Design Patterns as Language Constructs

Simon Walker
June 26th, 2025

This project stems from a search for *process*.

I'm motivated to,

find an architecting process that's

1. **iterative** // because plans change and mistakes happen
2. **multi-leveled** // it should be easy to see how high level changes reflect in an established code base
3. **collaborative** // a lone architect isn't necessary

But first,

to guide the exploration, let's be clear:

What is good code?

Washing Behind Your Ears: Principles of Software Hygiene

David M. Tilbrook: contributor of `QED` to unix, `vi` ancestor. And John McMullen

> *Like personal hygiene, **software hygiene** is most conspicuous in its absence.*

> *Often, quality considers only the development phase; the real picture is wider. The side of profession not usually highlighted in programming courses, is maintenance. It is widely **estimated that 70% of the cost of software is devoted to maintenance.** No discussion of software quality can be satisfactory if it neglects this aspect.*

What is good code?

And in my opinion, maintenance may be the right quality to focus on.

criteria for judging *quality*

1. **Extendibility** ease the modification of existing source and the addition or removal of source
   … which improves maintainability
2. **Reusability** promote and facilitate the sharing (i.e., reusability) of software components … which improves maintainability
3. **Compatibility** integrate diverse components in large systems … which improves maintainability
4. **Portability** operate agnostic to the specific system … which improves maintainability
5. **Testability** verify correctness before changes reach the product
   … which improves maintainability

The problem with design patterns

Design patterns have proven to be very useful,

~ for the design of object-oriented systems.
The power of design patterns stems from their ability to provide generic solutions to reappearing problems that can be specialized for particular situations

but,

... when the codebase gets large in a language like Java
... and regarding their maintainability from a programming & architecting perspective

The problem with design patterns

1. The traceability of a design pattern in the implementation is often insufficient; often the design pattern is "lost".
2. Since several patterns require an object to forward messages to other objects to increase flexibility, the self problem often occurs.
3. The pattern implementation is mixed with the domain class, the reusability of pattern implementations is often limited.
4. Implementing design patterns can present significant implementation overhead for the software engineer.

The problem with design patterns

1. The traceability of a design pattern in the implementation is often insufficient; often the design pattern is "lost".

hurts: maintainability, readability, testability

2. Since several patterns require an object to forward messages to other objects to increase flexibility, the self problem often occurs.

hurts: predictability, reliability

3. The pattern implementation is mixed with the domain class, the reusability of pattern implementations is often limited.

hurts: reusabiliity, maintainability

4. Implementing design patterns can present significant implementation overhead for the software engineer.

hurts: productivity, maintainability

The problem with design patterns

1. The traceability of a design pattern in the implementation is often insufficient; often the design pattern is "lost".
2. Since several patterns require an object to forward messages to other objects to increase flexibility, the self problem often occurs.
3. The pattern implementation is mixed with the domain class, the reusability of pattern implementations is often limited.
4. Implementing design patterns can present significant implementation overhead for the software engineer.

Bottom Line

1. It's hard to **1) iterate** if patterns are lost.
2. Particularly in a **2) collaborative way** since the high level and code are not always in sync; understanding the vision of another architect is not always obvious.
3. There's redundancy.

Can we make them more explicit?

Design patterns as language constructs

Jan Bosch

LayOM's (Layered Object Model) approach

LayOM's (Layered Object Model) approach

~~Class~~ → LayOM Object

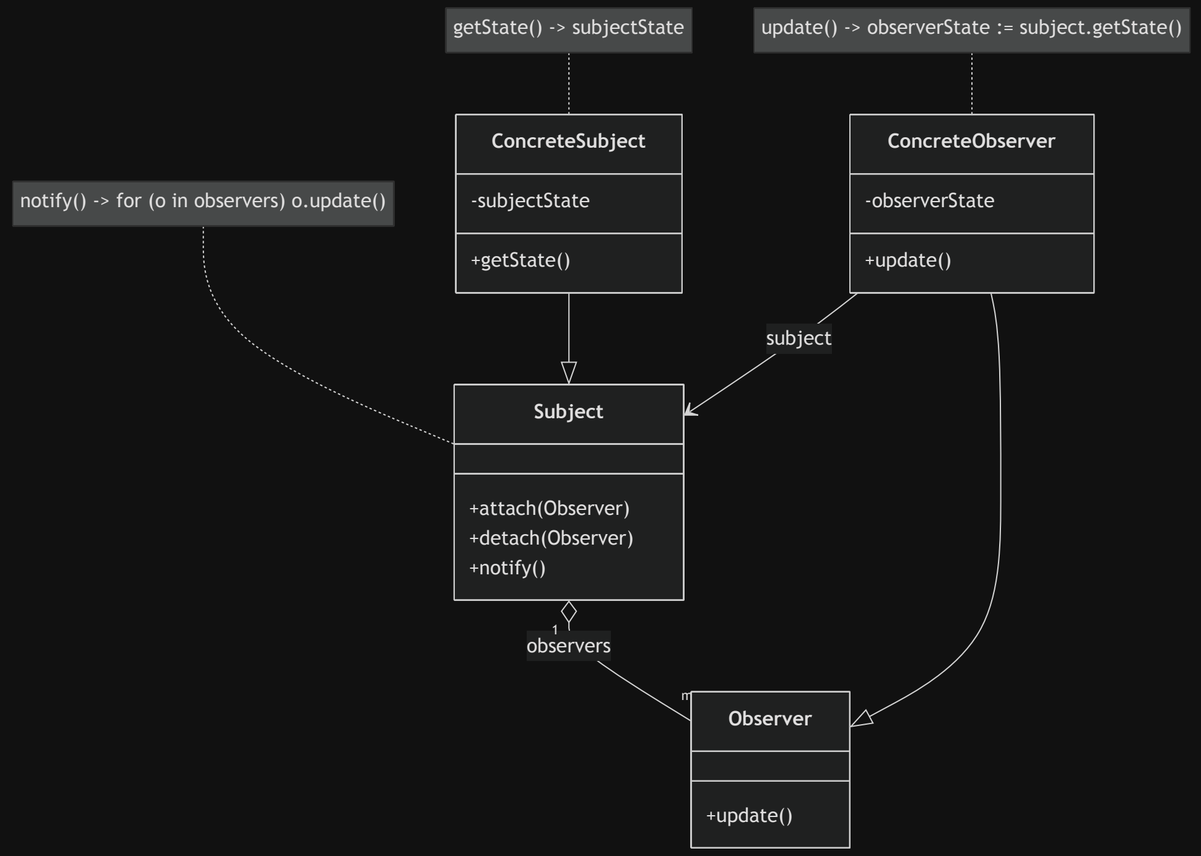
(which transpiles into C++)

Components of a LayOM Object

in addition to traditional instance variables and methods:

- **States:** An abstraction of the object's internal, or "concrete," state. This allows for a simplified, externally visible representation of the object's state.
- **Categories:** An expression that defines a specific group of client objects. This allows the object to treat a subset of its possible clients in a particular way.
- **Layers:** These encapsulate the object, intercepting all incoming and outgoing messages. Layers are organized into classes, with each layer class representing a specific concept, such as a design pattern.

LayOM's (Layered Object Model) approach
Observer Pattern



metaprogramming the

Observer Pattern

A **syntax template** for `Observer` -- built-in to LayOM.

```
<id> : Observer( notify [before|after] on <mess-sel>+ [on aspect <aspect>], ... );
```

For example,

```
priceWatcher : Observer( notify after on setPrice );
```

Observer Pattern

A **syntax template** for `Observer` -- built-in to LayOM.

`<id> : Observer( notify [before|after] on <mess-sel>+ [on aspect <aspect>], ... );`

We can use this in our code.

as a new Layer:

Example:

```
class ObservablePoint
    layers // customer layer faciliates the observer pattern
        st : Observer(notify after on setX on aspect “X-axis”, notify after on setY
                    on aspect “Y-axis”, notify after on moveTo on aspect “Location”);
    ...

    methods
        setX(newX : Location) returns Location
            begin ... end;
        setY(newY : Location) returns Location
            begin ... end;
        moveTo(move : Location2D) returns Location2D
            begin ... end;
    ...
end; // class ObservablePoint
```

1. Notice the resemblance to a standard `class`
2. Notice the `layers` section

But,

1. "Observer" had to be pre-defined and it's inconvenient to invent new **Layer** types. And patterns are relatively easy to implement so this overhead needs to be worth it.
2. It's very not mainstream -- new developers require training. Patterns have to be maintained in a meta-library.

I will admit,

1. It is relatively concise **to use**.
2. It solves the core issues
    1. Traceability
    2. Self-problem frequency
    3. Pattern implementation mixing
    4. Pattern implementation overhead

I will admit,

And its,

1. Powerful.
   An **extensible** paradigm.

- `<id> Adapter (accept <mess-sel>+ as <new-mess-sel>, ...);`
- `<id> Bridge(implement <mess-sel>+ as [<object>.]<method>,..);`
- `<id> Composite ([add is <mess-sel> and] ... multicast <mess-sel>+);`
- `<id> Facade (forward <mess-sel>+ to <object>, ...);`
- `<id> State(if <state-expr> forward <mess-sel>+ to [<mess-sel> <object>], ...);`
- `<id> Observer(notify [before|after] on <mess-sel>+ [on aspect <aspect>], ...);`
- `<id> Strategy(delegate [<mess-sel>+ to <class> [set by <mess-sel>]);`
- `<id> Mediator(forward <mess-sel>+ from <client> to <object>, ...);`

(to show a few)

2. Generalizable to far-reaching, multi-class patterns
3. Compatible with existing C++

we could stop here

But can we make it simpler?

Functional Programming

Stepping back, "can we make the *design* a part of the *language*?"
... perhaps the problem is OOP itself.

Functional programming makes the design process easy.

Patterns are not special techniques, they are the default, idiomatic way of writing code

Pattern or Principle	Functional Programming
Single Responsibility Principle	Functions
Open/Closed Principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory Pattern	Yes, functions
Strategy Pattern	Oh my, functions again!
Decorator Pattern	Functions

Pattern or Principle	Functional Programming
Visitor Pattern	Functions[]

Functional Programming

Patterns are not special techniques, they are the default, idiomatic way of writing code

Functional programming is well suited to the problem of representing design as language because all the challenges that OO design patterns address are related to state management

What is it?

FP is concerned with "pure" code — no side-effects; all variables are immutable; no state.

When systems have no state to protect, high quality design is easy

Functional Programming

We saw,

In **Design by Contract (DbC)**

- preconditions
- postconditions
- invariants

that this philosophy can be robust.

In FP,
Everything's immutable — the ultimate invariant.
Pure functions are ironclad contracts.

It requires a new way of thinking.
But we care because in FP,

Patterns are not special techniques, they are the default, idiomatic way of writing code

Patterns are not special techniques, they are the default, idiomatic way of writing code

For example,

Object Oriented

A trivial example of the **Strategy** pattern,

```
// The contract for the strategy
interface ICalculationStrategy {
    int execute(int a, int b);
}

// Concrete implementations
class AddStrategy implements ICalculationStrategy {
    public int execute(int a, int b) { return a + b; }
}

class SubtractStrategy implements ICalculationStrategy {
    public int execute(int a, int b) { return a - b; }
}
```

```
// The context that uses the strategy
class Calculator {
    private ICalculationStrategy strategy;

    public void setStrategy(ICalculationStrategy strategy) {
        this.strategy = strategy;
    }

    public int calculate(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

Patterns are not special techniques, they are the default, idiomatic way of writing code

For example,

Functional

```
// The "context" is just a higher-order function
function calculator(strategy_function, a, b) {
    return strategy_function(a, b);
}

// The "strategies" are just functions
const add = (a, b) => a + b;
const subtract = (a, b) => a + b;
```

```
// Usage
let result1 = calculator(add, 5, 3); // 8
let result2 = calculator(subtract, 5, 3); // 2
```

Yes, it's a typeless example -- functional **is very compatible** with strong types.
And sure, the code is shorter, but:

Patterns are not special techniques, they are the default, idiomatic way of writing code

Functional

```
// The "context" is just a higher-order function
function calculator(strategy_function, a, b) {
    return strategy_function(a, b);
}

// The "strategies" are just functions
const add = (a, b) => a + b;
const subtract = (a, b) => a + b;
```

```
// Usage
let result1 = calculator(add, 5, 3); // 8
let result2 = calculator(subtract, 5, 3); // 2
```

But notice

What we've gained

1. It is no longer necessary to encapsulate the strategy variable.
   In fact, there aren't any mutable variables whatsoever.
2. This implies ironclad **Design by Contract** -- impossible to violate; perfectly explicit.
3. The use of that pattern would be completely idiomatic to programmers in this language.

Complete idiomatic embedding of the pattern into the language.
No chance for the high level spec to become out of sync.

What's the catch?

Functional programming is, fundamentally, harder to make performant.
It's the classic story of a higher level language that's less performant but more abstract than the system that came before.

This means,
→ **We need a framework and new data structures to make it efficient**

But in many systems, the small overhead is a negligible price to pay.
And no mutable state means no `synchronized` (critical zones).

→ **Functional programs are automatically parallelizable**
And without risk of race conditions.

Thanks!

References

1. Jan Bosch, Design Patterns as Language Constructs, 1996
2. Radu Marinescu, Daniel Ratiu, Quantifying the quality of object-oriented design: The factor-strategy model, 2004
3. David M. Tilbrook, John McMullet, Washing Behind Your Ears: Principles of Software Hygiene, 1990
4. Reinhold Plösch et al, 'Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles', 2016
5. Scott Wlaschin, Functional Programming Design Patterns, 2014
6. Meyer Bertrand, Design by Contract, 1986

// TODO: need more FP refs
// and less dated refs would be nice

// TODO: pulled the "automatically parallelizable" FP code as anecdotal -- need refs