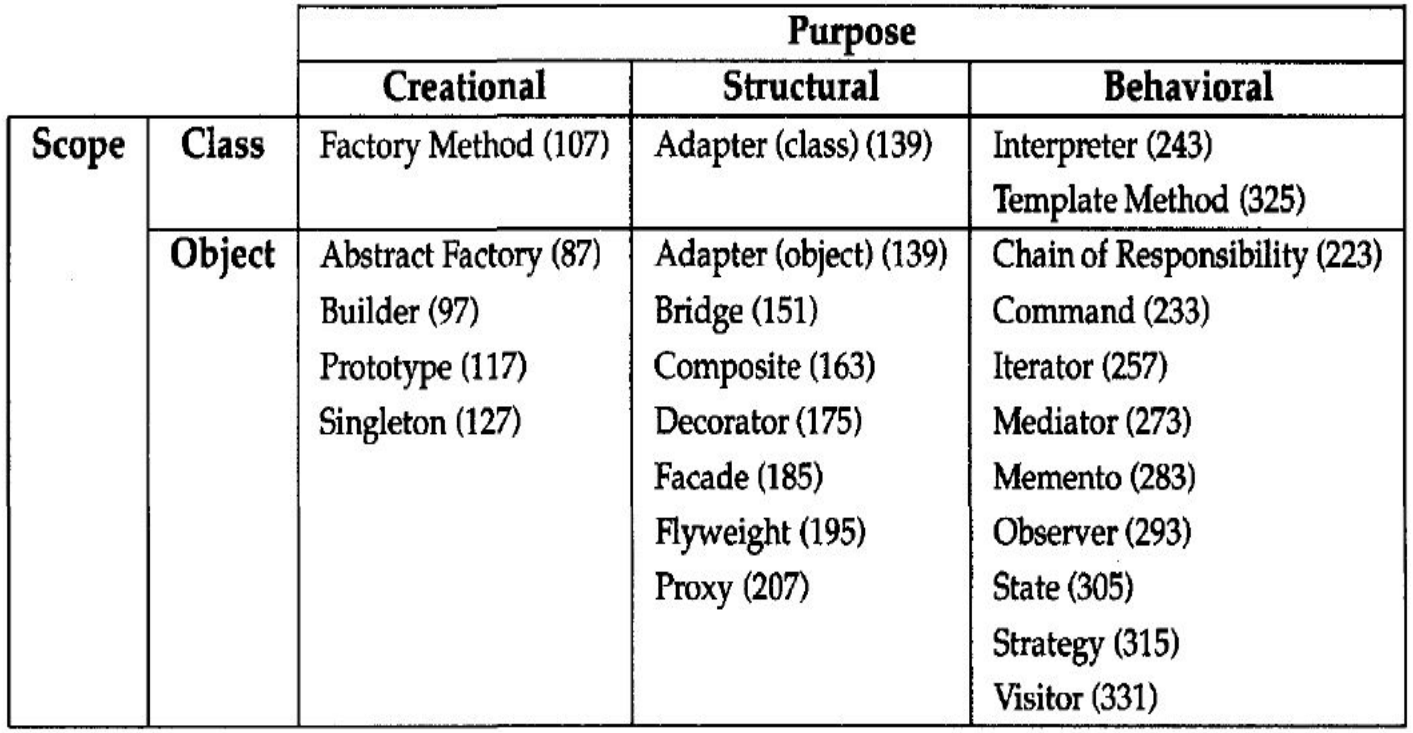
Behavioral Design Pattern

Instructor: Mehroze Khan

Behavioral Design Pattern

- Behavioral design patterns are concerned with algorithms and the **assignment of responsibilities** between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the **patterns of communication** between them.



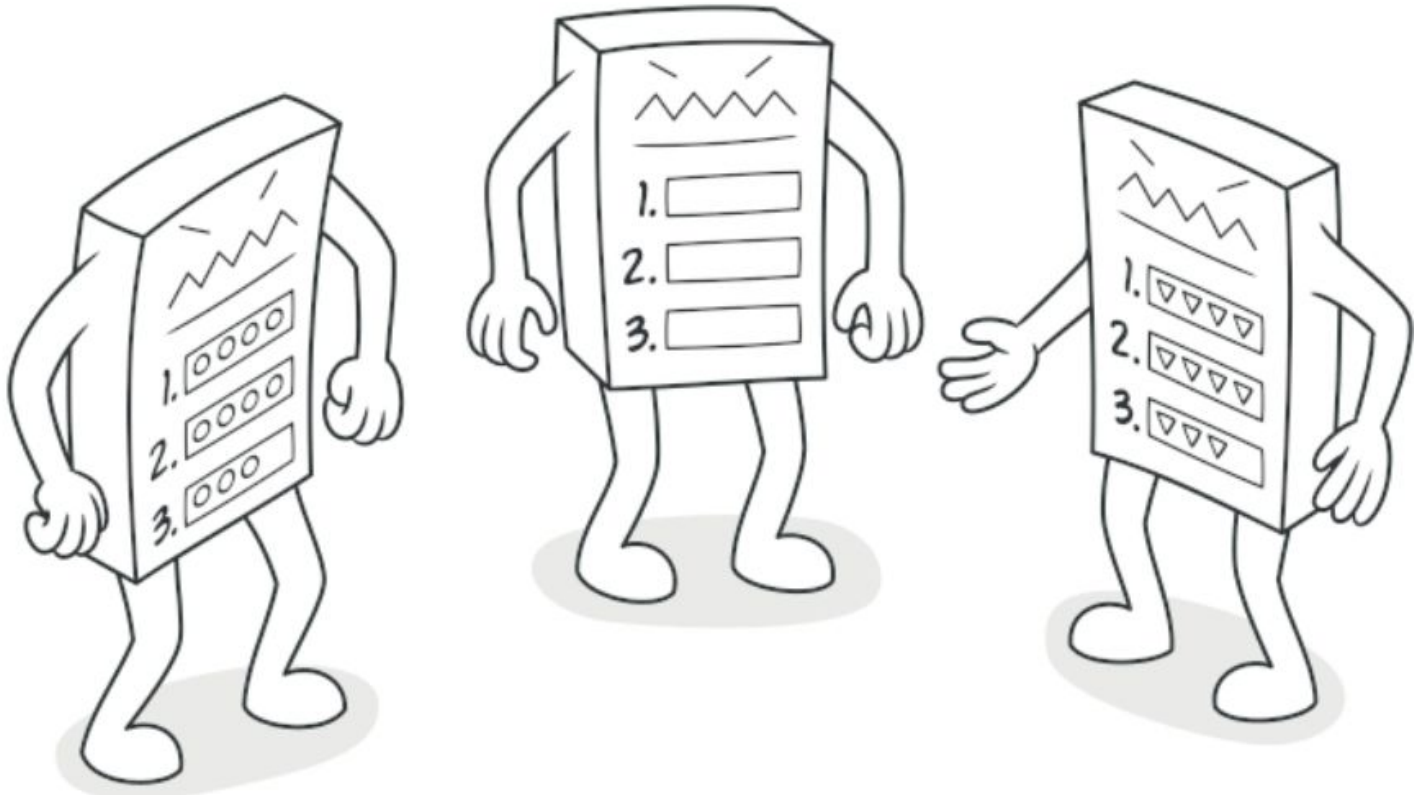
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243)
				Template Method (325)
	Object	Abstract Factory (87)	Adapter (object) (139)	Chain of Responsibility (223)
		Builder (97)	Bridge (151)	Command (233)
		Prototype (117)	Composite (163)	Iterator (257)
		Singleton (127)	Decorator (175)	Mediator (273)
			Facade (185)	Memento (283)
			Flyweight (195)	Observer (293)
			Proxy (207)	State (305)
				Strategy (315)
				Visitor (331)

Template Method Pattern

Template Method Pattern

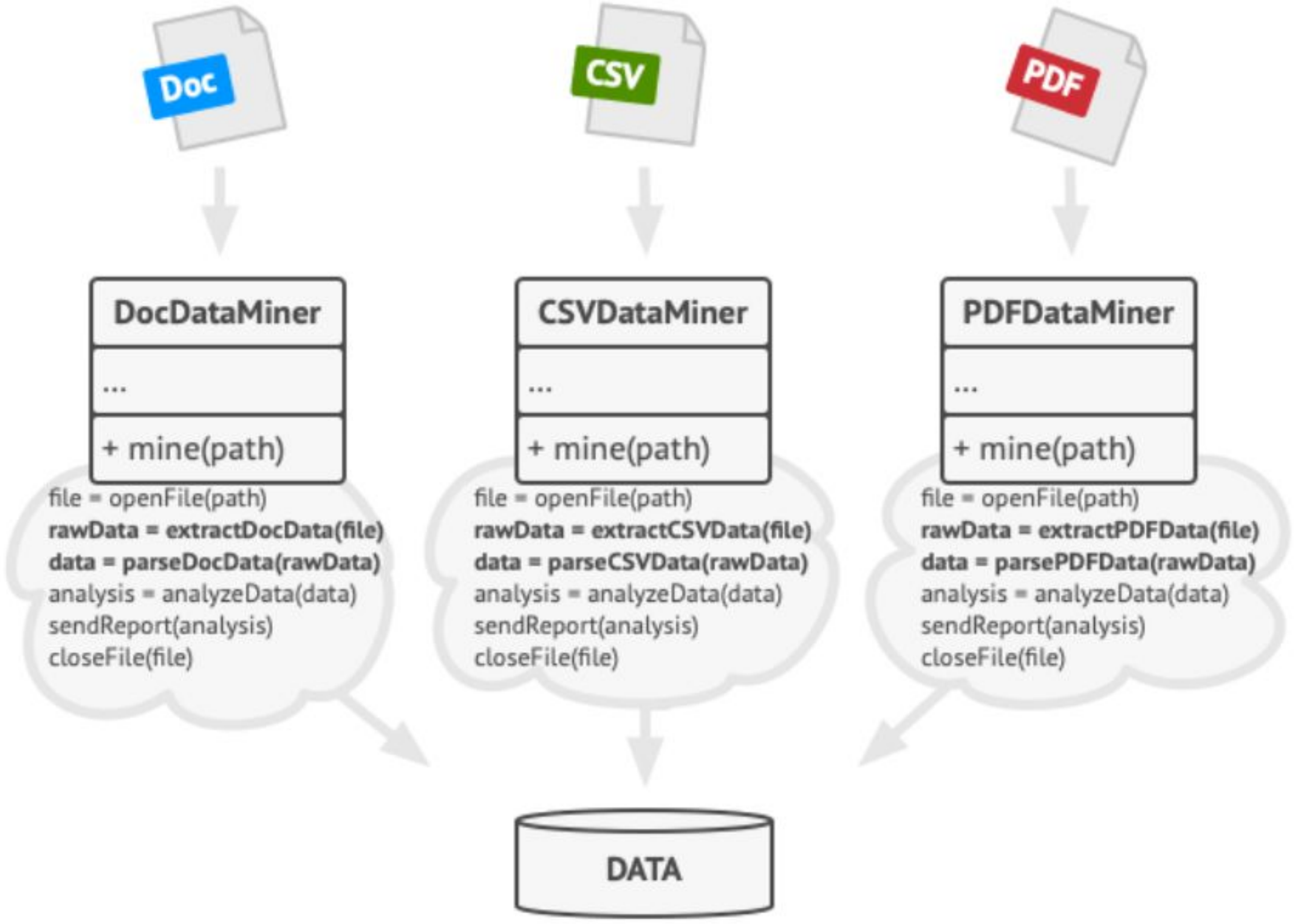
Intent:

- **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.



```
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

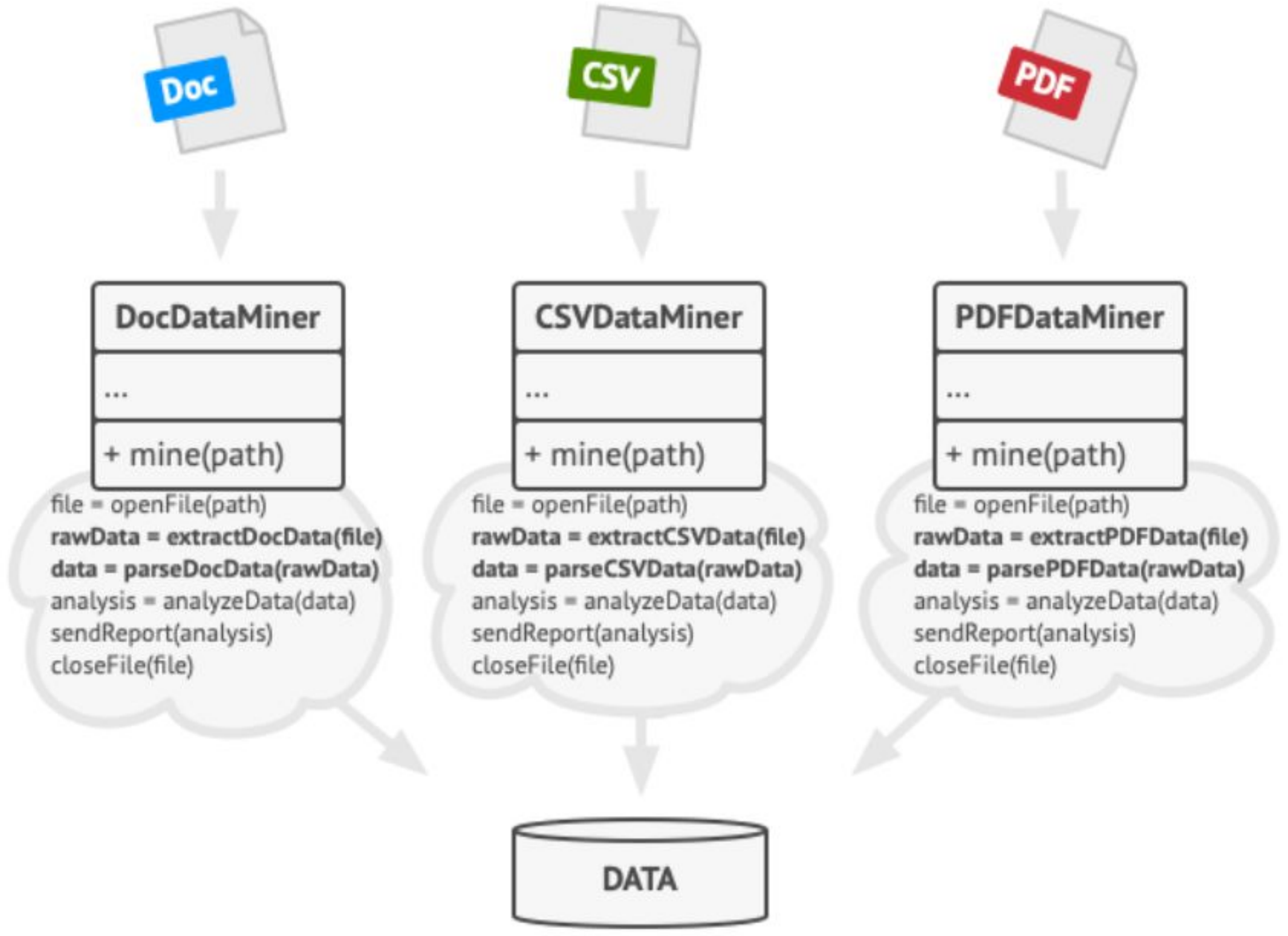
```
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

Data mining classes contained a lot of duplicate code.

Problem

- 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.



```
file = openFile(path)
rawData = extractDocData(file)
data = parseDocData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractCSVData(file)
data = parseCSVData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

```
file = openFile(path)
rawData = extractPDFData(file)
data = parsePDFData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
```

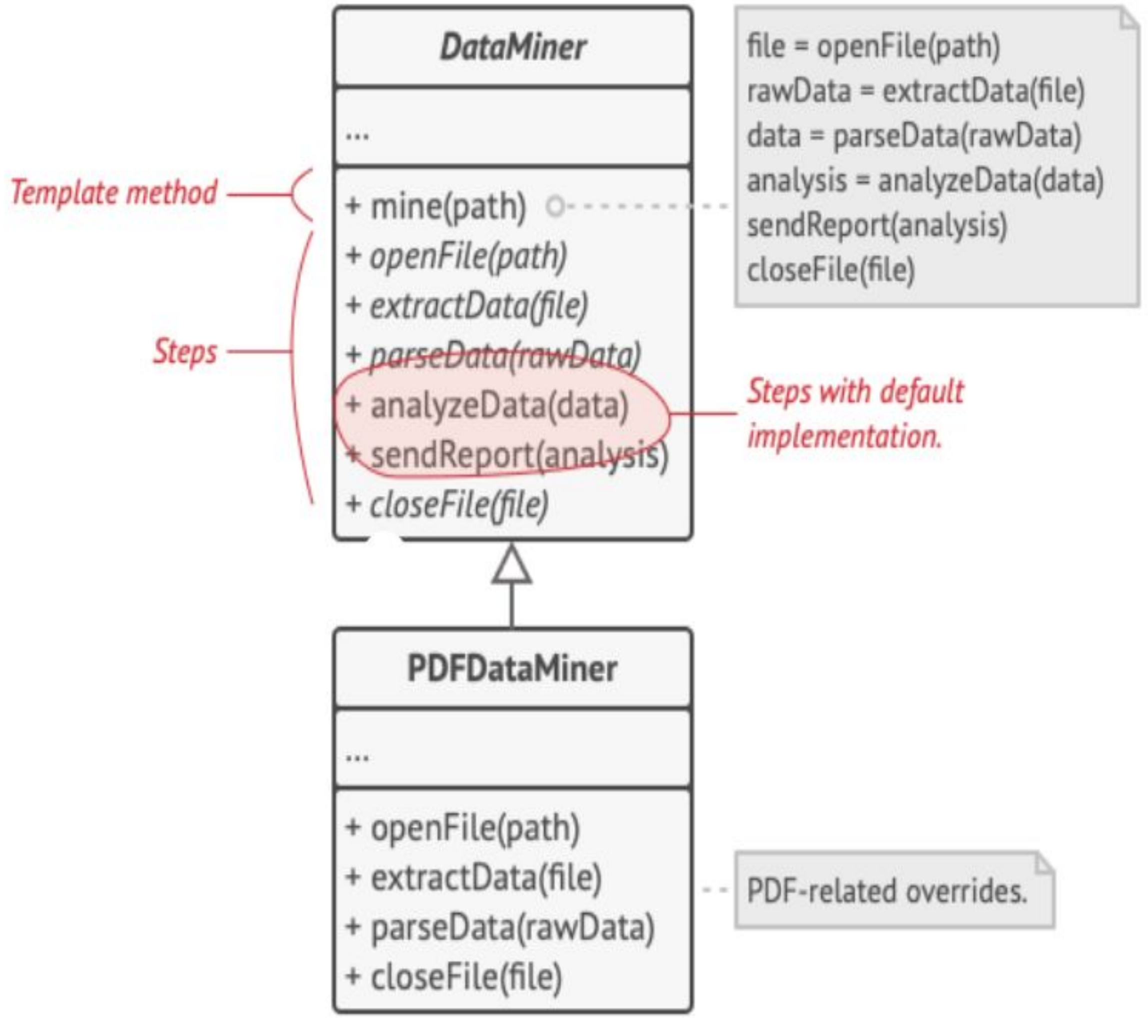
Data mining classes contained a lot of duplicate code.

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).

Solution

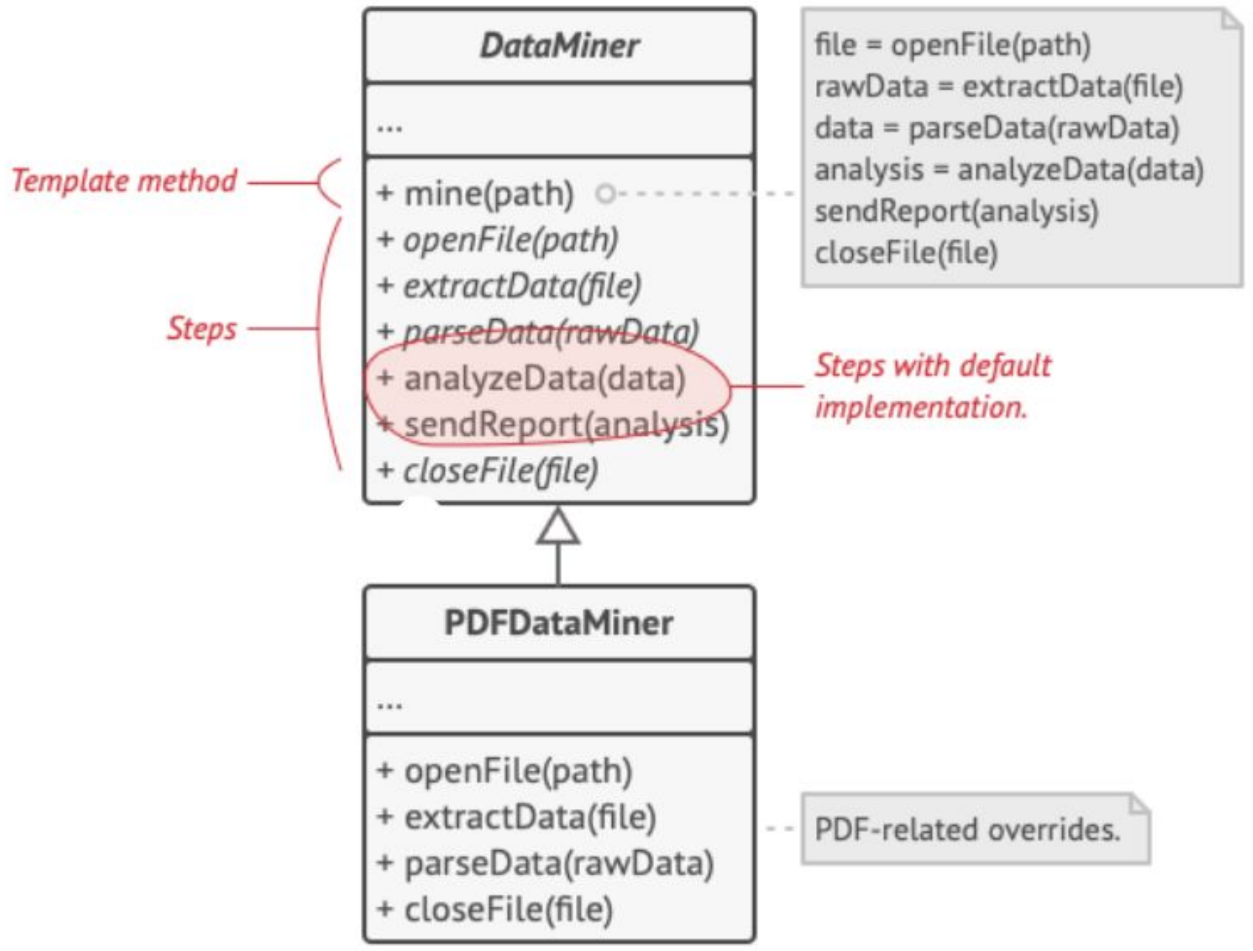
- We can create a base class for all three parsing algorithms.
- This class defines a template method consisting of a series of calls to various document-processing steps.



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

Solution

- 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.
- 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.



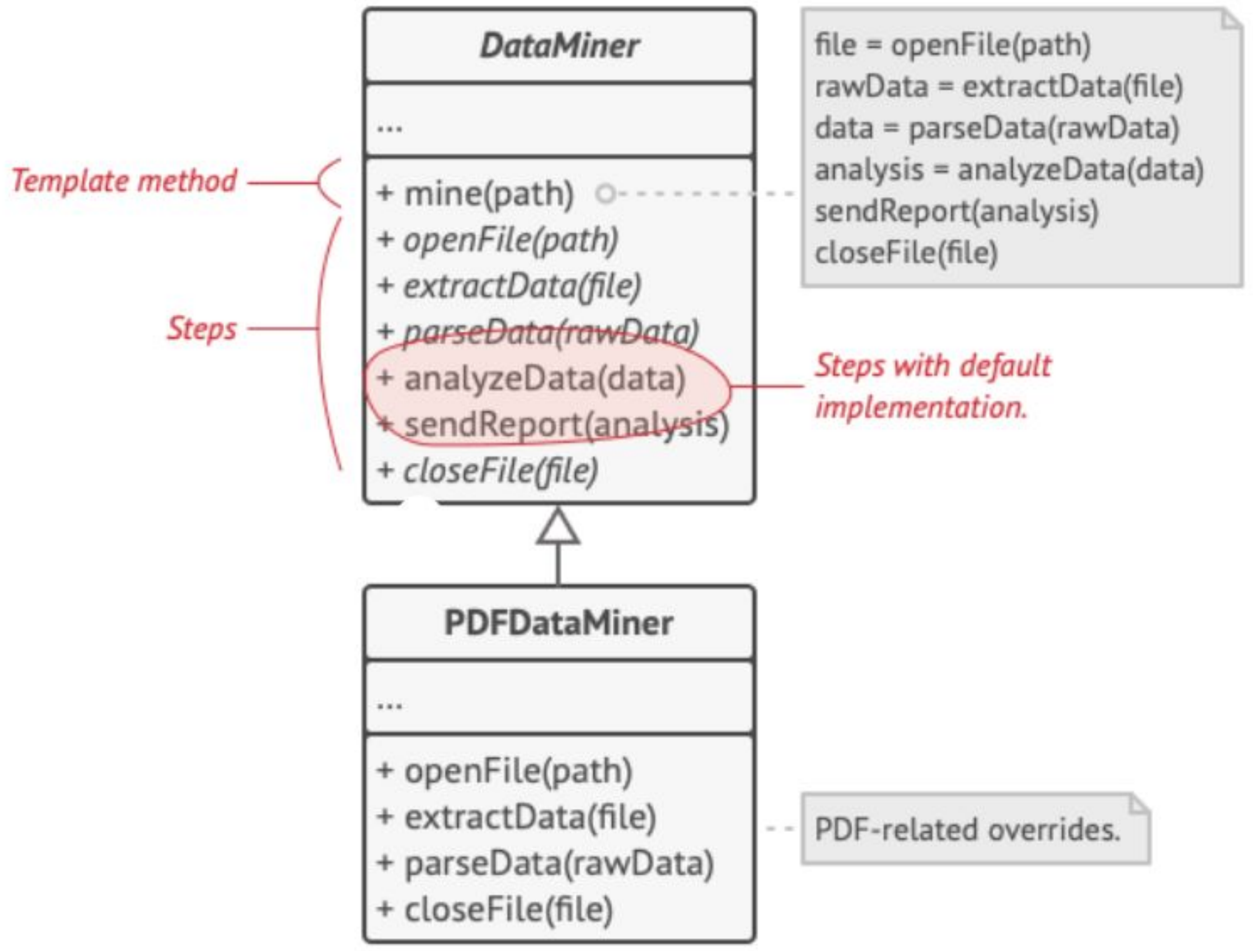
Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

Solution

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

1. Abstract steps must be implemented by every subclass
2. 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.

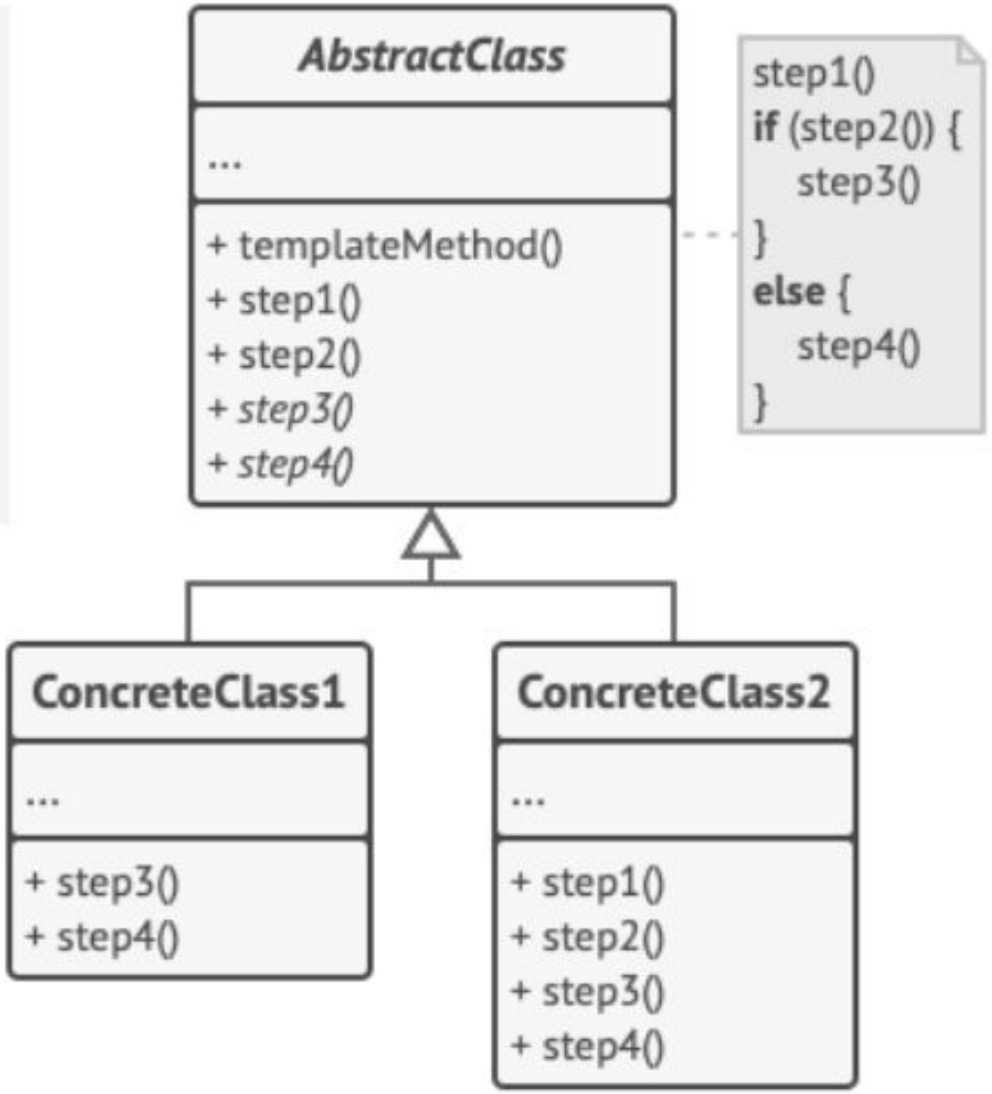


Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

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.

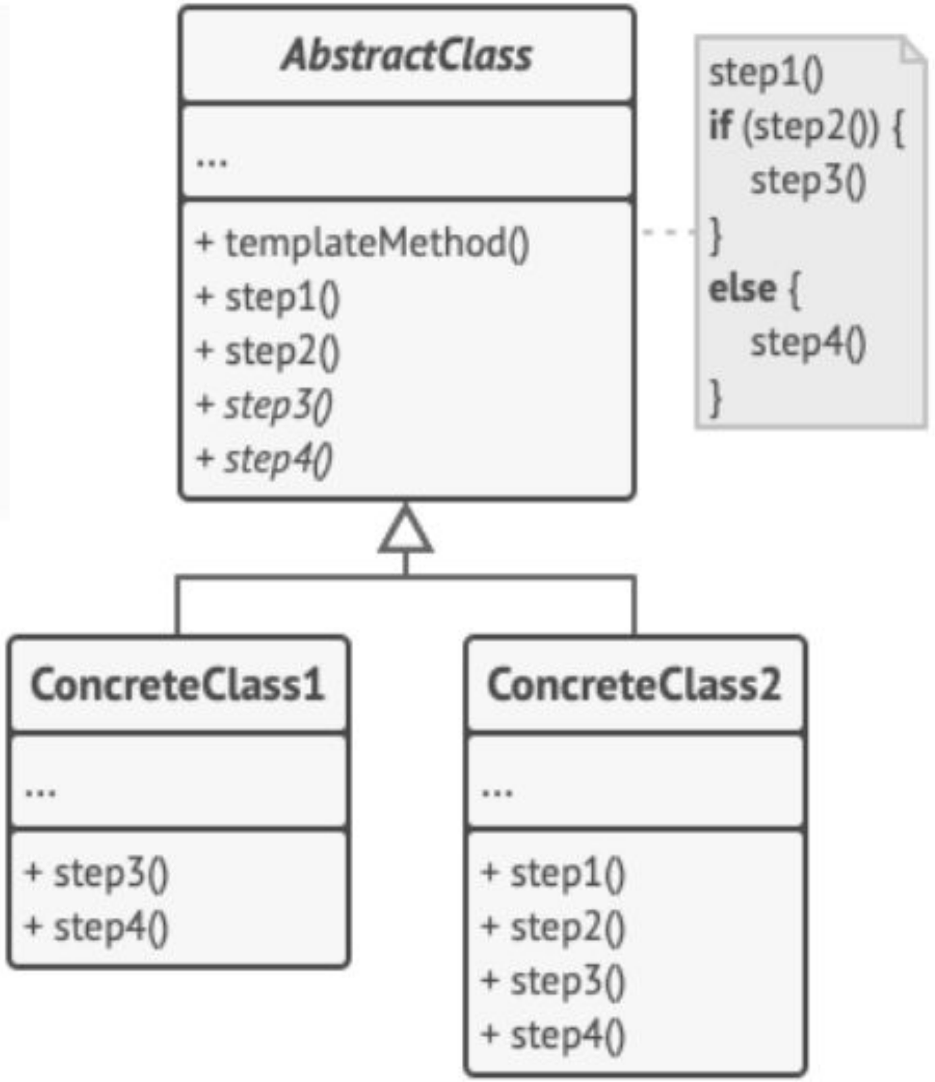
2 **Concrete Classes** can override all of the steps, but not the template method itself.



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.



```
step1()
if (step2()) {
   step3()
}
else {
   step4()
}
```

Example

```
// Abstract class defining the template method
class AbstractClass {
public:
   // Template method defining the algorithm
   void templateMethod() {
      step1();
      step2();
      commonStep();
      step3();
   }
   // Abstract methods to be implemented by subclasses
   virtual void step1() = 0;
   virtual void step3() = 0;
   // Default implementation of step2
   void step2() {
      cout << "Default Step 2" << endl;
   }
```

```
// Common step with a default implementation
   virtual void commonStep() {
      cout << "Common Step (Default implementation)" << endl;
   }
};
// Concrete class implementing the AbstractClass
class ConcreteClass : public AbstractClass {
public:
   void step1() override {
      cout << "Custom Step 1" << endl;
   }
   void step3() override {
      cout << "Custom Step 3" << endl;
   }
   // Overriding the commonStep method
   void commonStep() override {
      cout << "Custom Common Step" << endl;
   }    };
```

Example

```
int main() {
    AbstractClass* object = new ConcreteClass();

    // Calling the template method
    object->templateMethod();
}
```

// Output

```
Custom Step 1
Default Step 2
Custom Common Step
Custom Step 3
```

Another Example

```
class AbstractClass {
public:
  void TemplateMethod() const {
    BaseOperation1();
    RequiredOperations1();
    BaseOperation2();
    Hook1();
    RequiredOperation2();
    Hook2();
  }
protected:
  void BaseOperation1() {
    cout << "AbstractClass says: I am doing the bulk of the work\n";
  }
  void BaseOperation2() {
    cout << "AbstractClass says: But I let subclasses override some
operations\n";     }
```

```
  virtual void RequiredOperations1() = 0;
  virtual void RequiredOperation2() = 0;


  virtual void Hook1() {}
  virtual void Hook2() {}
};
class ConcreteClass1 : public AbstractClass {
protected:
  void RequiredOperations1() override {
    cout << "ConcreteClass1 says: Implemented Operation1\n";
  }
  void RequiredOperation2() override {
    cout << "ConcreteClass1 says: Implemented Operation2\n";
  }
};
```

Another Example

```
class ConcreteClass2 : public AbstractClass {
protected:
  void RequiredOperations1() override {
    cout << "ConcreteClass2 says: Implemented Operation1\n";
  }
  void RequiredOperation2() override {
    cout << "ConcreteClass2 says: Implemented Operation2\n";
  }
  void Hook1() override {
    cout << "ConcreteClass2 says: Overridden Hook1\n";
  }
};

void ClientCode(AbstractClass* c) {
  c->TemplateMethod();
}
```

```
 int main() {
 cout << "Same client code can work with different subclasses:\n";
 ConcreteClass1* concreteClass1 = new ConcreteClass1;
 ClientCode(concreteClass1);

 cout << "\n";

 cout << "Same client code can work with different subclasses:\n";
 ConcreteClass2* concreteClass2 = new ConcreteClass2;
 ClientCode(concreteClass2);
}
```

Another Example

```
int main() {
  cout << "Same client code can work with different subclasses:\n";
  ConcreteClass1* concreteClass1 = new ConcreteClass1;
  ClientCode(concreteClass1);

  cout << "\n";

  cout << "Same client code can work with different subclasses:\n";
  ConcreteClass2* concreteClass2 = new ConcreteClass2;
  ClientCode(concreteClass2);
}
```

// Output:

```
Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass1 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass1 says: Implemented Operation2


Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass2 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass2 says: Overridden Hook1
ConcreteClass2 says: Implemented Operation2
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Pearson, 1995.

Helping Links:

- https://refactoring.guru/design-patterns/