Software Design Patterns

Lecture 12

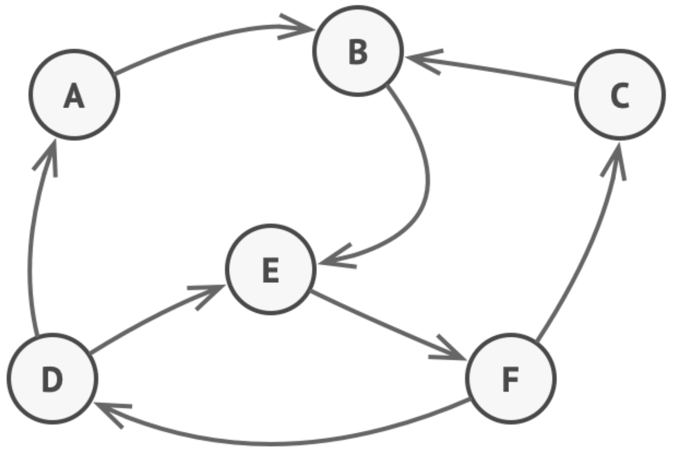
State, Strategy, and Template Method

Dr. Fan Hongfei

21 November 2024

State: Problem

- **Review: finite-state machine**

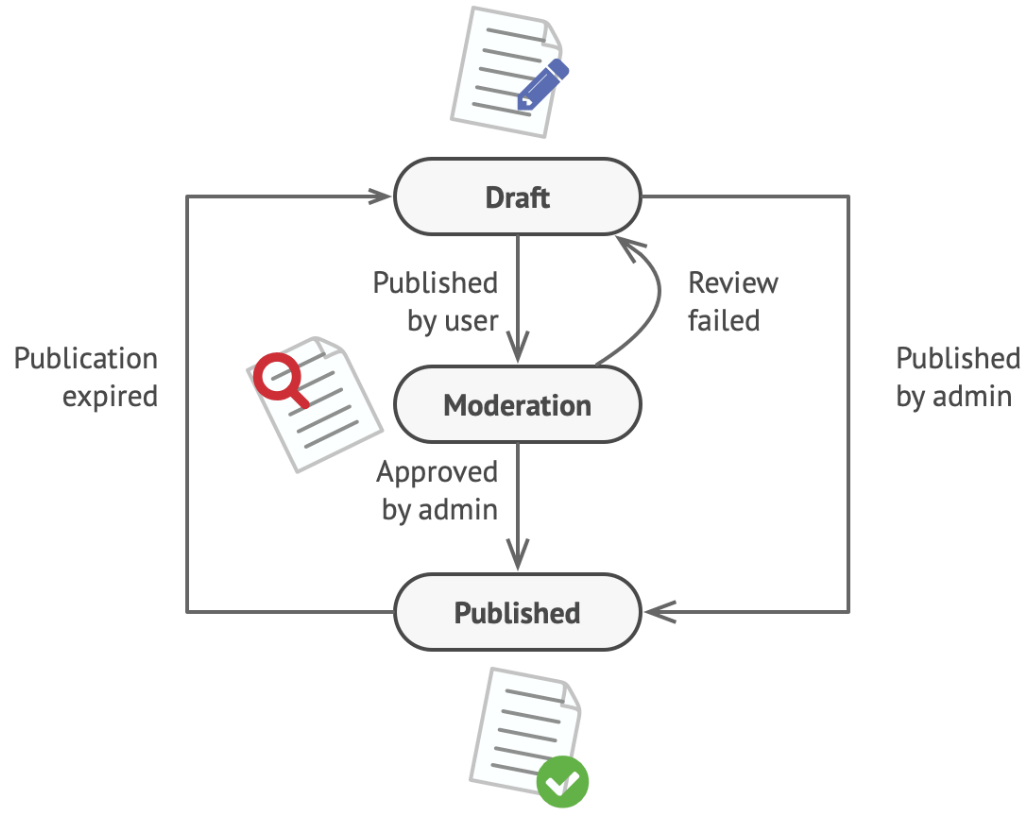


- A finite number of **states**
- The program behaves differently within a state
- Can be switched from one state to another, and switching rules (**transitions**) are also finite and predetermined

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/state]

State: Problem (cont.)

- **A document class**
  - **States: Draft, Moderation, Published**

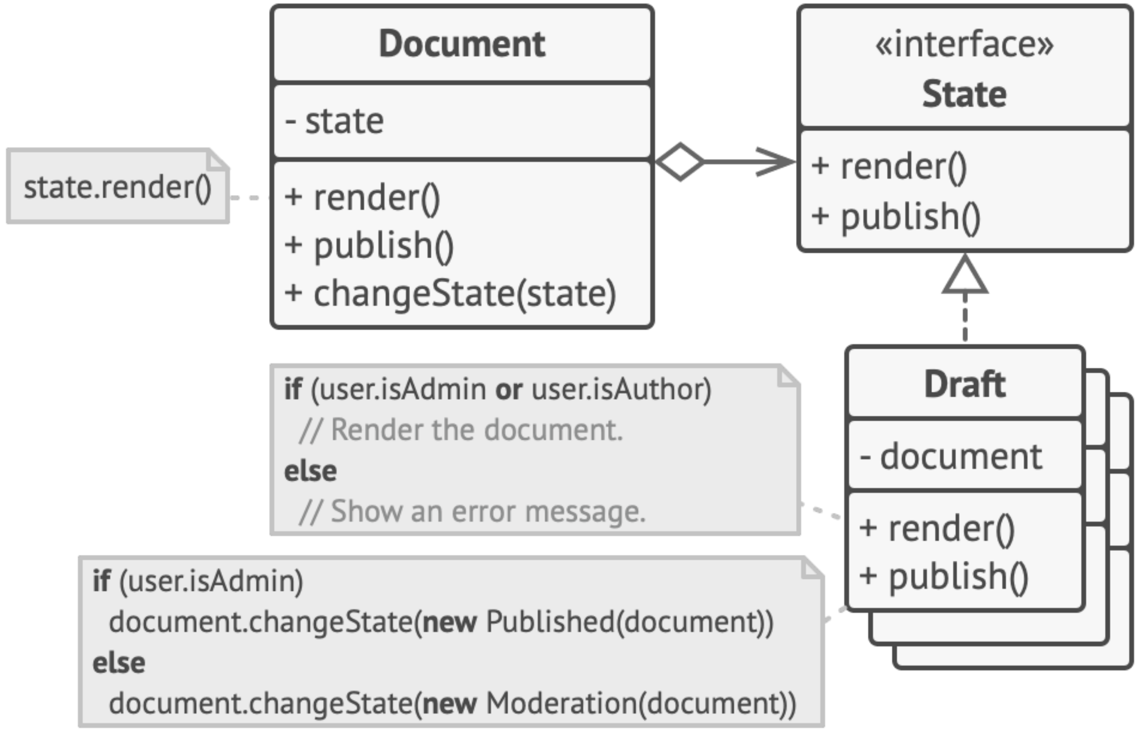


```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/state]

State: Solution

- Create new classes for all possible states, and extract all state-specific behaviors into these classes
- The original object: **context**
  - Storing a reference to the state
  - Delegating state-specific work to the state object
- State classes follow the **same interface**, and the context works with states through the interface

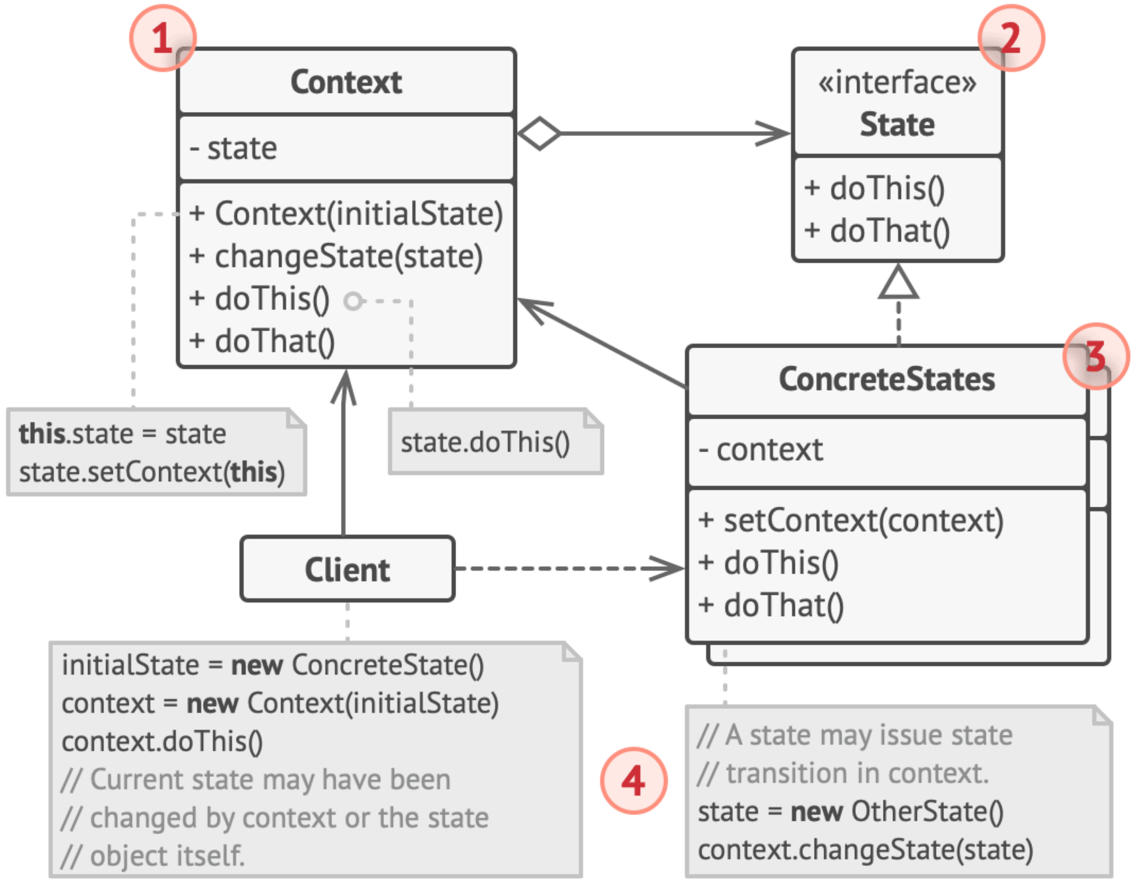


```
if (user.isAdmin or user.isAuthor)
  // Render the document.
else
  // Show an error message.
```

```
if (user.isAdmin)
  document.changeState(new Published(document))
else
  document.changeState(new Moderation(document))
```

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/state]

State: Structure



```
this.state = state
state.setContext(this)
```

```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

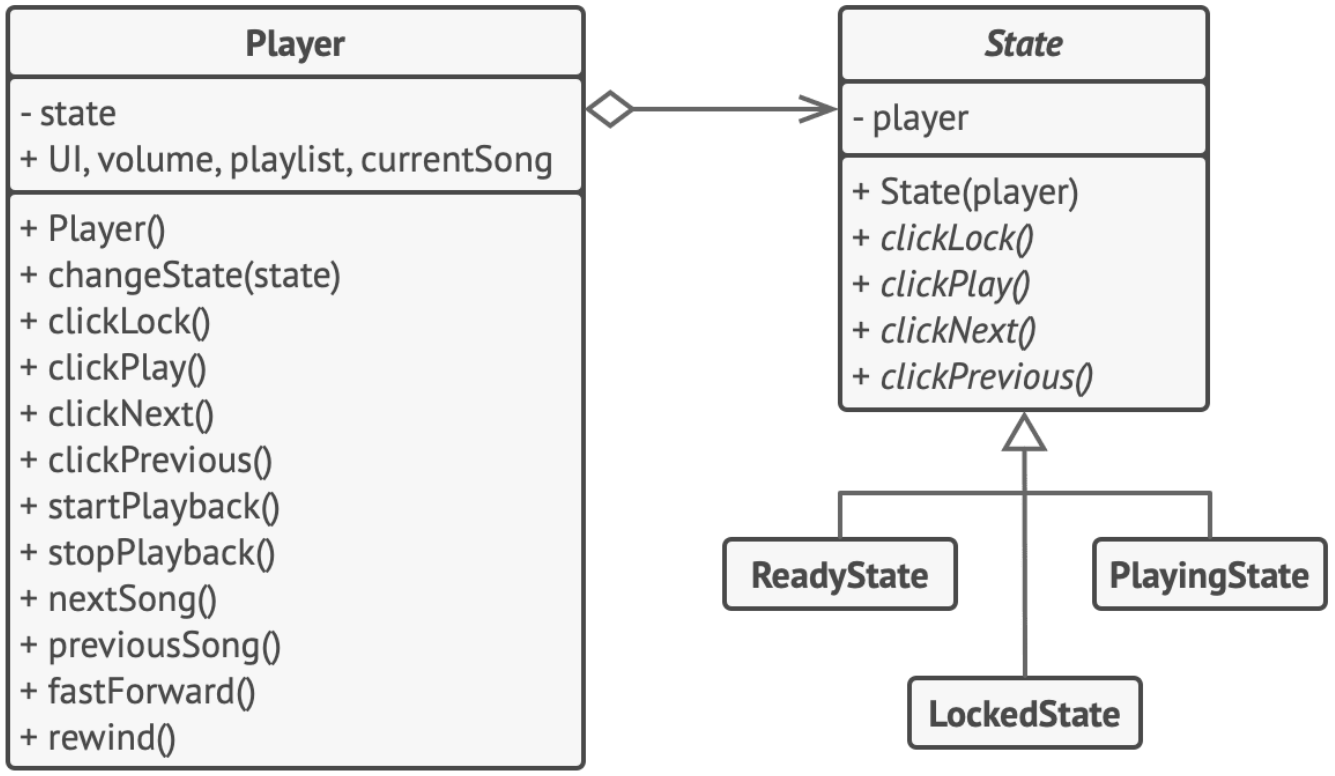
```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

1. **Context:** storing a reference to the state object, and exposes a setter for passing it a new state object
2. **State:** interface, declaring state-specific methods that make sense for all concrete states
3. **Concrete States:** implementations of state-specific methods
   – Optional: an intermediate abstract class
4. **Both** context and concreate states can execute the transition

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/state]

State: Example

- **Media player**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/state]

State: Applicability

- When an object behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently
  - Extract all state-specific code into a set of distinct classes, and add/change states independently
- When a class polluted with massive conditionals that alter how the class behaves according to the current values of the class' fields
  - Extract branches of these conditionals into methods of corresponding state classes
- When you have a lot of duplicate code across similar states and transitions of a condition-based state machine
  - Compose hierarchies of state classes and reduce duplication by extracting common code into abstract base classes

State: Implementation

1. Decide what class will act as the **context**: may be an existing class which has the state-dependent code, or a new class if state-specific code is distributed
2. Declare the **state interface**, with **state-specific behavior**
3. For every actual state, create a class that implements the state interface, and then go over the methods of the context and extract all code related to that state into the newly created class
   – If the code depends on private members of the context, consider:
      a) Make fields or methods public;
      b) Turn the behavior into a public method in the context; or
      c) Nest the state classes into the context class
4. In the context class, add **a reference** to the state and a public setter
5. Go over the method of the context, and replace state conditionals with calls to state objects
6. To switch the state, create an instance of the state class and pass it to the context (within the context, in various states, or in the client)

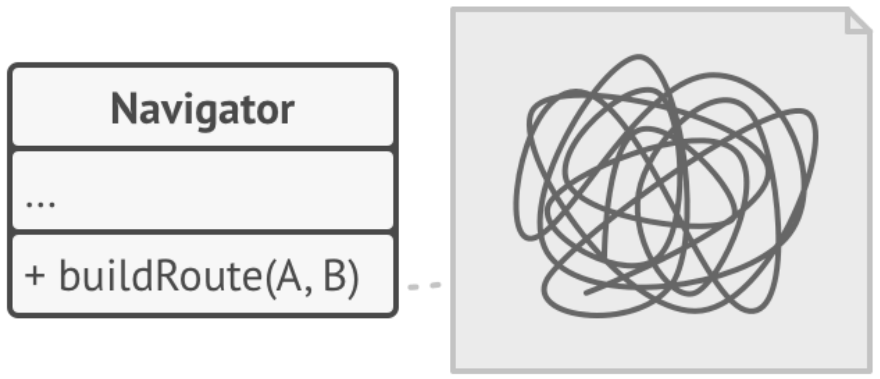
State: Pros and Cons

- **Pros**
  - Single Responsibility Principle: organizing the code related to particular states into separate classes
  - Open/Closed Principle: introducing new states without changing existing state classes or the context
  - Simplify the code of the context by eliminating state machine conditionals
- **Cons**
  - Can be overkill if a state machine has only a few states or rarely changes

Strategy: Problem

- **Example: route planning feature in a navigation app**

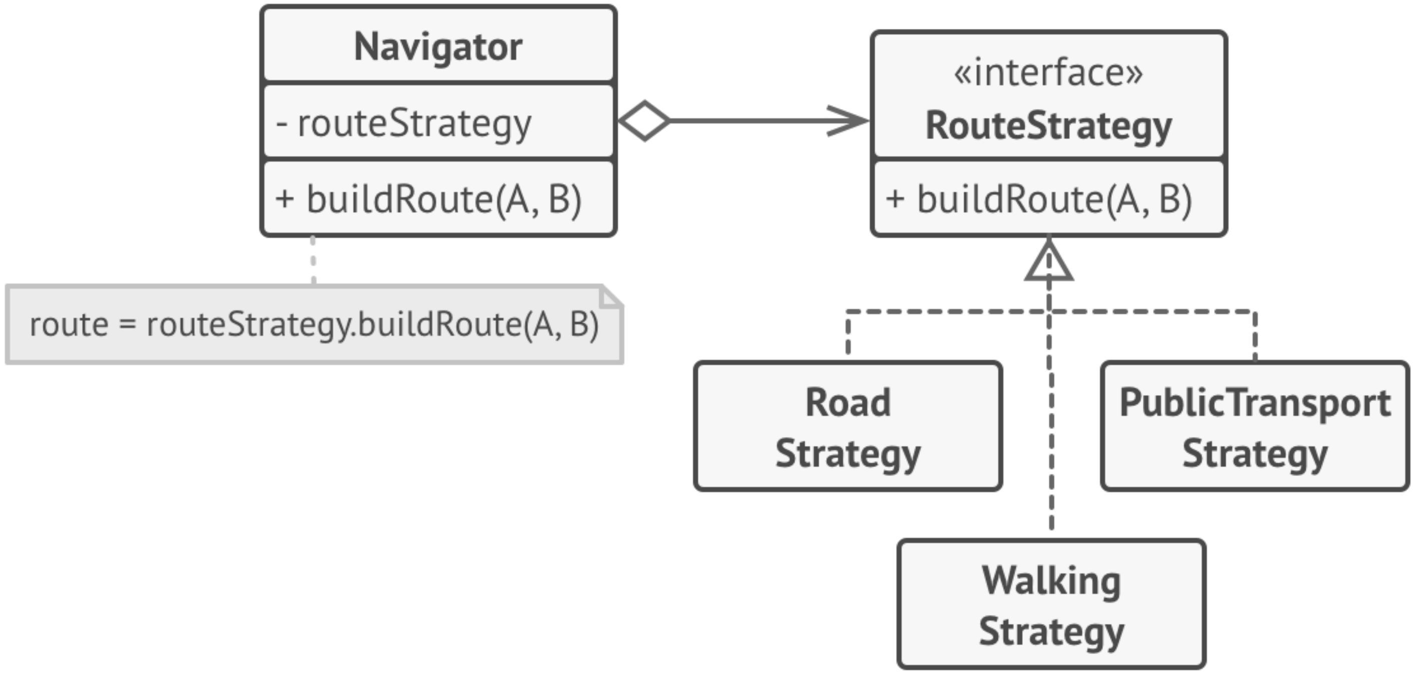
- V1: routes over roads
- V2: walking routes
- V3: public transport
- More...



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/strategy]

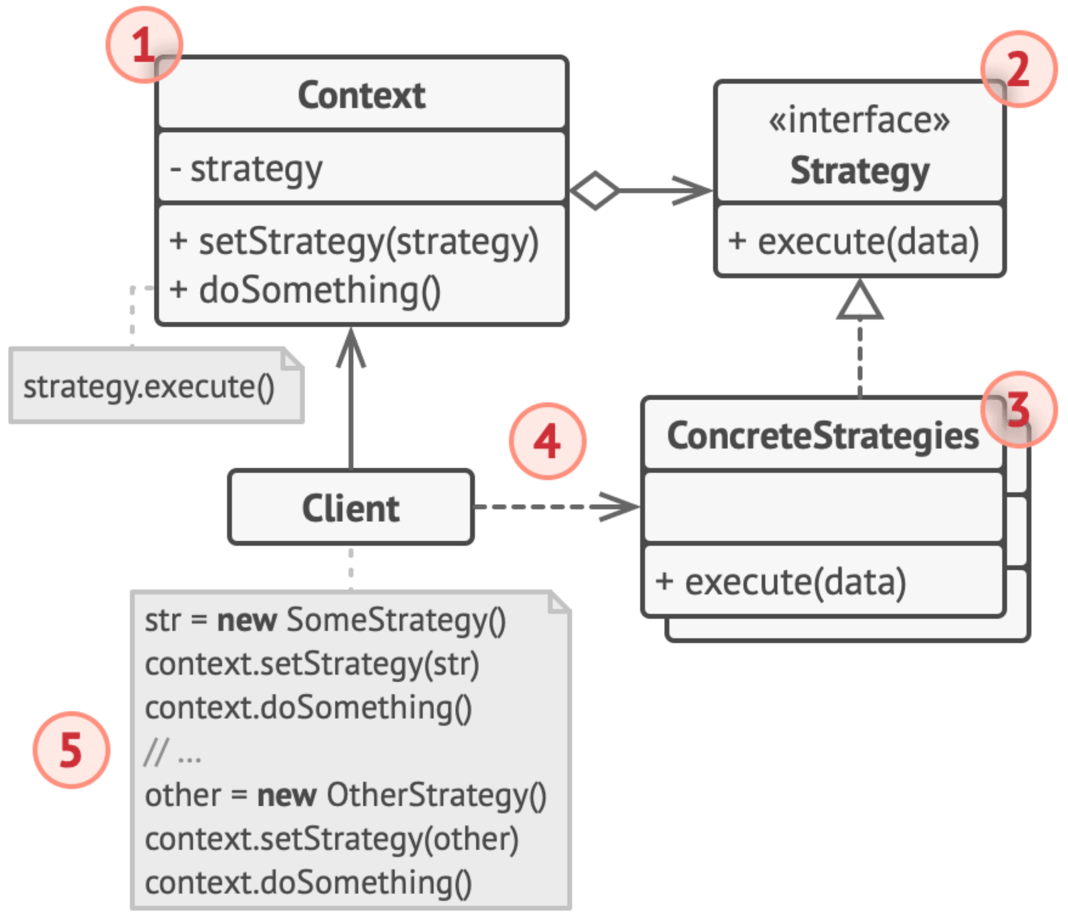
Strategy: Solution

- Extract the algorithms into separate classes: **strategies**
- The original class: **context**
  - Storing a reference to one strategy, and delegating the work to it
- The **client** is responsible for selecting the strategy
- Switching the strategy is possible **at runtime**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/strategy]

Strategy: Structure



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

1. **Context:** storing a reference to one concrete strategy, and communicating with it only via the interface
2. **Strategy:** interface, with a method for executing a strategy
3. **Concrete Strategies:** variations of an algorithm used by the context
4. The context does not know the concrete type of strategy used
5. Client creates the strategy and passes it to the context

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/strategy]

Strategy: Applicability

- To use different variants of an algorithm within an object, and be able to switch from one algorithm to another during runtime
  - Indirectly alter the object's behavior at runtime
- When you have a lot of similar classes that only differ in the way they execute some behaviors
  - Extract the varying behavior into a separate class hierarchy, and combine the original classes into one
- To isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic
  - Isolate the code, internal data, and dependencies of various algorithms from the rest of the code

Strategy: Implementation

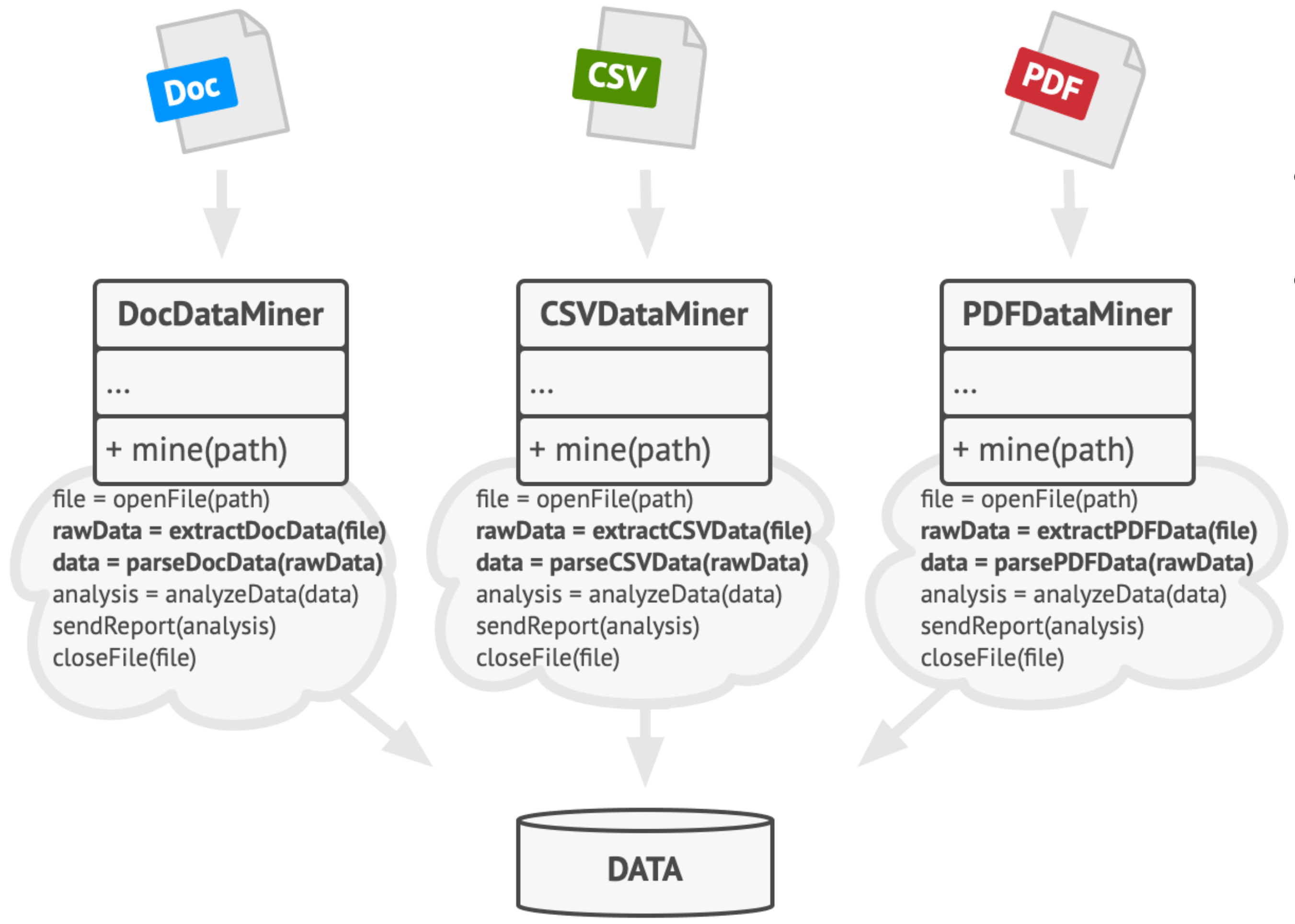
1. In the context class, **identify an algorithm** that is prone to frequent changes
2. Declare the **strategy interface** common to all variants of the algorithm
3. Extract all algorithms into their **own classes**
4. In the context class, add a field for storing a reference to a strategy object, and provide a setter
   – The context may define an interface which lets the strategy access its data
5. Clients of the context must associate it with a suitable strategy

Strategy: Pros and Cons

- **Pros**
  - Swap algorithms used inside an object at runtime
  - Isolate the implementation of an algorithm from the code that uses it
  - Replace inheritance with composition
  - Open/Closed Principle: introducing new strategies without changing the context
- **Cons**
  - If the algorithms rarely change, it is overcomplicating the program
  - Clients must be aware of the differences between strategies
  - Modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions, which can replace strategy objects

Template Method: Problem

- **Example: a data mining application that analyzes documents**



```
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)
```

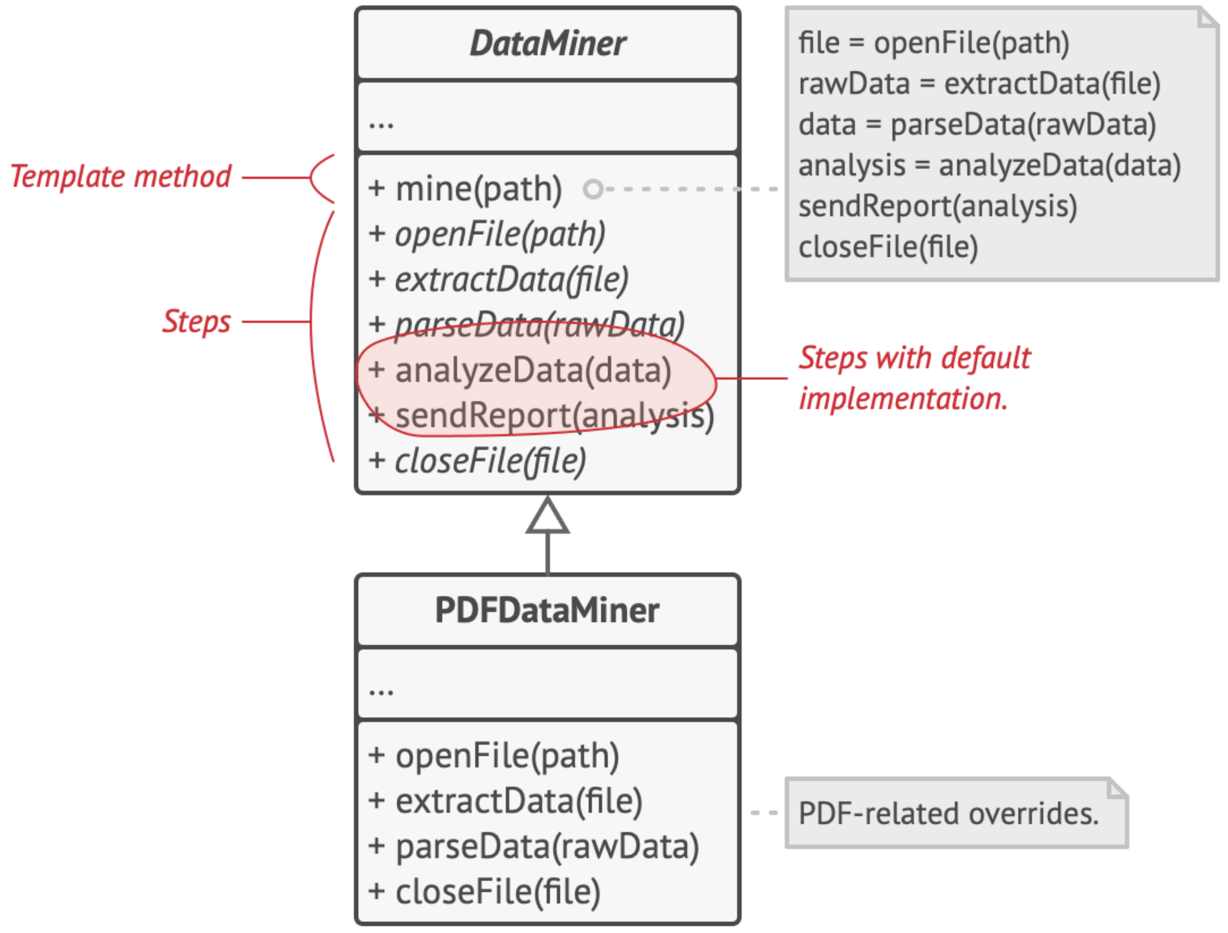
- **Problem 1:** a lot of similar code
- **Problem 2:** the client depends on the classes

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/template-method]

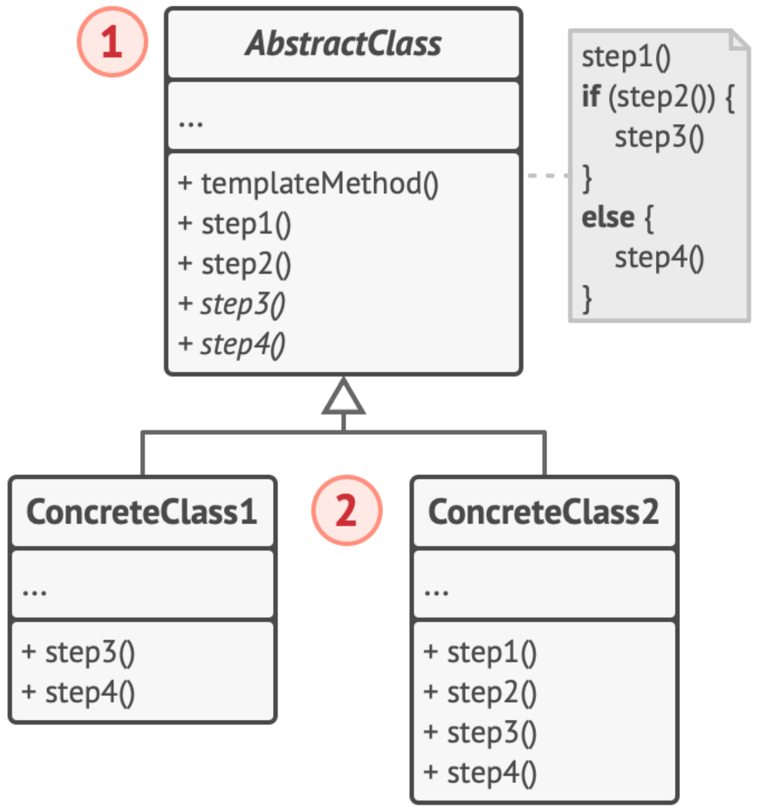
Template Method: Solution

- Break down the algorithm into **steps**
- Turn the steps into methods
- Call the methods in a single **template method**

- Steps may be abstract or have default implementation
- The subclass implements all **abstract steps**, and overrides **optional steps** (if needed)
- Optional: **hook**, with an empty body



Template Method: Structure

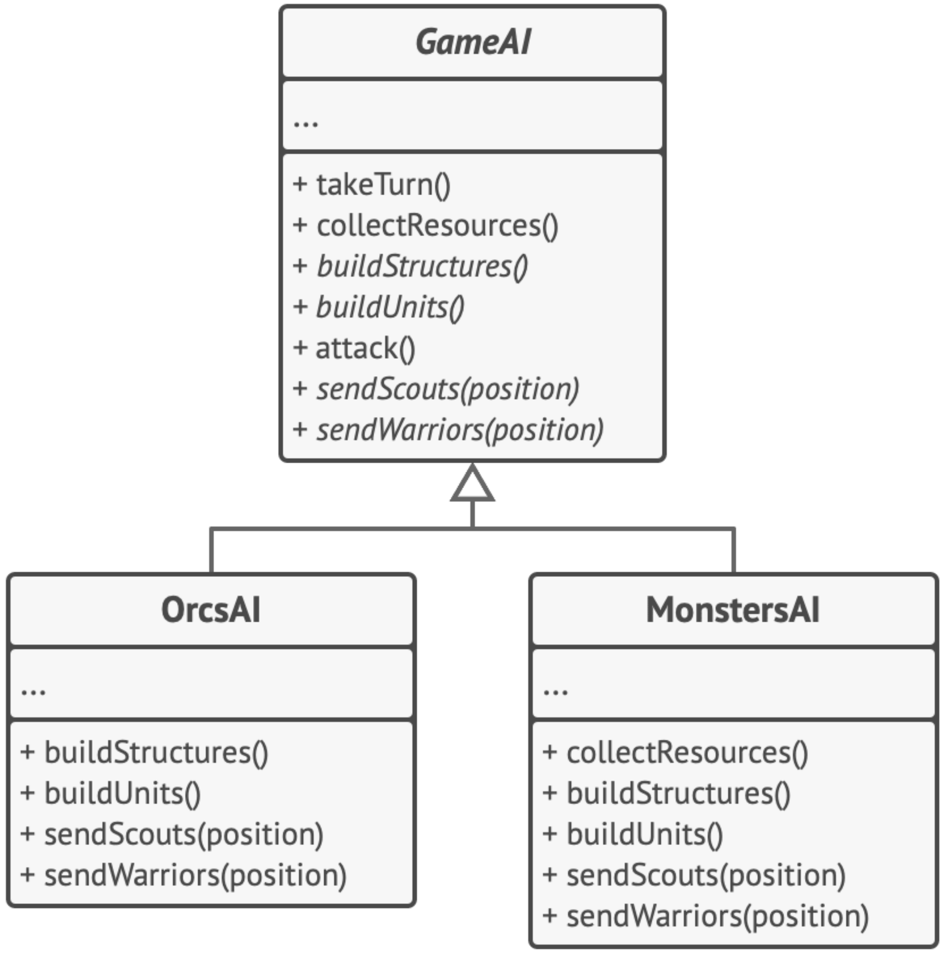


1. **Abstract Class:** template method + abstract steps + optional steps (with default implementation)
2. **Concrete Classes:** overriding the steps, but not the template method

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/template-method]

Template Method: Example

- **Providing a skeleton for various branches of AI in a strategy video game**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/template-method]

Template Method: Applicability

- To let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure
  - 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
- When you have several classes that contain almost identical algorithms with some minor differences
  - Turn such an algorithm into a template method, and pull up the steps with similar implementations into a superclass, eliminating code duplication

Template Method: Implementation

1. Analyze the target algorithm, break it into steps, and **consider which steps are common to all and which will be unique**
2. Create the **abstract base class** and declare the **template method** as well as a set of **abstract methods**
   – Outline the algorithm’s structure in the template method
   – Some steps might benefit from having a **default** implementation
   – Consider making the template method **final** to prevent subclasses from overriding it
3. Optional: add **hooks** between the crucial steps of the algorithm
4. For each variation of the algorithm, create a new concrete subclass

Template Method: Pros and Cons

- **Pros**
  - Let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm
  - Pull the duplicate code into a superclass
- **Cons**
  - Some clients may be limited by the provided skeleton of an algorithm
  - 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

Combinations and Comparisons

- **Bridge, State, Strategy, and Adapter**
  - All based on composition (delegating work to others), but solving different problems
- **Command and Strategy**
  - Both can be used to parameterize an object with actions
  - Command: convert any operation into an object, and the parameters become fields
  - Strategy: describe different ways of doing the same thing
- **Decorator and Strategy**
  - Decorator changes the skin of an object, while Strategy changes the guts
- **State and Strategy**
  - State can be considered as an extension to Strategy
  - Strategy objects are completely independent and unaware of each other
- **Template Method and Strategy**
  - Template Method: based on inheritance, working on the class level, static
  - Strategy: based on composition, working on the object level, switchable at runtime
- **Factory Method and Template Method**
  - Factory Method is a specialization of Template Method
  - A Factory Method may serve as a step in a large Template Method