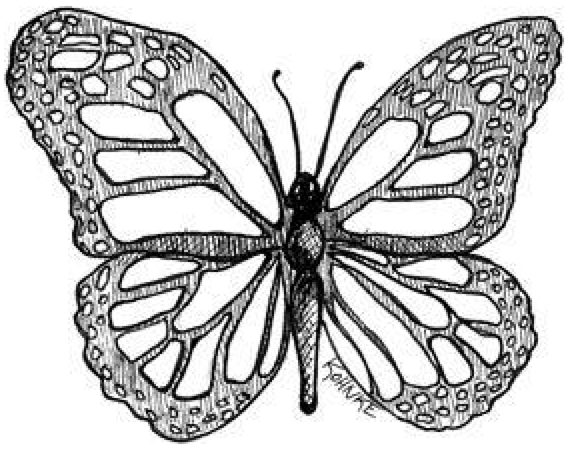
12 Emergence

by Jeff Langr



Getting Clean via Emergent Design

What if there were four simple rules that you could follow that would help you create good designs as you worked? What if by following these rules you gained insights into the structure and design of your code, making it easier to apply principles such as SRP and DIP? What if these four rules facilitated the *emergence* of good designs?

Many of us feel that Kent Beck's four rules of *Simple Design*[1] are of significant help in creating well-designed software.

According to Kent, a design is "simple" if it follows these rules:

- Runs all the tests
- Contains no duplication
- Expresses the intent of the programmer
- Minimizes the number of classes and methods

The rules are given in order of importance.

Simple Design Rule 1: Runs All the Tests

First and foremost, a design must produce a system that acts as intended. A system might have a perfect design on paper, but if there is no simple way to verify that the system actually works as intended, then all the paper effort is questionable.

A system that is comprehensively tested and passes all of its tests all of the time is a testable system. That's an obvious statement, but an important one. Systems that aren't testable aren't verifiable. Arguably, a system that cannot be verified should never be deployed.

Fortunately, making our systems testable pushes us toward a design where our classes are small and single purpose. It's just easier to test classes that conform to the SRP. The more tests we write, the more we'll continue to push toward things that are simpler to test. So making sure our system is fully testable helps us create better designs.

Tight coupling makes it difficult to write tests. So, similarly, the more tests we write, the more we use principles like DIP and tools like dependency injection, interfaces, and abstraction to minimize coupling. Our designs improve even more.

Remarkably, following a simple and obvious rule that says we need to have tests and run them continuously impacts our system's adherence to the primary OO goals of low coupling and high cohesion. Writing tests leads to better designs.

Simple Design Rules 2–4: Refactoring

Once we have tests, we are empowered to keep our code and classes clean. We do this by incrementally refactoring the code. For each few lines of code we add, we pause and reflect on the new design. Did we just degrade it? If so, we clean it up and run our tests to demonstrate that we haven't broken anything. *The fact that we have these tests eliminates the fear that cleaning up the code will break it!*

During this refactoring step, we can apply anything from the entire body of knowledge about good software design. We can increase cohesion, decrease coupling, separate concerns, modularize system concerns, shrink our functions and classes, choose better names, and so on. This is also where we apply the final three rules of simple design: Eliminate duplication, ensure expressiveness, and minimize the number of classes and methods.

No Duplication

Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity. Duplication manifests itself in many forms. Lines of code that look exactly alike are, of course, duplication. Lines of code that are similar can often be massaged to look even more alike so that they can be more easily refactored. And duplication can exist in other forms such as duplication of implementation. For example, we might have two methods in a collection class:

```
   int size() {}
boolean isEmpty() {}
```

We could have separate implementations for each method. The `isEmpty` method could track a boolean, while `size` could track a counter. Or, we can eliminate this duplication by tying `isEmpty` to the definition of `size`:

```
   boolean isEmpty() {
  return 0 == size();
}
```

Creating a clean system requires the will to eliminate duplication, even in just a few lines of code. For example, consider the following code:

```
   public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
    return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

  RenderedOp newImage = ImageUtilities.getScaledImage(
    image, scalingFactor, scalingFactor);
  image.dispose();
  System.gc();
  image = newImage;
 }
 public synchronized void rotate(int degrees) {
   RenderedOp newImage = ImageUtilities.getRotatedImage(
      image, degrees);
   image.dispose();
   System.gc();
   image = newImage;
 }
```

To keep this system clean, we should eliminate the small amount of duplication between the `scaleToOneDimension` and `rotate` methods:

```
 public void scaleToOneDimension(
     float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
     return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
  replaceImage(ImageUtilities.getScaledImage(
     image, scalingFactor, scalingFactor));
 }
 public synchronized void rotate(int degrees) {
   replaceImage(ImageUtilities.getRotatedImage(image, degrees));
 }
 privatex void replaceImage(RenderedOp newImage) {
   image.dispose();
   System.gc();
   image = newImage;
 }
```

As we extract commonality at this very tiny level, we start to recognize violations of SRP. So we might move a newly extracted method to another class. That elevates its visibility. Someone else on the team may recognize the opportunity to further abstract the new method and reuse it in a different context. This "reuse in the small" can cause system complexity to shrink dramatically. Understanding how to achieve reuse in the small is essential to achieving reuse in the large.

The TEMPLATE METHOD[2] pattern is a common technique for removing higher-level duplication. For example:

```
 public class VacationPolicy {
public void accrueUSDivisionVacation() {
  // code to calculate vacation based on hours worked to date
  // …
  // code to ensure vacation meets US minimums
  // …
  // code to apply vaction to payroll record
  // …
}

public void accrueEUDivisionVacation() {
  // code to calculate vacation based on hours worked to date
  // …
  // code to ensure vacation meets EU minimums
  // …
  // code to apply vaction to payroll record
  // …
}
}
```

The code across `accrueUSDivisionVacation` and `accrueEuropeanDivisionVacation` is largely the same, with the exception of calculating legal minimums. That bit of the algorithm changes based on the employee type.

We can eliminate the obvious duplication by applying the TEMPLATE METHOD pattern.

```
 abstract public class VacationPolicy {
public void accrueVacation() {
  calculateBaseVacationHours();


  alterForLegalMinimums();
  applyToPayroll();
  }

  private void calculateBaseVacationHours() { /* … */ };
  abstract protected void alterForLegalMinimums();
  private void applyToPayroll() { /* … */ };
}
public class USVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // US specific logic
  }
}

public class EUVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // EU specific logic
  }
}
```

The subclasses fill in the “hole” in the `accrueVacation` algorithm, supplying the only bits of information that are not duplicated.

Expressive

Most of us have had the experience of working on convoluted code. Many of us have produced some convoluted code ourselves. It’s easy to write code that *we* understand, because at the time we write it we’re deep in an understanding of the problem we’re trying to solve. Other maintainers of the code aren’t going to have so deep an understanding.

The majority of the cost of a software project is in long-term maintenance. In order to minimize the potential for defects as we introduce change, it’s critical for us to be able to understand what a system does. As systems become more complex, they take more and more time for a developer to understand, and there is an ever greater opportunity for a misunderstanding. Therefore, code should clearly express the intent of its author. The clearer the author can make the code, the less time others will have to spend understanding it. This will reduce defects and shrink the cost of maintenance.

You can express yourself by choosing good names. We want to be able to hear a class or function name and not be surprised when we discover its responsibilities.

You can also express yourself by keeping your functions and classes small. Small classes and functions are usually easy to name, easy to write, and easy to understand.

You can also express yourself by using standard nomenclature. Design patterns, for example, are largely about communication and expressiveness. By using the standard pattern names, such as COMMAND or VISITOR, in the names of the classes that implement those patterns, you can succinctly describe your design to other developers.

Well-written unit tests are also expressive. A primary goal of tests is to act as documentation by example. Someone reading our tests should be able to get a quick understanding of what a class is all about.

But the most important way to be expressive is to *try*. All too often we get our code working and then move on to the next problem without giving sufficient thought to making that code easy for the next person to read. Remember, the most likely next person to read the code will be you.

So take a little pride in your workmanship. Spend a little time with each of your functions and classes. Choose better names, split large functions into smaller functions, and generally just take care of what you've created. Care is a precious resource.

Minimal Classes and Methods

Even concepts as fundamental as elimination of duplication, code expressiveness, and the SRP can be taken too far. In an effort to make our classes and methods small, we might create too many tiny classes and methods. So this rule suggests that we also keep our function and class counts low.

High class and method counts are sometimes the result of pointless dogmatism. Consider, for example, a coding standard that insists on creating an interface for each and every class. Or consider developers who insist that fields and behavior must always be separated into data classes and behavior classes. Such dogma should be resisted and a more pragmatic approach adopted.

Our goal is to keep our overall system small while we are also keeping our functions and classes small. Remember, however, that this rule is the lowest priority of the four rules of Simple Design. So, although it's important to keep class and function count low, it's more important to have tests, eliminate duplication, and express yourself.

Conclusion

Is there a set of simple practices that can replace experience? Clearly not. On the other hand, the practices described in this chapter and in this book are a crystallized form of the many decades of experience enjoyed by the authors. Following the practice of simple design can and does encourage and enable developers to adhere to good principles and patterns that otherwise take years to learn.

Bibliography

[XPE]: *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.