and presume the existence of code you haven't yet created. When you write a lot of code that depends on the unknown -- code you haven't written yet -- you may find that the code you do write does not compile. For a long time.

You could write code for several hours without compiling and testing. That may seem like an efficent way to work! You don't have to break your mode of thinking. You can focus on writing code for a while, and then you can shift into a mode where you compile and debug for a while.

But that efficiency is probably an illusion. Your compile-and-debug phase will likely turn out to be longer than you expect, a grueling and humbling experience. The longer you write code without compiling, the more errors your code is likely to contain. And, what's worse, the errors can interact with one another to make more errors.

Taking small steps allows you to get feedback from compiling and running the program. You can learn a lot about your program that way, but you can also learn about the problem domain at the same time. The price of taking small steps is that you [... may have to tame the natural desire to race ahead ... need tools to help: JUnit].

Making a working program is fun! If you see yourself making progress, you'll feel better and be more motivated to move on.

Therefore, start with a single test case. See what objects are needed to satisfy the test. Then write just

enough code to make the test pass -- and no more!

Writing code that works is often difficult. But if you have a test that confirms the viability of your code, then you can have more confidence in the code you write. If you write your test first and then code to your test, then you can take small steps and still get where you want to go.

As an example, let's create a new VirtualPet class. You could start with the following trivial test case.

```
public void testVirtualPetExists(){
  VirtualPet pet = new VirtualPet();
  assertNotNull(pet);
}
```

As trivial as this test is, it indicates that VirtualPet is a concrete class that can be instantiated with no arguments. Create the corresponding production code:

```
class VirtualPet{
}
```

The test should now compile and pass. One advantage is that you can (if you are using JUnit) now refactor your test like this.

```
private VirtualPet pet;

protected void setUp(){
  pet = new VirtualPet();
}

public void testVirtualPetExists(){
  asserNotNull(pet);
}
```

This should also pass. Now suppose you want your pet to have a name. Add the following test.

```
public void testCanNamePet(){
  pet.setName("Igor");
  assertSame(pet.getName(), "Igor");
}
```

You make small changes to your VirtualPet class:

```
class VirtualPet{

  private String name ;

  public void setName(String name){
    this.name = name;
  }

  public String getName(){
    return name;
  }
```

```
}
```

What happens if you ask for the name of an unnamed VirtualPet? Before you write the test you have to consider what you would like to happen if you ask for the name of a pet that hasn't been named. Currently you will have a problem as the String has not been initialized. Perhaps you want to gave an empty String returned. Then test it like this.

```
public void testNameOfUnnamedPetIsEmptyString(){
  assertSame(pet.getName(),"");
}
```

This forces you to go back and initialize the String name in VirtualPet. Small tests, one at a time.

Leading test

You are writing code (finding objects?) by writing a test before code.

How do I know which test to write next?

Some tests are trivial, you know that your system ought to do them. Others are very hard, and will require a long time to implement. It is OK to write tests that you think will work; this is testing, not programming. It is not a good idea to write tests that will take a long time to implement because you will get frustrated and will go a long time with broken tests.

Therefore, pick a test that helps you know when you have successfully implemented the next piece of functionality. Your test may require some new code, but not a lot of new code. Favor easy tests over hard ones, and important operations over unimportant ones.

Sometimes it is a good idea to write a test that you know is going to fail, so that you can find problems in your tests. A test before code can be used to make a SpecificationBeforeYouCode.

Do not include implementation details too early. Think primarily about classes mentioned in the problem statement rather than implementation specific classes. It will be easier to write it later, than it is now. Just keep asking "what's next".

As an example, let's think about how we might begin implementing Conway's game of life. John Conway’s Game of Life takes place on an infinite grid with square cells. Each cell has as neighbors the eight cells that touch it. At time zero, you can set up the board in any state you wish and start the game. The state of the board at each discrete point in time is determined by the state of the board at the previous time by the following rules:

- A live cell stays alive only if it has two or three live neighbors, otherwise it dies.
- A dead cell comes to life only if it has three live neighbors, otherwise it remains dead.)

Typically, those new to programming start by creating the GUI and ensure that they are doing the right thing by clicking on a few cells and watching the board. You can start instead by implementing the Cells that are the model that underlie the GUI.

Here is a leading test to ensure that we can create a new cell and when it is created it will not be alive.

```
public void testNewCellIsDead(){
  Cell cell = new Cell();
```

```
  assertTrue(!cell.isAlive());
}
```

This leads us to create a Cell class that contains a method isAlive(). The easiest way to implement isAlive() at this point is to return false. The next smallest thing to check on is that we can set a cell to be alive.

```
public void testLiveCellIsAlive(){
  Cell cell = new Cell();
  cell.setAlive(true);
  assertTrue(cell.isAlive());
}
```

A more complicated test to set up might be this:

```
public void testLiveCellWithThreeLiveNeighborsStaysAlive(){
    cell.setAlive(true);
    for (int i = 0; i <3; i++){
      cell.incrementNumberOfLiveNeighbors();
    }
    assertTrue(!cell.needsToChange());
  }
```

Specification Before You Code

You have read the programming assignment ...

How do you transform a problem description into working code?

As you read through the initial description of what a program is supposed to do, you find you have many more questions than are answered in this initial document. Even when the prose appears to be clear and crisp, you find that it is unavoidably vague in many ways as you move to capture its meaning in code. You would like to clarify your idea of what you are being asked to do.

Therefore, write a specification before you begin to write code. This specification can be a more careful description of the solution in prose that enumerates classes and possible methods, you can begin to specify the solution with a UML (or less formal) diagram, or you can begin to write tests. The specification does not need to be complete before you start coding. You may want to try to determine classes from nouns or test before code to begin specifying the design.

Class

To be written

One major responsibility

To be written

Class captures an abstraction

To be written

Class must have behavior

A class must have some behavior that distinguishes it from other objects that currently exist before you decide it must exist.

Note, specialized comparisons represent behavior.

Public methods first

You are getting a good start. You've identified *a class* to write and have written a test before code.

Now you're ready to begin implementing the a class. What do you do next?

You could start by trying to figure out what the object needs to know, and then create the instance variables of the class. That may work, if the test you are working on is just right, but it conceals a danger: many of *a class*'s instance variables exist to support what the object does, and until you've tried to implement the object's behaviors you won't know what instance variables you need.

An object exists to serve a purpose to other objects in your program. The essential parts of any class are the methods that allow an object to do the things that other objects expect it to do.

A class may end up having private methods, but other objects won't know that. So any private methods you write will be of use only to the *other* methods in the class.

The public methods specify the behavior of the object. They implement the behavior that other objects will see. a test interacts with an object and so helps to specify the interface of the object.

Therefore, write the public methods of your class first.

Write just enough code to implement a method that satisfies a test. Create an instance variable only when you need it to implement some behavior.

Tie to test before code's Toaster tests and a test's VirtualPest tests and code.

Method

You are implementing *a class*. Perhaps you are writing the code to satisfy a test.

Objects do things, and objects know things. So a client that uses an object will send it messages asking it to do something, or asking about something it knows.

You can think of the methods in your class in terms of these two kinds of responsibility: methods that do things and methods that report what the object knows.

People find a program easier to understand when they can quickly tell whether a method is doing something or reporting something.

Sometimes you will want to write a method that does both. But consider the object that expects the

method only to report something. It may send the message and then be surprised that that state of the receiver has changes. Or consider the object that expects the method only to do something. It may send the message and not realize that the value returned carries important information -- such as whether the operation succeeded or not.

Therefore, design methods so that they either do something or return something, but not both.

If a method does something, give it a name that sounds like a command. Make its return type `void`.

If a method reports something, give it a name that describes its result. If its result is a boolean, then name it with a question or assertion. Make its return type match that of its result.

Implement the method body in one of several ways: *{is this too soon?}*

- Use Delegation to ask another *Object* to do part of the work.
- Use a *polymorphic message* to select among alternatives, or make a *choice*
- Use *helper method*s to implement several actions, to allow for future refinement of part of the method, or to document the meaning of the method body.

Examples: write code to implement test before code's Toaster tests.

Delegation

You are writing a method, probably to satisfy a test.

Each behavior in your program should reside in one place.

A program is a set of objects collaborating to solve a problem. Each object should do what it does best, and let other objects do what they are good at.

If two objects do the same thing, then you will have to write code that is identical or similar in two different parts of your program. Duplication of this sort is bad.

- When you decide to implement the behavior in a different way, then you have to remember to change it in two or more places.
- Programmers will have a harder time understanding your program, because they will see similar code in multiple places.

Therefore, ask another object to do a task if it knows how. Write new code (only?) for the part of the behavior that the other object cannot do.

The class has to know the name of the object to ask. You may store this reference in an instance variable. This is especially useful if you need to ask this object to perform a service many times. You may also require the sender of the message to pass an object along with its message. This is helpful when the sender knows better which object can help.

Helper method

To be written

Instance variable

Exposition assumed by this pattern:

- A method defines the behavior that an object uses to respond to a message. Instance variables represent the state of the object that (potentially) changes over its lifetime in support of its behavior.

- How does an object know things?
  - It knows whom to ask. (Delegation)
  - It knows it directly. (Instance Variable)
  - It computes it from other things it knows. (?????)

You are implementing a Class with one or more Methods.

When do I create an instance variable?

Instance variables should always be private, so outside users won't be able to tell whether a variable exists inside a class. Sometimes it is hard to tell whether an object has an instance variable. Perhaps the value is recomputed each time you need it. For example, perhaps the value is stored in another object. Are grades stored in a course for each student, or in a student for each course? Or maybe grades are stored in a gradebook, and not stored in either the student or the course.

Instance variables generally complicate the understanding of the class; so, create instance variables reluctantly. Sometimes, having an instance variable can simplify the interaction between methods.

Therefore, create an instance variable when multiple public methods need to share the same data, and the history of that data is important to all those methods.

Two common kinds of behaviors that require an object to maintain state are parameterized behavior and history-dependent behavior. A parameterized behavior is one in which one object differs from another on the basis of soem value, but that value doesn't change over time. Responding to a request for an object's name may be an example of parameterized behavior. A history-dependent behavior is one in which the response to a message depends on previous actions by the object or on previous requests made of it. Being able to answer queries about the contents of a collection depends on what objects have been previously added to the collection. This is an example of history-dependent behavior.

Thumbnails

This section will present thumbnails for patterns to which we refer but which we do not document fully in this paper.

References

This section will list other references for the reader.

Joe Bergin's *Coding at the Lowest Level: Coding Patterns for Java Beginners*, tells us a lot about what to do next after applying the patterns in this paper. See:
< http://csis.pace.edu/~bergin/patterns/codingpatterns.html>