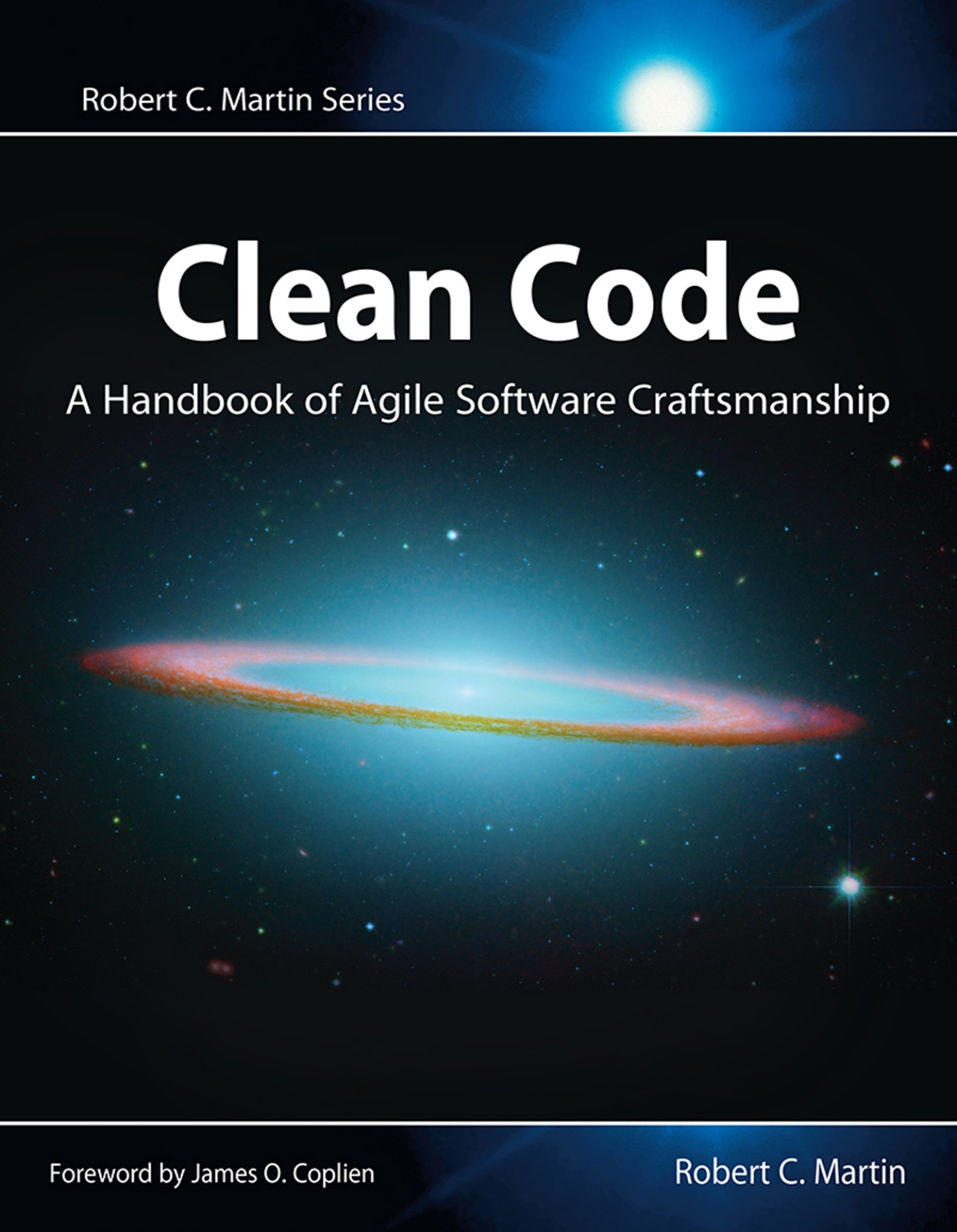
Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship



Foreword by James O. Coplien

Robert C. Martin

Clean Code

Let's look at an example of poor exception classification. Here is a `try-catch-finally` statement for a third-party library call. It covers all of the exceptions that the calls can throw:

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

That statement contains a lot of duplication, and we shouldn't be surprised. In most exception handling situations, the work that we do is relatively standard regardless of the actual cause. We have to record an error and make sure that we can proceed.

In this case, because we know that the work that we are doing is roughly the same regardless of the exception, we can simplify our code considerably by wrapping the API that we are calling and making sure that it returns a common exception type:

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

Our `LocalPort` class is just a simple wrapper that catches and translates exceptions thrown by the `ACMEPort` class:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
```

```
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

Wrappers like the one we defined for `ACMEPort` can be very useful. In fact, wrapping third-party APIs is a best practice. When you wrap a third-party API, you minimize your dependencies upon it: You can choose to move to a different library in the future without much penalty. Wrapping also makes it easier to mock out third-party calls when you are testing your own code.

One final advantage of wrapping is that you aren't tied to a particular vendor's API design choices. You can define an API that you feel comfortable with. In the preceding example, we defined a single exception type for `port` device failure and found that we could write much cleaner code.

Often a single exception class is fine for a particular area of code. The information sent with the exception can distinguish the errors. Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.

Define the Normal Flow

If you follow the advice in the preceding sections, you'll end up with a good amount of separation between your business logic and your error handling. The bulk of your code will start to look like a clean unadorned algorithm. However, the process of doing this pushes error detection to the edges of your program. You wrap external APIs so that you can throw your own exceptions, and you define a handler above your code so that you can deal with any aborted computation. Most of the time this is a great approach, but there are some times when you may not want to abort.

Let's take a look at an example. Here is some awkward code that sums expenses in a billing application:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

In this business, if meals are expensed, they become part of the total. If they aren't, the employee gets a meal *per diem* amount for that day. The exception clutters the logic. Wouldn't it be better if we didn't have to deal with the special case? If we didn't, our code would look much simpler. It would look like this:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Can we make the code that simple? It turns out that we can. We can change the `ExpenseReportDAO` so that it always returns a `MealExpense` object. If there are no meal expenses, it returns a `MealExpense` object that returns the *per diem* as its total:

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // return the per diem default
  }
}
```

This is called the SPECIAL CASE PATTERN [Fowler]. You create a class or configure an object so that it handles a special case for you. When you do, the client code doesn't have to deal with exceptional behavior. That behavior is encapsulated in the special case object.

Don't Return Null

I think that any discussion about error handling should include mention of the things we do that invite errors. The first on the list is returning `null`. I can't begin to count the number of applications I've seen in which nearly every other line was a check for `null`. Here is some example code:

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

If you work in a code base with code like this, it might not look all that bad to you, but it is bad! When we return `null`, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing `null` check to send an application spinning out of control.

Did you notice the fact that there wasn't a `null` check in the second line of that nested `if` statement? What would have happened at runtime if `persistentStore` were `null`? We would have had a `NullPointerException` at runtime, and either someone is catching `NullPointerException` at the top level or they are not. Either way it's *bad*. What exactly should you do in response to a `NullPointerException` thrown from the depths of your application?

It's easy to say that the problem with the code above is that it is missing a `null` check, but in actuality, the problem is that it has *too many*. If you are tempted to return `null` from a method, consider throwing an exception or returning a SPECIAL CASE object instead. If you are calling a `null`-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

In many cases, special case objects are an easy remedy. Imagine that you have code like this:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Right now, `getEmployees` can return `null`, but does it have to? If we change `getEmployee` so that it returns an empty list, we can clean up the code:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Fortunately, Java has `Collections.emptyList()`, and it returns a predefined immutable list that we can use for this purpose:

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

If you code this way, you will minimize the chance of `NullPointerExceptions` and your code will be cleaner.

Don't Pass Null

Returning `null` from methods is bad, but passing `null` into methods is worse. Unless you are working with an API which expects you to pass `null`, you should avoid passing `null` in your code whenever possible.

Let's look at an example to see why. Here is a simple method which calculates a metric for two points:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

What happens when someone passes `null` as an argument?

```
calculator.xProjection(null, new Point(12, 13));
```

We'll get a `NullPointerException`, of course.

How can we fix it? We could create a new exception type and throw it:

```
public class MetricsCalculator
{
```

```
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
  }
}
```

Is this better? It might be a little better than a `null` pointer exception, but remember, we have to define a handler for `InvalidArgumentException`. What should the handler do? Is there any good course of action?

There is another alternative. We could use a set of asser tions:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x - p1.x) * 1.5;
  }
}
```

It's good documentation, but it doesn't solve the problem. If someone passes `null`, we'll still have a runtime error.

In most programming languages there is no good way to deal with a `null` that is passed by a caller accidentally. Because this is the case, the rational approach is to forbid passing `null` by default. When you do, you can code with the knowledge that a `null` in an argument list is an indication of a problem, and end up with far fewer careless mistakes.

Conclusion

Clean code is readable, but it must also be robust. These are not conflicting goals. We can write robust clean code if we see error handling as a separate concern, something that is viewable independently of our main logic. To the degree that we are able to do that, we can reason about it independently, and we can make great strides in the maintainability of our code.

Bibliography

[Martin]: *Agile Software Development: Principles, Patterns, and Practices,* Robert C. Martin, Prentice Hall, 2002.

8

Boundaries

by James Grenning

We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code