SWE 621

SPRING 2021

DESIGN PATTERNS

LOGISTICS

- HW3 due today
- HW4 due in two weeks

IN CLASS EXERCISE

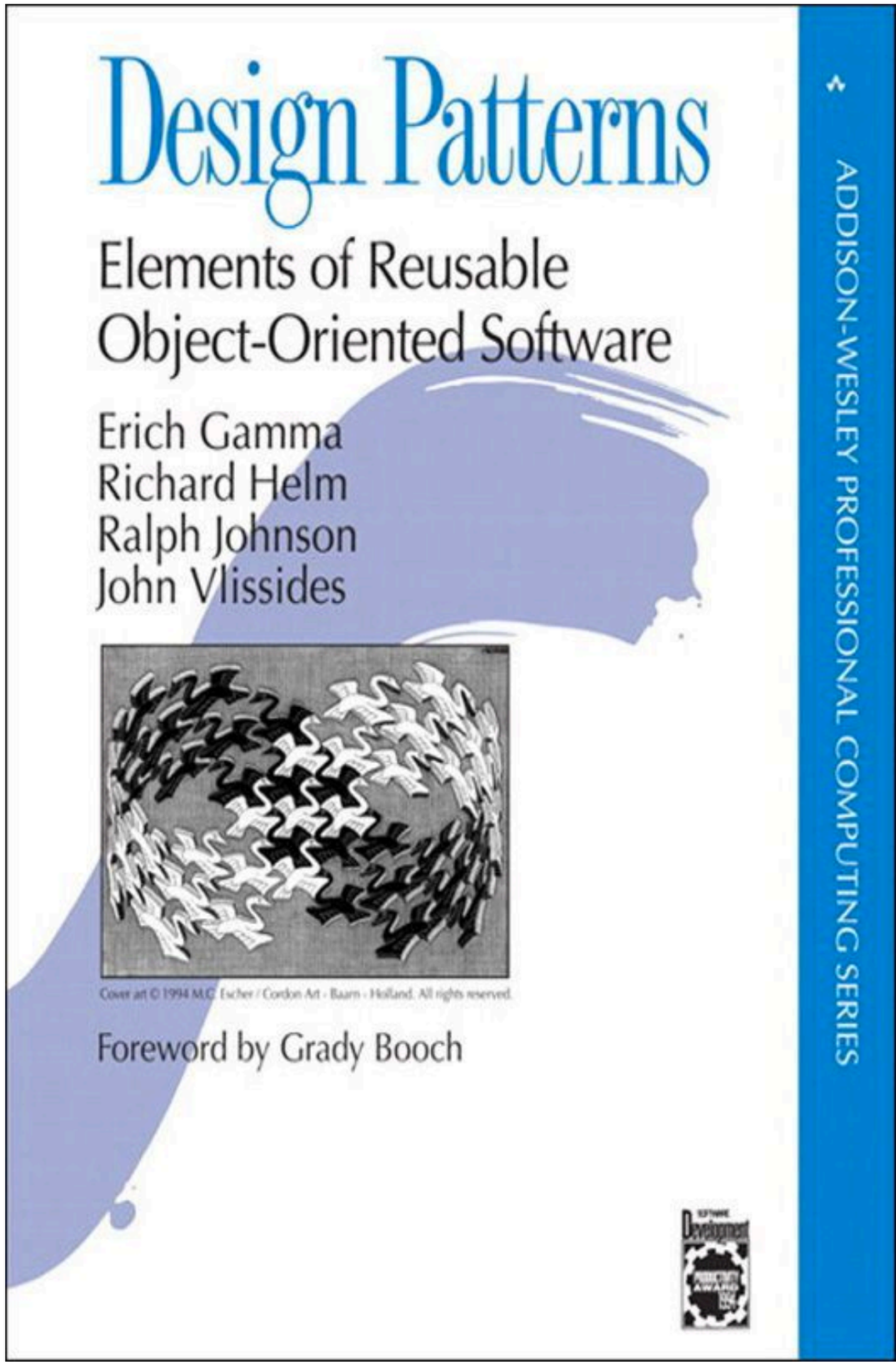
- What's a software design problem you've solved from an idea you learned from someone else?

DESIGN PATTERN

- Solution to a problem in a context
  - Rather than solving problems from scratch, borrow existing solution to a common design problems
  - Arrangement of elements that achieves particular quality attribute, often (but not always) extensibility
- Similar to architectural style or abstraction
  - Description of elements and their properties that is not tied to a specific implementation
  - Offers a name for a concept that makes concept easy to refer to
- But also different
  - Unlike architectural styles, implications are localized to a few elements
  - Design, not architectural; does NOT constraint how most elements in the system interact

DESIGN PATTERNS

- Idea popularized by "Gang of Four" (GOF) in the 1990s with their book Design Patterns
  - Sometimes abbreviated as "GOF patterns"
  - Today's first reading was immediate precursor of book
- Helped explain to developers how to take advantage of indirection facilities in OO to build systems that were more modular and maintainable by introducing indirection
- But… idea of design patterns is more general than GOF patterns
- Popular book that inspired **many** follow ons (e.g., Node.js design patterns)



What is the pattern's name and classification? The name should convey the pattern's essence succinctly. A good name is vital, as it will become part of the design vocabulary.

Intent

What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Motivation

A scenario in which the pattern is applicable, the particular design problem or issue the pattern addresses, and the class and object structures that address this issue. This information will help the reader understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can one recognize these situations?

Participants

Describe the classes and/or objects participating in the design pattern and their responsibilities using CRC conventions [5].

Collaborations

Describe how the participants collaborate to carry out their responsibilities.

Diagram

A graphical representation of the pattern using a notation based on the Object Modeling Technique (OMT) [25], to which we have added method pseudo-code.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What does the design pattern objectify? What aspect of system structure does it allow to be varied independently?

Implementation

What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there language-specific issues?

Examples

This section presents examples from real systems. We try to include at least two examples from different domains.

See Also

What design patterns have closely related intent? What are the important differences? With which other patterns should this one be used?

BENEFITS OF DESIGN PATTERNS

- Patterns enable reuse of design solutions
  - Capture knowledge of expert developers learned through trial and error
- Patterns improve communication, by offering a name and higher-level concept for something that commonly recurs
  - Rather than trying to describe how set of classes should be interacting, can simply reference concept

DESIGN FOR CHANGE

- Many GOF patterns designed make specific types of change easier
- How do you take some decision, hide it in a class, and enable that decision to change with minimal impact on rest of system?
- Enables many types of decisions to vary through **extension**, where **alternative** implementations can be written as planned extensions to system

EXAMPLES OF DECISIONS

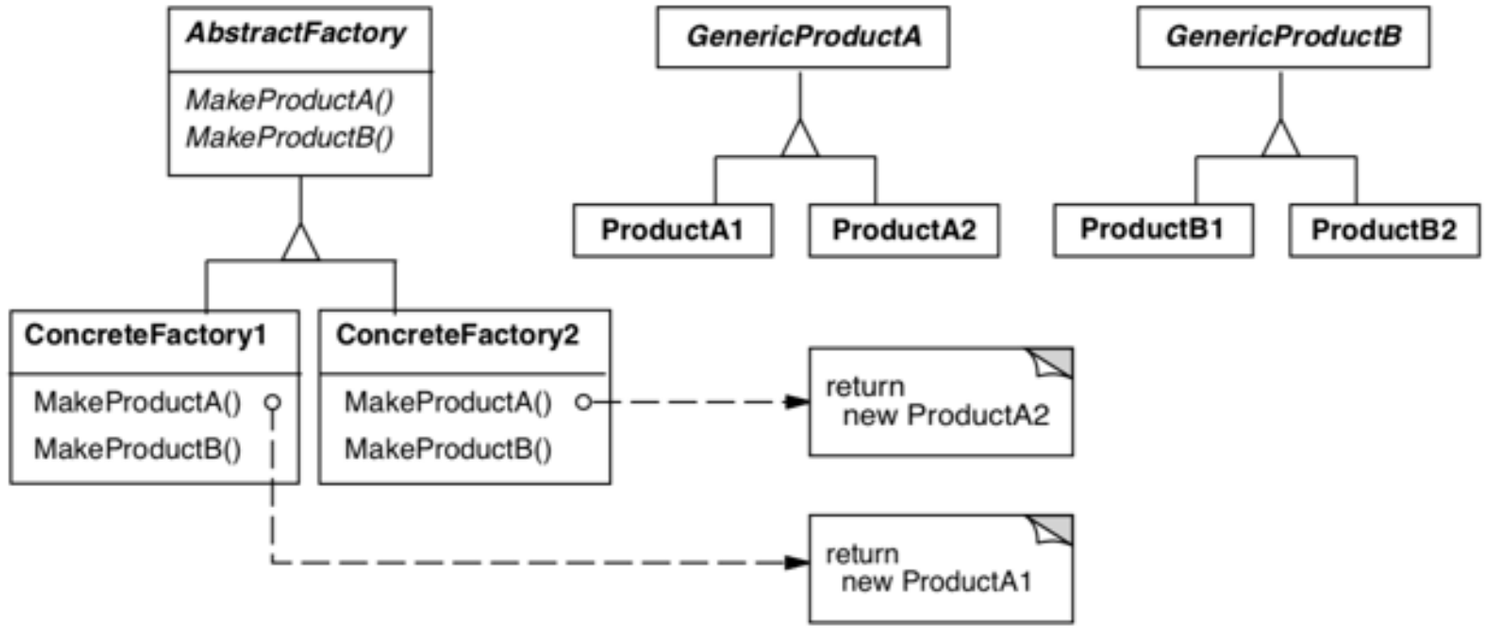
- The identity of a class
  - Want to commit only to an interface of class, not an implementation
- Specific operations
  - Want to commit to interface of an operation, not an implementation
- Specific algorithms
  - Want to enable alternative algorithms
- Data representation
  - Reduce client dependencies on how data is represented and stored

FORMS OF GOF PATTERNS

- Creational: how objects are instantiated
  - How can details about the type of element being created be hidden from clients?
- Structural: how objects are composed
  - How can objects be connected in way that reduce dependencies?
- Behavioral: how objects behave
  - How can objects encapsulate behaviors that may vary at runtime?

CREATIONAL PATTERNS

ABSTRACT FACTORY



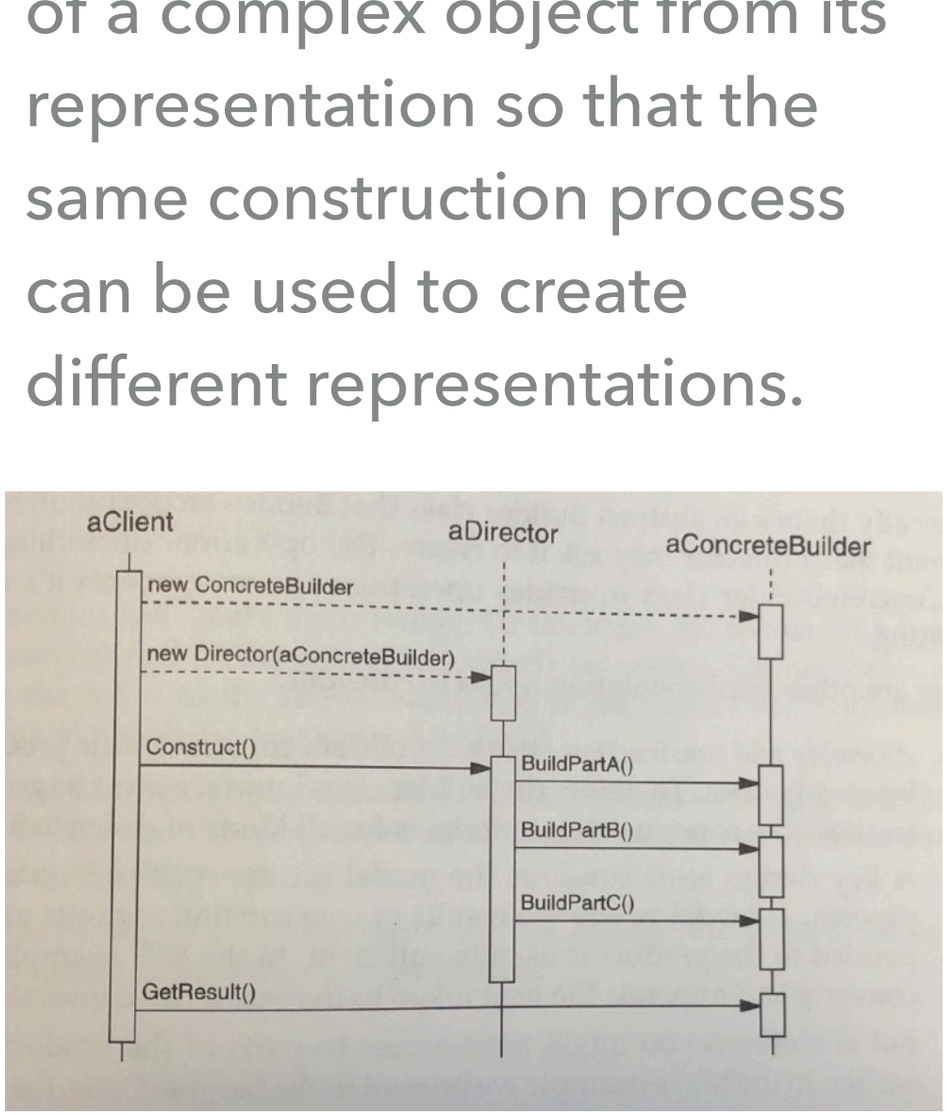
- Provide an interface for creating families of related objects without specifying their concrete classes
- Client knows they get a GenericProductA or B without knowing the particular implementation, which provider may vary without breaking clients

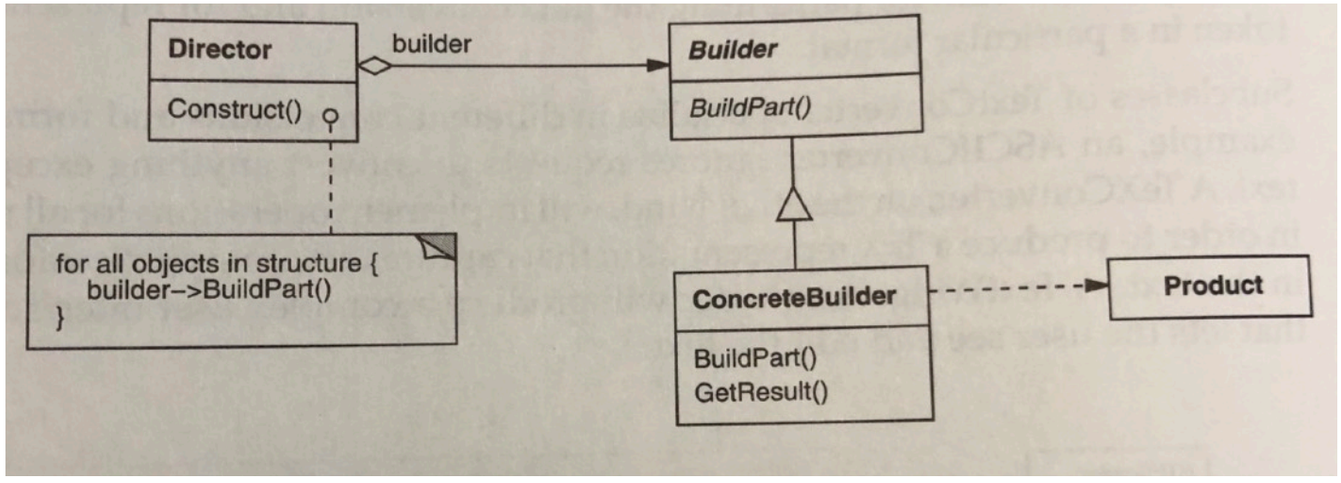
Participants

- **AbstractFactory**
  - declares a generic interface for operations that create generic product objects.
- **ConcreteFactory**
  - defines the operations that create specific product objects.
- **GenericProduct**
  - declares a generic interface for product objects.
- **SpecificProduct**
  - defines a product object created by the corresponding concrete factory.
  - all product classes must conform to the generic product interface.

BUILDER

- Separates the construction of a complex object from its representation so that the same construction process can be used to create different representations.

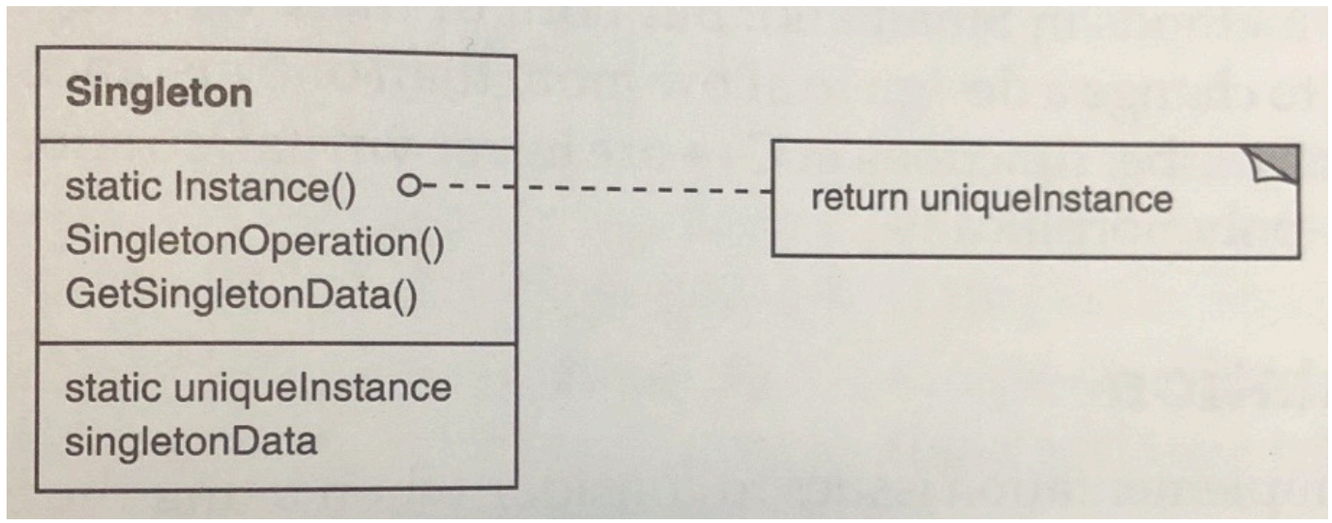




Participants

- Builder
  - Specifies an abstract interface for creating parts of a product object
- ConcreteBuilder
  - constructs and assemble parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
- Director
  - constructs an object using the builder interface
- Product
  - represents the complex object under construction

SINGLETON



- **Ensure** a class only has **one** instance, and provide a global point of access to it.

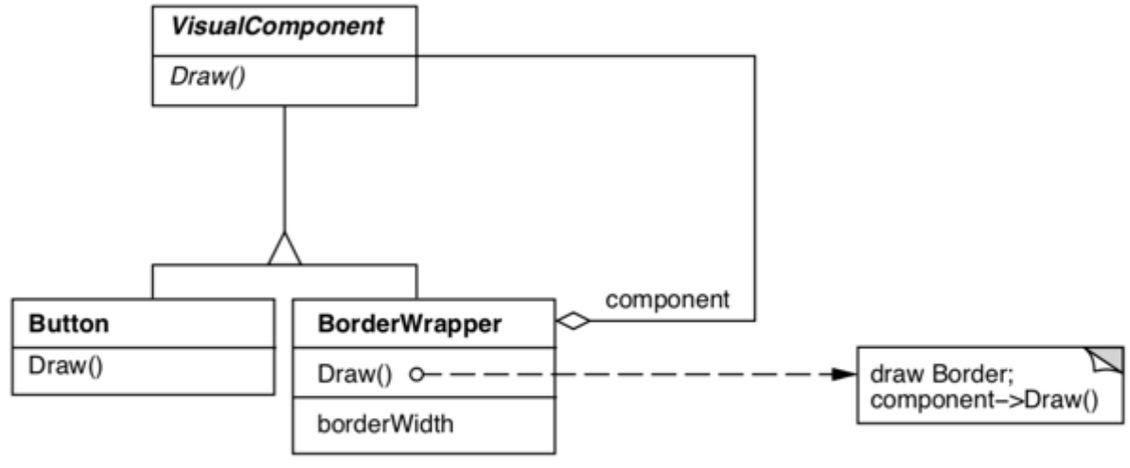
Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a static operation defined on the class rather than the instance
  - may be responsible for creating its own unique instance

STRUCTURAL PATTERNS

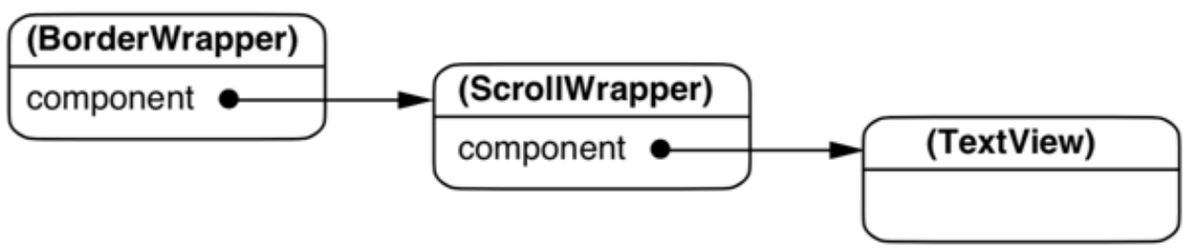
WRAPPER

- Attach additional properties or services to an object **without** having to subclass object
- Implements common interface (VisualComponent) rather than subclassing implementation (Button) which may not be hidden.
- Enables nesting wrappers, easily adding and removing at runtime



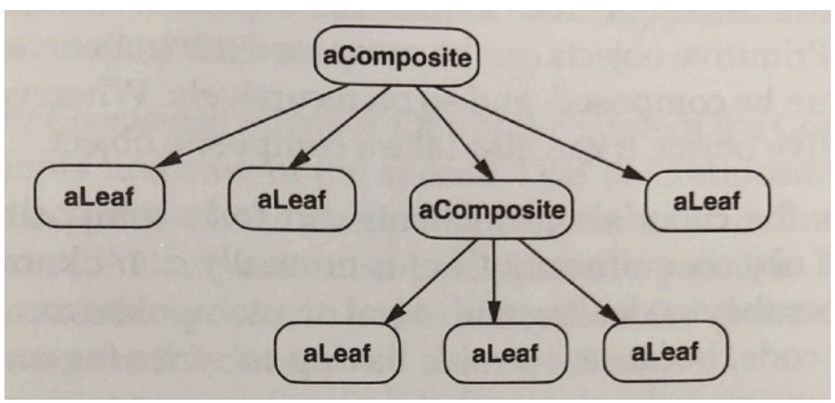
Participants

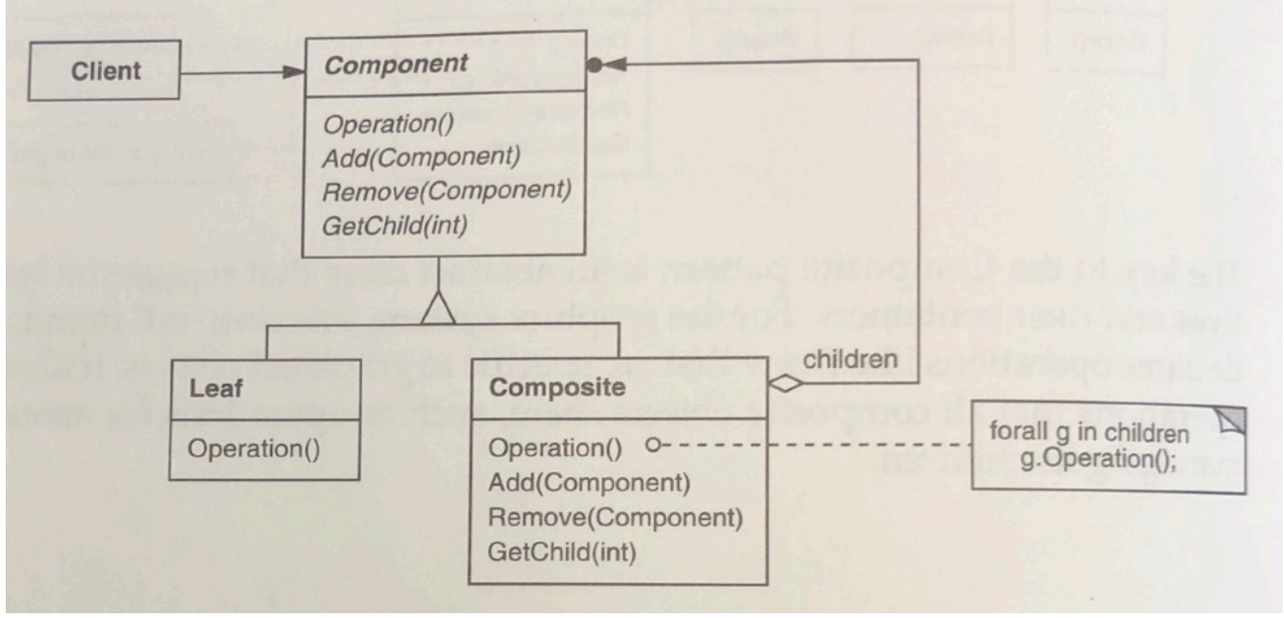
- **Component**
  - the object to which additional properties or behaviors are attached.
- **Wrapper**
  - encapsulates and enhances its Component. It defines an interface that conforms to its Component's.
  - Wrapper maintains a reference to its Component.



COMPOSITE

- Compose objects into tree structures to represent part-whole hierarchies.
- Lets clients treat individual objects and compositions uniformly



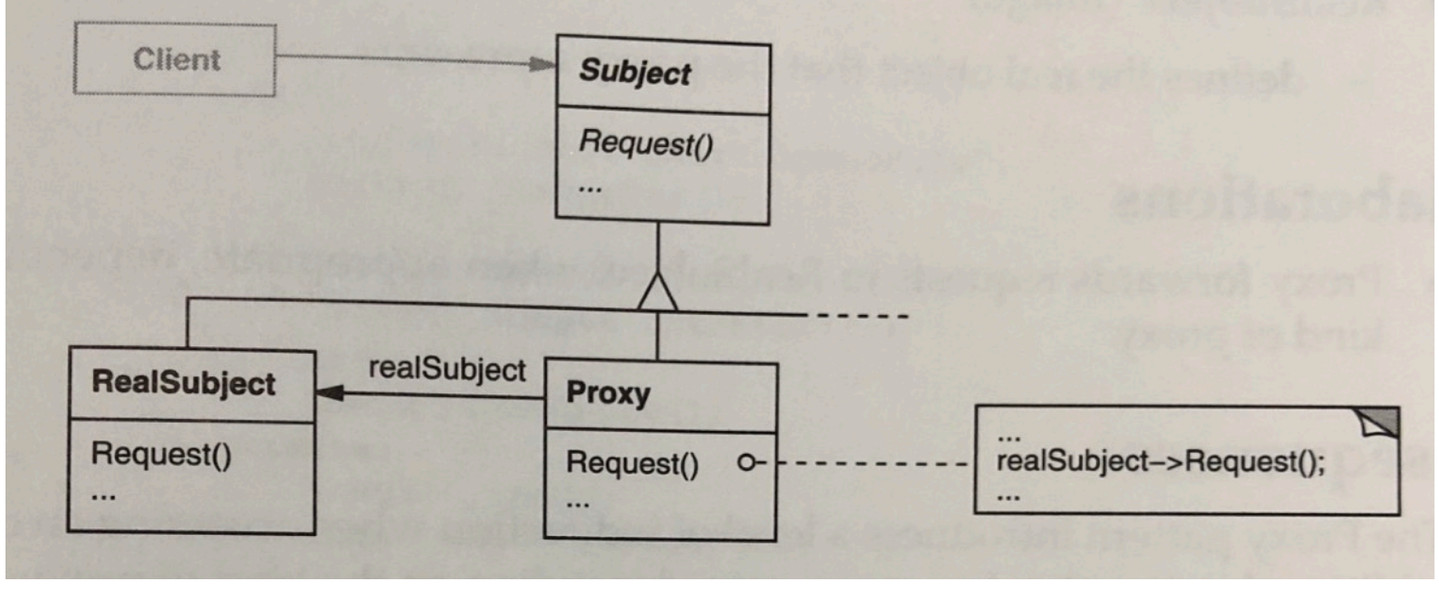


Participants

- Component
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes
  - declares interface for accessing and managing children
- Leaf (no children)
  - defines behavior for primitive objects in the composition
- Composite
  - defines behavior for components having children
  - stores children
  - implements child-related operations
- Client
  - manipulates objects in composition through Component interface

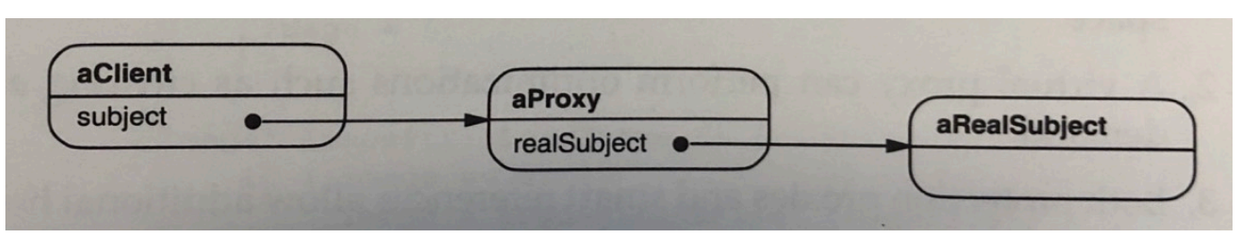
PROXY

- Provide a surrogate or placeholder for an object to control access to it
- Can be used to
  - avoid creating expensive objects unless really needed
  - check access rights
  - garbage collection



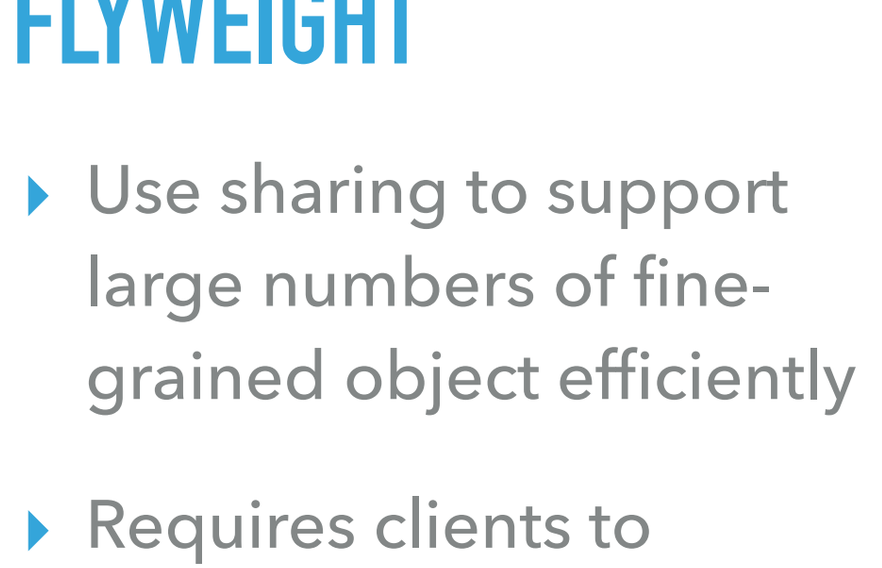
Participants

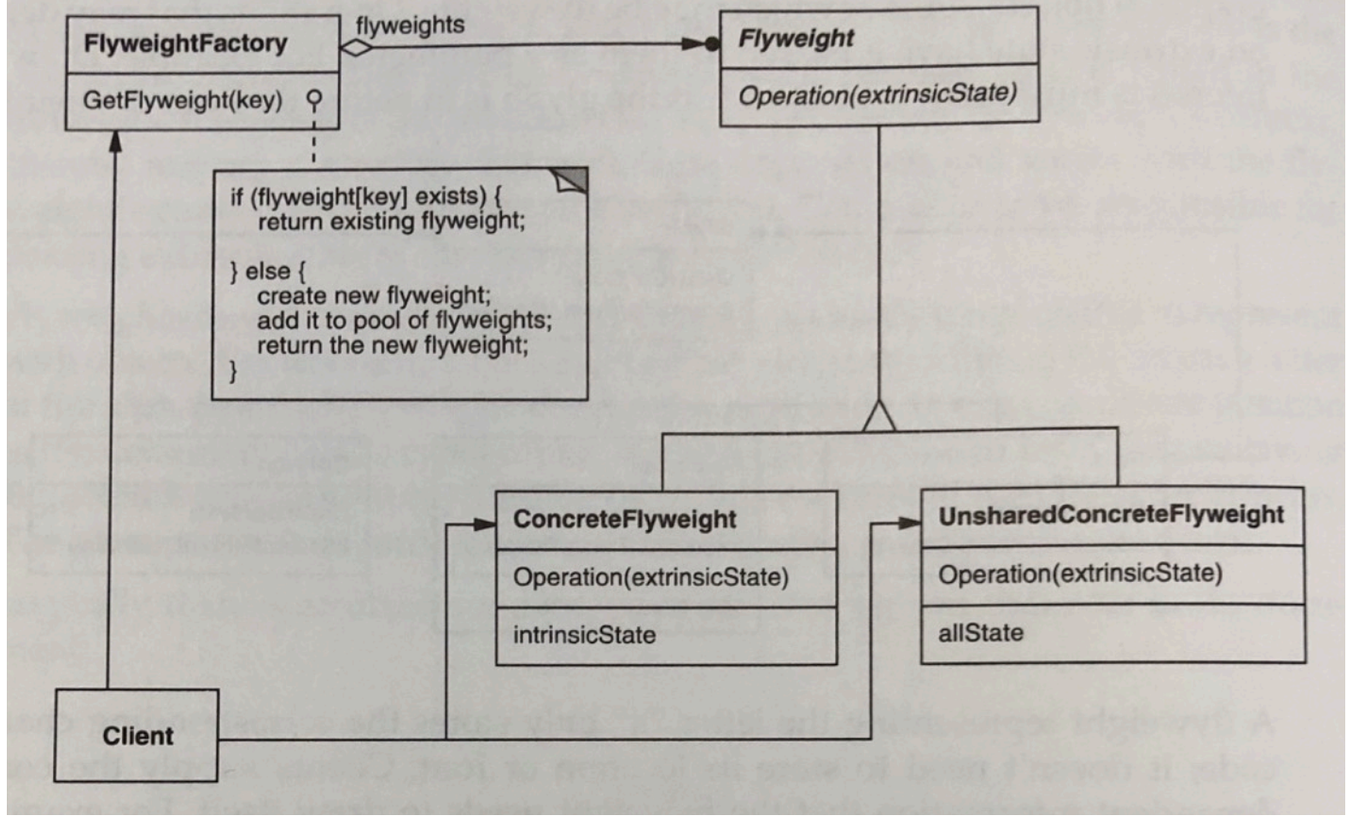
- Proxy
  - maintains reference that lets proxy access real subject
  - controls access to real subject, which may include creating and destroying it
- Subject
  - defines common interface
- RealSubject
  - defines the real object that proxy represents



FLYWEIGHT

- Use sharing to support large numbers of fine-grained object efficiently
- Requires clients to interact with objects only by value rather than identity

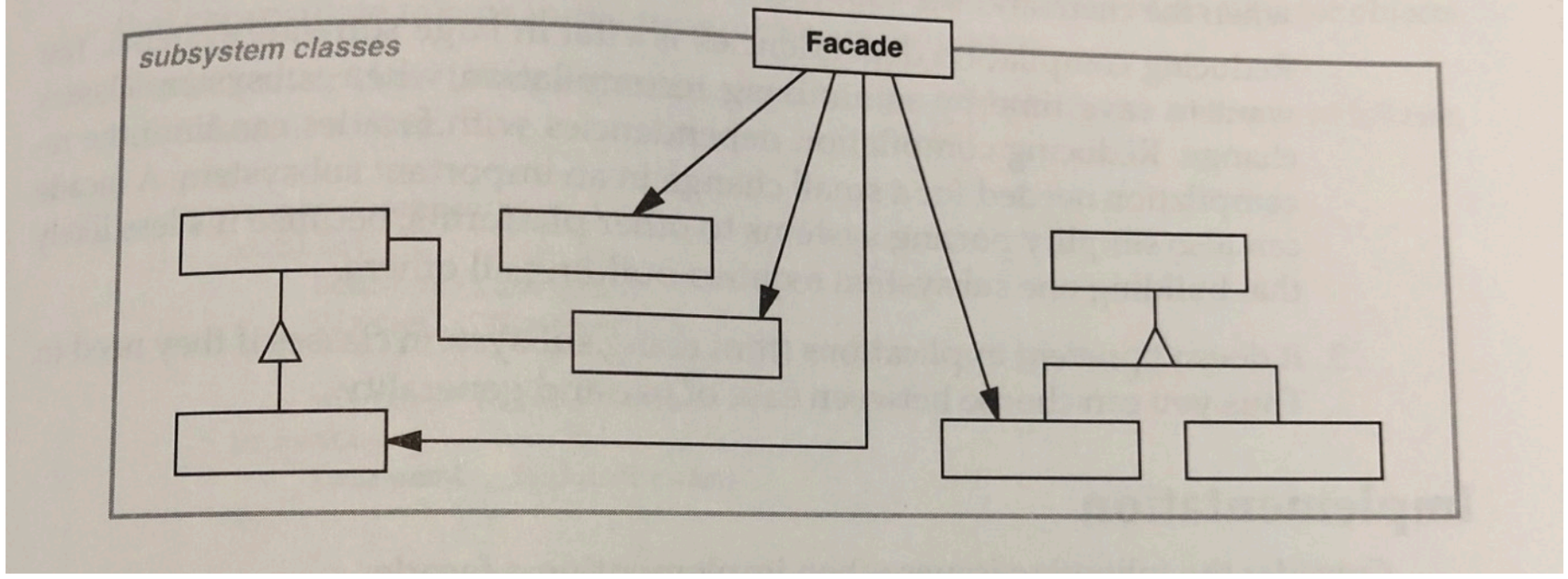




Participants

- Flyweight (common interface)
- ConcreteFlyweight
  - Implements interface, stores state
  - MUST be shareable
- FlyweightFactory
  - creates and mangoes flyweight objects
  - lazily creates instances, as necessary
- Client (uses flyweights)

FACADE



- Provide a higher-level, unified interface to a set of interfaces in a subsystem

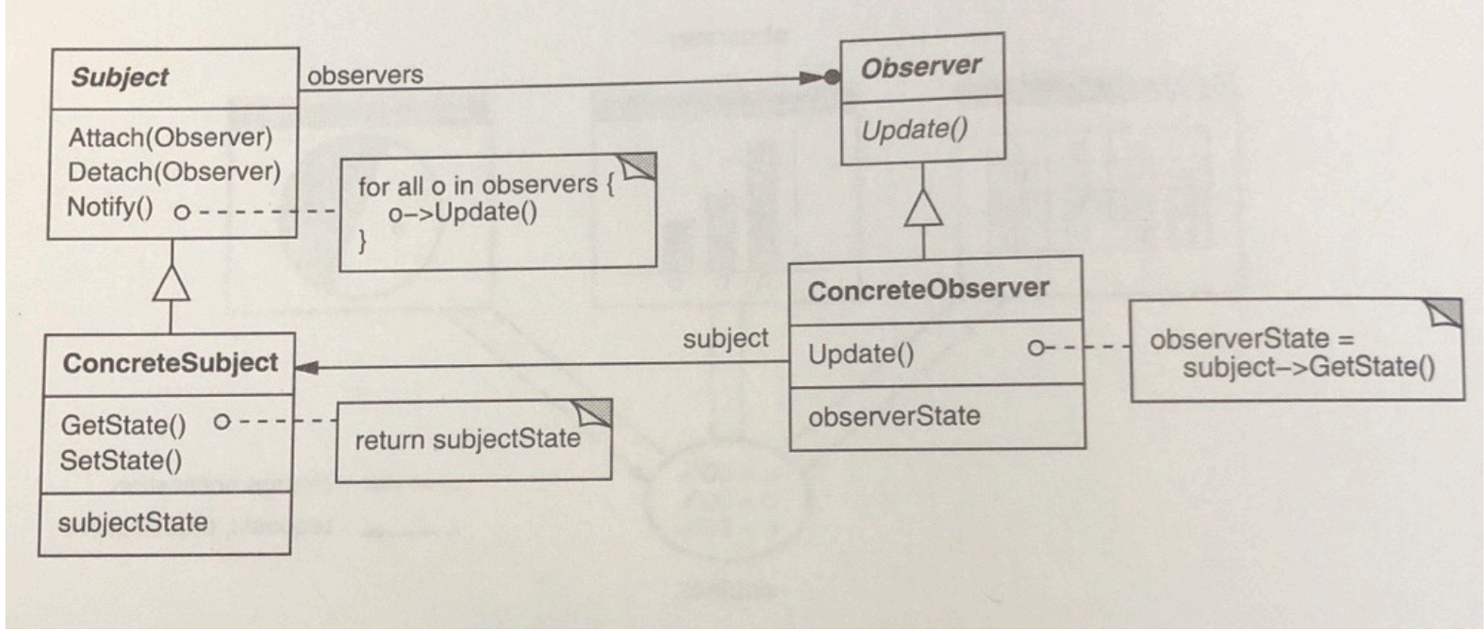
Participants

- Facade
  - knows which subsystem classes are responsible for a request
  - delegates client request to appropriate subsystem objects
- subsystem classes
  - implement subsystem functionality
  - handle work assigned by the Facade object
  - have **no** references to facade

BEHAVIORAL PATTERNS

OBSERVER

- Defines a one way one-to-many dependency, so that one object changes state all dependencies are notified automatically
- Lets subject emit events to observers without depending on observers



Participants

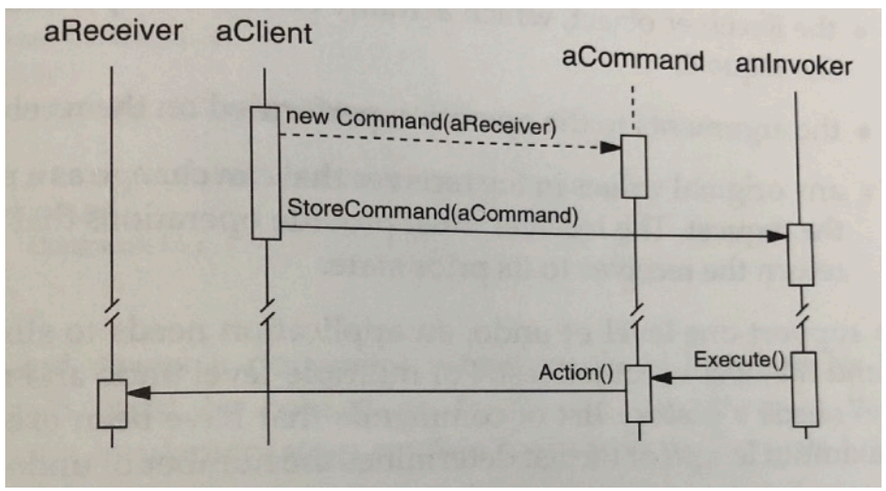
- Subject
  - stores and manages its observers, which may be any number
- Observer
  - defines an interface for updates
- ConcreteSubject
  - stores state of interest to Observers
  - sends notification to observers when state changes
- ConcreteObserver
  - maintains reference to ConcreteSubject object
  - stores state that is synchronized with subject

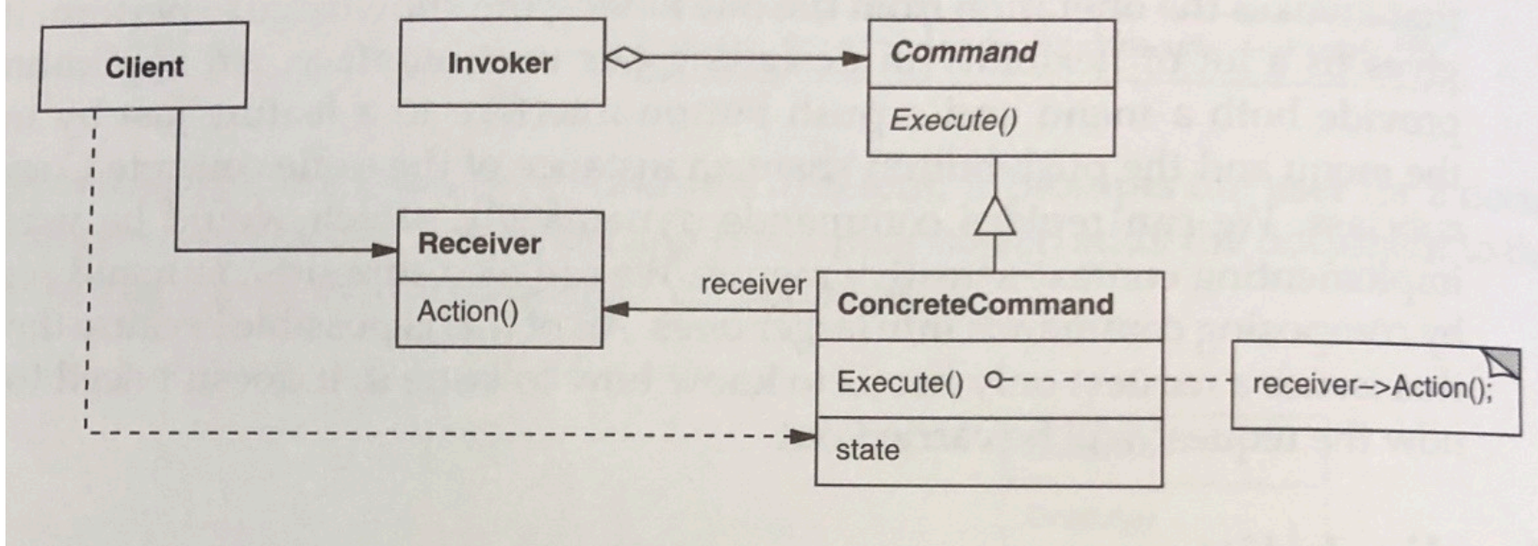
IN CLASS ACTIVITY: IMPLEMENT OBSERVER

- Pick an OO language (e.g., Java, C++, Python)
- Write an implementation of Observer
- Make sure your implementation lets subject emit events to observers without subjects depending on observers (e.g., can add and remove new types of observers without changing subject)

COMMAND

- Encapsulate a request as an object
- Enables parameterizing clients with requests, queuing and logging requests, undoable operations



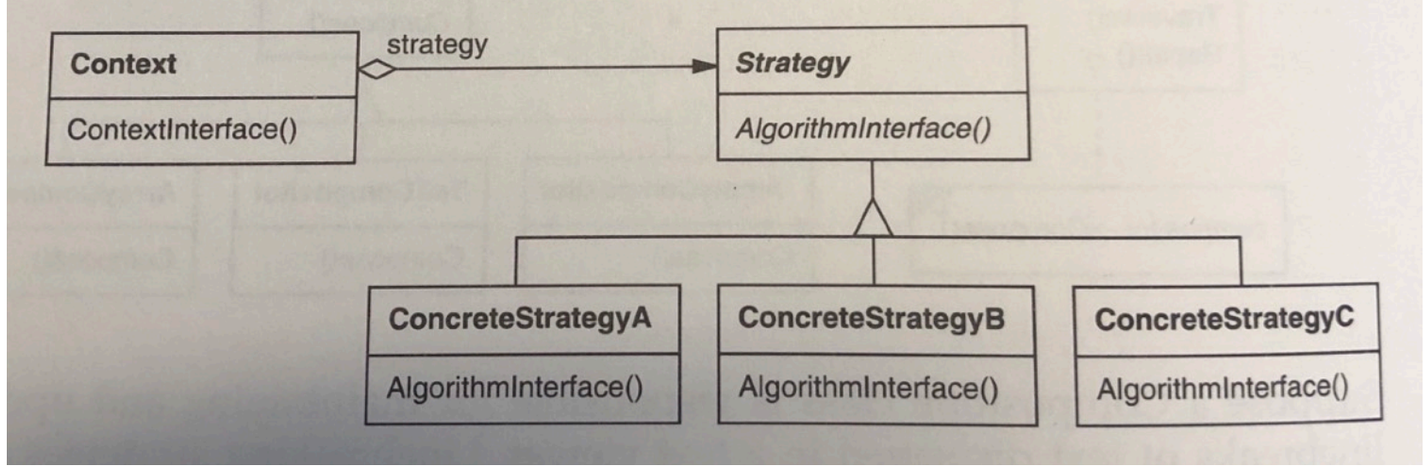


Participants

- Command
  - declares interface for executing an operation
- ConcreteCommand
  - implements execute by invoking corresponding operation on Receiver
- Client
  - creates ConcreteCommand object and sets its receiver
- Invoker
  - asks the command to carry out request
- Receiver
  - knows how to perform the operation associated with request

STRATEGY

- Transform an algorithm or behavior into an object, allowing it to vary independently
- Make it easy to change algorithm by swapping out an object

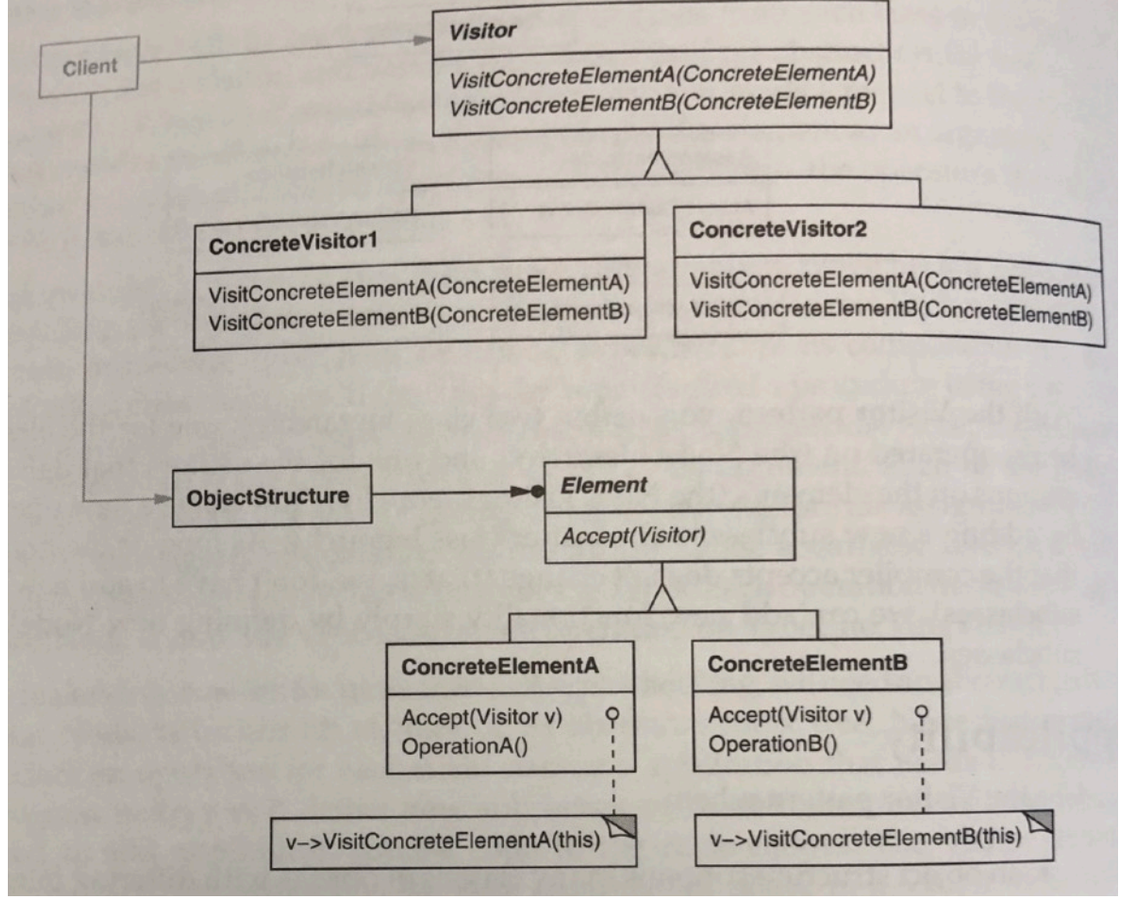


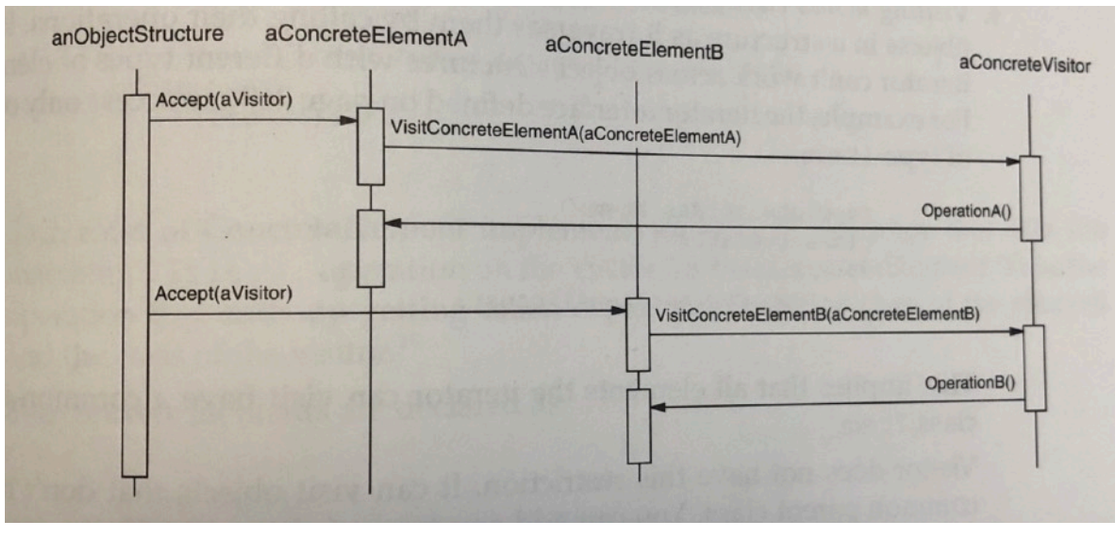
Participants

- Strategy
  - declares an interface common to all supported algorithms
- ConcreteStrategy
  - implements the algorithm
- Context
  - configured with a ConcreteStrategy object
  - maintains reference to strategy object
  - may define interface that lets Strategy access its data

VISITOR

- Represents an operation to be performed on elements of an object structure
- Enables defining new operations without changing implementation of elements on which it operates



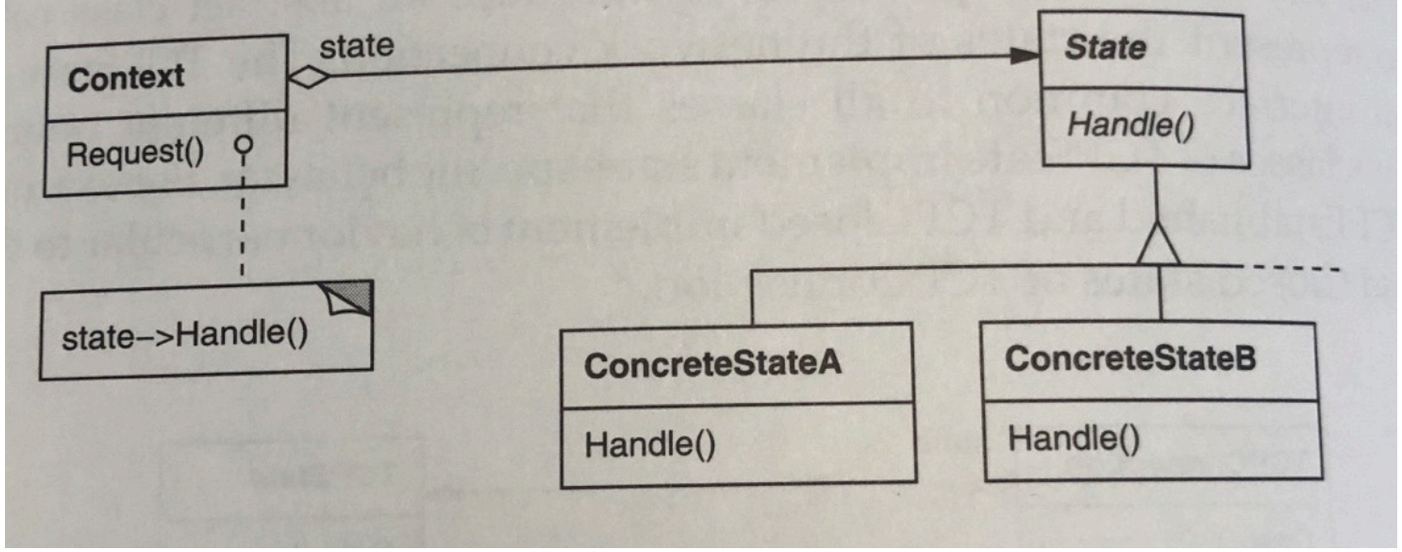


Participants

- Visitor
  - declares a Visit operation for **each** class of ConcreteElement
- ConcreteVisitor
  - implements each operation for corresponding object
  - accumulates state from visiting objets
- Element
  - defines Accept operation that takes visitor as argument
- ConcreteElement
  - implements an Accept operation

STATE

- Allows an object to alter its behavior when its internal state changes
- Object appears to change its class at runtime.



Participants

- Context
  - defines an interface of interest to clients
  - maintains an interface of a ConcreteState subclass that defines the current state
- State
  - defines an interface for encapsulating the behavior associated with a particular state
- ConcreteState subclasses
  - implements behavior associated with its state

WORKING WITH DESIGN PATTERNS

WORKING WITH DESIGN PATTERNS

- Useful patterns arise from practical experience
  - If you commonly see the same problem, pattern can describe a solution
  - Validating pattern comes from experience with it
    - Teams can create a process to author and disseminate their own patterns
- Patterns capture tradeoffs
  - Using a pattern brings both pros and cons, which can be captured in pattern
  - Important to understand context in which pattern can be useful

SUMMARY

- Design patterns offer a solution to a problem in a context
- GOF patterns offer solutions to how to design for change by enabling extensibility
  - Ways to encapsulate decisions that may change into classes decoupled from client code
- Design patterns broader than GOF patterns
  - Can have design patterns which describe technical solutions to variety of design problems that recur
  - Sometimes used to document how to teach how to use a new technology effectively (e.g., node.js design patterns)

IN CLASS ACTIVITY

IMPLEMENT COMPOSITE

- Pick an OO language (e.g., Java, C++, Python)
- Write an implementation of composite for a Drawing application
  - Implement common interface of Graphic
  - Primitive drawing elements: Line, Rectangle, Text.
  - Picture consists of one or more Graphic elements
- Code should focus only on portion of implementation relevant to Composite Pattern
  - e.g., do not need to write render function
- Deliverables:
  - Code implementing Composite for a drawing application