1 Introduction

This booklet provides you with some values, principles and design patterns to follow when writing your Java programs. These are based on “Implementation Patterns” by Kent Beck [2]. See the book “Objects First With Java” [1] for code examples and try to identify examples of the patterns.
There are three sections to this document, values, principles and patterns. The values give us some rules to always follow when writing code. The principles give us some rules to follow when trying to make decisions about the code we are writing. The patterns tell us how to implement something.
Software is always changing. We change code that has already been written for many reasons; when we maintain it, when we add new functionality, when we see a way to make the code easier to maintain. There is never a day when you can say “This software application is finished”. Following these values and principles lead to time savings and improved software design.

2 Values

There are three values that we follow when writing program code. These are communication, simplicity and flexibility.

1. **Communication:** Your code should be readable. You and other people need to be able to read your code while it is being developed and later when it is being maintained. Code is read more often than it is written. If your code is hard to explain, try to simplify it.

2. **Flexibility:** Code that can be easily understood is more flexible because it adds more options for changing it. Don’t add flexibility based on speculative design, by which I mean additional requirements that you think might be useful in future.

3. **Simplicity:** Keeping things simple makes things easier to understand. Some ways of keeping things simple are:

   (a) Look back on what you have written and try to simplify it.
   (b) Challenge requirements to find those that are essential.
   (c) Eliminate parts of code that do not add new information until nothing else can be deleted without losing information.

3 Principles

3.1 Put logic and data together

When we refer to data we mean variables or fields with data values (ie. inside an object). When we refer to logic we mean lines of program code, for example to implement a method or several methods.
Put the logic and the data that goes with it into the same method or class. Sometimes its not obvious where data and logic should go and you may find that they are in the wrong place. In this case there are four options:

1. Move the logic to the data

2. Move the data to the logic
3. Put the data and the logic into a helper class. Helper classes have a restricted role to perform certain calculations that perhaps occur in several places. They act as a central location for certain data and logic.
4. Don't know how to fix it. This is the case when the solution is not obvious and has to be derived after some thought or problem solving. The eventual solution will involve some combination of the other options. Note that the solution is not "give up".

3.2 Minimize repetition

Repeated code causes the programmer more work. Duplication is not always visible until later, see the value "Simplicity" point 3a above.
If you have the same code repeated in many places, then changing one occurrence usually means changing them all and that adds the challenge of finding all occurrences of a repeated section of code. Miss one and you might introduce a bug.

3.3 Declarative expression

We value communication so write code that explains what is going on. Don't make your reader construct complicated models in her head about what your code does. For each complicated step, start by giving it a name that describes what the implementation of that step will achieve. Use the name to create a private method and put the complicated code into the private method. Use the name again to call that private method.
Private methods exist to hide the details of a complicated or lengthy section of code. If we put all of the code into one place, the reader needs to build a big model in their head to understand it all. Therefore, break complicated code down into smaller parts and move the smaller parts into private methods.

3.4 Structure code so that changes have local consequences

If we need to change something in our code, we are aiming to limit the effects of that change to the same method or the same class.
This principle reduces the cost of making changes to your code. It also makes your code easier to understand because it allows individual sections to be understood before we need to move on an understand the next section of code.

3.5 Symmetry

Symmetry in code is conceptual and refers to the concepts you are implementing. Express the same idea in the same way everywhere it occurs. Do not mix concrete and abstract in the same method (check this in the book).

3.6 Rate of change

Sometimes we write code that implements regulations or rules that sometimes change (eg tax laws). If the regulation changes, the code needs to also change. Implementation details that are subject to change at the same rate should be grouped together.

4 Patterns for classes

4.1 Simple Class Name

Scenario: I want to find a name for my class.

Why: You want to communicate the role this class plays in your application. Class names are important.

Considerations: Sometimes good names are difficult to find. You may have to change the name later when you think of a better name for the class.

Solution: Pick a strong metaphor for the computation using as few words as possible. For example `Figure` is better than `DrawingObject`. Use a thesaurus if you have difficulty finding a name. Conversation is a good tool to find better names, so explain the purpose of the class to someone.

4.2 Class

Scenario: I have identified a component I want to model, it has some data and it has some behaviour.

Why: Data changes more frequently than logic. Each class is a declaration that says “This logic is fixed and goes with this data. These data values go together and are changed by the logic.” We effectively separate the declaration of the data and the logic, which is fixed, from the data values which change. Values are only assigned to the data when we construct an object from the class.

Considerations: Classes are relatively expensive design elements so a class should do something significant.

Solution: Create a class. It needs a name that describes its role. It needs data fields. It needs a constructor to initialize its data fields when we create an object from that class. It needs methods that implement the behaviour of objects constructed from this class. Classes are usually declared as `public`, followed by the keyword `class` and then the name of the class. Curly brackets are used to delineate the start and end of the code inside the class. For example (from Chapter 2 of [1]).

```
public class TicketMachine {
  // data fields
  // constructor
  // methods
}
```

5 Patterns for Object State

5.1 Role-Suggesting Name

Scenario: I want to hold some data in a field or variable. What name should I give to the variable.

Why: Communicate the role a variable plays through its name. Everything else, lifetime, scope and type are communicated through context.

Considerations: Avoid the temptation to abbreviate variable names. Variable names are read many times for each time they are written, abbreviating is a false economy. Optimize names for readability.

Solution: Come up with a short name. Don't include implementation detail such as the type or scope in the name as that is found by reading the code. If you struggle to find a name, it means you don't understand the role the variable plays. See also the simple class name pattern in Section 4.1.

5.2 Field

Scenario: I have a piece of data that is a component of a class.

Why: A field can be used to hold data or another object "owned" by this object.

Considerations: Make sure you have a name for a field before you try to declare it. See the role-suggesting name pattern in Section 5.1.

Solution: All fields should be declared `private`, with their data type and their name. For example, the data fields of a class to represent a circle may be defined as follows:

```java
public class Circle
{
    // The diameter of a circle
    private int diameter;
    // The color of the circle
    private String color;
    // Whether or not the circle is visible on the canvas
    private boolean isVisible
    // X coordinate of the center
    private int centerX;
    // Y coordinate of the center
    private int centerY;

    // constructor and methods not shown
}
```

Note that the type of a field may be a class. For example, if I create a class to represent a picture and I use a circle to show the sun, I would declare the class and field like this:

```java
public class Picture {
  // a circle to represent the sun
  private Circle sun;

  // the rest of the class is not shown
}
```

5.3 Parameter

Scenario: I need to communicate some data values to a method, so that the method can use those values in its computation.

Why: Often, when we write the code, we don't know the value of some of the data involved in the computation.

Considerations:

Solution: We use a method with a parameter. Assume we wanted to move a point horizontally across a canvas. We might implement that by adding some distance value to the X coordinate of the point. We have two options when it comes to implementing that. Option 1 is to write code that changes the X coordinate by a fixed amount. Option 2 is to use a parameter to allow the caller to specify what value should be added to the X Coordinate. For example:

```java
public class Point
{
    // X coordinate of the point
    private int xCoordinate;
    // Y coordinate of the point
    private int yCoordinate;

    // constructor not shown

    // Option 1: change by a fixed value
    public void moveHorizontal() {
      xCoordinate = xCoordinate + 10;
    }

    // Option 2: change using a variable value
    //communicated using a parameter
    public void moveHorizontal(int distance) {
      xCoordinate = xCoordinate + distance;
    }
}
```

The version without the parameter is less powerful because it only does one thing and it does that same thing every time – add 10 to the value of the data field representing the X coordinate. Occasionally that is what we need but in this example the version with the parameter is more flexible. It can also add 10 to the value of the data field representing the X coordinate, but significantly it can also add any other value too depending on the value we give to the parameter.

```java
Point p = new Point();
// the next line adds 10 to the X coordinate of the point
p.moveHorizontal();
// the next line adds 10 to the X coordinate of the point
p.moveHorizontal(10);
// the next line adds 55 to the X coordinate of the point
p.moveHorizontal(55);
```

5.4 Local Variable

Scenario: I need to store a data value as part of a computation in a method. Reasons may be:

- I need to keep a count
- I need to simplify a complicated expression
- I need to collect a data value for use later
- I need to store a value to avoid calculate it several times in the same calculation
- I need to hold one element from a collection of elements.

Why: Our first Principle is to put logic and data together. Following that principle, declare local variables just before they are used and in the innermost possible scope.

Considerations: None

Solution: Declare a local variable at the point you need it. For example, from Chapter 2 of Objects First with Java[1], refunding the balance from the ticket machine requires a local variable to be declared to temporarily hold the amount to refund:

```
public class TicketMachine {
    // The amount of money entered by a customer so far.
    private int balance;

    public int refundBalance() {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }

    // the rest of the code is omitted
}
```

In Chapter 3 of Objects first with Java, there is an example of a Mail system that contains a class called `MailClient` which is responsible for reading and sending mail:

```
public class MailClient
{
    // The server used for sending and receiving.
    private MailServer server;
    // The user running this client.
    private String user;

    // constructor ommitted

    public void printNextMailItem()
    {
        MailItem item = server.getNextMailItem(user);
        if(item == null) {
            System.out.println("No␣new␣mail.");
        }
        else {
            item.print();
        }
    }

    public void sendMailItem(String to, String message)
    {
        MailItem item = new MailItem(user, to, message);
        server.post(item);
    }
}
```

The methods to print a mail item and send a mail item both declare a local variable to reference an item of mail and then use that data later in the method.

In the same project, a class called `MailServer` uses local variables to keep a count and to hold elements from a collection of elements:

```
public class MailServer
{
    // Storage for the arbitrary number of mail
    // items to be stored on the server.
    private List<MailItem> items;

    public int howManyMailItems(String who)
    {
        int count = 0;
        for(MailItem item : items) {
            if(item.getTo().equals(who)) {
                count++;
            }
        }
        return count;
    }
}
```

6 Patterns for Methods

Program code is arranged into methods for a number of reasons[2]:

1. Putting all the code in one huge block is difficult to read which breaks Value 1 which says our code should be readable.
2. Most problems encountered during programming are not unique. Putting a solution to a problem into a method provides an easy way to re-use that solution by calling the same method as many times as you need it. You can't do that if all your code is in one huge block. This is consistent with Value 2 that says our code should be flexible.
3. Dividing our code into methods is consistent with Value 3 because it simplifies our code. We put logic that goes together into the same method. We put unrelated bits of logic into different methods. Doing this requires us to THINK about what we are writing before we actually write the methods.

6.1 Intention Revealing Name

Scenario: I am writing a method, what name should I give it?

Why: Think about where the method call appears in the code. The reader needs to know why this method was called and not some other method. The name of the method should provide that information.

Considerations: Kent Beck says[2] “Think about methods’ names based on how they look in calling code. That’s where readers are likely to first encounter the name. Why was this method invoked and not some other? That is a question that can profitably be answered by the name of the method. The calling method should be telling a story. Name methods so they help tell the story.”

Solution: Use a method name that conveys the purpose of that method. For example if we have a class called `AllCustomers` that provides a method that returns the details of a specific customer, a good name for that method would be `find`. We could call it `findCustomer` but the context (ie that the method is found in the `AllCustomers` class) tells us that we are finding a customer. This example also uses the local variable pattern (see Section 5.4 above) and the method return type pattern (see Section 6.3 below):

```
public class Customer {
  private String id;
  // other fields and methods not shown
}

public class AllCustomers {
  // fields and other methods not shown

  /**
   * Find the customer with the given customer ID
   */
  public Customer find(String customerID) {
    Customer customer;
    // details of how the customer is found are not shown
    // but eventually a value will be assigned to customer
    customer = ...
    // the method eventually returns a reference to the
    // customer that we are looking for
    return customer;
  }
}
```

The use of the `find` method allows our calling code to tell the story:

```
AllCustomers customers = ...
// find the customer with the ID c007
Customer theOneIWant = customers.find("c007");
```

6.2 Method Visibility

Scenario: I am writing a method, should I make it `public` or `private`?

Why: Making a method public says “this method will be useful to other programmers outside of this class”. Making a method private says “no one else needs to use this method”.

Considerations: If we change a method by renaming it or removing it, we need to find all places where that method is called and change them. Public methods may be called by other people who use this code and we may not have access to that code. Think carefully before you make a method public. Public methods provide the interface for other programmers to use your class. You need to think about what makes a good interface. There are more visibility options than `public` and `private`, but those are the two we consider here.

Solution: Declare a method as `public` when you are sure that for the foreseeable future, the method will be useful outside of the class. Making a method `public` means you accept responsibility to maintain it and let everyone know if you change it. Otherwise make the method `private`.

6.3 Method Return Type

Scenario: I am writing a method, should the method return something and, if so, what should it return.

Why: Some methods are designed to perform some calculation and return the result. These are sometimes called functions. Other methods, perform some calculation and store the result in a field. These methods are sometimes called procedures. inside the object.

Considerations: Return types often change as we evolve our code. Think carefully about what you are returning.

Solution: A mutator method will have a return type of `void` meaning “nothing is returned. This design is used because the mutator method stores its result by changing the state of the object. An accessor method will return a value of a specific type. The actual return type will depend on what data is being returned. This design is used when the result of the calculation is required in some other object. For examples of the method return type pattern see the setting method pattern in Section 6.6 and the getting method pattern in Section 6.5.

6.4 Complete Constructor

Scenario: When I am creating an object, which fields should I initialise.

Why: It is important to ensure that an object is always in a valid state. That means that all its data fields should have valid initial values so that it is ready to use.

Considerations:

Solution: Assign a valid initial value to every data field in the constructor. For example, in class that represents an item of mail (see chapter 3 of [1]), the data fields are the address of the sender, the address of the recipient and the message itself:

```
public class MailItem
{
    private String from;
    private String to;
    private String message;

    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }
}
```

Now when we are constructing a new `MailItem` we know what data should be provided in order to get an object that is ready to use without having to do any further initialisation:

```
MailItem item = new MailItem("John",
                             "Mary",
                             "There␣is␣a␣meeting␣today");
```

6.5 Getting Method

Scenario: I have an object and I want to get information about its state. By convention, in Java these methods have a name that is prefixed with the word “get”

Why: Data fields are private so if you have a need to use the value of a data field, you need a method to give you that value.

Considerations: Think about whether you need a public “get” method. Following the principle that data and logic should be together, think about whether or not the logic that goes with the get method should be in this class or whether the data field should move to the class that calls the “get” method.

Solution: Write a method that returns the value of a data field in the object. The return type (see the method return type pattern in Section 6.3) of the method should be the same type as the data field:

```
public class Person {
  private String name;

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

When using the `Person` class we can now write the following to obtain the value of the `name` data field. `Person` class:

```
Person person = ... //obtain a person object from somewhere
System.out.print("This␣person's␣name␣is␣");
System.out.println(person.getName());
```

6.6 Setting Method

Scenario: I want to change the value of one of the data fields that make up an object's state.

Why: An object has data fields that hold values. It is not unreasonable that from some objects those values should be changable.

Considerations: Not all objects need to allow data fields to be changed.

Solution: Create a method with the same name as the data field, prefixed with the word "set". The set method needs a parameter (see the parameter pattern in Section 5.3 above) because the new value needs to be communicated to the method. Here is an example:

```
public class Person {
  private String name;

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

The set method is used in this way:

```
Person person = ... //obtain a person object from somewhere
person.setName("Sarah");
```

References

[1] David J. Barnes and Michael Kölling. *Objects First with Java—A Practical Introduction using BlueJ*. There is a web site for the book. `http://www.bluej.org/objects-first/` – last viewed Sept 2013. Prentice Hall / Pearson Education, 2012. ISBN: 978-0-13-283554-1.

[2] Kent Beck. *Implementation Patterns*. See an interview with the author about the book here: `http://www.infoq.com/interviews/beck-implementation-patterns` – last viewed Sept 2013. Addison–Wesley, 2008. ISBN: 978-0-32-141309-3.