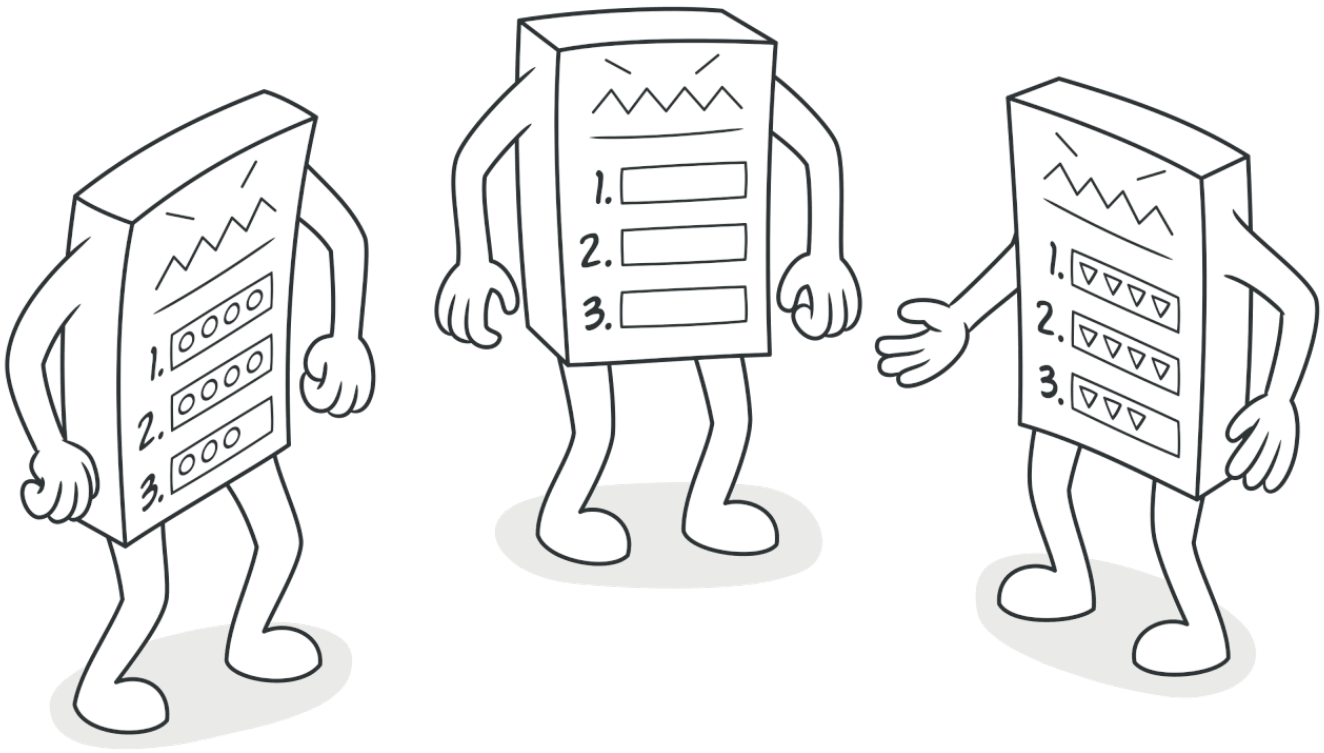
Template Method

Intent

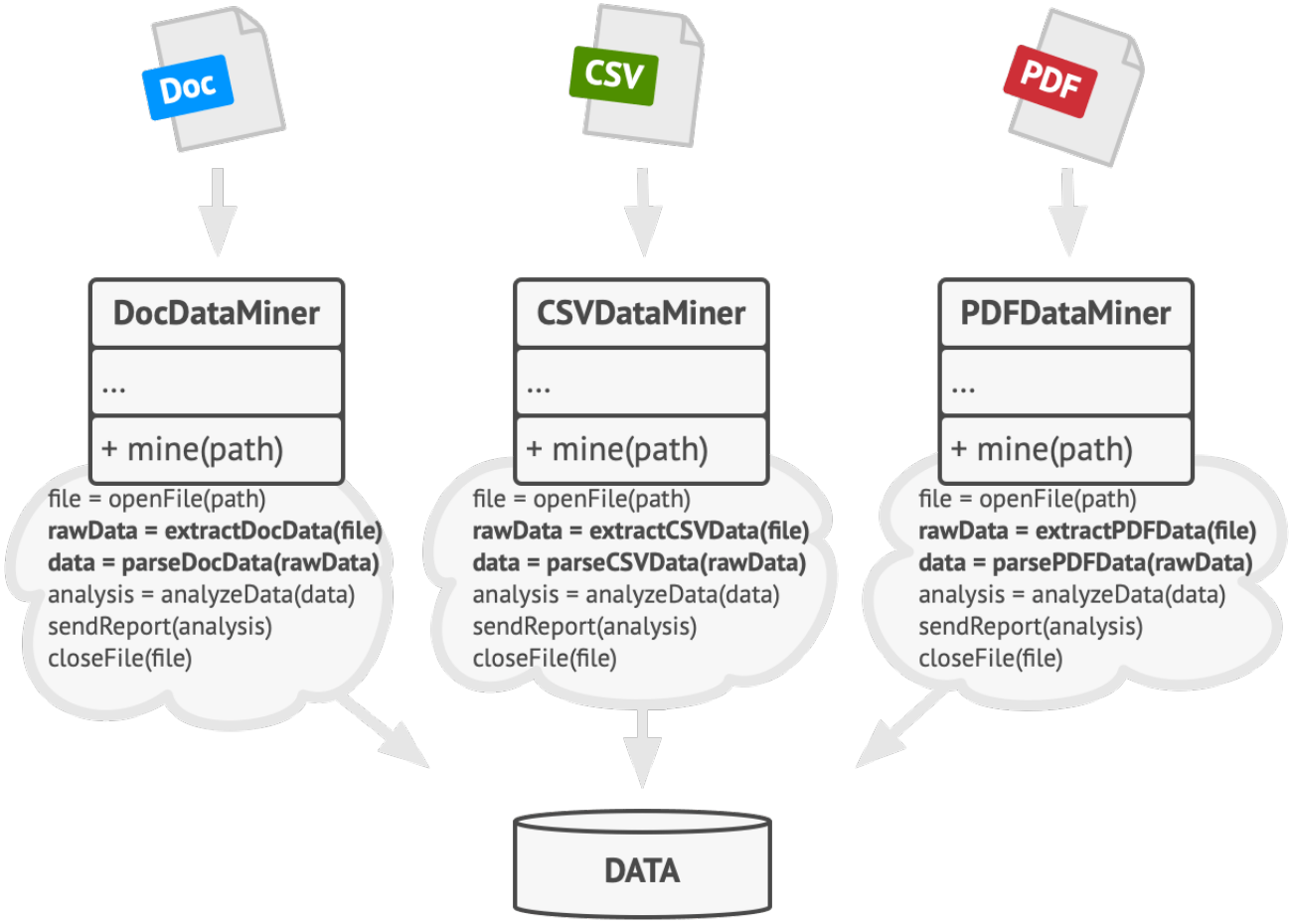
Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



Problem

Imagine that you’re creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.

The first version of the app could work only with DOC files. In the following version, it was able to support CSV files. A month later, you “taught” it to extract data from PDF files.



Data mining classes contained a lot of duplicate code.

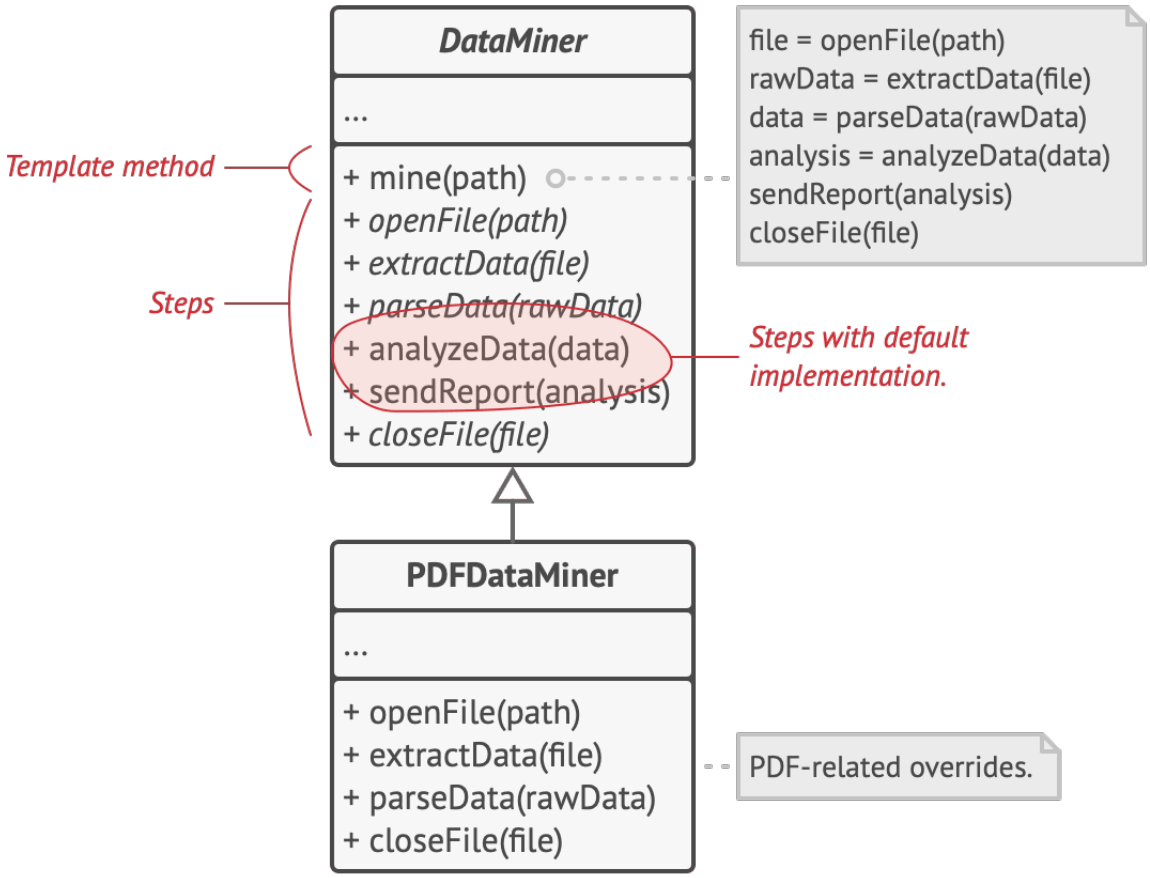
At some point, you noticed that all three classes have a lot of similar code. While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis is almost identical. Wouldn't it be great to get rid of the code duplication, leaving the algorithm structure intact?

There was another problem related to client code that used these classes. It had lots of conditionals that picked a proper course of action depending on the class of the processing object. If all three processing classes had a common interface or a base class, you'd be able to eliminate the conditionals in client code and use polymorphism when calling methods on a processing object.

Solution

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method*. The steps may either be `abstract`, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

Let's see how this will play out in our data mining app. We can create a base class for all three parsing algorithms. This class defines a template method consisting of a series of calls to various document-processing steps.



Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

At first, we can declare all steps `abstract`, forcing the subclasses to provide their own implementations for these methods. In our case,

subclasses already have all necessary implementations, so the only thing we might need to do is adjust signatures of the methods to match the methods of the superclass.

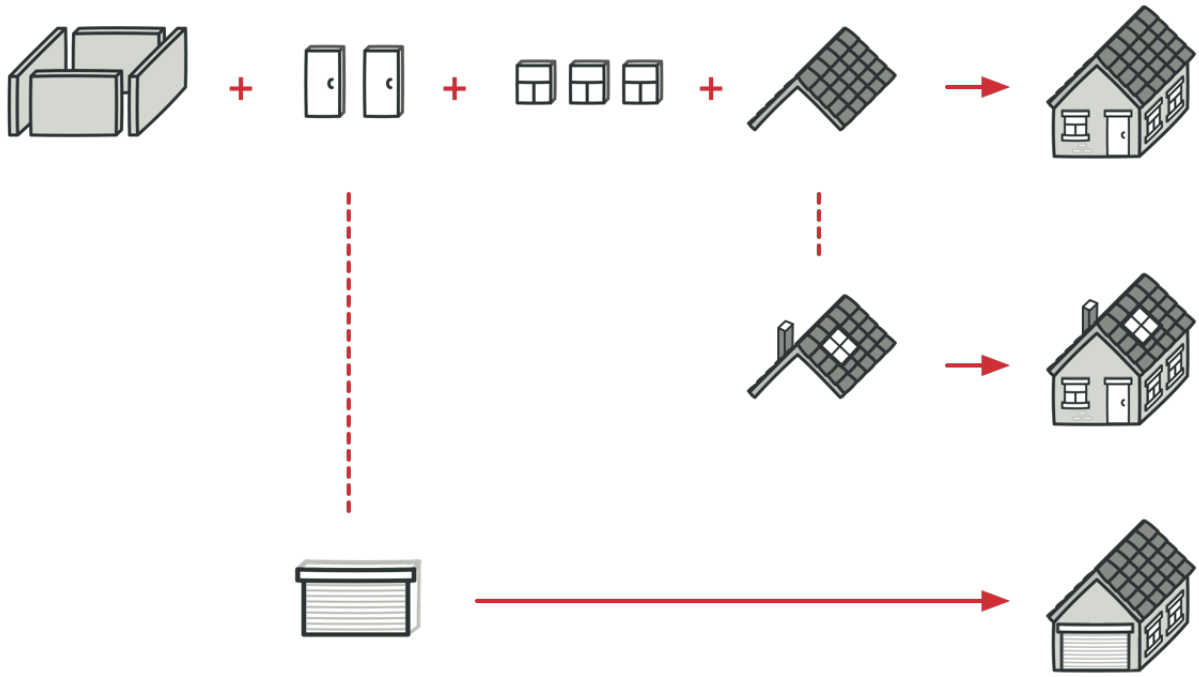
Now, let's see what we can do to get rid of the duplicate code. It looks like the code for opening/closing files and extracting/parsing data is different for various data formats, so there's no point in touching those methods. However, implementation of other steps, such as analyzing the raw data and composing reports, is very similar, so it can be pulled up into the base class, where subclasses can share that code.

As you can see, we've got two types of steps:

- *abstract steps* must be implemented by every subclass
- *optional steps* already have some default implementation, but still can be overridden if needed

There's another type of step, called *hooks*. A hook is an optional step with an empty body. A template method would work even if a hook isn't overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points for an algorithm.

Real-World Analogy

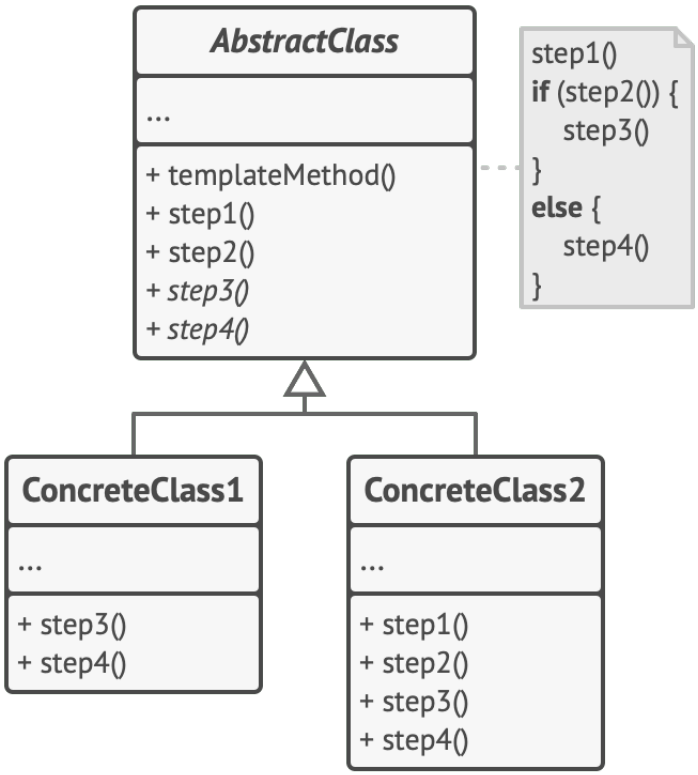


A typical architectural plan can be slightly altered to better fit the client's needs.

The template method approach can be used in mass housing construction. The architectural plan for building a standard house may contain several extension points that would let a potential owner adjust some details of the resulting house.

Each building step, such as laying the foundation, framing, building walls, installing plumbing and wiring for water and electricity, etc., can be slightly changed to make the resulting house a little bit different from others.

Structure

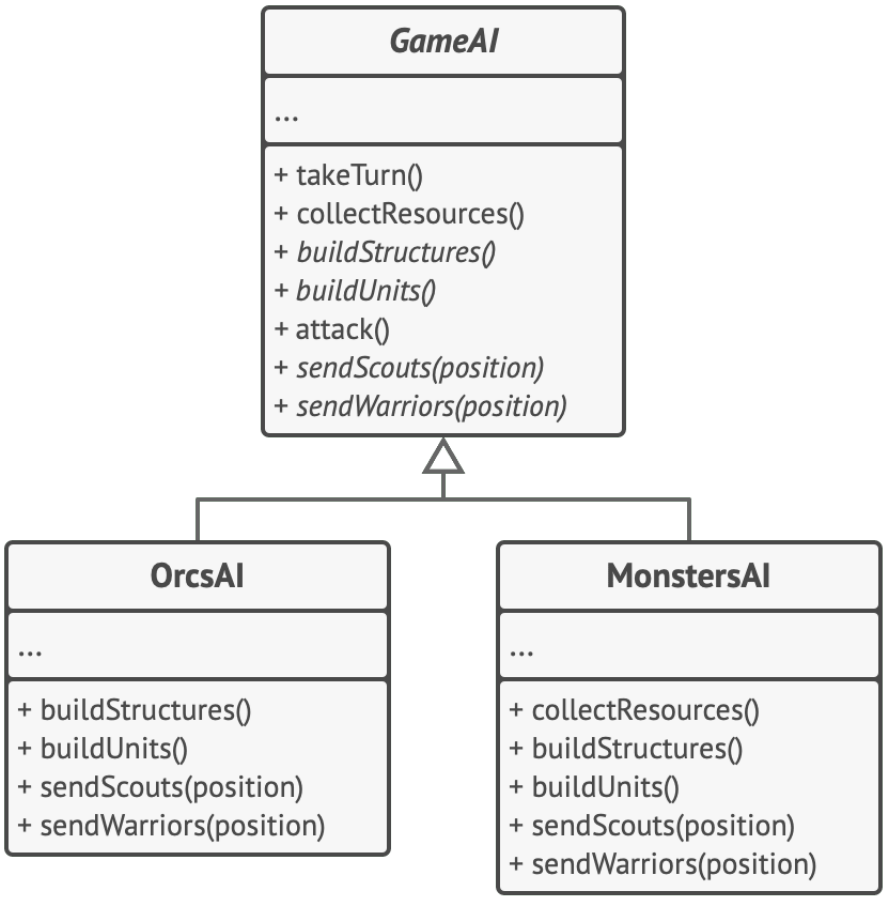


1. The **Abstract Class** declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared `abstract` or have some default implementation.

2. **Concrete Classes** can override all of the steps, but not the template method itself.

Pseudocode

In this example, the **Template Method** pattern provides a “skeleton” for various branches of artificial intelligence in a simple strategy video game.



AI classes of a simple video game.

All races in the game have almost the same types of units and buildings. Therefore you can reuse the same AI structure for various races, while being able to override some of the details. With this approach, you can override the orcs' AI to make it more aggressive, make humans more defense-oriented, and make monsters unable to build anything. Adding a new race to the game would require creating a new AI subclass and overriding the default methods declared in the base AI class.

```
// The abstract class defines a template method that contains a
// skeleton of some algorithm composed of calls, usually to
// abstract primitive operations. Concrete subclasses implement
// these operations, but leave the template method itself
// intact.
class GameAI is
    // The template method defines the skeleton of an algorithm.
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
```

```
            attack()

        // Some of the steps may be implemented right in a base
        // class.
        method collectResources() is
            foreach (s in this.builtStructures) do
                s.collect()

        // And some of them may be defined as abstract.
        abstract method buildStructures()
        abstract method buildUnits()

        // A class can have several template methods.
        method attack() is
            enemy = closestEnemy()
            if (enemy == null)
                sendScouts(map.center)
            else
                sendWarriors(enemy.position)

        abstract method sendScouts(position)
        abstract method sendWarriors(position)

    // Concrete classes have to implement all abstract operations of
    // the base class but they must not override the template method
    // itself.
    class OrcsAI extends GameAI is
        method buildStructures() is
            if (there are some resources) then
                // Build farms, then barracks, then stronghold.

        method buildUnits() is
            if (there are plenty of resources) then
                if (there are no scouts)
                    // Build peon, add it to scouts group.
                else
                    // Build grunt, add it to warriors group.

        // ...

        method sendScouts(position) is
            if (scouts.length > 0) then
                // Send scouts to position.

        method sendWarriors(position) is
            if (warriors.length > 5) then
                // Send warriors to position.

    // Subclasses can also override some operations with a default
    // implementation.
```

```
class MonstersAI extends GameAI is
    method collectResources() is
        // Monsters don't collect resources.

    method buildStructures() is
        // Monsters don't build structures.

    method buildUnits() is
        // Monsters don't build units.
```

Applicability

Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.

The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.

Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

When you turn such an algorithm into a template method, you can also pull up the steps with similar implementations into a superclass, eliminating code duplication. Code that varies between subclasses can remain in subclasses.

How to Implement

1. Analyze the target algorithm to see whether you can break it into steps. Consider which steps are common to all subclasses and which ones will always be unique.

2. Create the abstract base class and declare the template method and a set of abstract methods representing the algorithm's steps. Outline

the algorithm's structure in the template method by executing corresponding steps. Consider making the template method `final` to prevent subclasses from overriding it.

3. It's okay if all the steps end up being abstract. However, some steps might benefit from having a default implementation. Subclasses don't have to implement those methods.

4. Think of adding hooks between the crucial steps of the algorithm.

5. For each variation of the algorithm, create a new concrete subclass. It *must* implement all of the abstract steps, but *may* also override some of the optional ones.

Pros and Cons

You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
You can pull the duplicate code into a superclass.

Some clients may be limited by the provided skeleton of an algorithm.
You might violate the *Liskov Substitution Principle* by suppressing a default step implementation via a subclass.
Template methods tend to be harder to maintain the more steps they have.

Relations with Other Patterns

- Factory Method is a specialization of Template Method. At the same time, a *Factory Method* may serve as a step in a large *Template Method*.

- Template Method is based on inheritance: it lets you alter parts of an

algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. *Template Method* works at the class level, so it's static. *Strategy* works on the object level, letting you switch behaviors at runtime.