Design Patterns IV

Ergude Bao

Beijing Jiaotong University

Contents

- Structural Patterns II

Structural Patterns II

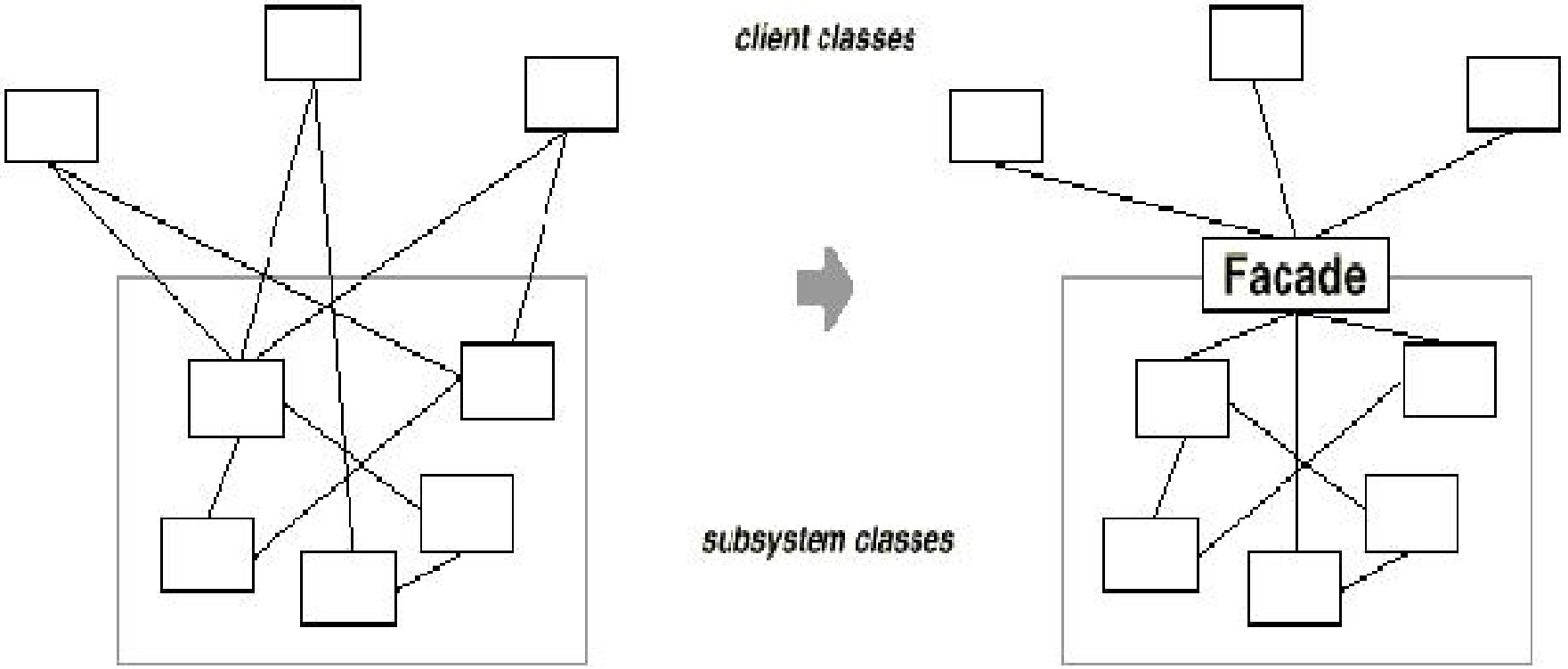
- Facade pattern
- Flyweight pattern
- Proxy Pattern

Facade Pattern

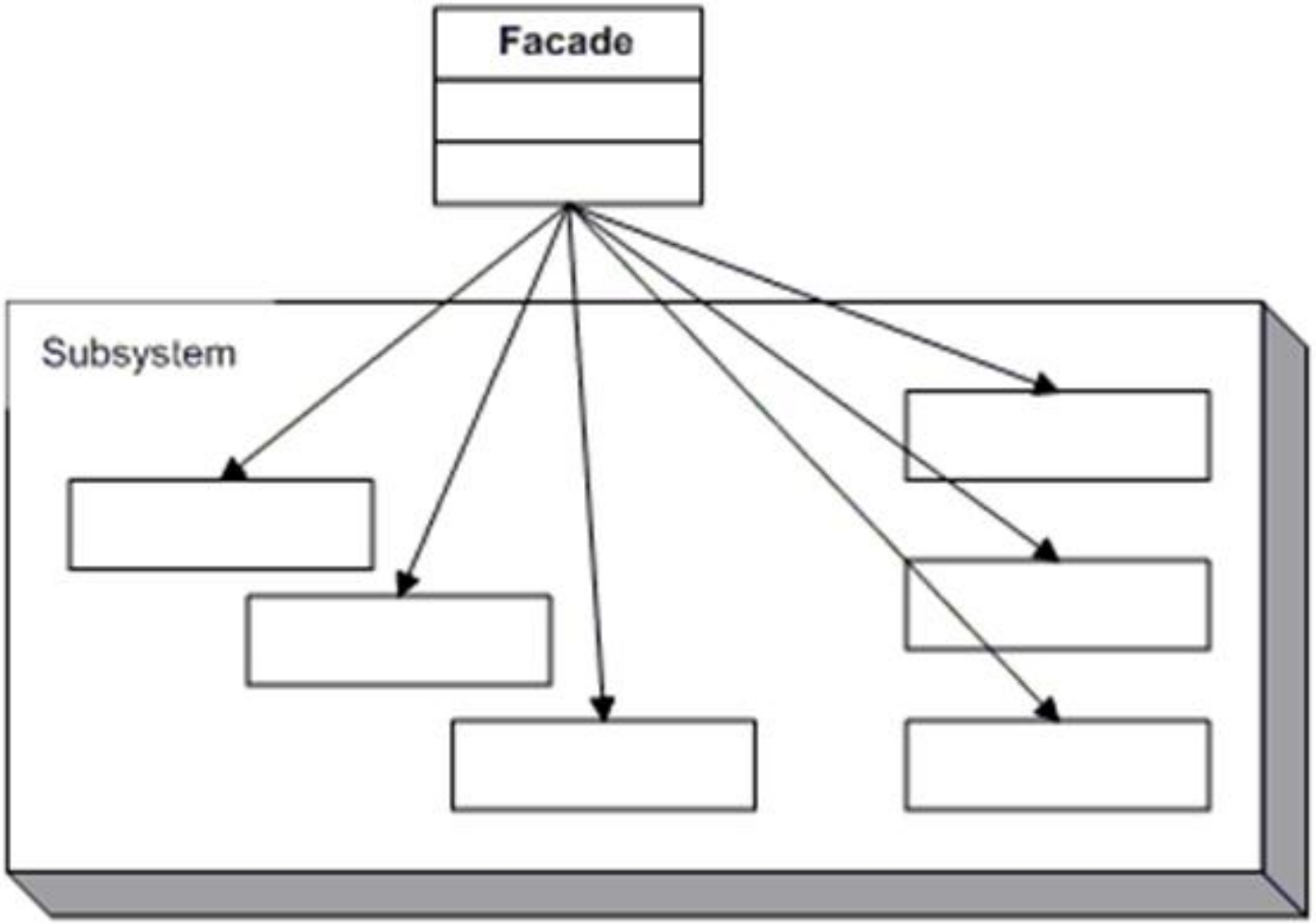
Facade Pattern

- Motivation
  - We want to reduce complexity of the many class interfaces in a subsystem for the usage by client
- Solution
  - Provide a unified interface to a set of class interfaces in a subsystem

Facade Pattern



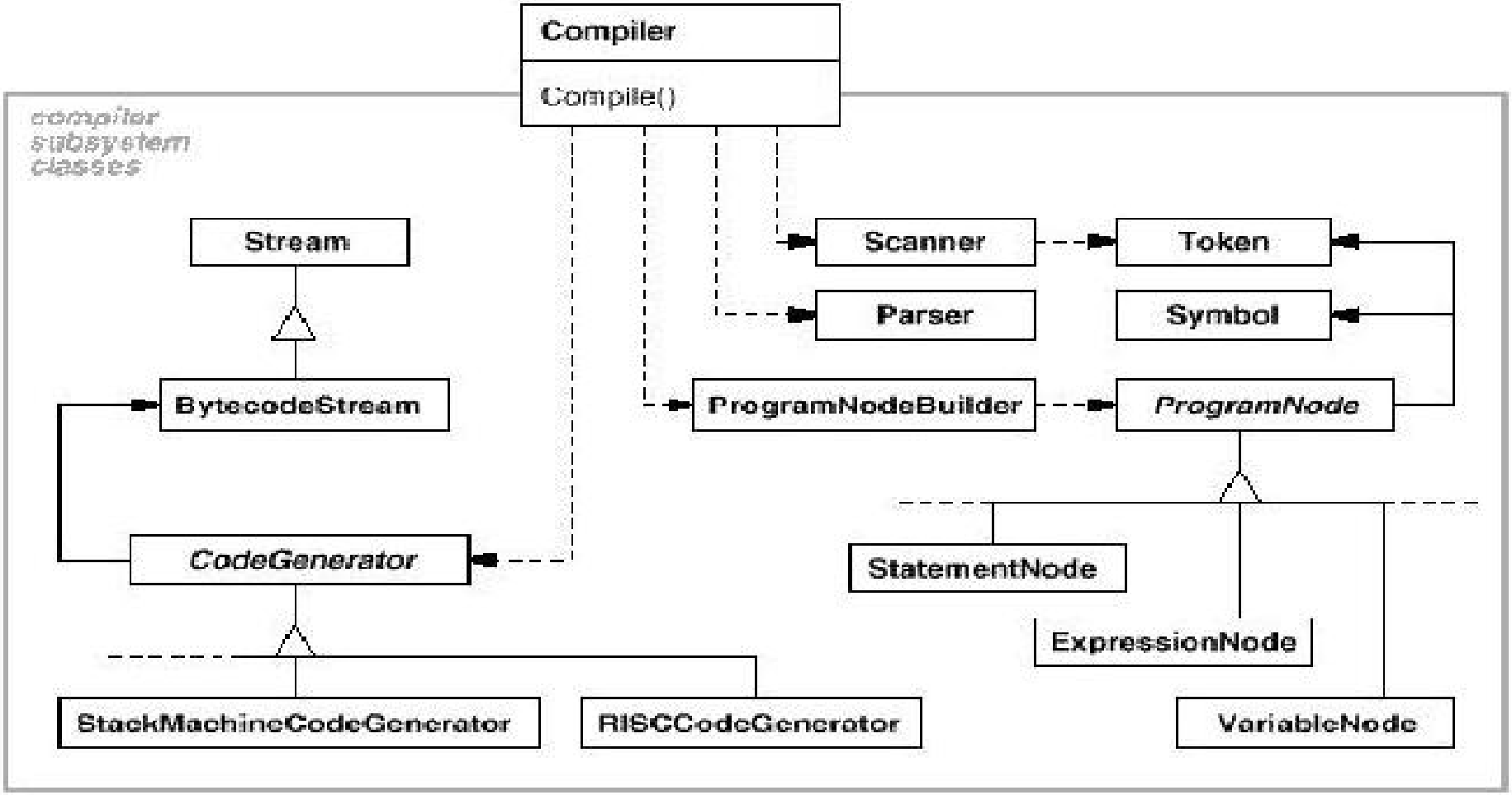
Facade Pattern



Participants

- Facade
  - Knows which classes are responsible for a request
  - Delegates the request to the classes
- Subsystem
  - Implements its own functionalities
  - Handles work assigned by the Facade
  - Has no knowledge of the Facade and keeps no reference to it

Example


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

Advantages

- Reduces complexity to use the subsystem
- Broadens the scale of clients to use the subsystem
- Reduces compilation dependencies
- Not add any additional functionality
- Not prevent client from accessing its classes

Flyweight Pattern

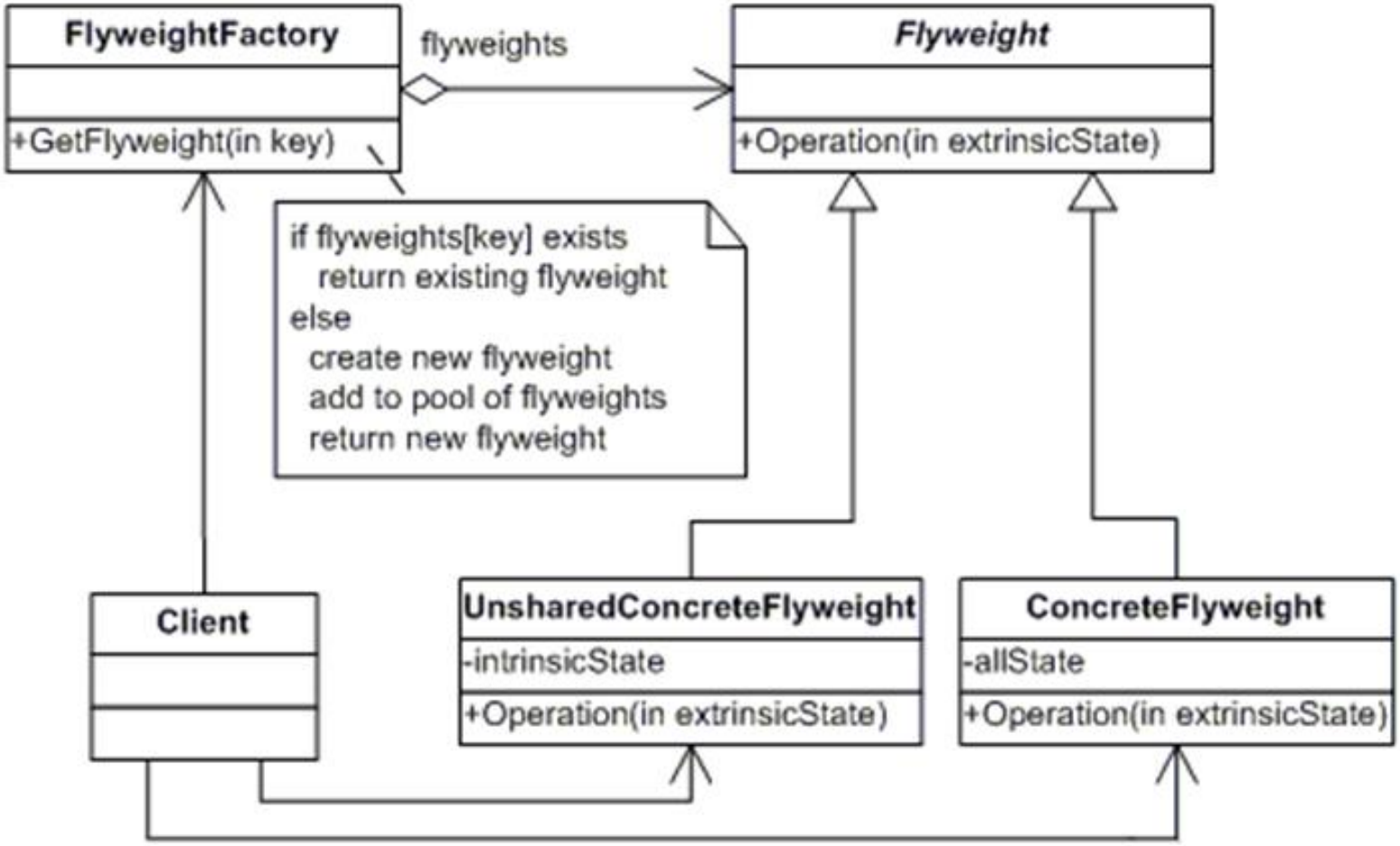
Flyweight Pattern

- Motivation
  - We want to share created objects as many as possible rather than newly create them
- Solution
  - Share created objects rather than newly create them
  - Simplify objects with only intrinsic states and add extrinsic states for specific uses

Intrinsic and Extrinsic States

- Intrinsic state is stored and shared in the Flyweight object
- Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked

Flyweight Pattern



Participants

- Flyweight
  - Declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight
  - Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context
- UnsharedConcreteFlyweight
  - Not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure

Flyweight Pattern

- FlyweightFactory
  - Creates and manages flyweight objects. Ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects assets an existing instance or creates one, if none exists
- Client
  - Maintains a reference to flyweight(s). Computes or stores the extrinsic state of flyweight(s)

Flyweight Pattern vs. Prototype Pattern

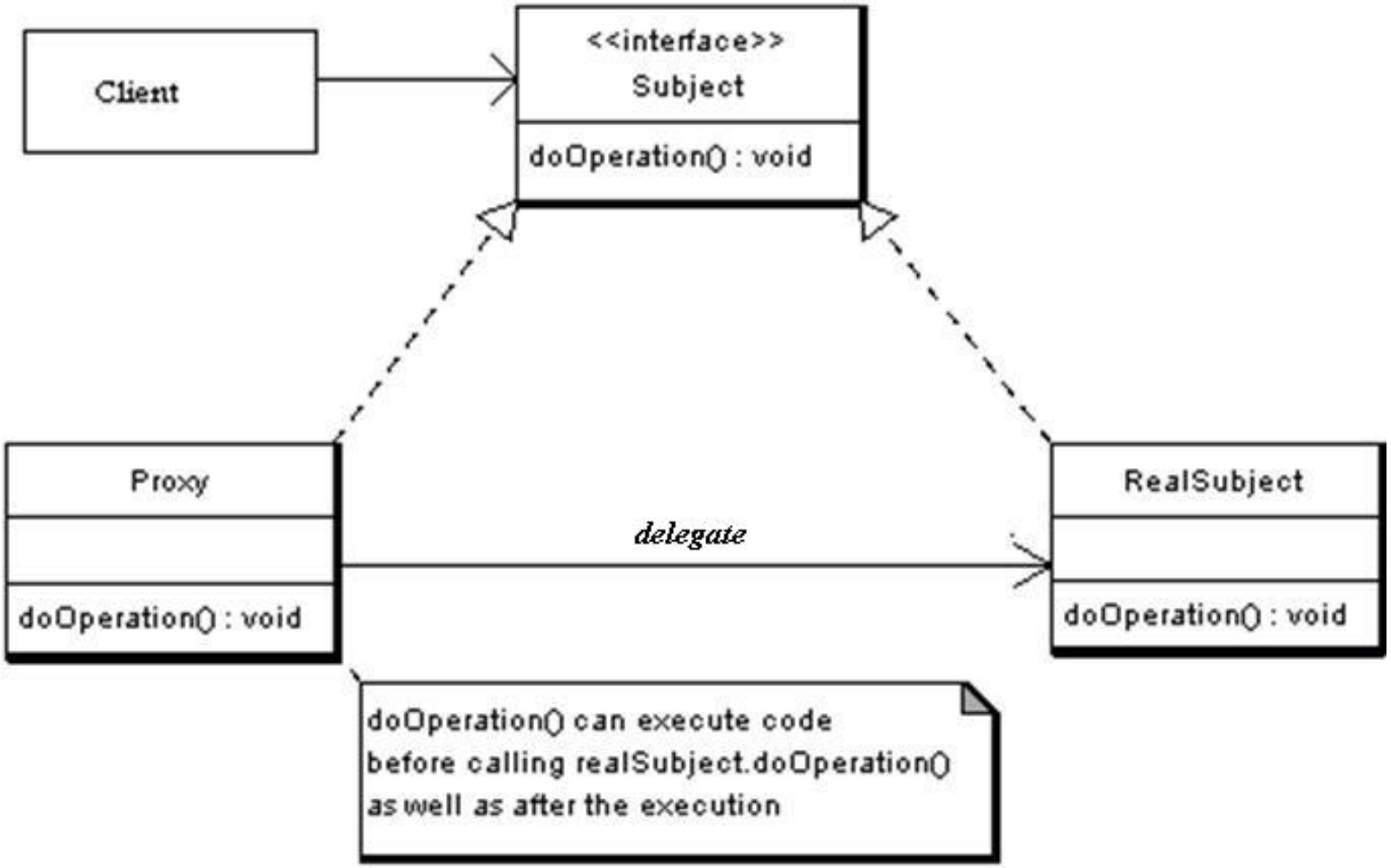
- Similarity: targeting at reducing time taken by creating objects
- Difference
  - Flyweight pattern either uses existing objects or creates new objects, while prototype pattern clones existing objects
  - Flyweight pattern divides each object into two parts with intrinsic and extrinsic states

Proxy Pattern

Proxy Pattern

- Motivation
  - We want to represent a complex object by a simpler one
  - We want to validate the access permission for an object
- Solution
  - Provide a proxy object for the target object

Proxy Pattern



Participants

- Subject
  - Defines the interface for RealSubject representing its services
- RealSubject
  - Implements the Subject interface
- Proxy
  - Maintains a reference to access the RealSubject
  - Implements the Subject interface as the RealSubject
  - Controls access to the RealSubject and may be responsible for its creation and deletion
  - May have other responsibilities depending on the kind of proxy

Proxy vs. Object Adapter

- Similarity: both use a new class to create objects of a target class and make changes
- Difference: proxy maintains the interface of the target class, while adapter changes the interface and adapts it to a different interface for the new class

Proxy vs. Decorator

- Similarity: both use a new class to create objects of a target class following the same interface
- Difference: proxy makes changes in itself, while decorator makes changes in subclasses