CS 247

Software Engineering Principles

Chapter 9

Introduction to Design Patterns

Victoria Sakhnini

Table of Contents

Someone Has Already Solved Your Problems 4
The One Constant: Change 5
Design Principle: Encapsulate What Varies 6
A motivating example: ducks that can suddenly fly 6
Applying the principle 8
Where to draw the seam 11
Design Principle: Program to an Interface, Not an Implementation 11
The animal example 11
Three meanings of "interface." 12
What it looks like when you violate it 12
The result 13
Design Principle: Favour Composition Over Inheritance 14
Why is inheritance so tempting? 14
Three real problems with inheritance 14
Composition: the alternative 14
When inheritance is still right 15
A second worked example: pizza toppings 16
The Power of a Shared Pattern Vocabulary 17
Five ways the vocabulary pays off 18
Patterns Are Not Libraries or Frameworks 18
What libraries and frameworks do 19
What patterns do (and don't) 19
But libraries and frameworks often use patterns 20
Definition: Pattern, Context, Problem, Solution 20
The context 20
The problem 21
The solution 21
The Gang of Four template 21
Three Categories of Patterns 23
Creational patterns 23
Structural patterns 24
Behavioural patterns 25
Class Patterns vs. Object Patterns 26

Class patterns 27
Object patterns 27
Why the distinction matters 28
Thinking in Patterns 28
Keep it simple 28
How to tell when a pattern is justified 29
Let the patterns emerge 29
Patterns are not set in stone 30
The role of OO basics 30
Summary 31

Someone Has Already Solved Your Problems

But knowing how the language works is not the same thing as knowing what to build. A small program with three classes has only a handful of plausible designs, and most of them are fine. A serious application has hundreds of classes, and the difference between a maintainable design and an unmaintainable one is the difference between an engineering team that ships features every week and a team that spends every sprint apologizing for last month's emergency fixes. The hard part of object-oriented design is not the syntax; it is deciding which classes should exist, which classes should depend on which, where the seams of the system should fall, and what should be allowed to change easily over time.

Here is the encouraging news: most of these design questions are not new. The same problems, "how do I add a new shape to the renderer without modifying the renderer?", "How do I notify a dozen unrelated widgets that the document has changed?" "How do I build an object whose construction needs five different optional steps in a defined order?" has been solved, refined, and re-solved in production code for forty years. The accumulated wisdom about how to solve them well has been collected into a catalogue of named, reusable solutions called *design patterns*. This chapter introduces what a design pattern is, what it is *not*, and the three small but powerful design principles that underlie almost every pattern in the catalogue.

Patterns are not a magic wand. They will not pick the right classes for you, write your code, or compensate for a confused understanding of the problem. What they give you is something more modest and more useful: a vocabulary of well-understood, battle-tested solutions, so that when you recognize that you are facing a problem someone has already solved, you do not have to reinvent the solution from first principles. You get to start where they finished.

A small warning before we begin. Beginning programmers, once they discover patterns, tend to use them everywhere. This is the wrong reflex. Patterns are a tool of last resort for an experienced designer who has spotted a recurring structure in a real codebase, not a starting point for a new design. Most of this chapter is about how patterns think; the chapters that follow are about specific patterns and when to reach for them. By the end, you should be able to read a pattern description, see the structural shape it solves, and recognize that shape in your own code. That recognition, not memorization of any specific pattern, is the goal.

Throughout the chapter, we will frame design patterns in terms of three ideas that recur in almost every pattern you will study:

- **Encapsulate what varies.** Find the part of your design that changes and isolate it behind an interface so the rest of the code does not have to change when it does.
- **Program to an interface, not an implementation.** Express dependencies in terms of abstract supertypes so that any conforming concrete type can be substituted at runtime.

- **Favour composition over inheritance.** Build complex objects by combining simpler ones at runtime rather than by extending a class hierarchy at compile time.

Each of these principles will get its own section, with motivating examples and the C++ machinery you need to apply them. We will then look at how the design-patterns community categorizes the catalogue (creational, structural, behavioural; class vs. object) and close with practice problems that ask you to reason about which principle applies in which situation. The patterns themselves, Strategy, Observer, Decorator, Singleton, and the rest, are the subject of the chapters that follow.

The One Constant: Change

If you asked a hundred working software engineers what the single most reliable fact about software is, true regardless of language, domain, company size, or decade, you would get a depressing variety of cynical answers, most of which boil down to the same thing: the requirements will change. The feature that was definitely the most important in the world on Monday morning will be quietly dropped by Wednesday afternoon. The third-party library you depended on will be deprecated. The data format the user told you was "absolutely fixed" will need a new field. The class that started out modelling a single Customer will, three releases from now, need to handle three different kinds of Customer, each with its own billing rule.

Software development is, in this sense, governed by exactly one law: change is the only constant. Almost every important principle of software design exists because of this law. Modularity exists to limit the radius of damage when something changes. Information hiding exists so the implementation can change without touching clients. Inheritance, composition, polymorphism, and templates exist so that pieces of behaviour can be swapped out without rewriting the surrounding code. The design patterns in this course are all answers to questions of the form: when this part of the system changes, how do I keep the change from rippling through the rest of the system?

> **? Question**
>
> Take a moment to think about the last non-trivial program you wrote, perhaps the Rational class or the Course Schedule application from earlier chapters. If you had to add a new feature today (a new operation, a new kind of input, a new output format), how many existing files would you need to open and edit?
>
> That number is a rough measure of how well isolated the change is in your design. A well-designed program is one in which most additions cost one new file and zero edits to existing files. A poorly-designed program is one in which every new feature touches everything. The difference between the two is mostly a matter of where the variation has been hidden.

Most design patterns address a specific axis of change. The Strategy pattern addresses change in the algorithm a class uses; the Observer pattern addresses change in who needs to be notified when something happens; the Decorator pattern addresses change in the responsibilities of a single object; the Factory pattern addresses change in which concrete subclass should be created. Once you internalize the idea that good designs absorb change cheaply, the patterns start to feel less like clever tricks and more like a catalogue of "where to put the seams", places where you can predict change will happen and have already prepared a graceful way to accept it.

> 💡 **Tip**
>
> **Read patterns from the back.** When you study an unfamiliar pattern, do not start with the diagram. Read the *intent* first, then the *motivation*, what problem does this pattern solve, and what would happen without it? Once you have that in your head, the diagram will make sense in five seconds. Without it, the diagram is just boxes and arrows.

Design Principle: Encapsulate What Varies

The first design principle is the most general, and many of the others are special cases of it:

Identify the aspects of your application that vary, and separate them from what stays the same.

That sentence is a mouthful, so let us unpack it. In any non-trivial program, some things are stable across many releases, while others change every few weeks. The class invariants of a Rational do not change; the algorithms used for billing customers do. The fact that the system has "orders" does not change; the rules for which orders qualify for free shipping change every holiday season. The principle says: locate the bits that are going to change and pull them into their own classes, behind their own interfaces, so that when they do change, you can swap out the implementation without touching the rest of the system.

A motivating example: ducks that can suddenly fly

Imagine a duck-pond simulation. You have a base class Duck with two non-virtual member functions, `quack()` and `swim()`, and two derived classes, `MallardDuck` and `RedheadDuck`, that each look like 35 lines of perfectly normal C++.

```
class Duck {
public:
    void quack() { std::cout << "Quack!\n"; }
    void swim()  { std::cout << "All ducks float on water\n"; }
    virtual void display() const = 0;     // looks different per duck
    virtual ~Duck() = default;
};

class MallardDuck : public Duck {
public:
    void display() const override { std::cout << "I look like a
Mallard\n"; }
};

class RedheadDuck : public Duck {
public:
    void display() const override { std::cout << "I look like a
Redhead\n"; }
};
```

Now your product manager appears and asks for a small feature: ducks should be able to fly. Easy enough, add a `fly()` function to the base class.

```
class Duck {
public:
    void quack() { std::cout << "Quack!\n"; }
    void swim()  { std::cout << "All ducks float on water\n"; }
    void fly()   { std::cout << "I'm flying!!\n"; }   // newly added
    virtual void display() const = 0;
    virtual ~Duck() = default;
};
```

It compiles, ships, and looks great in the demo. The next morning, someone files a bug: rubber ducks are flying around the screen at runtime. You had forgotten that a previous developer had added a `RubberDuck` subclass. Rubber ducks cannot fly. They also cannot quack; they squeak, but the inherited quack still prints "Quack!".

Your first instinct is to override the problematic functions in `RubberDuck`:

```
class RubberDuck : public Duck {
public:
    void display() const override { std::cout << "I am a rubber duck\n"; }
    void quack()                  { std::cout << "Squeak!\n"; }       //
override
    void fly()                    { /* do nothing, rubber ducks can't fly
*/ }
};
```

> ⚠ **Warning**
>
> This already "works" in a narrow sense, but it has set a trap. Every time a new subclass is added, the developer must remember to consider which inherited behaviours need to be overridden. If a future DecoyDuck (a wooden duck used by hunters) is added and the developer forgets to override quack, it will quack. The defaults are wrong by default, and silent defaults that are wrong are the worst kind of bug.

The next request comes in: please add a `ModelDuck` that doesn't fly today but might in some other configuration; and please add a few weird variations, a duck that flies with a rocket booster, a duck that flies the way mallards do, a duck whose flight behaviour changes over the course of the simulation. At this point, the override-locally strategy has collapsed: behaviours that should be reusable ("flies with wings") are duplicated across many subclasses, behaviours that should be selectable at runtime are baked into the class at compile time, and you can no longer answer the simple question "can this duck fly?" without reading every subclass.

Applying the principle

The principle says: identify what varies and separate it. What varies in this example? The flying and quacking behaviour varies from duck to duck; some fly with their wings, some don't fly at all, and some fly mechanically. The swimming behaviour and the basic identity of being a duck do not vary. So the design should be: pull flying and quacking out of the Duck class entirely, into their own little interfaces, and let each Duck object refer to one of each.

```
// The varying behaviours, each behind an interface.

class FlyBehaviour {
public:
    virtual void fly() const = 0;
    virtual ~FlyBehaviour() = default;
};
```

```
class QuackBehaviour {
public:
    virtual void quack() const = 0;
    virtual ~QuackBehaviour() = default;
};

// Concrete behaviour implementations.
class FlyWithWings : public FlyBehaviour {
public:
    void fly() const override { std::cout << "I'm flying with wings!\n"; }
};

class FlyNoWay : public FlyBehaviour {
public:
    void fly() const override { /* do nothing */ }
};

class Quack_ : public QuackBehaviour {
public:
    void quack() const override { std::cout << "Quack!\n"; }
};

class Squeak : public QuackBehaviour {
public:
    void quack() const override { std::cout << "Squeak!\n"; }
};

class MuteQuack : public QuackBehaviour {
public:
    void quack() const override { /* silence */ }
};

// Duck now composes its behaviour rather than inheriting it.
class Duck {
    std::unique_ptr<FlyBehaviour>   flyBehaviour;
    std::unique_ptr<QuackBehaviour> quackBehaviour;
public:
    Duck(std::unique_ptr<FlyBehaviour> fb, std::unique_ptr<QuackBehaviour>
qb)
        : flyBehaviour(std::move(fb)), quackBehaviour(std::move(qb)) {}
    void performFly()   const { flyBehaviour->fly(); }
    void performQuack() const { quackBehaviour->quack(); }
    void swim()         const { std::cout << "All ducks float on water\n";
}
    virtual void display() const = 0;
    virtual ~Duck() = default;
};
```

```
class MallardDuck : public Duck {
public:
    MallardDuck()
      : Duck(std::make_unique<FlyWithWings>(), std::make_unique<Quack_>())
{}
    void display() const override { std::cout << "I look like a
Mallard\n"; }
};

class RubberDuck : public Duck {
public:
    RubberDuck()
      : Duck(std::make_unique<FlyNoWay>(), std::make_unique<Squeak>()) {}
    void display() const override { std::cout << "I am a rubber duck\n"; }
};
```

With this design, the original embarrassments simply cannot occur. A `RubberDuck` never has the ability to fly because it never holds a `FlyWithWings`, it composes a `FlyNoWay` instead. Adding new behaviours (rocket-powered flight, a robotic quack) means creating a new class that implements the right interface; the existing Ducks need no change. And because the behaviour pointers are non-const, you can swap a duck's flight behaviour at runtime:

```
auto duck = std::make_unique<MallardDuck>();
duck->performFly();             // I'm flying with wings!
                                // later, somewhere in the simulation:
duck->setFlyBehaviour(std::make_unique<FlyWithRocket>());
duck->performFly();             // Whoosh, with the rocket!
```

This is, almost word for word, the Strategy pattern from the Gang of Four catalogue. We will study it formally in a later chapter; for now, observe that all we did was take the principle (encapsulate what varies) literally, and the design emerged.

> 💡 **Tip**
>
> **Naming convention.** In examples, we use the suffix Behaviour for these little extracted interfaces (FlyBehaviour, QuackBehaviour, BillingStrategy, SortStrategy). The point is to make the role explicit: this object exists *only* to encapsulate a behaviour that may vary.

Where to draw the seam

Deciding what to encapsulate is the genuinely hard part. Pull out too little, and you have not bought any flexibility; pull out too much, and you have a hundred tiny interfaces and a configuration file longer than the code. Three rules of thumb help:

- **Encapsulate things that already differ.** If two subclasses already override the same operation in incompatible ways, that operation is varying and should probably be pulled into a behaviour class.
- **Encapsulate things you have changed recently.** If you have edited the same five lines four times this quarter, those lines are varying. Predict the next edit by extracting them now.
- **Do not encapsulate the speculative.** If you cannot point to a concrete second case that wants different behaviour, the abstraction is premature. Wait until you have two concrete needs before designing for variation.

Design Principle: Program to an Interface, Not an Implementation

The second principle is closely related to the first, but it is about what your code says, not just about where the seams are drawn:

Program to an interface, not an implementation.

In C++ terms, this is the instruction to write your variables, function parameters, return types, and member fields in terms of *supertypes*, abstract base classes (or, where appropriate, templates), rather than in terms of specific concrete classes. The principle does not say "use inheritance everywhere." It says: when one piece of code holds onto another, the type it holds should be the most general type that supports the operations it actually needs.

The animal example

Recall the Duck story. The first design hard-coded `MallardDuck` and `RubberDuck` everywhere a `Duck` appeared. Any client function had to choose, at compile time, which concrete duck it was operating on. With the refactored design, the same client code can be written against the supertype:

```
void duckSimulation(const Duck& d) {
    d.display();
    d.performQuack();
    d.performFly();
    d.swim();
}
```

```
int main() {
    MallardDuck m;
    RubberDuck  r;
    duckSimulation(m);   // works
    duckSimulation(r);   // also works, same code path
}
```

The function `duckSimulation` never mentions a specific kind of duck. It takes a const Duck&, a reference to the abstract supertype, and the dynamic dispatch through display, performQuack, performFly, and the contained behaviour objects do the rest. New kinds of duck can be added without ever opening `duckSimulation` again.

Three meanings of "interface."

The word interface is overloaded in C++ in a way that confuses students coming from Java. There are at least three things that go by that name in this principle, and the principle applies to all of them:

- **Abstract base class with pure virtual functions.** The closest analogue to a Java interface. Holds no data, declares only virtual functions, and has a virtual destructor. This is what we mean by "interface" most often in the rest of this course.
- **Abstract base class with some default behaviour.** A class that has some pure virtual functions but also some implemented ones, providing a partial template for derived classes. Still acceptable as the "supertype" of the principle.
- **A template type parameter constrained by a concept (or by use).** In modern generic C++, the "interface" is the set of operations the template requires. std::sort programs to the iterator "interface", anything that supports ++, *, and < works.

The first two are runtime polymorphism; the abstract supertype is real, and the concrete type is decided when the object is created. The third is compile-time polymorphism: there is no abstract supertype at all, but the principle still applies because the template body uses only operations that any conforming type provides. Both are valid ways to satisfy this design principle. The question is whether the choice of concrete type needs to vary at run time (favour virtual functions) or only at compile time (favour templates).

What it looks like when you violate it

The smell of code that violates this principle is *downcasting*, code that takes an abstract reference, asks at runtime which concrete subtype it really is, and branches:

```
void duckSimulation(Duck& d) {           // BAD example, do not write this
    if (auto m = dynamic_cast<MallardDuck*>(&d)) {
        m->mallardSpecificQuack();
    } else if (auto r = dynamic_cast<RubberDuck*>(&d)) {
        r->squeak();
    } else {
        d.performQuack();
    }
}
```

⚠ Warning

If you find yourself using dynamic_cast to determine which concrete subclass you have and then calling subclass-specific functions, the abstract interface is missing some operations. Either promote the missing operation to the supertype (so every duck has a performQuack), or recognize that the operation is polymorphic along a different axis (the Visitor pattern, covered later, addresses this). What you must not do is sprinkle dynamic_cast throughout the codebase. Each cast is a place where adding a new subclass forces an edit to existing code, and the variation has effectively escaped its encapsulation.

The result

Following this principle has two payoffs. First, the same client code can be reused with any current or future implementation, including, crucially, mock or test implementations supplied during unit testing. Second, when a change to one implementation is required, the change does not propagate beyond that implementation's source file.

💡 Tip

"Program to a supertype" is often the more accurate slogan. The word "interface" tempts readers familiar with Java to think that this principle is about declaring interfaces. In C++ the principle is broader: variables, parameters, and return types should be declared at the highest level of abstraction that still supports the operations the code needs. Sometimes that level is an abstract base class; sometimes it is an STL concept like InputIterator; sometimes, for a leaf utility that genuinely operates on a specific type, it is the concrete type itself, and that is fine.

Design Principle: Favour Composition Over Inheritance

The third principle is the one that surprises students the most, because it pushes against the way most introductory object-oriented courses teach you to think:

Favour composition over inheritance: HAS-A can be better than IS-A.

Inheritance is the headline feature of object-oriented programming, so it is tempting to use it whenever two classes seem related. This principle says: stop. Before you make B inherit from A, ask whether B *is* an A in the strict, all-the-time, can-be-substituted-for sense, or whether B merely *has* some A-like behaviour. The two situations call for different mechanisms.

Why is inheritance so tempting?

Inheritance offers genuine benefits. It lets you define a common base class once and have many derived classes reuse it. It makes substitution easy: every Mallard is a Duck, so anywhere a Duck is expected, a Mallard fits. And it gives you a single name (the base class) for a family of related types, which is good for documentation and for polymorphism.

In our Duck story, all four ducks really were ducks, so deriving them from a common Duck base was sensible. The mistake was not *using* inheritance; it was using inheritance to encode the *flying* and *quacking* behaviours, which were the parts that varied. Inheritance is appropriate for the parts of the design that genuinely are stable IS-A relationships; it is wrong for the parts that vary.

Three real problems with inheritance

Inheritance is a single, very strong commitment with three uncomfortable consequences.

- **It is fixed at compile time.** A Mallard is a Duck the moment it is constructed and remains one for the rest of its life. You cannot turn a Mallard into a RubberDuck at runtime by re-parenting it. If the variation you want is dynamic, inheritance is the wrong tool.
- **It is a leaky contract.** Derived classes do not just inherit operations; they inherit the implementation of those operations, including any assumptions baked into the base. Change the base, and you may break every derived class. The base class is, in this sense, the most fragile part of the system.
- **It is composed badly.** If a Duck has two independently varying behaviours (flying style and quacking style), the inheritance hierarchy needs one subclass for every *combination*, a Cartesian explosion that grows quadratically with each new axis of variation. Composition only adds new pieces along the axis that actually changed.

Composition: the alternative

Composition is a relationship in which one class holds another as a member. The duck design from Section 3.2 was a composition design: each Duck *had* a FlyBehaviour and *had* a QuackBehaviour, rather than *being* a particular flying-quacking variety. The result was a small, flexible kit: any flying behaviour

can be combined with any quacking behaviour, and either can be replaced at runtime without touching the Duck class.

Notice how Chapter 8 prepared you for this. The UML notation distinguishes association (*uses*), aggregation (*part-of, shared*), and composition (*part-of, exclusive*), all of which are HAS-A relationships, from generalization (*is-a*). The principle we are now stating is that, in a tie, the HAS-A relationships are usually the better design choice. Composition is the mechanism that the Strategy, State, Decorator, Bridge, Observer, and many other patterns rely on.

When inheritance is still right

This is not a blanket prohibition. Inheritance is the right answer when:

- **The relationship is genuinely IS-A.** A RoyaltyFreeAccount is a kind of Account; a Mallard is a kind of Duck. A Square is *not* a kind of Rectangle if you can resize the Rectangle, a famous example of how IS-A in mathematics is not always IS-A in software.
- **The base provides default behaviour you actively want.** If most subclasses share the same implementation of an operation and only a few override it, inheritance lets you write the default once. With pure composition, you would resupply it every time.
- **Polymorphic substitution is the whole point.** If clients are expected to treat all the subclasses uniformly through the base, as in the Duck& parameter of duckSimulation, then the base class is exactly the abstraction the clients depend on, and inheritance is how you build it.

The slogan "favour composition over inheritance" does not say *never inherit*; it says *prefer the lighter commitment when the design admits it*. Composition can be added incrementally and reversed cheaply. Inheritance is a permanent architectural decision. When in doubt, start with composition.

Tricky

Consider this hierarchy:

```
class Stack : public std::vector<int> {
    // Reuse std::vector's implementation; provide push, pop, top
};
```

What is wrong with it? It compiles, the operations work, and you saved typing the storage management. The problem is that Stack IS-A vector, clients can call operator[], erase(iterator), insert, everything. The whole point of a Stack, that elements come out in LIFO order through push/pop/top, has been silently violated by inheriting the full interface. The correct design is a composition: Stack has a private std::vector<int> member and exposes only the three operations of the Stack abstraction. This is exactly how std::stack is implemented.

A second worked example: pizza toppings

Suppose we are building a pizza ordering system. A pizza has a base cost, and each topping adds to the price and the description. Let us try the inheritance design first:

```
class Pizza { public: virtual double cost() const = 0; virtual std::string
desc() const = 0; };
class CheesePizza : public Pizza { /* base price */ };
class CheeseMushroomPizza : public CheesePizza { /* +mushroom */ };
class CheeseMushroomOlivePizza : public CheeseMushroomPizza { /* +olive */
};
class CheeseOlivePizza : public CheesePizza { /* +olive */ };
// ... and on, and on
```

With *n* toppings, the number of classes is 2^*n*. For 10 toppings, we would need over 1,000 classes. The HAS-A approach is much cheaper:

```
class Pizza {
public:
    virtual double cost() const = 0;
    virtual std::string desc() const = 0;
    virtual ~Pizza() = default;
};

class CheesePizza : public Pizza {
public:
    double cost() const override { return 8.0; }
    std::string desc() const override { return "Cheese pizza"; }
};

class ToppingDecorator : public Pizza {
protected:
    std::unique_ptr<Pizza> wrapped;
public:
    explicit ToppingDecorator(std::unique_ptr<Pizza> p) :
wrapped(std::move(p)) {}
};

class Mushroom : public ToppingDecorator {
public:
    using ToppingDecorator::ToppingDecorator;
    double cost() const override { return wrapped->cost() + 1.0; }
    std::string desc() const override { return wrapped->desc() + ",
mushroom"; }
};
```

```
class Olive : public ToppingDecorator {
public:
    using ToppingDecorator::ToppingDecorator;
    double cost() const override { return wrapped->cost() + 0.5; }
    std::string desc() const override { return wrapped->desc() + ",
olive"; }
};

// Usage:
std::unique_ptr<Pizza> order =
    std::make_unique<Olive>(
        std::make_unique<Mushroom>(
            std::make_unique<CheesePizza>()));
std::cout << order->desc() << " : $" << order->cost();
// Cheese pizza, mushroom, olive : $9.5
```

Adding a new topping requires one new class. Adding a new base pizza also requires one new class. The number of combinations a customer can order is unlimited, while the number of classes grows linearly with the number of variations. This is the Decorator pattern, which we will study in detail later in the course. The only point now is that none of it would have been possible if every topping had to be expressed as a subclass of a particular pizza.

The Power of a Shared Pattern Vocabulary

So far, we have argued that patterns matter because they capture good design. There is a second, equally important reason patterns matter: they give development teams a shared vocabulary.

Consider what happens when two designers, both fluent in patterns, discuss a problem. One says: "The simulator subsystem has too many code paths that depend on the type of vehicle being simulated. I think we should extract the steering and acceleration logic as Strategy objects." The other replies: "Agreed, and we will need a Factory in the harness to construct them." In two short sentences, the entire shape of the proposed refactor is communicated: what is being moved, into which structural arrangement, and with which adjustments at the boundary. Anyone fluent in patterns can read that exchange and immediately picture the resulting class diagram.

Compare that to the equivalent conversation between two designers who are not pattern-fluent: "So I think we need to take all the if-statements out of the simulator and put each branch in its own class, then give the simulator a pointer to one of these classes and choose which one when the vehicle is built. We cannot just construct them directly in the simulator, because that would reintroduce the dependency, so we need something else to make the right one for us." Same idea, four times the words,

much more room for miscommunication, and much less confidence that the listener actually understood what was meant.

Five ways the vocabulary pays off

- **You communicate qualities, not just names.** When you say "Observer," you are also saying that there is a one-to-many dependency, that dependents update themselves rather than being polled, and that adding a new dependent does not require changing the subject. Five sentences of design context, compressed into one word.
- **Designs are pinned down precisely.** Three different developers, asked to draw the design that goes with the name "Strategy," will produce essentially the same diagram. The pattern name picks out a specific structural shape, not an approximate intention.
- **Discussion stays at the design level.** Pattern talk lets you reason about how classes are arranged without diving into implementation details. "Should this be a Strategy or a Template Method?" is a higher-leverage conversation than "Should we use a virtual function or a function pointer?"
- **Teams move faster.** A team well-versed in patterns can sketch a non-trivial design on a whiteboard in twenty minutes, agree on it, and disperse. Without that vocabulary, the same conversation often takes two hours and ends with everyone politely pretending they remember what was decided.
- **Junior developers ramp up.** New hires see senior developers confidently using pattern names and start learning the catalogue. Once a junior developer knows what an Observer is, an entire genre of bugs, "this widget did not refresh when the model changed", becomes recognizable and fixable.

> 💡 **Tip**
>
> **Build the vocabulary deliberately.** In a code review, name the pattern: "this is essentially a Decorator," or "the underlying shape here is Template Method." That single sentence teaches more than ten paragraphs of "you should refactor." Soon, the rest of the team is naming patterns back at you, and the team's collective design ability rises faster than any one individual's.

Patterns Are Not Libraries or Frameworks

A natural reaction at this point is: if design patterns are so well understood, why can't someone just write them down once as a library so we can call them instead of re-implementing them every time? It is a fair question, and the answer is the central point of this section: patterns operate at a different level of abstraction than libraries or frameworks do.

What libraries and frameworks do

A *library* is a bag of pre-written code that solves specific, concrete problems. The C++ standard library, <vector>, <string>, <algorithm>, <thread>, these are libraries. They give you concrete classes and concrete functions with well-defined behaviour. You call them; they execute; control returns to you. Libraries are an excellent way to reuse *code*.

A *framework* is a more ambitious arrangement: a partially-built application with holes you fill in. Frameworks invert the control direction; your code is called by the framework, not the other way around. Qt and Boost.Asio are frameworks of various sizes; the Hollywood principle "don't call us, we'll call you" describes the framework-user relationship. Frameworks are an excellent way to reuse *architecture*.

What patterns do (and don't)

A *design pattern*, in contrast, is not code at all. It is a documented description of how classes and objects should be arranged to solve a recurring design problem. The arrangement is precise, the same names, the same connections, the same responsibilities, but the actual class names, member function bodies, and data types are determined by the specific application that uses the pattern. Patterns are an excellent way to reuse *experience*.

To make the difference concrete:

	Library	Framework	Pattern
What you get	Concrete classes & functions you call	An application skeleton that calls you	A diagram and a description
When applied	At link time	At build time (you fill in callbacks)	When you write the code
Reuse target	Code	Architecture	Design experience
Example	std::vector	Qt event loop	Strategy, Observer, Singleton
You adapt by	Choosing arguments	Subclassing & overriding	Rewriting in your own classes

The crucial row is the last one. You do not *import* a pattern; you *apply* one. When you use the Observer pattern in your stock-ticker application, you will write your own Stock, StockObserver, Display, and EmailAlert classes. The pattern tells you how to connect them; it does not give you the classes. That is why no one has "written a library of patterns"; the moment you generalize a pattern enough to put in a library, you have lost the part that made it useful in your specific application.

But libraries and frameworks often use patterns

This is a separate observation worth absorbing: many production libraries and frameworks are themselves built using design patterns. The iostream hierarchy uses the Decorator pattern (an ofstream wraps a filebuf wraps a streambuf base). The STL's std::function uses the Type Erasure pattern. Qt's signal/slot mechanism is an Observer. Once you know patterns, the source code of major libraries becomes much easier to read because you recognize the familiar shapes instead of seeing a soup of classes.

> **? Question**
>
> Which of these is a design pattern, which is a library, and which is a framework?
>
> - 1. std::sort, a function template that sorts a range using a user-supplied comparator.
> - 2. The std::stack container adaptor, which exposes only push/pop/top over a sequence container.
> - 3. "A function object is held by another class and used to vary an algorithm's behaviour."
> - 4. A unit-test runner that discovers your TEST(name) macros, constructs the test objects, and calls them in order.
>
> Answers: (1) library, concrete code you call. (2) library, though the technique it uses internally is the Adapter pattern. (3) pattern, a verbal description of a design shape (Strategy). (4) framework, you supply the test code, and the framework runs the application.

Definition: Pattern, Context, Problem, Solution

It is time to define a design pattern precisely. The standard definition, due to Christopher Alexander (an architect, not a software engineer, whose 1977 book is the source of the whole movement) and adopted by the software design pattern community, is short:

A pattern is a solution to a problem in a context.

That sentence contains three deliberately chosen words, and a pattern description is only as good as its account of all three.

The context

The context is the situation in which the pattern applies. It describes the environment, the constraints, and the recurring shape of the problem. "You have a class hierarchy whose subclasses differ mostly in how a particular operation is implemented" is a context. "You have an object that should never be instantiated more than once" is a context. "You have a third-party data source whose interface does not match what your code expects" is a context. A pattern that does not match the context does not apply, even if the diagram appears superficially relevant.

Patterns explicitly describe their context because the same diagram can be the right answer in one situation and the wrong answer in another. A singleton instance is appropriate when there really is exactly one in the universe (the system logger, the configuration registry) and inappropriate as a generic "global access" technique. Reading the context section of a pattern description is how you learn when not to apply it.

The problem

The problem is the goal you are trying to achieve in this context, along with the constraints that govern it. Goals are easy to write; constraints are where the pattern description earns its keep. "I want to make this algorithm pluggable" is a goal. The constraints are usually variations of: it has to be selectable at runtime; clients of the algorithm should not change when a new algorithm is added; the algorithm should not have access to the client's private state; and so on. Taken together, these constraints narrow the design space to the point where only a small family of solutions is plausible.

Two patterns can share a goal but differ in their constraints, which is exactly why they are distinct patterns. Strategy and Template Method both let an algorithm vary, but Strategy lets it vary per-object at runtime, whereas Template Method lets it vary per-subclass at compile time. The choice between them depends on which set of constraints fits the application.

The solution

The solution is what you came for: a general structural arrangement of classes and objects, usually presented as a UML class diagram, sometimes with a sequence diagram, sometimes with participant descriptions, and sometimes with code skeletons, that anyone can adapt to fulfil the stated goal under the constraints. The word "general" is doing important work. A pattern solution is not concrete code; it is a template you customize. In your application, the participants will have application-specific names; their methods will perform application-specific work. What you reuse is the arrangement.

The Gang of Four template

When you read a pattern in the Gang of Four ("GoF") book, by Gamma, Helm, Johnson, and Vlissides, published in 1994 and still the authoritative reference, each pattern is documented using the same template, of which the most important fields are:

Field	Purpose
Pattern Name	A single noun phrase that becomes part of the design vocabulary (Strategy, Observer, Singleton…)
Intent	A one-sentence summary of what the pattern does
Motivation	A worked example that illustrates the problem
Applicability	When to use the pattern, the contexts in which it applies
Structure	A UML class diagram showing the participants
Participants	The roles each class plays in the diagram
Collaborations	How the participants work together at runtime (often a sequence diagram)
Consequences	The trade-offs of using the pattern, what you gain, what you give up
Implementation	Language-specific notes on how to realize the pattern in code
Sample Code	Often C++ or Smalltalk, occasionally pseudocode
Known Uses	Real systems that use the pattern
Related Patterns	Patterns that are similar, often-paired-with, or sometimes-confused-with this one

When you read a new pattern, the four fields that matter most for understanding are *Intent*, *Motivation*, *Applicability*, and *Consequences*. The Structure diagram is easier to absorb once those four are in your head; the Sample Code is for when you are ready to apply the pattern in your own work.

> 💡 **Tip**
>
> In this course, when we present a pattern in later chapters, we will deliberately follow a structure close to the GoF template: name, intent, motivating problem, the design before and after the pattern, structure, participants, consequences, and a worked C++ example. The names and the structure are the same across patterns, so as you read more of them the cognitive cost of reading each new one decreases.

Three Categories of Patterns

The original GoF catalogue contains twenty-three patterns. To make so many patterns easier to navigate, the authors classify them along two independent axes. The first, and more famous, axis is the *purpose* of the pattern: creational, structural, or behavioural.

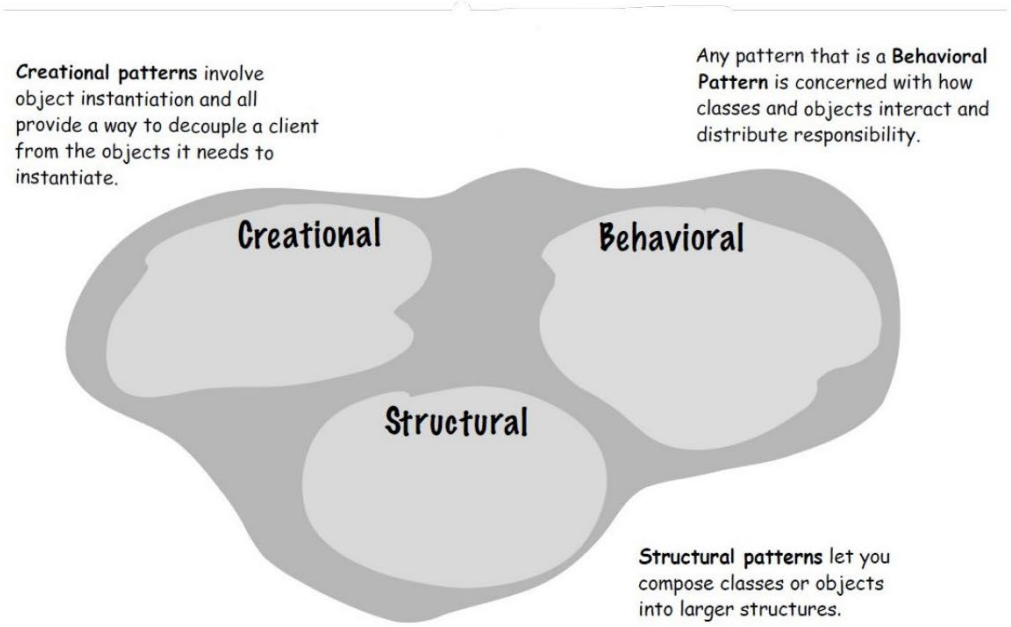

Figure 2: The three pattern categories by purpose. Creational patterns make new objects. Structural patterns compose objects. Behavioural patterns govern interactions among objects.

Creational patterns

Creational patterns are about object instantiation. They decouple the code that *uses* an object from the code that *creates* it, so that the client does not need to know which concrete class is being instantiated. This is the most direct application of "program to an interface": rather than the client writing `std::make_unique<MallardDuck>()`, which names a concrete class, the client asks a factory for "a duck" and gets whichever concrete duck the application is currently configured to produce.

Five patterns sit in this category:

Pattern	One-line intent
Singleton	Ensure a class has exactly one instance, and provide a global access point to it.
Factory Method	Let a base class declare an operation that creates an object, but let subclasses decide which concrete class to instantiate.
Abstract Factory	Provide an interface for creating families of related objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, so the same construction process can create different representations.
Prototype	Create new objects by copying an existing prototype rather than by calling a constructor directly.

Creational patterns share a single observation: deciding which concrete class to make is itself a thing that can vary, and pulling that decision behind an interface is just one more application of the encapsulate-what-varies principle.

Structural patterns

Structural patterns are about how classes and objects are composed to form larger structures. They are the patterns you reach for when you have classes that almost-but-not-quite work together: one class's interface is wrong for the other, one class's responsibilities have grown too large, one class needs to stand in for another. Structural patterns are mostly about adjusting interfaces and relationships, so the pieces fit.

Seven patterns sit here:

Pattern	One-line intent
Adapter	Convert a class's interface into a different interface that clients expect.
Bridge	Decouple an abstraction from its implementation so they can vary independently.
Composite	Compose objects into tree structures so clients can treat individual objects and compositions uniformly.
Decorator	Attach additional responsibilities to an object dynamically, without subclassing.
Facade	Provide a single, simple interface to a complex subsystem.

Pattern	One-line intent
Flyweight	Share many fine-grained objects efficiently by separating intrinsic state from extrinsic.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Behavioural patterns

Behavioural patterns describe how classes and objects interact and distribute responsibility. They focus on communication: which object talks to which, in what order, and with what knowledge of the other. Behavioural patterns are the largest group, with eleven entries, and they include some of the patterns you will find most immediately useful in everyday coding.

Pattern	One-line intent
Chain of Responsibility	Pass a request along a chain of handlers until one of them handles it.
Command	Encapsulate a request as an object, so it can be parameterized, queued, logged, or undone.
Interpreter	Given a language, define a representation for its grammar and an interpreter for sentences in that language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its internals.
Mediator	Define an object that encapsulates how a set of objects interacts, so they need not refer to one another directly.
Memento	Capture and externalize an object's internal state so it can be restored later.
Observer	Define a one-to-many dependency so that when one object changes, all its dependents are notified.
State	Allow an object to alter its behaviour when its internal state changes.
Strategy	Define a family of algorithms, encapsulate each, and make them interchangeable.
Template Method	Define the skeleton of an algorithm in a base class, letting subclasses override individual steps.

Pattern	One-line intent
Visitor	Represent an operation to be performed on the elements of an object structure, without changing those classes.

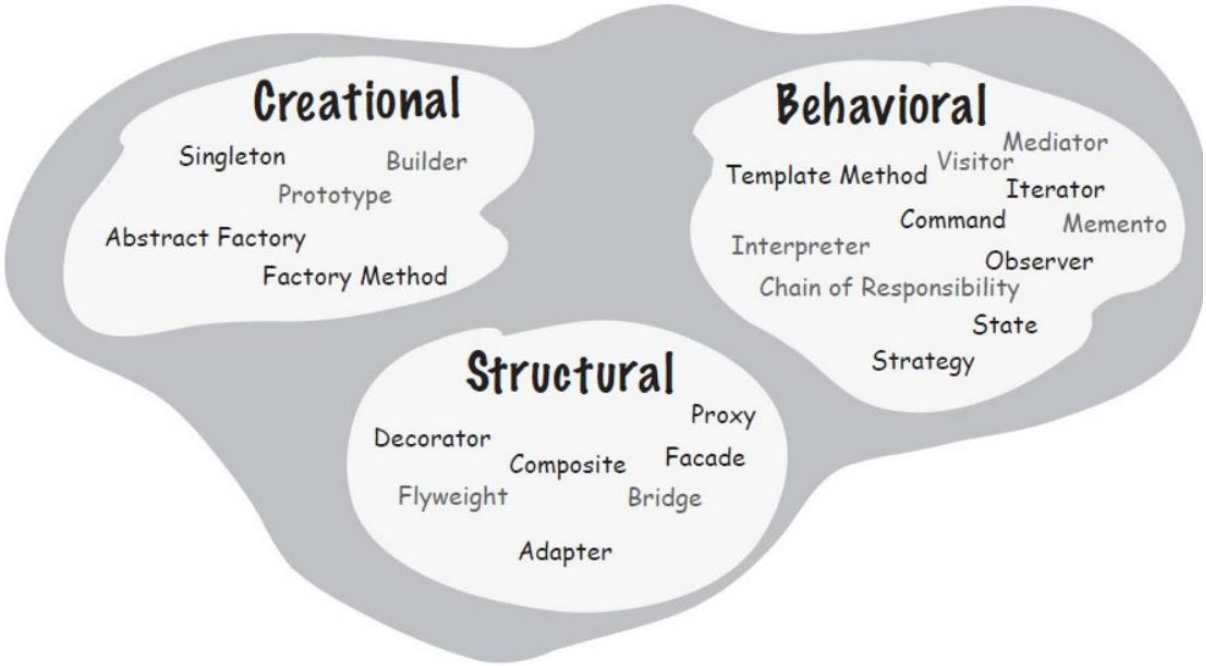

Figure 3: The twenty-three GoF patterns grouped by purpose.

You are not expected to memorize this catalogue today. The names will appear and reappear throughout the rest of the course, each in its own chapter; by the end of CS 247, most of these names will be familiar. For now, the names are an index. When you are stuck on a design problem, scan the names for one whose intent matches your needs.

Class Patterns vs. Object Patterns

The second axis of classification splits the catalogue along a different dimension: does the pattern arrange its participants by inheritance, or by composition?

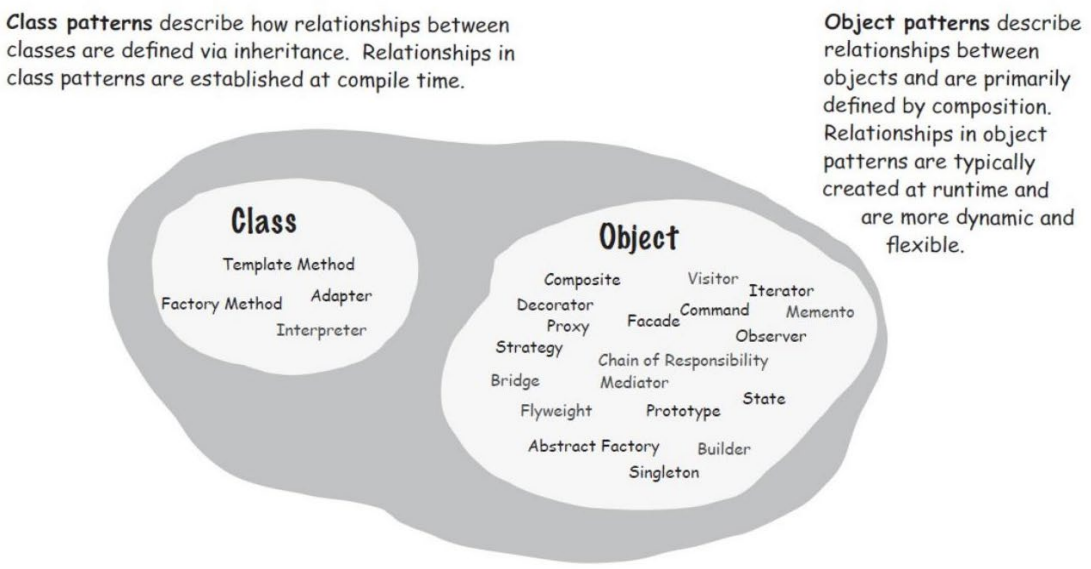

Figure 4: The same twenty-three patterns regrouped by scope. Class patterns wire their participants by inheritance (compile-time); object patterns wire them by composition (runtime). Most patterns are object patterns, composition is the more flexible mechanism.

Class patterns

Class patterns describe how classes are related *by inheritance*. Their relationships are wired up at compile time; once you have inherited from a class, that relationship is fixed for the rest of the program's life. The cost of inheritance (rigid, leaky, hard to combine) and the benefit (clear, statically checked, often more efficient) both apply.

Only four of the GoF patterns are class patterns: Template Method, Factory Method, Adapter (the class form), and Interpreter. Notice how few there are; this is because the design community came to prefer composition after seeing what the alternatives offered.

Object patterns

Object patterns describe how objects are related *by composition*, one object holds a reference to another, often through a smart pointer, and dispatches through that reference at runtime. The relationships can be changed during the lifetime of the program; new participants can be plugged in without recompiling, sometimes without restarting; and combinations of behaviours are achieved by nesting objects rather than by subclassing combinations.

The remaining nineteen patterns are object patterns. The exact list is in Figure 4, but the takeaway is that the design pattern community internalized "favour composition over inheritance" so completely

that the catalogue itself reflects it: most of the patterns are about how to compose objects flexibly, not how to extend classes.

Why the distinction matters

The two axes, purpose (creational/structural/behavioural) and scope (class/object), together give you a small grid of twelve possible cells, and looking at where a pattern sits in the grid tells you a lot about how it works before you have read a word of its description. A behavioural object pattern, for example, is overwhelmingly likely to involve one object holding a polymorphic pointer to another and dispatching some method through it. A creational class pattern (Factory Method) is overwhelmingly likely to involve a base class with a virtual function that returns a newly-constructed object whose type is decided by the subclass.

As you read each pattern in the chapters that follow, ask yourself which row and which column it lives in. The answer should become obvious within a few minutes of looking at the diagram, and that obviousness is a sign that you are starting to internalize the design pattern way of thinking.

> 💡 **Tip**
>
> **How to read a pattern grid.** Pick the row first (creational, structural, behavioural), it tells you what the pattern is trying to do at a high level. Then pick the column (class, object), it tells you what mechanism the pattern uses to do it. The combination narrows the pattern's shape enough that the actual diagram becomes a confirmation rather than a surprise.

Thinking in Patterns

This chapter has spent ten sections promoting design patterns. The honest follow-up is also important: how do experienced designers use patterns in practice, and how do they avoid the trap of patterns-everywhere?

Keep it simple

The single most important rule about patterns: **keep it simple**. The acronym KISS, "keep it simple, stupid", is a near-universal heuristic, and it applies here with full force. Design patterns are not a magic bullet. They are not even a bullet. If something simpler than a pattern will solve your problem, use it.

Most production code does not need patterns. A class that does one thing well, exposes a sensible interface, and is used in three or four places does not need a Strategy, a Visitor, or an Observer. It needs

to be a class that does one thing well. Adding a pattern to a problem that does not call for it is not engineering; it is showing off, and it inflicts a maintenance cost on everyone who reads the code afterwards.

> ⚠ **Warning**
>
> **Pattern overuse is a recognized antipattern.** Code reviews from the early 2000s are full of unhappy stories about new graduates discovering Design Patterns and immediately retrofitting every project with Factory Factories. The result is code that is simultaneously more abstract and less flexible than the original, and harder to read, debug, and change. The patterns are not at fault; the inability to ask "Is the simpler solution sufficient?" is.

How to tell when a pattern is justified

Three signs collectively suggest a pattern may be the right tool:

- **The simpler solution has been tried and is causing measurable pain.** Not predicted pain, actual pain. The current code has a bug that recurs whenever a new subclass is added. Every new requirement triggers an edit in five places. The same condition appears in eight files. These are concrete signs that the design has the wrong shape.
- **The pain matches the shape of a recognized pattern.** Once you know the catalogue, problems start to look like patterns. "This algorithm should vary at runtime" is the Strategy shape. "This object holds a list of dependents that need to learn about changes" is the Observer shape. Recognition is not magic; it is the result of having a vocabulary large enough to spot familiar contours.
- **The full pattern, not just half of it, applies to your situation.** A common error is to apply the structural part of a pattern without its constraints. If your problem genuinely is one-to-many notification, Observer is correct; if it is many-to-many with prioritized ordering, Observer is not, and forcing it produces a worse design than starting from scratch.

Let the patterns emerge

The most experienced designers we know describe their pattern-recognition process the same way. They start a design without thinking about patterns. They write the code in whatever form seems most direct. They notice that the same structure keeps recurring: three places using the same conditional, two classes with duplicated state, and every new feature touching the same file. Only then do they ask: Is there a pattern that captures this structure? Often there is. Sometimes there isn't, and that is fine. Either way, the pattern is the answer to a question the code is already asking, not a decision imposed from outside.

This is what "let patterns emerge" means. It does not mean refusing to use patterns; it means using them only when the design calls for them, not before. A good way to develop this instinct is to write a feature simply, deliberately resist the urge to apply any pattern, then ask: where did this go wrong? The places where the simple design broke down are exactly where a pattern would have helped, and now you know which one because you have seen the specific failure it would have prevented.

Patterns are not set in stone

The diagrams you will see throughout the rest of the course are reference shapes, not canonical truth. Real applications often need to bend a pattern to fit local constraints. Maybe the Strategy's interface needs one more method. Maybe the Observer's notify call should pass a typed event object instead of arguments. Maybe you need two Singletons that talk to each other (a notorious edge case the GoF book warns about). All of these are fine; the test is not "does my diagram match the book?" but "is the pattern's structural idea still doing useful work in my code?"

Once you understand a pattern well enough, you should feel free to adapt it. Until you understand it well, copy the book's version faithfully. The version in the book has been chewed over for decades, and the deviations it makes from a more obvious design are almost always there for good reason.

The role of OO basics

A point worth ending the section on: knowing the syntax of object-oriented programming does not make you a good object-oriented designer. You have spent most of CS 247 learning the syntax: constructors, copy semantics, virtual functions, templates, exceptions, smart pointers, and the STL. All of these are necessary, but none is sufficient. Good OO design is about reusability, extensibility, and maintainability, qualities that emerge from how the pieces are arranged, not from which language features are used.

Patterns are among the best teachers of those qualities. The patterns themselves are not invented; they are discovered. Real systems built by serious teams in many languages over decades kept arriving at the same arrangements when faced with the same problems. The GoF book was largely a documentation effort: writing down what was already happening so other teams could benefit. When you study a pattern, you are receiving the distilled experience of those teams. That is a much better deal than rediscovering it yourself.

> 💡 **Tip**
>
> **Pattern checklist before you commit.** Before you apply a pattern, ask yourself four questions: (1) Have I tried the simple solution? (2) Did the simple solution actually fail in a concrete way I can describe? (3) Does the pattern's stated context match my context? (4) Am I willing to live with the pattern's listed consequences, including the ones I would rather not?

> If all four are yes, apply the pattern. If any are no, the pattern is probably premature. The check takes thirty seconds and saves weeks of regret.

Summary

This chapter introduced design patterns, what they are, why they matter, and the small set of underlying principles that almost all of them apply. The patterns themselves are the subject of the chapters that follow.

The key points to take with you:

- **Change is the only constant.** Almost every design principle and pattern exists because requirements change, and good designs absorb change cheaply.
- **Encapsulate what varies.** Find the part of your design that changes and isolate it behind an interface. The Duck simulation went from override-everywhere to one-pluggable-behaviour by applying this principle.
- **Program to an interface, not an implementation.** Hold references to the most abstract type that supports the operations you need. The client of an abstraction should not name a concrete subclass.
- **Favour composition over inheritance.** HAS-A is a lighter, more flexible commitment than IS-A. Use inheritance when the relationship is genuinely IS-A and substitution is the goal; use composition when behaviour might vary or be combined.
- **Patterns provide a shared vocabulary.** Naming the pattern compresses a sentence's worth of design intent into a single word, and the team that shares the vocabulary moves faster and miscommunicates less.
- **Patterns are not libraries or frameworks.** A library is code you call; a framework is code that calls your code; a pattern is a documented arrangement of classes that you re-create in your application. You apply patterns; you do not import them.
- **A pattern is a solution to a problem in a context.** All three words matter. The context tells you when the pattern applies; the problem tells you what it solves; the solution is the general arrangement of classes and objects.
- **The catalogue is organised two ways.** By purpose (creational, structural, behavioural) and by scope (class or object). The intersection tells you most of what a pattern does before you read its description.
- **Keep it simple.** Patterns are powerful when the design has asked for them. Reaching for a pattern before the design is overengineering. Let them emerge.
- **Patterns capture experience, not code.** They are the distilled wisdom of many teams that solved the same problem; using them is how you start where they finished.

In the next chapters, we will study individual patterns in depth, Strategy, Observer, Decorator, Factory, Singleton, Adapter, Template Method, and others, each in roughly the GoF format and each connected back to the principles we have just introduced. As you read them, watch for the principles in action; the patterns will become much easier to remember once you can see them as concrete applications of three or four short ideas.