It's natural that object-oriented programming is focused on objects and their relationships, but the code within an object is important too. Classic books like Jon Bentley's *Programming Pearls* or Brian Kernighan and P. J. Pike's *Elements of Programming Style* can help inspire you to write good, clean code.

Conditional logic is often the trickiest part of such code.

- It's hard to reason about, since we have to consider multiple paths through the code.
- It's tempting to add special-case handling rather than develop the general case.
- Conditional logic sometimes is used as a weak substitute for object-oriented mechanisms.

### Smells Covered

- Null Check
- Complicated Boolean Expression
- Special Case
- Simulated Inheritance (Switch Statement)

### Null Check

**Symptoms**

- There are repeated occurrences of code like this:
  
  ```java
  if (xxx == null) {
  ...
  ```
CONDITIONAL LOGIC

Causes
Someone decides, “We’ll use null to mean the default.” This may let them avoid the trouble of initializing certain fields or of creating certain objects, or it may be an afterthought for an unexpected case. The null check may be introduced to work around a null-pointer bug (without addressing the underlying cause).

What to Do
• If there’s a reasonable default value, use that.
• Otherwise, Introduce Null Object to create a default object that you explicitly use.

Payoff
Reduces duplication. Reduces logic errors and exceptions.

Contraindications
• If the null check occurs in only one place (e.g., in a factory method), it is usually not worth the effort to create a separate Null Object.
• Null objects need to have safe behavior for the methods they provide. They often act like identity objects (as 0 does relative to addition). If you can’t define a safe behavior for each method, you may not be able to use a Null Object.
• Watch out for a case where null means two or more different things in different contexts. (You may be able to support this with different Null Objects.)

EXERCISE 20  Null Object.

Consider the code in Exercise 4 (Chapter 3).

A. Some of the null checks are checks for null strings. One alternate approach would be to use empty strings. What are the downsides of this approach (taking into account the test and all the other client classes you don’t see here)?
Complicated Boolean Expression

Symptoms

• Code has complex conditions involving and, or, and not.

Causes

The code may have been complicated from the beginning, or it may have picked up additional conditions along the way.

What to Do

• Apply DeMorgan’s Law:
  
  ! (a && b) => (!a) || (!b)

  and

  ! (a || b) => (!a) && (!b)

You may find that some variables will communicate better if they change names to reflect their flipped sense.

• Introduce Explaining Variable to make each clause clearer.
• Use guard clauses to peel off certain conditions; the remaining clauses get simpler.
• Decompose Conditional to pull each part into its own method.

Payoff

Improves communication.

EXERCISE 20  Null Object. (Continued)

B. What’s another approach to this problem?

C. Extract a Bin class, and introduce a Null object.

See Appendix A for solutions.
Contraindications
You may be able to find other ways to simplify the expressions, or you may find that the rewritten expression communicates less well.

EXERCISE 21  Conditional Expression.

Consider this code fragment:

```java
if (!((score > 700) ||
    ((income >= 40000) && (income <= 100000)
    && authorized && (score > 500)) ||
    (income > 100000)))
    reject();
else
    accept();
```

A. Apply DeMorgan’s Law to simplify this as much as possible.

B. Starting from the original, rewrite the condition by introducing explaining variables.

C. Starting from the original, flip the if and else clauses, then break the original into several if clauses. (You’ll call accept() in three different places.)

D. Consolidate Conditional Expression by extracting a method to compute the condition.

E. Which approach was the simplest? The clearest? Can you combine the techniques?
EXERCISE 21  Conditional Expression. (Continued)

F. Describe the conditions in tabular form. Base the rows and columns on three variables: one for the three score ranges, one for the income ranges, and one for the authorized flag. The cells in the table should say either “accept” or “reject.”

See Appendix A for solutions.

Special Case

Symptoms

• Complex if statements
• Checks for particular values before doing work (especially comparisons to constants or enumerations)

Causes

Someone realizes a special case is needed.

What to Do

• If the conditionals are taking the place of polymorphism, Replace Conditional with Polymorphism.
• If the if and then clauses are similar enough, you may be able to rewrite them so that the same code fragment can generate the proper results for each case; then the conditional can be eliminated.

Payoff

Improves communication. May expose duplication.

Contraindications

• In a recursive definition, there is always a base case that will stop the recursion; you can’t expect to eliminate these.
• Sometimes an if clause is just the simplest way to do something.
Simulated Inheritance (Switch Statement)

Symptoms

- Code uses a switch statement (especially on a type field).
- Code has several if statements in a row (especially if they're comparing against the same value).
- Code uses instanceof (or its equivalent) to decide what type it's working with.

Causes

This smell is often caused by laziness in introducing new classes. The first time you need conditional behavior, you might use an if or switch statement rather than a new class. It's not a big problem at this point because it occurs only once. Then say you need another condition based on the same type code, and you introduce a second switch instead of fixing the lack of polymorphism. Sometimes the lack of polymorphism is hidden behind a series of if statements instead of an explicit switch statement, but the root problem is the same.

What to Do

Don't simulate inheritance—use mechanisms built into the programming language.

- If a switch statement on the same condition occurs in several places, it is often using a type code; replace this with the polymorphism built into objects. It takes a series of refactorings to make this change:
  1. Extract Method. Pull out the code for each branch.
  2. Move Method. Move related code onto the right class.
  3. Replace Type Code with Subclass or Replace Type Code with State/Strategy. Set up the inheritance structure.
  4. Replace Conditional with Polymorphism. Eliminate the conditionals.

- If the conditions occur within a single class, you might be able to replace the conditional logic via Replace Parameter with Explicit Methods or Introduce Null Object.

Payoff

Improves communication. May expose duplication.
Contraindications

Sometimes a switch statement is the simplest way to express the logic. If the code is doing something simple, in one place, you may not feel the need for a separate class. This may be especially common for places that are interfacing with non-object-oriented parts of the system. Michael Feathers says, “I’m okay with switches if they convert data into objects.”

A single switch statement is sometimes used in a Factory or Abstract Factory (for more information, see Gamma’s Design Patterns). This one place decides how to configure the whole factory. You can sometimes replace it with Class.forName() (i.e., dynamic class loading), but that is not always attractive for security, communication, or performance reasons.

Sometimes a switch statement is used in several related places to control a state machine. Decide whether it makes more sense as is, or whether the State pattern (see Gamma’s Design Patterns) is more appropriate.

EXERCISE 22  Switch Statement.

Consider this code:

```java
public void printIt(Operation op) {
  String out = "?";
  switch (op.type) {
    case '+': out = "push"; break;
    case '-': out = "pop"; break;
    case '@': out = "top"; break;
    default: out = "unknown";
  }
  System.out.println("operation=" + out);
}

public void doIt(Operation op, Stack s, Object item) {
  switch (op.type) {
    case '+': s.push(item); break;
    case '-': s.pop(); break;
  }
}
```

A. What would you do?

See Appendix A for solutions.
EXERCISE 23  Factory Method. (Challenging).

Consider this class structure:

```
DriverFactory
    -- for constructor:
    +useMemoryDriver: int
    +useDebugDriver: int
    +useProductionDriver: int

Factory(int driverType)
    getDriver(): Driver

Driver
    find()
    save()

MemoryDriver

DebugDriver

ProductionDriver
```

A. Write code for the factory according to the implied design. Note: One of the three constants is passed to the constructor; this determines what type driver will be returned by `getDriver()`.

B. Your code probably smells of a `switch` statement (even if it's implemented using `if`). Is this conditional logic justified?

C. Some factories use dynamic class loading—via `Class.forName()`—to load the correct class on demand. Modify your factory to accept a string argument (the name of the driver class) and load the instance dynamically. Your code no longer mentions the types explicitly. What are some advantages to that? (Hint: Think of the impact on building the system.)

D. What are some disadvantages to this new arrangement?

---

See Appendix A for solutions.
INTERLUDE 3 DESIGN PATTERNS

INTERLUDE I3.1 Patterns.

Following is a list of design patterns described in Gamma's *Design Patterns*. What refactorings might you use to evolve to some of these patterns?

Creational patterns

___ Abstract Factory
___ Builder
___ Factory Method
___ Prototype
___ Singleton

Structural Patterns

___ Adapter
___ Bridge
___ Composite
___ Decorator
___ Façade
___ Flyweight
___ Proxy
Behavioral Patterns

___ Chain of Responsibility
___ Command
___ Interpreter
___ Iterator
___ Mediator
___ Memento
___ Observer
___ State
___ Strategy
___ Template Method
___ Visitor

See Appendix A for solutions.