3

First Principles

If you read this chapter and grasp all the principles within, then you'll be 90% of the way toward cleaning code. The ideas presented here are old and well established. They have stood the test of time and should be part of every programmer's regimen of discipline.

Keep in mind that these are not strict laws. They are not rules that must be followed. They are, more or less, guidelines. Each should be applied in the context of the problem at hand, and every programmer should feel free to take whatever license is necessary to solve that problem.

But, when all else is equal, and all the variables are averaged out, these are the principles that I believe emanate from well-cleaned code.

If, as you read what follows, you encounter concepts that are foreign or novel, be assured that all will be explained in subsequent chapters. Indeed, when you have finished reading this book, you may wish to complete the circle by rereading this chapter.

Everything Small, Well Named, Organized, and Ordered

If you stop now and read nothing more than that heading, you'll have much of the essence of what it takes to clean code.

Keep everything small. Choose names for those small elements that communicate to other programmers. Define and maintain a structure and organization that present the software in such a way that others can easily consume it. Keep the elements within that organization well-ordered so that one concept follows another in a rational and sensible way.

Remember that your own understanding of the code will likely bias your attempts at providing good names and structures; so work hard to put yourself in the headspace of those who do not understand the code.

Functions

Functions should be small. Most should be just a handful of lines. This allows them to have nice, descriptive names and promotes the separation of concerns. It ensures that each function does one, and only one, thing.

Let's start with the name. If you see a snippet of lines that does something that can be reasonably named, then that snippet should be moved into a function named for what that snippet does. The name should be a verb that says what the function does and that allows the calling sequence to read "like well-written prose."[1]

---

1. Grady Booch, private email.

As an example, consider this function, adapted from the `JCommon` library. It finds the date of a specified weekday before a given date. The weekday codes are integers 1 (Sunday) through 7 (Saturday).

```
public static Date getPreviousDayOfWeek(int weekday,
  // check arguments…
  if (!isValidWeekdayCode(weekday)) {
    throw new IllegalArgumentException(
      "Invalid day-of-the-week code."
    );
```

```
    }

    // find the date…
    final int adjust;
    final int baseDOW = from.getDayOfWeek();
    if (baseDOW > weekday) {
      adjust = Math.min(0, weekday - baseDOW);
    } else {
      adjust = -7 + Math.max(0, weekday - baseDOW);
    }

    return Date.addDays(adjust, from);
  }
```

There are at least two snippets of code that we could extract into smaller and better-named functions. And those extractions will lead to some simplifications.

```
  public static Date getPreviousDayOfWeek(int weekday,
    checkWeekdayArgument(weekday);
    return addDays(-daysBefore(weekday, from), from);
  }

  private static void checkWeekdayArgument(int weekday)
    if (!isValidWeekdayCode(weekday))
      throw new IllegalArgumentException(
        "Invalid day-of-the-week code.");
  }

  private static int daysBefore(int targetWeekday, Date
    int fromWeekday = from.getDayOfWeek();
    int diff = fromWeekday - targetWeekday;
    return diff + (diff > 0 ? 0 : 7);
  }
```

Notice how the simplified `getPreviousDayOfWeek` function reads. It's short, and it's obvious. If you need to know more, all you have to do is look down to see the extracted functions. However, most often you would not need to know more. You'd simply look at that function, agree with it, and move on. It may not read like the prose of a Crichton novel, but it reads pretty well.

Future Bob:

I don't like the leading minus sign. It would have been better to create a `subtractDays` function in order to get rid of it. That's a small thing, but small things matter.

A More Significant Example

Here I present the Uncle Bob Conference Room system. Scan through it. It's not very complicated—probably.

```
——Statement.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class Statement {
  public enum CatalogItem {SMALL_ROOM, LARGE_ROOM,
                           PROJECTOR, COFFEE, COOKIES

  public record RentalItem(CatalogItem type,
                           int days,
                           int unitPrice,
                           int price,
                           int tax) {
  }

  public record Totals(int subtotal, int tax) {
  }

  private String customerName;
  private int subtotal = 0;
  private int tax = 0;
  private List<RentalItem> items = new ArrayList<>();

  public Statement(String customerName) {
    this.customerName = customerName;
  }

  public void rent(CatalogItem item, int days) {
```

```
  int unitPrice = switch (item) {
    case SMALL_ROOM -> 100;
    case LARGE_ROOM -> 150;
    case PROJECTOR -> 50;
    case COFFEE -> 10;
    case COOKIES -> 15;
  };

  boolean eligibleForDiscount = switch (item) {
    case SMALL_ROOM, LARGE_ROOM -> days == 5;
    case PROJECTOR, COFFEE, COOKIES-> false;
  };

  int price = unitPrice * days;

  if (eligibleForDiscount) price = (int) Math.round

  subtotal += price;
  int thisTax = switch (item) {
    case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
         (int) Math.round(price * .05);
    case COFFEE,COOKIES -> 0;
  };
  tax += thisTax;
  items.add(new RentalItem(item, days, unitPrice, p
}

public RentalItem[] getItems() {
  List<RentalItem> items = new ArrayList<>(this.ite
  boolean largeRoomFiveDays = items.stream().anyMat
    item -> item.type() == CatalogItem.LARGE_ROOM &
  boolean coffeeFiveDays = items.stream().anyMatch(
    item -> item.type() == CatalogItem.COFFEE && it
  if (largeRoomFiveDays && coffeeFiveDays)
    items.add(new RentalItem(CatalogItem.COOKIES, 5
  return items.toArray(new RentalItem[0]);
}

public String getCustomerName() {
  return customerName;
}

public Totals getTotals() {
  return new Totals(subtotal, tax);
```

```
    }
  }
```

I'm sure you figured it out. Users can rent small rooms, large rooms, coffee, projectors, and cookies. (How do you rent cookies?) Small rooms rent for $100 per day. Large rooms are $150. You get a 10% discount if you rent a room for the whole week. There's a 5% (rounded up) tax on all nonfood items. Coffee is $10 per day. Cookies are $15 per day. Projectors are $50 per day. And if you rent a large room and coffee for a whole week, you get one day of free cookies.

The `Statement` class represents a sales statement that includes each item rented, the price of the item, the number of days rented, the subtotal, and the tax. The `Statement` also contains the subtotals of all the items and all the tax.

Easy peasy.

This is not too hard to understand, but you can tell that it was slapped together. It's a bit unkempt, but not terrible. We could probably clean it up a bit, but would that be worth the effort?

But let me ask you. Have you ever seen a system that started out simple and then got much, much more complicated? If you've been in this business for more than a year or so, I *know* you have. Indeed, if you've been in the business for five years or more, you've likely seen a system that began a bit unkempt but grew into a horrible mess, becoming a living hell for the developers and the organization.

So let's make this example a bit more real. You don't suppose that this program started out in its current form, do you? No, at first it just rented one small room. Then, many small rooms. Then, large rooms with a different unit price. Then, coffee, with no sales tax. Then, the discount for a full week was added. And then cookies, and the bonus for renting a large room for a week.

In other words, this program *grew*.

Let's suppose that we've just gotten back from an all-hands meeting at which the CEO gave us a glowing report about how the company is

expanding. There are new markets, new states, new laws to contend with. There will be new discounts and new promotions, and new tax regulations. It's going to be great for the business; but you look at this little `rent` function and you think … it's going to continue to grow, isn't it? And as it grows, it will become ever more tangled and confusing. It will degrade, like a piece of rotting meat.

So let's stop that from happening. Let's figure out a way to get ahead of the coming business growth and let this function grow without rotting from the inside out. Let's tidy it up first.[2]

---

2. [Tidy].

Look back at the `rent` function and ask yourself what you don't like about it. Personally, I think it's a bit large and disorganized. It does several things that I can pull apart into functions that do one thing.

So first, let's just use the extract method refactoring to create functions that do one thing, and gather together the things that are related and separate the things that are different.[3]

---

3. The Single Responsibility Principle.

```
——Statement.java——
…
  public void rent(CatalogItem item, int days) {
    int unitPrice = getUnitPrice(item);
    int price = calculatePrice(item, days, unitPrice)
    int thisTax = getTax(item, price);
    items.add(new RentalItem(item, days, unitPrice, p
    subtotal += price;
    tax += thisTax;
  }

  private int getUnitPrice(CatalogItem item) {
    return switch (item) {
      case SMALL_ROOM -> 100;
      case LARGE_ROOM -> 150;
      case PROJECTOR -> 50;
```

```
      case COFFEE -> 10;
      case COOKIES -> 15;
    };
  }

  private int calculatePrice(CatalogItem item, int da
    boolean eligibleForDiscount = isEligibleForDiscou
    int price = unitPrice * days;
    if (eligibleForDiscount) price = (int) Math.round
    return price;
  }

  private boolean isEligibleForDiscount(CatalogItem i
    return switch (item) {
      case SMALL_ROOM, LARGE_ROOM -> days == 5;
      case PROJECTOR, COFFEE, COOKIES-> false;
    };
  }

  private int getTax(CatalogItem item, int price) {
    return switch (item) {
      case SMALL_ROOM, LARGE_ROOM, PROJECTOR ->
           (int) Math.round(price * .05);
      case COFFEE,COOKIES -> 0;
    };
  }
```


Future Bob:

Having used Grok3 on the `fromRoman` module in the last chapter, I thought I'd do the same here. So I told Grok3 to improve my initial implementation. It was not particularly surprising that its solution was nearly identical to mine.

Perhaps you look at this and think that it's more code. But it's not more *executable* code; it's just more names and more structure. That extra structure makes room for growth. For example, if the tax rules become more complicated, it is not the `rent` function that will grow; instead, in all likelihood, it will be the `getTax` function that grows.

This is an example of the Single Responsibility Principle (SRP)[4] in action. The stakeholders of our system who are most concerned about taxation will most probably be the only ones asking for changes to the `getTax` function, whereas the stakeholders who are interested in discounts will most likely be the only ones affecting the `calculatePrice` and `isEligibleForDiscount` functions.

---

4. See Chapter 19, "The SOLID Principles."

This is helpful to us because, after this change, we are not very likely to break the discounts when the taxes change, or break the taxes when the price calculation changes. The farther we can keep these different stakeholders' responsibilities apart and isolated, the safer our code will be when changes are made.

If we ignore the SRP, then we risk the symptom of *fragility*. A fragile system breaks in unexpected ways; for example, when a stakeholder asks us to modify taxes and we inadvertently break discounts. Such occurrences are terrifying to our stakeholders, managers, and users because it gives them a glimpse of what's under the hood; and they don't like what they see.

Our stakeholders have the right to expect that a change to taxes will not break discounts. When we violate that expectation, the only conclusion they can draw is that we've lost control of the system and don't really know what the hell we are doing.

Now look at that new code. What don't you like about it? One thing we might focus on are those `switch` statements.

`Switch` statements aren't intrinsically bad. However, if the number of cases is likely to grow, then they violate the Open–Closed Principle (OCP).

The OCP suggests that when we add a new feature, we should add that new feature in one place, not in many places. Right now, if we were to add a new `CatalogItem`, like `NotePads`, we'd have to add those changes to as many as five different places in the code. We can reduce this down to two by turning the `CatalogItem`s into data structures and replacing the `switch` statements with accesses of those data structures.

```
——Statement.java——
…
public enum CatalogItem {
  SMALL_ROOM(100, .05),
  LARGE_ROOM(150, .05),
  PROJECTOR(50, .05),
  COFFEE(10, 0),
  COOKIES(15, 0);

  private double taxRate;
  private int unitPrice;

  CatalogItem(int unitPrice, double taxRate) {
    this.unitPrice = unitPrice;
    this.taxRate = taxRate;
  }
}
…
public void rent(CatalogItem item, int days) {
  int unitPrice = item.unitPrice;
  int price = calculatePrice(item, days, unitPrice);
  int thisTax = (int) Math.round(price * item.taxRate
  items.add(new RentalItem(item, days, unitPrice, pri
  subtotal += price;
  tax += thisTax;
}
…
```

This is better—I have to say that I like that Java's `enum` s are actually classes, and that each enumerator is an instance. Now when we add `NotePads` , we'll only have to change the enum, and, perhaps, the `getItems` function. However, the OCP tells us to keep new changes out of old modules. So it would be better, from an OCP point of view, to get that `enum` out of the `Statement` module altogether so that when we add `NotePads` , we can leave the `Statement` module untouched.

```
——CatalogItem.java——
package ubConferenceCenter;

public enum CatalogItem {
  SMALL_ROOM(100, .05),
  LARGE_ROOM(150, .05),
```

```
    PROJECTOR(50, .05),
    COFFEE(10, 0),
    COOKIES(15, 0);

    public double taxRate;
    public int unitPrice;

    CatalogItem(int unitPrice, double taxRate) {
      this.unitPrice = unitPrice;
      this.taxRate = taxRate;
    }
  }
```

We're doing pretty well here. We've isolated the `CatalogItem enum` s, and we've gotten rid of the `switch` statements. The SRP and the OCP are pretty happy. But there's another principle that is sounding an alarm.

What modules ought to be recompiled if we add `NotePads` to `CatalogItem` ? Answer: `CatalogItem.java` (of course) and probably[5] `Statement.java` because it depends upon `CatalogItem.java` . Now ask yourself whether `Statement.java` *should have* to be recompiled.

---

5. The Java compiler is often smart enough to determine whether or not upstream modules need to be recompiled based on downstream changes. The safe bet, however, is to assume that those modules will be recompiled.

I contend that at least the `rent` function of `Statement.java` should not have to be recompiled when we add `NotePads` , because nothing within that function needs to know anything about `NotePads` .

This is a violation of the Dependency Inversion Principle (DIP). This principle is violated when high-level policies directly depend on low-level details. The addition of `NotePads` is a low-level detail that the `rent` function should not depend upon.

You may not think this is a big issue. Recompiling a module is quick and easy, and why should you care if you compile more modules than is required? For small projects, you could be right to make that decision. But for larger projects, the recompile and redeployment burden can get pretty

large. This is especially true for JavaScript applications where redeployments are downloaded into browsers. There's also the cognitive burden of trying to remember which modules depend on which others. So let's assume that this project is becoming large enough for those issues to be a concern. How can we invert the dependencies to reduce the recompilation, redeployment, and cognitive burdens?

The SRP will help us here. The `Statement.java` module is doing two major things. The `rent` method adds `CatalogItem` s, and the `getItems` method totals them and computes the cookie bonus. Hmmm, `getItems` isn't the best name, is it? We'll take care of that momentarily. First, we should separate those two major behaviors.

The first step is to pull out the `RentalItem` record into its own module.

```
——RentalItem.java___
package ubConferenceCenter;

public record RentalItem(CatalogItem type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

Next, we'll create a new module for the `ItemList` .

```
——ItemList.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class ItemList {
  private List<RentalItem> items = new ArrayList<>();

  public void add(CatalogItem item, int days, int uni
                  int price, int thisTax) {
    items.add(new RentalItem(item, days, unitPrice, p
  }
```

```
  public RentalItem[] getItems() {
    boolean largeRoomFiveDays = items.stream().anyMat
      item -> item.type() == CatalogItem.LARGE_ROOM &
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type() == CatalogItem.COFFEE && it
    if (largeRoomFiveDays && coffeeFiveDays)
      items.add(new RentalItem(CatalogItem.COOKIES, 5
    return items.toArray(new RentalItem[0]);
  }
}
```

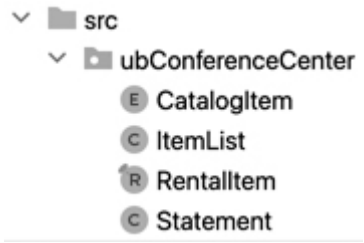
We'll have to come back to this module because that `getItems` method gives me the willies.[6] It smells of an SRP violation. But for now, all that remains is to make a few adjustments to the `Statement.java` module.

---

6. An archaic term that means nervous, uneasy, or creeped out. (According to Grok.)

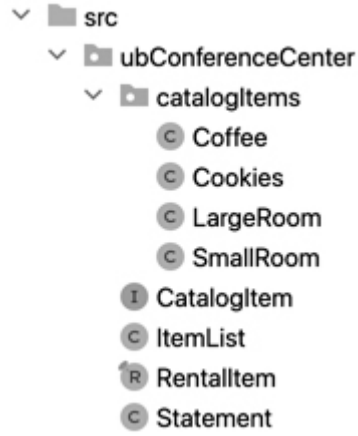
```
…
public class Statement {
  …
  private ItemList items = new ItemList();
  …

  public RentalItem[] getItems() {
    return items.getItems();
  }
  …
}
```

Again, you may be concerned that we're chopping this up into too many pieces, and that all those pieces will make the code harder to understand. But our assumption has been that this code is going to grow, and that we're trying to make room for that growth. In any case, if you look at the directory structure, you'll see that there's a nice road map that will help anyone understand the organization.



Looking at this, you can see that the names may not be perfect, but we'll come back to that when we understand more about where this reorganization is going. In the meantime, we still have to reduce the recompile and redeployment burden.

To do this, we'll change the `CatalogItem` from an `enum` to an `interface`, with each enumeration becoming a derived class. This creates several new classes and makes the directory look like this.




```
——CatalogItem.java——
package ubConferenceCenter;

public interface CatalogItem {
  boolean isEligibleForDiscount(int days);
  int getUnitPrice();
  double getTaxRate();
  String getName();
}
```

This represents a significant change in the way the application works. Now instead of using `switch` and `if` statements to check for enumerators, we defer to the polymorphic methods of the `CatalogItem` interface. The `getTaxRate` and `getUnitPrice` methods are familiar, but the other two are new. You can see how they are used in the `Statement.java` and `ItemList.java` modules.

```
——Statement.java——
package ubConferenceCenter;

public class Statement {
  …
  public void rent(CatalogItem item, int days) {
    int unitPrice = item.getUnitPrice();
    int price = calculatePrice(item, days, unitPrice)
    int thisTax = (int) Math.round(price * item.getTa
    items.add(item, days, unitPrice, price, thisTax);
    subtotal += price;
    tax += thisTax;
  }

  private int calculatePrice(CatalogItem item, int da
{
    boolean eligibleForDiscount = item.isEligibleForD
    int price = unitPrice * days;
    if (eligibleForDiscount) price = (int) Math.round
    return price;
  }
 …
}

——ItemList.java——
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class ItemList {
  private List<RentalItem> items = new ArrayList<>();

  public void add(CatalogItem item, int days, int uni
                  int price, int thisTax) {
    items.add(new RentalItem(item.getName(), days,
              unitPrice, price, thisTax));
  }

  public RentalItem[] getItems() {
    boolean largeRoomFiveDays = items.stream().anyMat
      item -> item.type().equals("LARGE_ROOM") && ite
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type().equals("COFFEE") && item.da
```

```
    if (largeRoomFiveDays && coffeeFiveDays)
      items.add(new RentalItem("COOKIES", 5, 0, 0, 0)
    return items.toArray(new RentalItem[0]);
  }
}
```

Notice that the `type` field of the `RentalItem` record has been changed from the `enum` to a `String`. This was necessary because, when we abandoned the `enum`, we needed some kind of token to represent the `CatalogItem` type; and a `String` seemed the most obvious choice. However, using that string means that we've had to sacrifice a bit of static type safety. The compiler cannot check that those names are correct. This minor loss of static type safety always accompanies the effort to reduce the recompile and redeployment burden, and the isolation of low-level details from high-level policy.

```
——RentalItem.java——
package ubConferenceCenter;

public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

With the exception of the `getItems` method of the `ItemList` class, the last few listings above represent the high-level policy of the application. We're going to have to do something about that `getItems` method. For now, let's look at how the low-level details have been isolated in the derivatives of `CatalogItem`.

```
——SmallRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class SmallRoom implements CatalogItem {
  public String getName() {
    return "SMALL_ROOM";
  }
```

```
  public boolean isEligibleForDiscount(int days) {
    return days == 5;
  }

  public int getUnitPrice() {
    return 100;
  }

  public double getTaxRate() {
    return 0.05;
  }
}

———LargeRoom.java———
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return days == 5;
  }

  public int getUnitPrice() {
    return 150;
  }

  public double getTaxRate() {
    return 0.05;
  }

  public String getName() {
    return "LARGE_ROOM";
  }
}

———Coffee.java———
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return false;
```

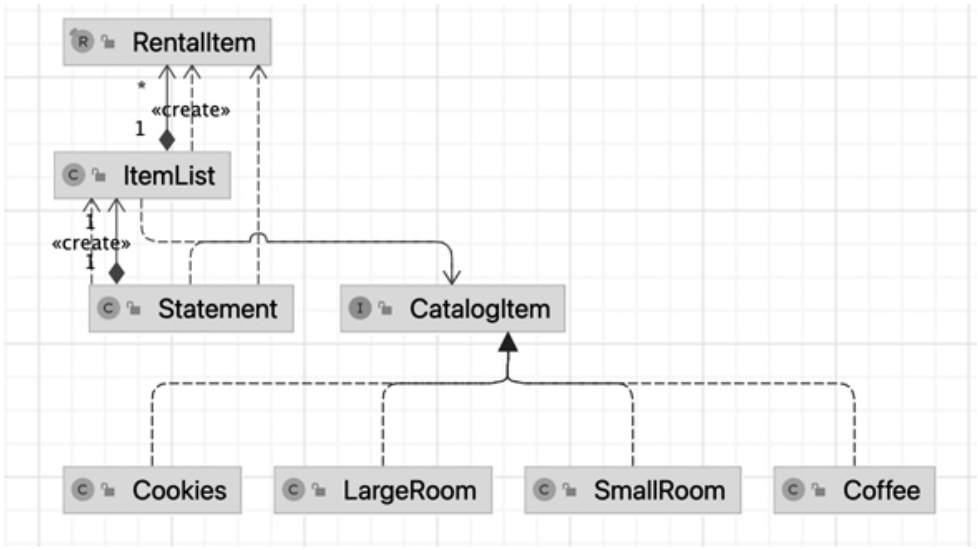
```
  }

  public int getUnitPrice() {
    return 10;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COFFEE";
  }
}

——Cookies.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
  public boolean isEligibleForDiscount(int days) {
    return false;
  }

  public int getUnitPrice() {
    return 15;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COOKIES";
  }
}
```

One of the people who commented on the first edition of this book looked at code like this and called it "*just dreadful*." You might be feeling that now too. But remember, we are expecting business growth, and we have decided that it is necessary to make room in this code for that growth.

Some folks might charge us with violating YAGNI. They think it stands for You Aren't Going to Need It. But the original intent of YAGNI was to ask yourself: "What if you aren't going to need it?" It was a way of asking us to count the cost before we invested in making room in our code for things we might not need.

But given the enthusiasm of the CEO and the kinds of markets he is courting, we have decided we need to make this room. So, in our case, we've decided to answer YAGNI's question with: *"Yes, we are going to need it."*

Why is this code better? Let's look at the dependency diagram generated by my IDE.



The four classes that represent our high-level policy— `RentalItem` , `ItemList` , `Statement` , and `CatalogItem` —are tied together with a rat's nest of dependencies. We'll deal with that later. But look at how well isolated the low-level details are. The previous dependencies have been inverted. Nothing within the high-level policy depends upon the low-level details. The low-level details depend only on the `CatalogItem` . Just say this to yourself, over and over: *High-level policy should not depend on low-level details.*

This means that if any of the low-level details are changed, nothing in the high-level policy ought necessarily to be recompiled or redeployed. Nor will changes to the high-level policy force recompilation and redeployment of the low-level details.

Let me say that differently. If the unit price of `Cookies` changes, or if the tax rate of `Coffee` changes, or if the discount for `SmallRoom` changes, none of the classes in the high-level policy will need to be recompiled or redeployed. Changes to those low-level details are completely isolated from the high-level policy.

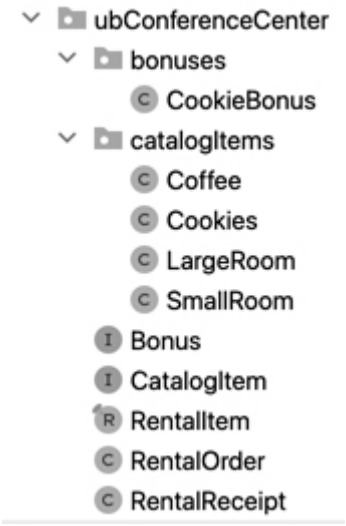
Indeed, the low-level details have become a *plug-in* to the high-level policy. Or, at least they are in a position to become such a plug-in.

But now we need to do something about the rat's nest that's caused by the `ItemList`.

We called this class `ItemLis`t because initially it contained our list of items, and we needed a place to put that bonus code. Now I want to take that bonus code out of there and do to it what we did to the `CatalogItems`. So I'll create a `Bonus` interface and have the `CookieBonus` implement it. The `ItemList` can contain a list of `Bonus` instances and simply iterate through all of them to add all the bonuses. That means that the `ItemList` *finalizes* all the bonuses. And that suggests that the `ItemList` should be renamed something like `RentalReceipt`, since it represents the final state of the order.

That name change suggests that `Statement` should be renamed something like `RentalOrder`. These name changes are important. As we pick apart the design and start making room for growth, we gain insights into what's really going on. When the problem is small, names can afford to be inconsistent or vague because there aren't that many concepts to keep track of. But as the problem grows, names become much more important because they help us keep the concepts straight in our heads.

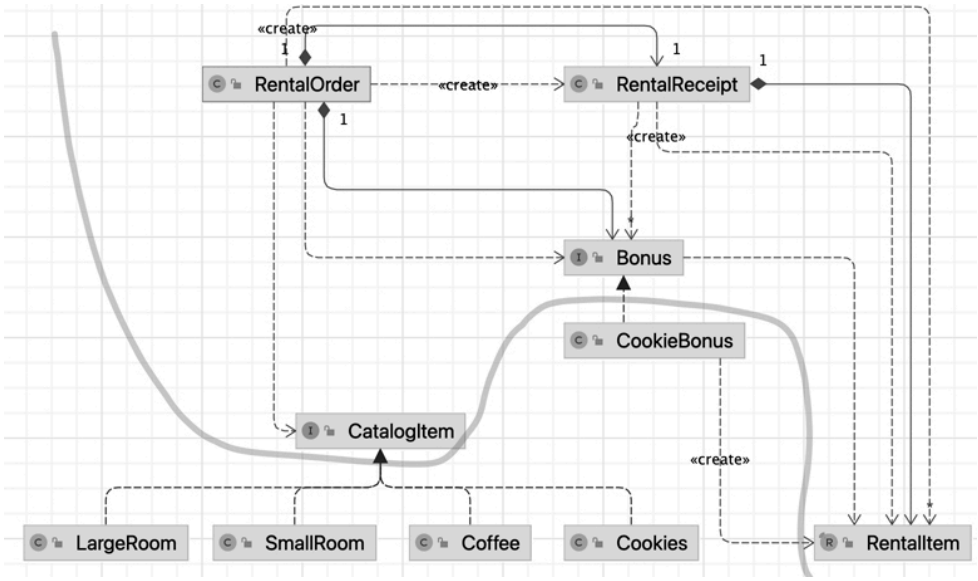
So with those changes in place, here's the directory structure:



The diagram is shown below. The `RentalOrder` depends on the `RentalReceipt`, but not vice versa. Both depend upon the `Bonus`, but the `CookieBonus` is an independent low-level detail.

The lowest-level detail is the `RentalItem` record. That's a bit of a concern because it is so concrete. If it gets changed in any way, then pretty much everything has to recompile and redeploy. There are ways to resolve that—but I think that's a battle for another day.

Notice the curvy border line. That line divides the project into two components. One contains the high-level policy, and the other contains the low-level details. All dependencies cross that line in one direction, toward the higher-level component. That line is an *architectural boundary* that separates those two components—and the rule for architectural boundaries is that dependencies cross only toward the higher-level side.



First, let's look at the code within the high-level component.

```
———RentalOrder.java———
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class RentalOrder {
  public record Totals(int subtotal, int tax) {
  }

  private String customerName;
  private int subtotal = 0;
  private int tax = 0;
  private RentalReceipt receipt = new RentalReceipt()
  private List<Bonus> bonuses = new ArrayList<>();

  public RentalOrder(String customerName) {
    this.customerName = customerName;
  }

  public void addBonus(Bonus bonus) {
    bonuses.add(bonus);
  }

  public void rent(CatalogItem item, int days) {
    int unitPrice = item.getUnitPrice();
    int price = item.getDiscountedPrice(days);
    int thisTax = (int) Math.round(price * item.getTa
    receipt.add(new RentalItem(item.getName(), days,
             unitPrice, price, thisTax));
    subtotal += price;
    tax += thisTax;
  }

  public RentalItem[] getReceipt() {
    return receipt.finalize(bonuses);
  }

  public String getCustomerName() {
    return customerName;
  }

  public Totals getTotals() {
    return new Totals(subtotal, tax);
```

```
  }
}
```

Notice that virtually all of the business rules have fled this module. The only ones that remain are the tax and total calculations. But the discount and bonus calculations have been moved into the appropriate derivatives of `CatalogItem` and `Bonus`.

```
———RentalReceipt.java———
package ubConferenceCenter;

import java.util.ArrayList;
import java.util.List;

public class RentalReceipt {
  private List<RentalItem> items = new ArrayList<>();

  public void add(RentalItem item) {
    items.add(item);
  }

  public RentalItem[] finalize(List<Bonus> bonuses) {
    List<RentalItem> finalItems = new ArrayList<>(thi
    finalItems.addAll(addBonuses(bonuses));
    return finalItems.toArray(new RentalItem[0]);
  }

  private List<RentalItem> addBonuses(List<Bonus> bor
    List<RentalItem> bonusItems = new ArrayList<>();
    for (Bonus bonus : bonuses)
      bonus.checkAndAdd(items, bonusItems);
    return bonusItems;
  }
}
```

The `RentalReceipt` class walks through and applies the bonuses; but it does not know any of the details about those bonuses. It creates a finalized receipt with the bonuses added to all the items.

Next come the two interfaces that inverted the dependencies between the high-level policy and the low-level detail.

```
——CatalogItem.java——
package ubConferenceCenter;

public interface CatalogItem {
  int getDiscountedPrice(int days);
  int getUnitPrice();
  double getTaxRate();
  String getName();
}

——Bonus.java——
package ubConferenceCenter;

import java.util.List;

public interface Bonus {
  void checkAndAdd(List<RentalItem> items, List<Renta
}
```

The last module in the high-level component is the `RentalItem`. This is a simple concrete data structure.

```
——RentalItem.java——
package ubConferenceCenter;

public record RentalItem(String type,
                         int days,
                         int unitPrice,
                         int price,
                         int tax) {
}
```

As previously stated, this module is somewhat problematic. Any changes made to this data structure will force recompilation and redeployment of both components. And changes to this data structure are not at all unlikely.

There are ways of dealing with this. For example, we could turn the `RentalItem` into a hash map. But that's probably going well beyond our current goal of making room for growth. We'll keep that idea in our back pocket for now.

Now let's look at the low-level component.

```
——SmallRoom.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class SmallRoom implements CatalogItem {
  public String getName() {
    return "SMALL_ROOM";
  }

  public int getDiscountedPrice(int days) {
    double discountRate = (days == 5) ? 0.9 : 1.0;
    return (int) Math.round(getUnitPrice() * days * d
  }

  public int getUnitPrice() {
    return 100;
  }

  public double getTaxRate() {
    return 0.05;
  }
}

——LargeRoom.java___
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class LargeRoom implements CatalogItem {
  public int getDiscountedPrice(int days) {
    double discountRate = (days == 5) ? 0.9 : 1.0;
    return (int) Math.round(getUnitPrice() * days * d
  }

  public int getUnitPrice() {
    return 150;
  }

  public double getTaxRate() {
    return 0.05;
  }
```

```
  public String getName() {
    return "LARGE_ROOM";
  }
}

——Coffee.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Coffee implements CatalogItem {
  public int getDiscountedPrice(int days) {
    return getUnitPrice() * days;
  }

  public int getUnitPrice() {
    return 10;
  }

  public double getTaxRate() {
    return 0;
  }

  public String getName() {
    return "COFFEE";
  }
}

——Cookies.java——
package ubConferenceCenter.catalogItems;

import ubConferenceCenter.CatalogItem;

public class Cookies implements CatalogItem {
  public int getDiscountedPrice(int days) {
    return getUnitPrice() * days;
  }

  public int getUnitPrice() {
    return 15;
  }

  public double getTaxRate() {
    return 0;
```

```
  }

  public String getName() {
    return "COOKIES";
  }
}

——CookieBonus.java——
package ubConferenceCenter.bonuses;

import ubConferenceCenter.Bonus;
import ubConferenceCenter.RentalItem;

import java.util.List;

public class CookieBonus implements Bonus {
  public void checkAndAdd(List<RentalItem> items,
                          List<RentalItem> bonusItems
    boolean largeRoomFiveDays = items.stream().anyMat
      item -> item.type().equals("LARGE_ROOM") && ite
    boolean coffeeFiveDays = items.stream().anyMatch(
      item -> item.type().equals("COFFEE") && item.da
    if (largeRoomFiveDays && coffeeFiveDays)
      bonusItems.add(new RentalItem("COOKIES", 5, 0,
  }
}
```

As you can see, all the business rules have been tucked away into their corresponding derivatives. If a new discount is needed for coffee, we can place it in the `Coffee` class. If a new bonus is needed for small rooms, we can create a `Bonus` derivative for that. If we want to add `NotePads` or `Projector`s, we can do that by adding those classes to the low-level component. None of those changes will cause the high-level component to be recompiled or redeployed. We have achieved that isolation by conforming to the DIP.

You might be concerned about one thing, though. Perhaps you are wondering where all those derivatives are getting instantiated. I never showed you my tests, did I? That's where all that work was done. Here they are.

```
———RentalOrderTest.java———
package ubConferenceCenter;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import ubConferenceCenter.RentalOrder.Totals;
import ubConferenceCenter.bonuses.CookieBonus;
import ubConferenceCenter.catalogItems.Coffee;
import ubConferenceCenter.catalogItems.Cookies;
import ubConferenceCenter.catalogItems.LargeRoom;
import ubConferenceCenter.catalogItems.SmallRoom;

import static org.junit.jupiter.api.Assertions.assert

public class RentalOrderTest {
  private final SmallRoom SMALL_ROOM = new SmallRoom(
  private final LargeRoom LARGE_ROOM = new LargeRoom(
  private final Coffee COFFEE = new Coffee();
  private final Cookies COOKIES = new Cookies();
  private RentalOrder order;

  @BeforeEach
  void setUp() {
    order = new RentalOrder("Customer Name");
    order.addBonus(new CookieBonus());
  }

  private void assertTotalAndTax(int subtotal, int ta
    Totals totals = order.getTotals();
    assertEquals(subtotal, totals.subtotal());
    assertEquals(tax, totals.tax());
  }

  private void assertTypeDaysUnitTotalTax(RentalItem
                                          int days, i
                                          int price,
    assertEquals(type, item.type());
    assertEquals(days, item.days());
    assertEquals(unitPrice, item.unitPrice());
    assertEquals(price, item.price());
    assertEquals(tax, item.tax());
  }

  @Test
```

```
public void oneSmallRoomForOneDay() throws Exceptic
  assertEquals("Customer Name", order.getCustomerNa
  order.rent(SMALL_ROOM, 1);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTotalAndTax(100, 5);
}

@Test
public void oneSmallRoomForTwoDays() throws Excepti
  order.rent(SMALL_ROOM, 2);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTotalAndTax(200, 10);
}

@Test
public void oneLargeRoomForThreeDays() throws Excep
  order.rent(LARGE_ROOM, 3);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROC
  assertTotalAndTax(450, 23);
}

@Test
public void oneSmallRoomForOneWeek() throws Excepti
  order.rent(SMALL_ROOM, 5);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(1, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTotalAndTax(450, 23); /* 10% discount */
}

@Test
public void twoSmallRoomsForOneDay() throws Excepti
  order.rent(SMALL_ROOM, 1);
  order.rent(SMALL_ROOM, 1);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTypeDaysUnitTotalTax(receipt[1], "SMALL_ROC
  assertTotalAndTax(200, 10);
```

```
}

@Test
public void oneSmallRoomAndCoffeeForOneDay() throws
  order.rent(SMALL_ROOM, 1);
  order.rent(COFFEE, 1);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTypeDaysUnitTotalTax(receipt[1], "COFFEE",
  assertTotalAndTax(110, 5); /* No tax on coffee */
}

@Test
public void oneSmallRoomAndCookiesForOneDay() throw
  order.rent(SMALL_ROOM, 1);
  order.rent(COOKIES, 1);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(2, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTypeDaysUnitTotalTax(receipt[1], "COOKIES",
  assertTotalAndTax(115, 5); /* No tax on cookies *
}

@Test
public void oneSmallRoomCookiesCoffeeForFiveDays()
  order.rent(SMALL_ROOM, 5);
  order.rent(COFFEE, 5);
  order.rent(COOKIES, 5);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(3, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "SMALL_ROC
  assertTypeDaysUnitTotalTax(receipt[1], "COFFEE",
  assertTypeDaysUnitTotalTax(receipt[2], "COOKIES",
  /* 10% discount on room but not on coffee */
  assertTotalAndTax(575, 23);
}

@Test
public void oneLargeRoomAndCoffeeForWeekGetsCookies
  order.rent(LARGE_ROOM, 5);
  order.rent(COFFEE, 5);
  RentalItem[] receipt = order.getReceipt();
  assertEquals(3, receipt.length);
  assertTypeDaysUnitTotalTax(receipt[0], "LARGE_ROC
```

```
      assertTypeDaysUnitTotalTax(receipt[1], "COFFEE",
      assertTypeDaysUnitTotalTax(receipt[2], "COOKIES",
    }
  }
```

These tests were written before any of the refactoring in this chapter had begun. At each step of that refactoring, I kept these tests passing. Some of the changes made to the production code caused changes to the tests, but I was able to minimize them by keeping the tests and production code decoupled and by isolating the details of the production code from the details of the tests. For example, the final fields named for the `CatalogItems` replaced the old enumerations without too much fuss.

Independent Deployability

Our design now has a strong architectural boundary that separates it into two components. Those two components can be independently deployed. They can be compiled into two separate `jar` files. When one component changes, only that `jar` changes, and the other does not need to be redeployed.

Imagine that this code was written in JavaScript as opposed to Java. Imagine that it will be sent to a browser to run there. The fact that we have divided it into two components has a very specific advantage. If one of those two components changes, and if the browser maintains a cache of those components, then only the component that changed needs to be reloaded into the browser. On slow network connections, that could be a big advantage.

A Final Thought

This major cleaning will greatly facilitate the growth of the project. The time it took will be paid back many times over simply because there's a clear and obvious place to put the needed changes; and those changes aren't likely to interfere with each other or with the overall flow of the system.

However, the new structure is not cost free. Those who complain that it is harder to read because of all the different modules and all the indirection are not without a point. Yes, adding structure adds complexity—and that complexity is not free. There's a cognitive price to pay. But we knew that

the cognitive load was going to grow, and so our goal was to minimize that growth by creating an organization that would accept it gracefully.

What about performance? Is the new structure slower than the old one? Probably so. There are a few more references that need to be followed, and `switch` statements are easier for compilers to optimize than polymorphic dispatch. But the difference in performance is probably so tiny that you could not easily measure it. For most applications, that tiny decrease in performance would be of no concern at all.

Future Bob:

Grok3, good as it is, did not drive the design of the original code to the level of depth we just saw. Grok3 did not invert the dependencies, nor did it attempt to separate high-level policy from low-level detail. Large language models (LLMs)/AIs have their uses; but they are not human and are not adept at understanding higher architectural goals.

This chapter, and the chapter before, were a whirlwind tour through the low-hanging fruit of clean code. But now that tour is over, and it’s time to get down to the basics. And, of course, we begin with names.