# CS 247

Software Engineering Principles

---

Chapter 14

# Template Method Pattern

*Victoria Sakhnini*

# Table of Contents

# The Shape of a Repeating Algorithm

Open up any non-trivial codebase and look for procedures that all begin the same way, do something different in the middle, and end the same way. You will find dozens. A web framework receives an HTTP request, parses headers, dispatches to a handler, formats a response, and writes it to the socket, and the only piece that ever changes between routes is the handler in the middle. A unit-testing framework sets up a fixture, runs a test method, and tears the fixture down, and the only piece that changes between tests is the test method itself. A coffee shop's drink-preparation procedure boils water, brews the leaves or grounds, pours into a cup, and adds condiments, and only the brewing step and the condiments differ between coffee and tea. A turn-based game initializes the board, runs a play loop, and finalizes the score, and only the rules differ between Football and Cricket.

In every one of these examples, the same algorithmic skeleton, the same ordered list of steps with the same control flow between them, is being replayed dozens, hundreds, or millions of times with only a few of the steps swapped out. The skeleton is stable; the meat varies. The interesting design question is: where does the skeleton live? In each individual variant, or in one shared place?

## The naive answer: copy-paste the skeleton

The simplest thing that could possibly work is to write each variant from scratch, repeating the shared structure every time. A Football class implements its own play() method as initialize() / startPlay() / endPlay(); a Cricket class implements its own play() method with the same three calls in the same order; if you add a third sport, you copy the same three calls again. Each class is self-contained, there is no inheritance, no abstract base, no indirection, and a junior developer can read any one of them top-to-bottom without ever looking at another file.

The cost arrives the day someone wants to add a step. Imagine the rule changes: every game must now log its outcome to a central audit service after the play ends. With three copies of the skeleton, you make the change three times, in three files. If you forget one, that sport silently fails to log. If you get the change subtly wrong in one copy, perhaps a typo, perhaps a different order, the bug is undetectable in unit tests because each file passes its own tests in isolation. The longer this pattern lives in the codebase, the worse it gets: variants drift apart, each picking up its own local twist, and the "shared structure" exists only as a fading memory in the team's collective head.

## The right answer: write the skeleton once

The Template Method Pattern is the disciplined fix. Instead of repeating the shared structure across every variant, you write it *once*, in an abstract base class, as a single method that lays out the order of operations. The variant-specific steps are declared as separate methods on the same base class, initially abstract, with no implementation, and each concrete subclass fills in only those step methods. The

skeleton lives in exactly one place; the variation lives in exactly one place per variant; and the two are bound together by the language's ordinary virtual-call machinery.

The pay-off is immediate. To add the audit-logging step, you change the skeleton method in the base class. Every subclass, every sport, every test, every drink, every request handler, now logs automatically, with no further code changes. To add a fourth sport, you write a new subclass containing only the steps that distinguish it from the others; the play loop is inherited. To remove the audit step, you delete a line in one file. The shape of the algorithm and the contents of its steps are now under independent change control, which is exactly what we have been chasing in every chapter of this book.

> 💡 **Tip:**
>
> **The pattern in one sentence.** The Template Method Pattern says: when many classes share an algorithm whose overall shape is the same but whose individual steps differ, put the shape in a base class and let subclasses override the steps.

## A note on the name

The word *template* in "Template Method" has nothing to do with C++'s `template<typename T>` facility. The pattern was named in the 1990s, and at that time, the word "template" carried its everyday English meaning: a fixed outline that you fill in. A blank tax form is a template; the rows and columns are stamped on the page once, and individual filers write their own numbers in. The Template Method is the same idea applied to code: the abstract class defines a fixed sequence of method calls, and individual subclasses implement the blanks. The collision with the generic-programming sense of "template" is purely a linguistic accident; the two ideas are unrelated.

# The Template Method Pattern

With the motivation in hand, here is the formal definition, in the wording of the original Design Patterns text:

> *Template Method: Define the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*

Three phrases in that definition deserve unpacking because, together, they specify everything the pattern does and does not do.

## "The skeleton of an algorithm in a method"

The skeleton is a single, named, callable thing. Not a comment in the source. Not a paragraph in the design document. Not a set of conventions that maintainers are supposed to remember. A real method, with a real signature, that takes the same arguments every variant takes and produces the same kind of result. The skeleton method's body lists the steps of the algorithm in their canonical order, sometimes with conditionals, sometimes with loops, but always with the overall control flow visible and reviewable in one place.

Because the skeleton is itself a method of the abstract class, every subclass inherits it. The subclass does not re-declare it, does not override it, and (in the textbook version of the pattern) cannot override it even if it wanted to. The subclass inherits the skeleton in the same sense that it inherits any other concrete method from a base class: the code is reused as-is, by reference, with no duplication.

## "Deferring some steps to subclasses"

Not every step is deferred. The whole point of the pattern is that *some* steps are universal, the same in every variant, and are implemented once, in the base class, as ordinary concrete methods. Other steps are variable; they differ across subclasses and are declared as pure virtual functions on the base class to be implemented by each subclass. The base class is therefore a mixture: a concrete skeleton, some concrete steps, and some abstract steps.

The art of applying the Template Method is deciding which steps go in which bucket. Steps that are the same across every variant you can think of, and that you cannot imagine ever varying, become concrete methods in the base class; the subclass inherits them without thinking. Steps that vary, or that you can see varying in plausible future subclasses, become pure virtual; the subclass is forced to make a deliberate choice. Steps in between, where there is a reasonable default but a subclass might want something different, are the *hooks* we will meet in Section 5.

### "Without changing the algorithm's structure"

This is the constraint that makes the pattern worth using. The subclass cannot reorder the steps. The subclass cannot skip a step. The subclass cannot add a step in the middle. The subclass can change *what* each variable step does, but never *when* the steps happen relative to each other. Anyone reading the base class's skeleton method is reading the definitive specification of the algorithm's shape, for every subclass that exists today and for every subclass that will ever exist.

This is also the source of the pattern's friction. Real systems sometimes want a subclass to insert one extra step between the third and fourth steps of the algorithm. The textbook Template Method does not permit that. The escape hatches, adding a hook between every pair of steps, or breaking the algorithm into smaller template methods, exist, but they have costs in clarity, and they erode the simplicity that made the pattern attractive in the first place. We will return to this tension in Section 10.

> ❓ **Question:**
>
> Suppose you have a `ReportGenerator` with a template method `generate()` that calls `fetchData()`, `formatBody()`, and `sign()` in that order. A new requirement says PDF reports must also include a watermark, which must be drawn *after* the body but *before* the signature. A junior developer suggests overriding `sign()` in the PDF subclass so it draws the watermark and then signs. Is that a good idea?
>
> **Answer.** No. Smashing two responsibilities into a single method (sign now means "watermark + sign") breaks the contract the `sign` advertises and confuses anyone reading the PDF subclass. The Template Method discipline would say: extend the base class's skeleton to include a `watermark()` step between the body and the signature, provide a default empty implementation in the base class (a hook), and override it only in the PDF subclass. The algorithm's shape changes once, in the base class, where everyone can see it; the PDF subclass changes only the watermark step.

## Anatomy: AbstractClass and ConcreteClass

The Template Method Pattern's class structure is the simplest among the patterns in the Gang of Four catalogue. There are exactly two participants: one abstract class and one or more concrete subclasses, connected by a single inheritance arrow.

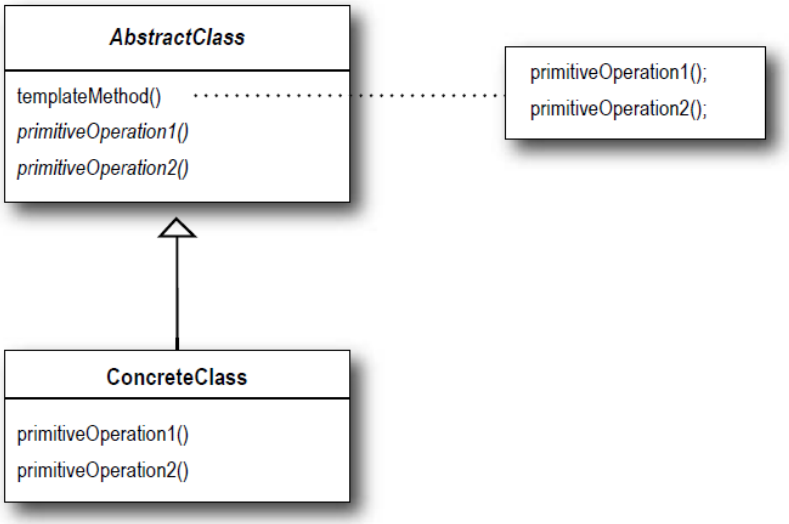


*Figure 3.1: The Template Method Pattern.*

## Reading the diagram

Three things in Figure 3.1 are worth noticing, because they encode the pattern's whole content. First, the `templateMethod()` is in normal type on `AbstractClass`, not italicized. That is UML notation for "this method is *not* abstract; it has a concrete implementation in this class." The base class contains code for the template method, which defines the algorithm's structure.

Second, the `primitiveOperation1()` and `primitiveOperation2()` methods on `AbstractClass` are in italics. That is UML notation for "this method is abstract; the class declares it but does not implement it." In C++ terms, these are pure virtual functions, declarations ending in `= 0`. The base class commits to the algorithm's shape but is silent on what the variable steps actually do.

Third, the dashed line from `templateMethod()` to the box on the right is a UML note attachment. The note's contents, `primitiveOperation1() and primitiveOperation2()`, are a sketch of what the `templateMethod` body actually does. The note is not strictly part of the structural diagram; it is a hint to the reader about behaviour. The real implementation, of course, is the source code of the abstract class, not the picture.

The inheritance arrow from `ConcreteClass` to `AbstractClass`, the open triangle, says exactly what you would expect: `ConcreteClass` inherits everything `AbstractClass` has. The `templateMethod` comes along for free, with its body intact. The `primitiveOperation1` and `primitiveOperation2` appear as abstract declarations that ConcreteClass is now obliged to implement; the diagram shows it doing so in normal type in the lower box.

### The pieces in the C++ vocabulary

Translating the UML into C++ terms gives a checklist for implementing the pattern correctly:

- **AbstractClass becomes an abstract C++ class.** Concretely: it has at least one pure virtual function. You cannot construct an instance of it directly; clients must construct a concrete subclass.
- **templateMethod becomes a non-virtual member function.** Or, equivalently, a virtual member function declared `final`. Either form prevents subclasses from overriding it. The intent is to lock the algorithm's structure so it cannot drift on a class-by-class basis.
- **primitiveOperation1, primitiveOperation2 become pure virtual functions.** Declared with `= 0`. Each is typically protected, not public; they are implementation details of the algorithm, not part of the abstract class's outward-facing interface. Clients call `templateMethod`; the template method calls the primitive operations.
- **AbstractClass has a virtual destructor.** Any class designed to be inherited from must have a virtual destructor, or deleting a derived object through a base-class pointer is undefined behaviour. This is a C++ requirement, not a Template Method one, but it must not be forgotten.
- **ConcreteClass marks its overrides with** `override` and itself with `final`. Marking each override with the `override` keyword turns a typo (overriding the wrong signature) into a compile error. Marking the class itself `final` tells readers and the compiler that nothing further derives from it; the inheritance hierarchy is exactly two levels deep.

> 💡 **Tip:**
>
> **Two-level hierarchies are a feature.** Template Method is clearest when there is exactly one abstract base class and one layer of concrete subclasses. Three-level hierarchies, where a concrete subclass becomes the base for further subclasses, are almost always a sign that you should refactor: either the middle layer should be folded into the base, or the variation it captured should be expressed by composition (a Strategy object) rather than further inheritance.

## A Worked Example: The Game Framework

Imagine you are building educational software that lets students play simple versions of various sports. Each sport has a different set of rules, but the overall lifecycle of a single play session is the same in all of them: set up the field and players, run the play, and tear down afterwards. We will encode that lifecycle once, in an abstract Game class, and let Football and Cricket fill in only what differs.

## The abstract base class

Here is the Game class. Read the body of play() first, three method calls, in a fixed order, and then look at the rest of the class to see how that body is supported:

```
// Abstract base class defining the template method
class Game {
public:
    // Template method - non-virtual to prevent override
    void play() {
        initialize();
        startPlay();
        endPlay();
    }
    // Virtual destructor with default implementation
    virtual ~Game() = default;

    // Deleted copy operations (games shouldn't be copied)
    Game(const Game&) = delete;
    Game& operator=(const Game&) = delete;
    // Default move operations
    Game(Game&&) = default;
    Game& operator=(Game&&) = default;
protected:
    // Protected constructor - only derived classes can instantiate
    Game() = default;

    // Pure virtual methods to be implemented by subclasses
    virtual void initialize() = 0;
    virtual void startPlay()  = 0;
    virtual void endPlay()    = 0;
};
```

Several deliberate choices appear in this class, and each of them is worth understanding because each one fixes a real problem that would otherwise bite later.

**Why play() is non-virtual**

The `play()` method is the template method. It is declared *non-virtual*, which in C++ means a subclass cannot override it. (We could equivalently have written `virtual void play() final`; the effect is the same.) The intent is to say: the order of the steps in play is set; subclasses customize the steps, not the order. A Football class that wanted to call endPlay before `startPlay`, perhaps by mistake, cannot do so by overriding `play`; the only way it could rearrange the order is to override `play` itself, which the compiler now forbids.

**Why the destructor is virtual**

The `~Game()` destructor is *virtual*. This is unrelated to the Template Method Pattern proper; it is a baseline C++ rule about polymorphic deletion. Any class that has at least one virtual function and is intended to be derived from must have a virtual destructor, because clients will routinely hold derived objects through `Game*` or `std::unique_ptr<Game>`, and the destruction of those handles must dispatch correctly to the derived destructor. Forgetting `virtual` here is the textbook way to leak resources from a derived class.

**Why is `copy` deleted, and is `move` defaulted**

The copy constructor and copy assignment operator are `= deleted`. The intent is that a Game instance represents a single ongoing play session; there is exactly one Football game in flight, with its own state, and copying it makes no sense. Move operations are `= defaulted` because moving (transferring ownership of the unique session) is harmless and occasionally useful (for example, returning a game from a factory function by value). This combination, delete copy, default move, is sometimes called the "move-only" idiom, and it is the right default for resource-holding polymorphic classes.

**Why is the constructor `protected`**

The default constructor is declared `protected` rather than public. The class is abstract, it has pure virtual functions, so the language already forbids constructing a `Game` directly. But putting the constructor in the protected section makes the intent explicit to readers and slightly improves the error messages when someone tries to write `Game g;` anyway. Subclasses can call it (because they are inheriting); outside code cannot.

**Why are the primitives `protected`, not `public`**

The three pure virtual methods, `initialize`, `startPlay`, `endPlay`, are declared in the `protected` section. The reason is the same one we will see repeated throughout the chapter: they are implementation details of the play algorithm, not the abstract class's contract with the outside world. Clients call `play()`, which is the public interface, and play() calls the primitive operations along the way. Making the primitives protected means a misuser cannot bypass the template method by calling, say, `game->startPlay()` directly and getting the steps out of order.

## The concrete subclasses

Now the variants. Football and Cricket each fill in only the three primitive operations. Notice what they do not do: they do not override play(), they do not redeclare the destructor, they do not duplicate the play loop. The shape of the algorithm is wholly inherited; only the contents of the steps are local.

```
// Concrete class implementing the primitive operations
class Football final: public Game {
protected:

    void initialize() override {
        std::cout << "Football Game Initialized! Start playing.\n";
    }
    void startPlay() override {
        std::cout << "Football Game Started. Enjoy the game!\n";
    }
    void endPlay() override {
        std::cout << "Football Game Finished!\n";
    }
};

// Another concrete class implementing the primitive operations
class Cricket final: public Game {
protected:
    void initialize() override {
        std::cout << "Cricket Game Initialized! Start playing.\n";
    }
    void startPlay() override {
        std::cout << "Cricket Game Started. Enjoy the game!\n";
    }
    void endPlay() override {
        std::cout << "Cricket Game Finished!\n";
    }
};
```

The `final` keyword on the class itself says "nothing derives from `Football`." Together with the `override` keyword on each method, this means a typo, say, `void initialize() override` with British spelling, or `void initialize() const override` with a stray `const`, fails to compile rather than silently leaving the base class's pure virtual function unimplemented. The compiler is your friend here; let it catch these errors at build time, not at midnight when a user reports that Cricket games never finish.

Crucially, neither class mentions the order of `initialize`, `startPlay`, and `endPlay`. A reader who wants to know what happens when a Football game runs has to look in `Game::play`, because that is where the order lives. This is the inversion-of-control pattern the code produces: the variant code knows about the steps, the framework code knows about the structure, and the two pieces of knowledge live in separate files that change for different reasons.

## Putting it together

Finally, the client code. A user of the framework constructs a Game and calls play() on it, no awareness of which subclass it is, no awareness of the three internal steps:

```
// Factory function for creating games (optional but idiomatic)
template<typename GameType>
auto createGame() {
    return std::make_unique<GameType>();
}

int main() {
    // Playing a football game using smart pointers
    {
        auto game = std::make_unique<Football>();
        game->play();
    } // Automatic cleanup
    std::cout << '\n';
    // Playing a cricket game
    {
        auto game = std::make_unique<Cricket>();
        game->play();
    } // Automatic cleanup
    // Alternative: using the factory function
    std::cout << "\nUsing factory function:\n";
    auto footballGame = createGame<Football>();
    footballGame->play();

    return 0;
}
```

Running the program produces the expected output, with the three lines of each game's lifecycle appearing in their canonical order:

```
Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!

Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Using factory function:
Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

Three observations about this client code. First, `main` never says the word `initialize`, `startPlay`, or `endPlay`. The internal steps of a play session are private to the algorithm; the client only sees the high-

level operation `play`. Second, the program uses `std::unique_ptr<Game>`, not `Game*` with a raw `new` and `delete`; the curly-brace scope blocks are there precisely so the destructor fires at the end of the scope and tears down the Game. Third, the factory function `createGame<GameType>` is a convenience: it returns a `unique_ptr<GameType>`, which implicitly converts to `unique_ptr<Game>` wherever a `Game` is wanted. None of this is essential to the Template Method Pattern; you could write the example with raw pointers, but it is what well-written modern C++ would actually look like.

> **Tricky:**
> Why is `play()` *public* in the abstract class, while `initialize`, `startPlay`, and `endPlay` are *protected*? Because they have different audiences. `play` is the contract the abstract class offers to outside code: "call this and a game will be played." The three primitives are the contract that the abstract class offers to its subclasses: "override these to specialize the algorithm." Outside code has no business touching the primitives directly; doing so would allow it to call endPlay before initialize, repeat `startPlay` twice, or simply skip the lifecycle and produce nonsense state. Making the primitives protected is how the language enforces that boundary.

## Three Kinds of Operations: Abstract, Concrete, and Hook

The Game example used only pure virtual primitives; every step of the algorithm was deferred to the subclass. Real applications of the Template Method Pattern almost always have a mixture. To use the pattern fluently, you need a vocabulary for the three kinds of operations an abstract class can offer, and a sense for when each kind is appropriate.

| Kind | Declaration in base | Subclass obligation | Use when... |
|---|---|---|---|
| **Abstract operation** | Pure virtual (`= 0`) | Must override | The step varies across subclasses; no sensible default exists. |
| **Concrete operation** | Non-virtual member function | Cannot override | The step is the same across all subclasses; changing it would be a bug. |
| **Hook** | Virtual with a default body (often empty) | May override | The step has a reasonable default, but some subclasses might customize it. |

## Abstract operations: the variable steps

Abstract operations are the easy ones to identify. They are the steps that have no meaningful default, the steps that, by their nature, require knowing which variant of the algorithm you are running. In the game framework, "initialize the field" depends on whether the field is a football pitch or a cricket pitch, and there is no neutral version of the operation that makes sense in the abstract. In a beverage-preparation framework, "brew the leaves or grounds" depends on whether it is coffee or tea; you can't write a default "brew" that works for both.

In C++, an abstract operation is a pure virtual function. Declaring it pure virtual has two effects: it commits the abstract class to having the step in its algorithm (the template method can call it), and it forces every concrete subclass to provide an implementation. If a developer adds a new subclass, say `Hockey`, and forgets to implement one of the pure virtual operations, the class is itself abstract and cannot be instantiated, and the compiler will say so at the line where someone tries to construct a `Hockey`. The mistake is caught at build time, not at runtime.

## Concrete operations: the invariant steps

Concrete operations are steps that are the same across all subclasses. The classic example is logging the start and end of the operation: every variant of `processRequest` wants the same opening and closing log lines; the only thing that varies is the work in between. A concrete operation is implemented as a non-virtual member function in the abstract base class, with a body that performs the work directly. Subclasses inherit it without overriding it.

It is tempting to leave concrete operations virtual "just in case", what if some subclass eventually wants to do its own logging? The Template Method discipline says: don't. A virtual concrete operation is an invitation to subclasses to drift, and once one of them does, the invariant becomes a fiction. If the day really comes when a subclass needs different behaviour, you can promote the operation to a hook (Section 5.3) at that point, the change is local and small. Until then, the operation is concrete, non-virtual, and its body is the definitive specification of what the step does.

Concrete operations are how the abstract class achieves real code reuse. Without them, the Template Method Pattern would only specify the *order* of steps; with them, the pattern can specify both the order and the implementation of any step that does not vary. Most well-designed template-method frameworks have more concrete operations than abstract ones: a long, stable spine of inherited behaviour, with a few abstract sockets where subclasses can plug in their own logic.

## Hooks: the optionally variable steps

Hooks sit between the two extremes. A hook is a virtual member function on the abstract base class with a default implementation, often an empty body, sometimes a sensible default, that subclasses *may* but *need not* override. The template method calls the hook at a specific point in the algorithm;

subclasses that want to do something at that point override the hook, and subclasses that do not want to do anything inherit the empty default.

Hooks are how you add optional steps to a template method without forcing every subclass to opt in. Consider the report-generator example from Section 2.3: we wanted to add a watermark step between body and signature, but only for PDF reports. The solution is a hook:

```
class ReportGenerator {
public:
    void generate() {
        fetchData();
        formatBody();
        watermark();     // hook, default does nothing
        sign();
    }
    virtual ~ReportGenerator() = default;
protected:
    virtual void fetchData()  = 0;   // abstract
    virtual void formatBody() = 0;   // abstract
    virtual void sign()       = 0;   // abstract

    // Hook, has a default empty implementation;
    // subclasses override it only if they need a watermark.
    virtual void watermark() { /* nothing by default */ }
};

class PdfReport final : public ReportGenerator {
protected:
    void fetchData()  override { /* ... */ }
    void formatBody() override { /* ... */ }
    void sign()       override { /* ... */ }
    void watermark()  override { drawDiagonalGreyText("DRAFT"); }
};

class HtmlReport final : public ReportGenerator {
protected:
    void fetchData()  override { /* ... */ }
    void formatBody() override { /* ... */ }
    void sign()       override { /* ... */ }
    // No override for watermark(), the empty default applies.
};
```

From the outside, `PdfReport` and `HtmlReport` look interchangeable; both are `ReportGenerators`, both respond to `generate()`, and the client cannot tell which is which. Internally, the PDF variant adds a watermark step, and the HTML variant does not. The structure of the algorithm, fetch, format, optionally watermark, sign, is in one place, in the base class, where everyone can see it.

**Boolean-returning hooks**

A particularly useful kind of hook returns a `bool` that the template method uses to decide whether to run a step. The base class supplies a default (*true* or *false*), and subclasses override the hook to flip the answer when needed. The CaffeineBeverage example from the original *Head First Design Patterns* book uses this for condiments: the template method asks `if (customerWantsCondiments()) addCondiments()`, and the default implementation of `customerWantsCondiments()` always returns *true*; subclasses (or a customized instance) override it to return *false* when, say, the user has specified plain black coffee. The pattern lets the template method express "sometimes we do this, sometimes we don't" without the subclass needing to know which step is skipped or the order in which the remaining steps run.

## How to decide which kind to use

When you are designing a new template method, the choice between abstract, concrete, and hook for each step comes down to two questions. The first is whether the step has a sensible default: if it does not, the step is abstract; if it does, you continue to the second question. The second is whether subclasses are likely to need to customize it: if not, the step is concrete; if so, it is a hook.

Beginners overuse abstract operations. Every step is declared pure virtual; the subclasses end up reimplementing triviality, and the abstract class fails to provide much value. The correction is to ask of each pure virtual: "Is there a default that works for most subclasses?" If yes, promote it to a hook and let subclasses opt out.

Beginners also overuse hooks. Every step gets declared virtual with a default implementation, even steps that have no real reason to vary. The correction is the mirror image: "Do I have a concrete example of a subclass that would want to override this?" If no, if you are adding the hook "just in case", the step should be a concrete operation, not a hook. You can always promote concrete to hook later; it's a small change. Going the other way, discovering that a hook subclass depends on cannot actually be customized because some invariant must hold, is much harder.

**? Question:**

You are designing a template method for a `DataExporter` base class with steps: `openOutput`, `writeHeader`, `writeRows`, `writeFooter`, `closeOutput`. Two subclasses exist so far: `CsvExporter` and `JsonExporter`. The `openOutput` and `closeOutput` steps just deal with the destination file and are the same in both. The other three differ. Which steps are abstract, which are concrete, and which (if any) are hooks?

**Answer.** `openOutput` and `closeOutput` are **concrete**; the file-handling code is identical and can live in the base class. `writeHeader`, `writeRows`, and `writeFooter` are **abstract**; each format writes them differently, and there is no neutral default (`writeRows` in particular has no possible

default body). None of them are hooks, because no step has a sensible default that some-but-not-all subclasses would override. If a third subclass, say, XmlExporter, appeared that wanted an optional "pretty-print" step inserted between the header and rows, then we would add a hook, prettyPrint(), with an empty default. Don't add the hook until that need is concrete.

## Locking Down the Algorithm

The Template Method Pattern's whole value proposition rests on one promise: the algorithm's structure is fixed. Every subclass plays the same sequence of steps in the same order; the only thing that varies is what each step does. If a subclass can override the template method itself, change the order, add steps, or remove steps, that promise is empty. So the pattern requires a mechanism to prevent overriding the template method, and C++ provides two equivalent ways to do so.

### Option 1: non-virtual

The simplest way to make a method un-overridable in C++ is not to declare it virtual in the first place. A non-virtual member function is not part of the dynamic dispatch table; the call is resolved statically, by the type of the pointer or reference, not by the object's actual class. This is what we did in the Game example: play() was declared without the virtual keyword, and so it was an ordinary, non-virtual member function.

```
class Game {
public:
    // Non-virtual: subclasses cannot override.
    void play() {
        initialize();
        startPlay();
        endPlay();
    }
    // ...
};
```

There is a small subtlety to understand here. Non-virtual does not mean "cannot be redefined in a subclass"; a subclass *can* declare its own method called `play`, and the language will accept it. What it cannot do is *override* the base class's `play` through dynamic dispatch. If a client holds a `Game*` pointing to a `Football`, calling `play` on that pointer will always run `Game::play`, even if `Football` has its own `play`. The subclass version is name-hidden, not overriding. In practice, this is enough; clients use base-

class handles and so see the inherited template method, but readers of the subclass might be momentarily confused.

### Option 2: virtual final

The more recent and arguably clearer option is to declare the template method `virtual` and `final`:

```
class Game {
public:
    // Virtual but final: subclasses cannot override.
    virtual void play() final {
        initialize();
        startPlay();
        endPlay();
    }
    // ...
};
```

The `final` keyword, introduced in C++11, turns attempts to override into compile errors. A subclass that writes `void play() override` gets a diagnostic at compile time, not a silent name-hiding. This is the same protection that `final` gives the Java pattern that inspired the original Gang of Four discussion: it makes the intent explicit and turns a mistake into a build failure rather than a runtime surprise.

Why pick this option over a plain non-virtual one? Because it documents the intent at the site of the declaration. A reader who sees `virtual void play() final` knows immediately that `play` is dispatched dynamically, meaning the template method calls the primitive operations through the v-table, and that it cannot be replaced by a subclass. A reader who sees a plain non-virtual `void play()` has to know enough about the pattern to recognize that this is the deliberate choice; otherwise, they might assume the absence of `virtual` was an oversight. In code reviews, `virtual ... final` is the more obviously-intentional form.

> ⚠ **Warning:**
> Whichever option you pick, *pick one consistently* across your project. A code base that uses non-virtual in some template methods and `virtual final` in others sends mixed signals to readers about whether the choice was deliberate.

### What if you genuinely want subclasses to extend the algorithm?

Sometimes the requirement is that a subclass really does need to do work outside the existing step boundaries, perhaps a third sport needs a step between `initialize` and `startPlay` that none of the others want. Three options exist, in order of preference:

- **Add a hook to the base class.** If the variation might apply to other subclasses too, give the algorithm a hook between the relevant steps with an empty default body, and let subclasses override it. The algorithm's shape changes once, in the base class, where the change is visible and reviewable. This is the right answer most of the time.
- **Split the template method.** If the variation is large, multiple new steps, or a different control structure, consider breaking the algorithm into two template methods that the new subclass can sequence differently. This trades the simplicity of "one template method per class" for the flexibility of having two algorithm shapes. Use sparingly.
- **Reconsider whether Template Method is the right pattern.** If your subclasses are routinely fighting against the fixed structure, perhaps the structure isn't actually fixed, perhaps the algorithm is really a sequence of independently-composable steps, and what you want is the Strategy or Pipeline pattern, not Template Method. The cost of switching is real, but staying with the wrong pattern is more expensive in the long run.

## The Hollywood Principle: Inversion of Control

The Template Method Pattern is not just a code-reuse trick; it embodies a design principle that recurs throughout software engineering. The principle has a name with a memorable slogan attached to it:

**Don't call us, we'll call you.**

This is the Hollywood Principle, named (allegedly) after the standard rejection casting directors give to hopeful auditioning actors. In software, it expresses a particular shape of dependency between high-level and low-level components, a shape that the Template Method Pattern produces almost as a side effect.

### The naive direction of calls

In a procedural codebase the natural shape of dependencies is bottom-up. Low-level utility functions exist first: string manipulation, file I/O, and network sockets, with high-level business logic calling into them. The dependency arrows in the architecture diagram all point downwards: the high-level module names depend on the low-level module, but the low-level module knows nothing of the high-level one. A library is the textbook example: your code calls into *malloc* and *printf*; *malloc* and *printf* know nothing about your code.

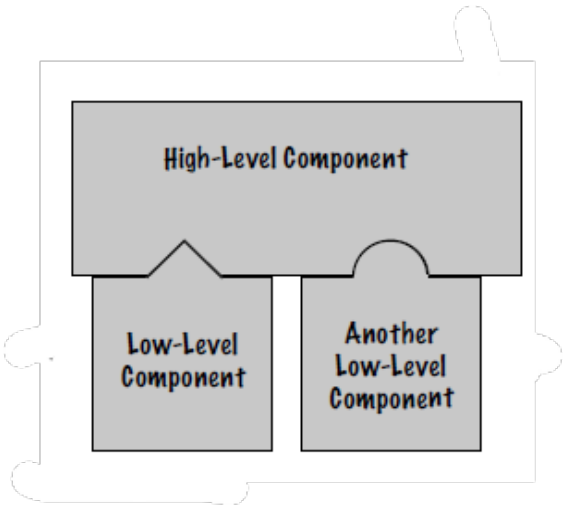


*Figure 7.1: The Hollywood Principle.*

This bottom-up shape works for utility libraries because utilities have no business decisions to make. *malloc* doesn't need to know whether the calling program is a web server or a video game; the contract is the same in both cases. But the moment the low-level component needs to make a decision that depends on which high-level context it is running in, what to do when the user wants something custom, when to fire a callback, or how to handle an error, the bottom-up dependency starts to fail. The low-level component cannot ask the high-level one for guidance without naming it, and naming it inverts the dependency.

## What inversion of control means

The Hollywood Principle says that, in well-designed object-oriented systems, the dependency arrow points the other way. The high-level component owns the algorithm, the order in which things happen, the conditions under which steps run, and the points at which decisions are made. The low-level components contribute to the contents of individual steps; they have no opinions about when or whether they are called. Decision-making lives at the top; participation lives at the bottom. The principle does not forbid the low-level components from doing anything interesting; it forbids them from calling upwards into the algorithm to ask what happens next.

"A low-level component never calls a high-level component directly", is what "don't call us, we'll call you" actually says. The job of the low-level component is to make itself *available* to be called: to provide a method with a known signature that the high-level component can invoke when it needs to. The job of the high-level component is to decide *when* and *whether* to invoke it. The two roles are kept rigorously separate, and the dependency graph is therefore acyclic.

## Template Method as Hollywood in practice

Now look back at the Game example. Which class is high-level, and which is low-level? `Game`, the abstract class with the template method, is at a high level. It owns the algorithm; `play` is the decision-making code. `Football` and `Cricket`, the concrete subclasses with the primitive operations, are low-level. They contribute behaviour pieces (`startPlay`, `endPlay`), but they make no decisions about the order of events.

Notice the direction of the calls in the running program. When a client calls `game->play()`, control enters `Game::play`, the high-level component. `Game::play` then calls `initialize()`, which dispatches through the v-table to `Football::initialize`. Control briefly visits the low-level component, then returns to `Game::play`, which calls `startPlay()`, another brief excursion to the low-level component, and so on. The high-level component is in charge throughout; the low-level component is invoked, does its small piece of work, and returns. At no point does `Football::initialize` call back into `Game::play` or any other high-level method. The dependency direction is preserved.

This is exactly the shape every well-designed framework has. A framework is by definition a high-level component that lives in a library; the application code you write is the low-level participant; the framework calls your code, not the other way around. When you write a class that derives from `HttpServlet` and overrides `doGet`, your `doGet` is a primitive operation, and the framework's request-handling pipeline is the template method. When you write a JUnit test class whose methods are annotated `@Test`, your test methods are primitive operations, and JUnit's `setUp` / `runTest` / `tearDown` loop is the template method. The Hollywood Principle is what makes a framework *feel* different from a library: a library you call, a framework calls you.

## The dependency-graph test

The Hollywood Principle gives you a concrete tool for evaluating designs. Draw the dependency graph, which modules call which? Are the arrows all pointing in one direction (low-level called by high-level), or do they go both ways (low-level calling back into high-level)? Bidirectional arrows are a smell. They mean that two modules are mutually entangled and cannot evolve independently: changing the high-level interface forces changes in the low-level, and changing the low-level forces changes in the high-level. Acyclic graphs, by contrast, let each module change at its own pace within the contract it offers to the other side.

> 💡 **Tip:**
>
> **Cyclic dependencies are not always wrong.** But they are always worth a second look. Often, the two modules represent two parts of a single concept and should be merged, or the dependency should be inverted (by passing a callback or an interface implemented by the high-level module).

> The Template Method Pattern is one of several tools, alongside Observer, Strategy, and dependency injection, for breaking accidental cycles.

## Template Method vs Strategy vs Factory Method

Three of the patterns in the Gang of Four catalogue concern themselves with parameterizing an algorithm: Template Method, Strategy, and Factory Method. They are easy to confuse; all three involve a base class with abstract or virtual operations and concrete subclasses that implement them, and the way to keep them straight is to be clear about which axis of variation each one captures.

### Template Method vs Strategy

Template Method varies *parts of an algorithm* by subclassing; Strategy varies *the whole algorithm* by composition. The difference is structural, and it has consequences.

Recall how Strategy works (from Chapter 13, or the original Duck example): the class that has the varying behaviour holds a pointer or reference to a Strategy object, and delegates the varying step to that object. Different strategies are objects of different classes that implement the same abstract Strategy interface. The class can swap its strategy at runtime, a `Duck` can change its `FlyBehavior` from `FlyWithWings` to `FlyNoWay` simply by assigning a new behaviour object to its member field.

Template Method does the same job through inheritance rather than composition. Different variants of the algorithm are different subclasses, not different member objects. A `Football` is a `Game` with built-in football-specific operations; it cannot turn into a `Cricket` at runtime by reassigning a field, because the variation is baked into the object's type.

So which to pick? The rules of thumb:

- **Use Template Method** when there is a small, fixed set of variants and the algorithm has a stable shape with a few customization points. Inheritance is the right tool when the variation is at compile time, the variants do not need to be swapped, and the steps are tightly bound to a particular kind of object.
- **Use Strategy** when the variations need to be swappable at runtime, when the variations are independent of the object that uses them (so a `FlyBehavior` can be reused across `Duck`, `Bird`, and `Plane`), or when the same variant might apply to many different host classes.
- **Use both** when one part of an algorithm needs Template-Method-style parameterization and another part needs Strategy-style swappability. The two patterns compose: a template method can perfectly well delegate to a Strategy object for one of its steps.

| Aspect | Template Method | Strategy |
|---|---|---|
| Mechanism | Inheritance, subclasses override primitive operations. | Composition, the host object holds a Strategy object. |
| Variation point | Per-step, inside one algorithm. | Whole algorithm, swapped as a unit. |
| Binding time | Compile time (object's type is fixed). | Runtime (Strategy field can be reassigned). |
| Number of variants | Each variant is a class. | Each variant is an object; many can co-exist. |
| Coupling between variant and host | Tight variants subclass the host. | Loose, variants implement an interface unrelated to the host. |
| When to prefer | Fixed steps with fixed order; small, stable set of variants. | The algorithm itself varies; runtime swapping is needed. |

## Template Method and Factory Method

Factory Method (Chapter 12) is, surprisingly, a special case of Template Method. The classic Factory Method setup has an abstract `Creator` class with two methods: a concrete `orderProduct()` method that performs the work, and an abstract createProduct() method that subclasses override to produce the actual product. Look at `orderProduct`: it is a template method. It defines the order in which `createProduct` is called and what is done with the result. The Factory Method pattern is what you get when you apply Template Method to the special case of "the only step that varies between subclasses is the construction of the product object."

This connection is not just trivia. It tells you when each pattern is the right name to use. If the variation in your algorithm is genuinely "different subclasses construct different products, but everything else is the same," call it Factory Method, the name communicates more. If the variation is broader, multiple steps differ, of which one might happen to be construction, call it Template Method, with one of its primitive operations happening to be a factory. The patterns shade into each other, and the right name to use is the one that best signals to readers what is varying.

> **? Question:**
> Suppose a colleague describes their design as follows: "I have a base class `PaymentProcessor` with a method `processPayment()`. The method validates the amount, calls a `charge()` method that each subclass implements differently (CreditCard, PayPal, Crypto), and then logs the result. The

validation and logging are the same for every subclass." Is this Template Method, Strategy, or Factory Method?

**Answer.** Template Method. There is a base class with a fixed algorithm (`processPayment`) whose middle step is deferred to subclasses; validation and logging are concrete operations, and `charge` is an abstract operation. It is *not* a Factory Method because the deferred step does not produce an object; it performs an action. It is *not* a Strategy because the variation lives in subclasses of the host, not in separate objects passed in by composition. If you wanted runtime-swappable payment methods, you could redesign the same logic as Strategy by extracting `charge` into a separate `ChargeStrategy` interface and having `PaymentProcessor` hold one, and that redesign is exactly what you would do if the requirements changed to support multi-provider payments, where one transaction might cascade through several charge methods.

## Real-World Uses

Once you know the pattern, you start to see it everywhere. It is one of the most widely used patterns in production code because it captures the natural shape of any framework: "we handle the boilerplate, you fill in the interesting parts." A small selection of examples, grouped roughly by domain:

### Web and network frameworks

- **HTTP request handling.** Java's `HttpServlet` exposes a `service()` template method that dispatches to `doGet`, `doPost`, `doPut`, `doDelete`, and friends, primitive operations that the application overrides. The framework handles authentication checks, content negotiation, error mapping, and access logging around the application's per-verb code. Almost every web framework in any language has the same structure.
- **Spring's JdbcTemplate.** The `execute(...)` method opens a database connection, sets up a transaction, runs a callback supplied by the caller (this part is the primitive operation, although it is passed in as a lambda rather than an overridden method), then commits or rolls back, and closes the connection. The connection-management boilerplate, which used to be the source of countless leak bugs, lives in the framework; the application code is just the SQL.

### Testing frameworks

- **JUnit's test runner.** For each test method, the runner calls `setUp()` (or `@BeforeEach`-annotated methods), runs the test, calls `tearDown()` (or `@AfterEach`), and reports the outcome. The skeleton is rigid, pre, run, post, and the application code never has to write the pre/post sequence by hand. In older versions of JUnit, `setUp` and `tearDown` were literal overridable methods on a `TestCase` base class, textbook Template Method.

- **Google Test fixtures.** A `TestFixture` class overrides `SetUp()` and `TearDown()`; each `TEST_F` macro produces a class that derives from the fixture and adds the test body as a third primitive operation. The framework drives the lifecycle.

## Game engines and simulators

- **Game loops.** Engines like Unity (in C#) and Unreal (in C++) expose a fixed update-render-cleanup loop and let game code override callbacks: `OnUpdate`, `OnRender`, `OnDestroy`. The engine's main loop is the template method; the user-written game logic is a collection of primitive operations.
- **Discrete-event simulators.** The skeleton is "draw next event from the queue, advance simulation clock, dispatch event, repeat"; the per-event behaviour is the primitive that the application provides.

## Data processing

- **ETL pipelines.** Extract-transform-load frameworks (Apache Beam, dbt, etc.) define the overall "read source / apply transformations / write sink" structure and let users plug in transformations. Different sources and sinks are different concrete subclasses or strategy objects, but the shape of the run is fixed.
- **File parsers.** Format-agnostic parsing frameworks expose an `openInput` / `readHeader` / `readRecords` / `closeInput` pipeline; CSV, JSON, and XML parsers fill in only the format-specific steps.

## GUI frameworks and component lifecycles

- **React component lifecycle.** Although JavaScript does not have classical inheritance in the same way C++ does, the React class-component model is a template method in spirit: the framework calls `componentDidMount`, `render`, `componentDidUpdate`, `componentWillUnmount` in a fixed order, and the application code overrides whichever ones it cares about.
- **Win32 application loop.** Even at the OS level, the message-pump pattern is template-method-shaped: the framework drains messages and dispatches them to handlers provided by the application.

## Crypto, OAuth, and payment workflows

- **OAuth / SSO flows.** The token-acquisition workflow has a fixed shape: redirect to the provider, receive a callback, exchange the code for a token, and fetch the profile. That is the same for every provider, but the per-step API URLs and payloads are provider-specific. Libraries like Spring Security define the workflow as a template and let provider integrations supply the primitives.
- **Payment processing.** Validate → authorize → capture → notify is the canonical e-commerce transaction shape. The payment library or PSP integration defines the order; the application or the provider plug-in supplies the per-step actions.

A useful exercise the next time you read framework documentation is to find the abstract base class, usually called `AbstractSomething`, `BaseSomething`, or just `Something`, and identify which methods are pure virtual, which are concrete, and which are hooks. You will almost always find a clean Template Method structure underneath, even when the documentation does not use the name.

## Caveats: When Template Method Is the Wrong Tool

The Template Method Pattern is genuinely useful, but it has costs and limits. Reaching for it reflexively, every time a class has steps in it, produces designs that are rigid where they should be flexible, deep where they should be flat, and confusing where they should be straightforward. Three failure modes appear often enough to be worth naming.

### The fragile-base-class problem

Inheritance creates a tight coupling between the base class and the subclass. Any change to the base class, even one that does not change its publicly-documented behaviour, risks breaking subclasses in surprising ways. Suppose a Game subclass relies on the fact that `initialize` is called exactly once per `play`, and uses that fact to set up a counter. Later, the base class is "optimized" to call `initialize` twice in some edge case. The subclass's counter is now off by one, and the bug is not visible in the base class's own tests, because those tests don't know about the counter.

This is the *fragile base class* problem: subclasses make assumptions about the base class's internal behaviour that are not part of any explicit contract, and the base class cannot evolve without risking those assumptions. The textbook Template Method Pattern is particularly vulnerable because the whole point is that subclasses participate in the algorithm's internal sequencing; they see, by definition, the order in which the base class calls its own protected methods. The base class cannot easily refactor without breaking what subclasses see.

Mitigations: keep the template method short (fewer steps, fewer dependencies for subclasses to lean on), document the call order as part of the contract (so changes are visible breakages), and prefer composition (Strategy) over inheritance when you anticipate that the algorithm's internal structure might evolve.

### The hook explosion

As subclasses accumulate, each one wants its own customization point in some new corner of the algorithm. "Just add one more hook" seems harmless each time, but five years later the abstract class has fourteen hooks, half of them with empty default bodies, half overridden by exactly one subclass each, and the template method itself is a fifty-line tour through the hook table. New subclasses must understand the entire tour to determine which hooks they need; the algorithm's shape is no longer visible at a glance, and the line between essential structure and accumulated detail has been blurred.

The smell is real, and it is a smell of the pattern, not just of one particular use. When you find yourself adding the fourth or fifth hook, stop and consider whether the algorithm really has that many natural variation points. Often, what is happening is that the abstract class is trying to be everything to everyone, three or four different algorithms wearing the same name, and what you actually need is two or three smaller abstract classes, each with a tighter template method and fewer hooks. Splitting one bloated abstract class into several focused ones is a substantial refactor but usually pays for itself within months.

### The deep hierarchy

Template Method is a two-level pattern: one abstract base and one layer of concrete subclasses. The moment a concrete subclass becomes the base for further subclasses, you have a deep hierarchy, and the pattern's invariants start to break down. The intermediate class has to be both concrete enough to be subclassed and abstract enough not to be instantiated; the template method's call order has to make sense at all three levels; and the rules about which methods are abstract, concrete, and hook get fuzzy.

The textbook advice is simple: don't do that. Three- and four-level inheritance hierarchies are almost always a sign that something else is going on, most commonly that the variation captured in the deeper layers should be expressed by composition (each subclass holds a Strategy or a configuration object) rather than further inheritance. If you find your Template Method hierarchy growing past two levels, that is the point to step back and consider a different design.

> ⚠ **Warning:**
> **The hierarchy gets deep when nobody is watching.** The fragile base class, the hook explosion, and the deep hierarchy all tend to creep in incrementally, one well-meaning change at a time, and only become visible when a senior engineer joins the team and notices the whole shape at once. The defence is an occasional, deliberate review of the inheritance hierarchy at the design level: ask not "is each individual change good?" but "is the abstract class still doing what we designed it to do, or has it become a kitchen sink?"

## When NOT to use Template Method

- **When the algorithm's order itself varies.** If different variants want different orders of the same steps, not just different contents, then the fixed-skeleton constraint is wrong for the problem, and Strategy, the Command pattern, or a small pipeline framework will fit better.
- **When the variants need to be composed at runtime.** Template Method binds a variant to a type. If the same object needs to behave like Football at 10 am and Cricket at 11 am, the same instance, different rules, you want Strategy, not Template Method.

- **When there is only one or two subclasses.** Two variants do not justify the indirection of an abstract base class with primitive operations. Write the two classes directly, factor out shared helper functions if useful, and revisit only when a third variant arrives.
- **When the steps don't really form an algorithm.** If the "template method" is just a sequence of unrelated method calls with no inherent reason to be in that order, the pattern is dressing up incidental code. Refactor the methods into independent operations that the client can call directly, or into a small pipeline framework.

## Summary

The Template Method Pattern is one of the smallest, oldest, and most widely used patterns in the Gang of Four catalogue, and it captures something fundamental about how frameworks are built. The pattern in seven sentences:

- **Skeleton in one place.** Define the order of steps of an algorithm in a single method, the *template method*, that lives in an abstract base class.
- **Variable steps as virtual methods.** Declare the steps that vary between subclasses as pure virtual member functions; each concrete subclass implements them.
- **Invariant steps as concrete methods.** Implement the steps that are the same across all subclasses directly in the base class as non-virtual member functions; subclasses inherit them.
- **Optional steps as hooks.** Use virtual member functions with empty (or default) bodies for steps that have a sensible default but might be customized by some subclasses.
- **Lock the template method.** Make the template method itself non-virtual, or virtual and `final`, so subclasses cannot reorder the algorithm by overriding it.
- **Hollywood Principle.** The pattern produces the inversion-of-control shape: the high-level abstract class calls down into the low-level concrete subclasses, never the other way around.
- **Two levels, deliberate use.** Keep the inheritance shallow (one abstract base, one layer of concrete subclasses), use the pattern only when the variation is genuinely per-step and per-subclass, and switch to Strategy when the algorithm itself needs to vary or be swapped at runtime.

In a sentence, *the Template Method Pattern defines the skeleton of an algorithm in one place and lets subclasses fill in the holes, and the whole point is that the holes are the only things subclasses get to decide.* Master that single idea, and you can read most production frameworks at a glance.

## Practice Problems

### Problem 1

Given the Game / Football / Cricket code in Section 4, list the exact sequence of method calls produced by the following client code:

```
auto g1 = std::make_unique<Football>();
auto g2 = std::make_unique<Cricket>();
g1->play();
g2->play();
```

For each method call, state whether it dispatches statically (resolved at compile time) or dynamically (through the v-table). Why is `play` static dispatch but `initialize` dynamic?

### Problem 2

Write an abstract base class `CaffeineBeverage` whose template method `prepareRecipe()` calls, in order: `boilWater`, `brew`, `pourInCup`, and `addCondiments`. Decide which of the four operations are concrete and which are abstract. Then write two concrete subclasses, `Tea` and `Coffee`, that fill in only what they have to.

### Problem 3

Extend your CaffeineBeverage class from Problem 2 so that `addCondiments` is only called if a hook `customerWantsCondiments()` returns *true*. The default implementation should return *true*. Show how a `BlackCoffee` subclass would suppress condiments by overriding the hook.

### Problem 4

A colleague writes the following base class:

```
class FileExporter {
public:
    virtual void exportFile() {
        openFile();
        writeContent();
        closeFile();
    }
    virtual void openFile()     { /* open */ }
    virtual void writeContent() { /* default */ }
    virtual void closeFile()    { /* close */ }
};
```

Identify at least three things wrong with this design from the perspective of the Template Method Pattern, and propose fixes. (Hints: think about which methods should be virtual, which should be final, and which should be abstract.)

### Problem 5

For each of the following framework snippets, decide whether it is Template Method, Factory Method, or Strategy, and explain why:

- A `UnitTest` base class with virtual `setUp`, `runTest`, and `tearDown`; a `run()` method that calls them in order.
- A `Sorter` class that takes a `Comparator` object in its constructor and stores it as a member.
- A `Logger` base class with a virtual `createFormatter()` method; its concrete subclasses (`ConsoleLogger`, `FileLogger`) each return a different `Formatter` instance.
- A `HttpServlet` base class whose `service()` method dispatches to overridable `doGet`, `doPost`, etc.

### Problem 6

Design a `DataPipeline` base class with a template method `run()` that calls `extract`, `transform`, and `load` in order. The `transform` step is the only one that varies across the three subclasses `CsvPipeline`, `JsonPipeline`, and `XmlPipeline`. Now suppose that each pipeline also needs a runtime-swappable validation policy (`Strict`, `Lenient`). Refactor the design to use Template Method for the `transform` step and Strategy for the validation policy. Sketch the resulting class diagram.

### Problem 7

Read the following subclass and identify what it does wrong relative to the textbook Template Method discipline:

```
class StreamingFootball final : public Game {
public:
    void play() {                        // hides Game::play
        broadcastStart();
        initialize();
        startPlay();
        broadcastUpdate();
        endPlay();
        broadcastEnd();
    }
protected:
    void initialize() override { /*...*/ }
    void startPlay()  override { /*...*/ }
    void endPlay()    override { /*...*/ }
private:
    void broadcastStart()  { /*...*/ }
    void broadcastUpdate() { /*...*/ }
    void broadcastEnd()    { /*...*/ }
};
```

Why is this design problematic? Propose two alternative ways to add the broadcast steps without violating the pattern, one using hooks, one using composition.

### Problem 8

Look at a piece of software you have worked on in the last few months. Pick one module that is roughly a framework, your code calls into it, it calls back into your code. List, for that module, the calls that go from your code into the framework, and the calls that go from the framework back into your code. Are the dependencies acyclic? If not, where do they form cycles, and what would it take to break them?

### Problem 9

Each of the following situations is a poor fit for Template Method. For each, explain why, and propose a more appropriate design pattern:

- A Shape hierarchy where each shape has an `area()` method but the methods do not share any algorithmic structure.
- A user-interface widget that needs to switch between five different rendering strategies (light theme, dark theme, high-contrast, large-text, mobile) at runtime based on user preferences.
- A class with exactly one subclass that overrides exactly one method.
- A request handler whose middle three steps need to be executable in any order chosen by configuration.

### Problem 10

Write a complete C++ program (a single `.cpp` file is fine) that defines an abstract `Game` base class with the Section 4 structure, plus three concrete subclasses: `TicTacToe`, `Chess`, and `RockPaperScissors`. Each subclass should:

- Implement `initialize`, `startPlay`, and `endPlay` with a short message that names the game.
- Add a hook `displayInstructions()` called between `initialize` and `startPlay`, with a default empty body, that `Chess` overrides to print "Chess instructions: see page 1 of the FIDE laws."
- Be exercised by a `main` function that constructs each game through a `std::unique_ptr<Game>` and calls `play()`.

Confirm that the broadcast/instruction step appears in the correct place in the output and that the empty hook is invisible for the games that do not override it.