Some Design Patterns

CS 345 Winter 2018

Chris Reedy

A Catalog of Object Design Patterns (from Wikipedia, "Software Design Pattern")

- Creational
  - Abstract Factory
  - Builder*
  - Factory Method
  - Lazy Initialization
  - Multiton
  - Object Pool
  - Prototype
  - Resource Acquisition is Initialization
  - Singleton*
- Structural
  - Adapter (Wrapper, Translator)
  - Bridge*
  - Composite*
  - Decorator*
  - Façade
  - Flyweight
  - Front Controller
  - Marker
  - Module
  - Proxy
  - Twin
- Behavioral
  - Blackboard
  - Chain of Responsibility
  - Command*
  - Interpreter*
  - Iterator*
  - Mediator
  - Memento
  - Null Object
  - Observer* (Publish/Subscribe)
  - Servant
  - Specification
  - State*
  - Strategy*
  - Template Method*
  - Visitor

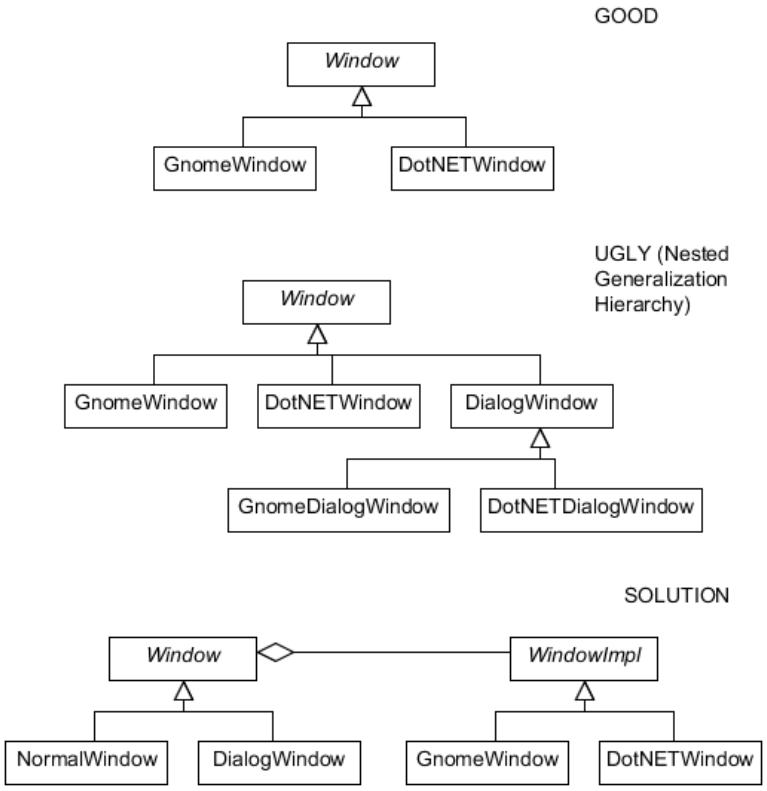
*Discussed in this lecture.

Structural Pattern

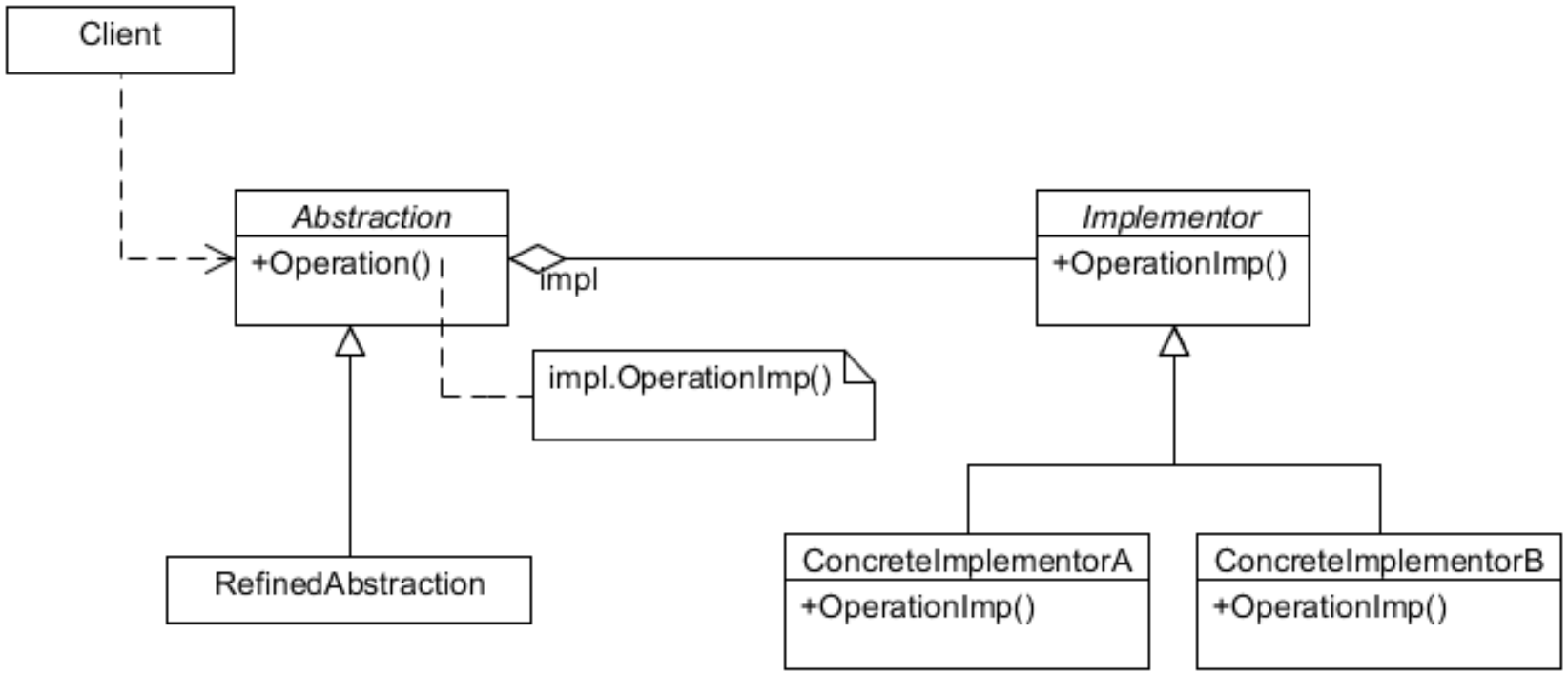
Bridge

Bridge Pattern Motivation

- AKA Handle/Body
- What do you do when an abstraction (a behavior) can have multiple implementations?
  - Usual approach: Abstraction is an interface or an abstract class. Implementations inherit from the abstraction.
- Problem: What if there are both multiple abstractions and multiple implementations?



Bridge Pattern Structure



Bridge Applicability

- Avoid a permanent binding between an abstraction and its implementation.
  - Implementation can be changed dynamically.
- Both Abstraction and Implementor can be extended using inheritance.
  - Can combine different abstractions and implementations.
- Changes in Implementor have no effect on clients of Abstraction.
  - No recompilation required
- Avoids “nested generalization” hierarchies.
- Potentially share Implementors.

Bridge: Consequences and Implementation

- Consequences
  - Decoupling interface and implementation
    - Can configure implementation at run-time
    - Eliminate compile time dependencies
  - Improved extensibility
    - Can extend Abstraction and Implementor independently
- Implementation
  - You can make Abstraction or Implementor concrete if there is only one.
  - How do you pick the correct Implementor:
    - Have Abstraction pick the Implementor
    - Use a default and change it later
    - Delegate to another class
  - Sharing Implementors: Use reference counting if no garbage collection

Bridge Examples
Ropes (1)

- A Rope is a heavyweight String. There are multiple implementations of Ropes, corresponding to Strings and Concatenations, Substrings, etc. of Ropes.
  - For example: A substring of a Rope is simply a Rope that points to its parent and contains the start and end indices for the substring.
  - Note that Ropes share structure.
  - Why? Answer: Ropes make for fast operations when building a string. Getting the underlying string, for example, for output, is O(length of rope).

Bridge Examples
Ropes (2)

- You can use the Bridge pattern with a single Rope object which points to multiple kinds of Rope implementation objects.
  - Rope implementations are inherently value types.
    - Mandatory due to shared structure.
  - Rope may or may not be a value type.

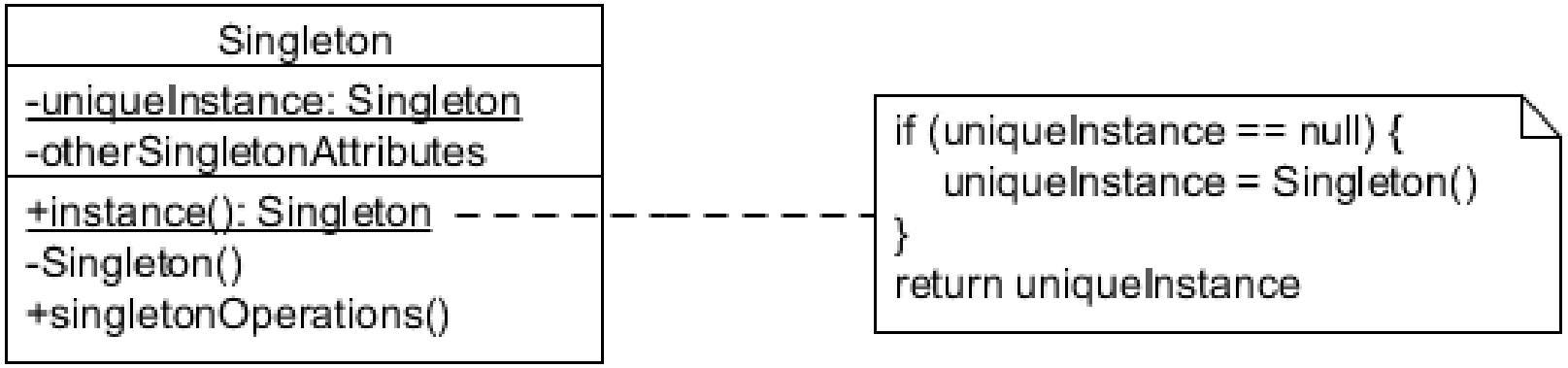
Creational Pattern

Singleton

Singleton Motivation

- There are a lot of situations where there should be at most one instance of a class. Examples:
  - Database connection
  - Window manager
  - The application
- How do you control/guarantee that no more than one instance of a class is ever created?

Singleton Structure



Singleton Applicability

- When there must be (no more than) one instance of a class and it must be accessible from a well-known access point.
- When the sole instance should be accessible by a subclass.
  - One implementation: Make uniqueInstance protected and have the first call to instance() be on the subclass
  - Clients can use the extended instance without code modification.

Singleton: Consequences

- Controlled access to the sole instance
  - instance method can control access
- Reduced name space
  - The class eliminates need for an additional global variable
    - The class is a global variable!
- More flexible than static operations
- Can be modified to permit a variable number of instances

Singleton: Implementation

- Singleton instance may be subclassed
  - Must be configured. Configuration can be done via:
    - Initialization logic in the main program
    - Linking in the correct class (C++ easy, Java tricky)
    - Registry of singletons: instance method looks up correct class in registry
- If operating in multi-threaded environment, you must:
  - Do instance creation as part of initialization, or
  - Put locks around the instance creation.

Singleton Consequences: Major Liability

- The singleton pattern creates a strong coupling between the using code and the Singleton class
  - Singleton class cannot be an interface: `instance` method and `uniqueInstance` attributes are static
- Use of singleton interferes with automated testing
  - Cannot replace singleton class for testing purposes
    - Example: Use a different database connection object when running tests
- Use of singleton interferes with dependency injection
  - Dependency injection framework handles this problem directly without use of singleton pattern

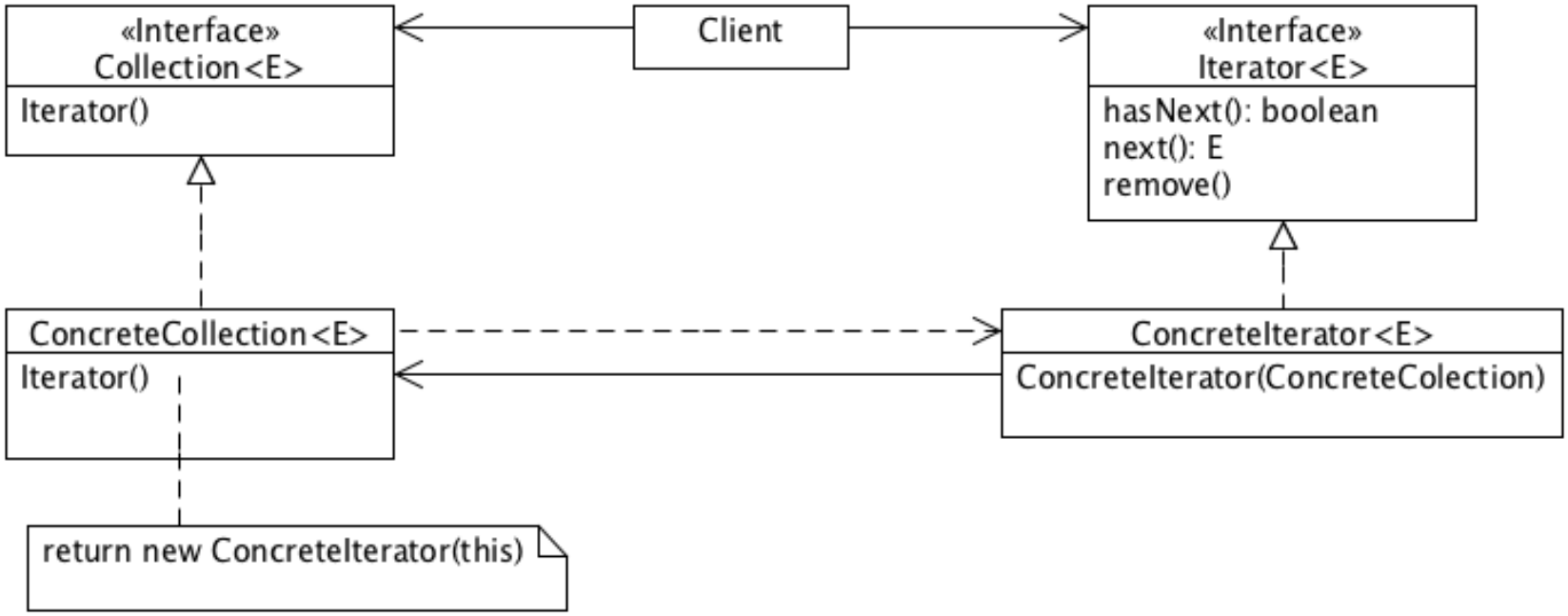
Behavioral Pattern

Iterator

Iterator Motivation

- A collection (e.g. List) needs a way to provide access to its elements without exposing implementation details
- You might want to traverse a collection in different ways
- You want to keep the collection interface as simple as possible and not burden it with traversal operations
- You might want to have more than one traversal active at the same time on the same collection

Iterator: Structure



Note: This shows Java collections framework version of Iterator.
Other languages use alternative operations.

Iterator
Applicability and Consequences

- Applicability
  - Access a collection's contents without exposing implementation
  - Support multiple "simultaneous" traversals of a collection
  - Provide a uniform interface for traversing different kinds of collections
- Consequences
  - Supports variations in the traversal (e.g. forward and backward)
  - Simplifies the collection interface
  - Allows more than one simultaneous traversal

Iterator Implementation

- Does Client or Iterator control iteration?
  - Client: external iterator (Java does this)
  - Iterator: internal iterator (Java 8 also does this)
    - Client must provide the operation to be performed to an internal iterator
    - Internal iterators do not support simultaneous iteration (e.g. merging two lists as in a merge sort)
- Does Collection or Iterator control traversal?
  - In Java, Iterator controls
  - When Collection controls:
    - Iterator is referred to as a Cursor
    - Traversal operations are part of the Collection interface

Iterator Implementation

- Can you modify the Collection while maintaining the Iterator?
  - “Robust” Iterator allows for modification
  - Java provides a limited form of robust iterator
    - You can’t directly modify the collection
    - You can use the iterator to modify the collection
    - You can’t have more than one iterator active when modifying a collection
- Does the Collection have a reference to the Iterator
  - Generally needed for robust Iterators
  - If yes, this can interfere with garbage collection
    - May have to “close” the iterator

Java 8: Internal Iterators Example

`roster` is a Collection of persons.

```
roster.stream()
      .filter( p -> p.getAge() >= 18 && p.getAge() <= 25)
      .map(p -> p.getEmailAddress())
      .forEach(email -> System.out.println(email));
```

Equivalent to:

```
for (Person p : roster) {
    if (p.getAge() >= 18 && p.getAge() <= 25) {
        EmailAddress email = p.getEmailAddress();
        System.out.println(email);
    }
}
```

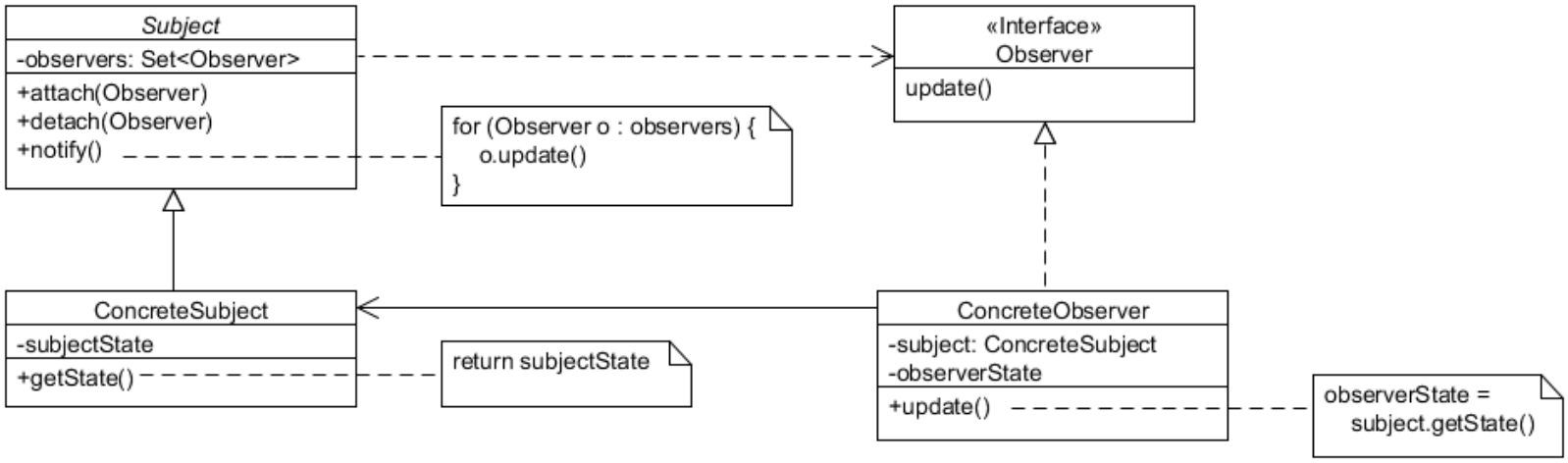
Behavioral Pattern

Observer

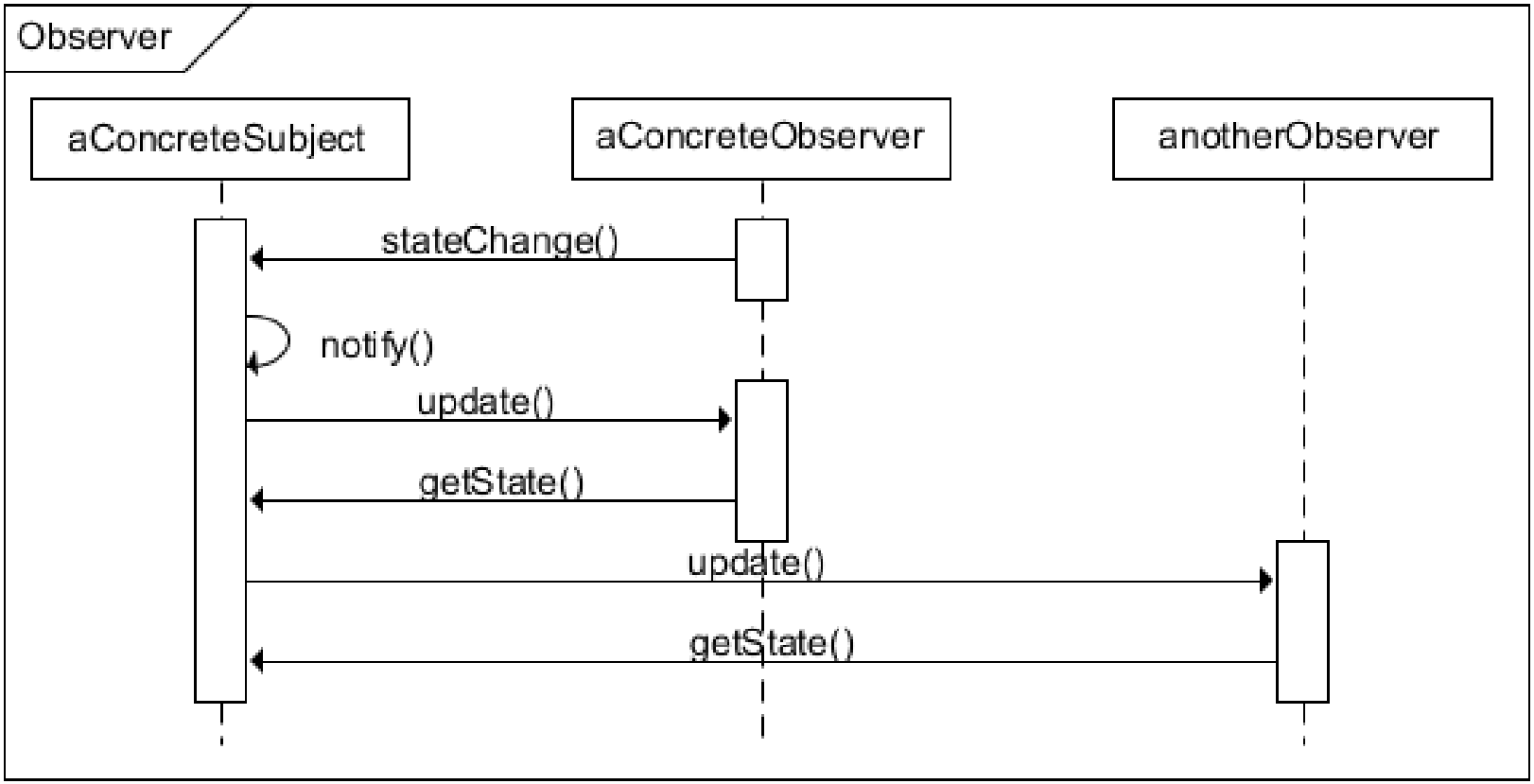
Observer
Motivation

- AKA: Dependents, Publish-Subscribe
- Need to maintain consistency between related objects.
  - Consequence of partitioning system into a collection of cooperating classes.
- Example: Spreadsheet
  - If you change the value of a cell you may need to:
    - Update the displayed value
    - Update any other cells that depend on that cell
    - Update any charts, graphs, etc. that display the value of the cell

Observer Structure



Observer: Collaboration



Note that the ConcreteObserver that initiated the state change in the ConcreteSubject waits to update its copy of the subject's state until it is informed of the change.

Observer Applicability

- Encapsulating separate aspects of an abstraction when one is dependent on the other.
  - Separate computational aspects of a spreadsheet from the display
- You don’t know how many objects need to respond to a change in some object.
- Avoid tight coupling of Subject to notified objects.
  - You can notify other objects without having to make assumptions about those objects.
- In general: To avoid tightly coupling an abstraction to dependent abstractions.

Observer Consequences

- Subject is not coupled to observer.
  - This is good.
- Subject can broadcast notification to multiple observers.
  - This may be good. However,
- Multiple observers have no knowledge of each other.
  - A seemingly simple change can result in cascading updates.
    - An observer can also be a subject!
  - Circular dependencies can arise (especially if the developer doesn't control dependencies).
    - This can lead to problems with infinite loops of updates and spurious updates.

Observer Implementation (1 of 2)

- Mapping subjects to observers
  - Usual approach: Subject maintains set of observers
  - Alternative: Hash table mapping subjects to observers
    - Advantage: No memory required for subjects with no observers
- Observing more than one subject
  - Change update() protocol so that subject identifies itself
- Does subject or client call notify
  - Subject: Clients don't have to remember to call notify
  - Client: Notify can be delayed until after a series of related updates
- Subject state must be self-consistent before notify is called
  - Document when subject classes call notify

Observer Implementation (2 of 2)

- Push versus pull models of updates
  - Pull model:
    - Observers ask subject for details of current state
    - Observers must determine what, if anything, changed
  - Push model:
    - Subject provides information about what changed, whether observer wants it or not
    - Assumes subject has knowledge of what information is needed by observers
- Explicitly specifying information of interest
  - Subjects can provide notification for multiple change events and observers register for specific events of interest
- When things get particularly messy:
  - Consider a Change Manager – a separate class that encapsulates complex update semantics.

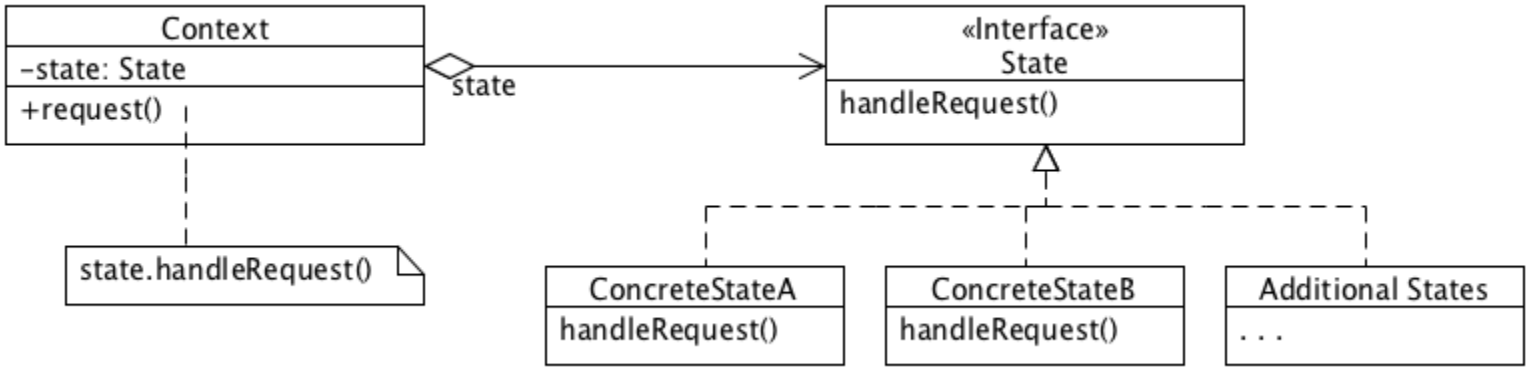
Behavioral Pattern

State

State
Motivation

- Objects can have state
  - For example, an email can be received, sent, in composition (maybe more)
- The behavior of an object can depend on the state
  - Code for methods contains (maybe complex) conditionals based on the objects state
- Key idea: The actual behavior of the object depends on the state of the object

State Structure



State – Applicability, Consequences, Implementation

- Applicability
  - An object must change behavior at run-time based on its state
  - Avoid operations with complex conditionals (if or switch statements) testing object's state.
- Consequences
  - Localizes state specific behavior (good)
    - Distributes behavior for different states among multiple classes (may not be good)
  - Makes state transitions explicit
- Implementation
  - Context or State can define state transitions
  - Does State have instance variables?
    - If not, can share State objects among Contexts

Behavioral Pattern

Strategy

Strategy
Applicability and Consequences

- Applicability
  - Configure a class with one of many behaviors
  - You need different variants of an algorithm
  - Encapsulate complex, algorithm specific data structures
- Structure – see State
- Consequences
  - Alternative to subclassing
  - Provide a choice of implementations for the same behavior
  - Clients may need to be aware of different strategies
  - Communication overhead between strategy and context

Strategy Implementation

- How does Context pass data to Strategy?
  - Context passes relevant data as parameters to Strategy operations
    - What is relevant may change between strategies
  - Context passes itself as argument
    - Context data must be accessible by Strategy That is, it can't be private.
  - Strategy maintains a reference to the Context
    - Context must provide a more elaborate interface to its data

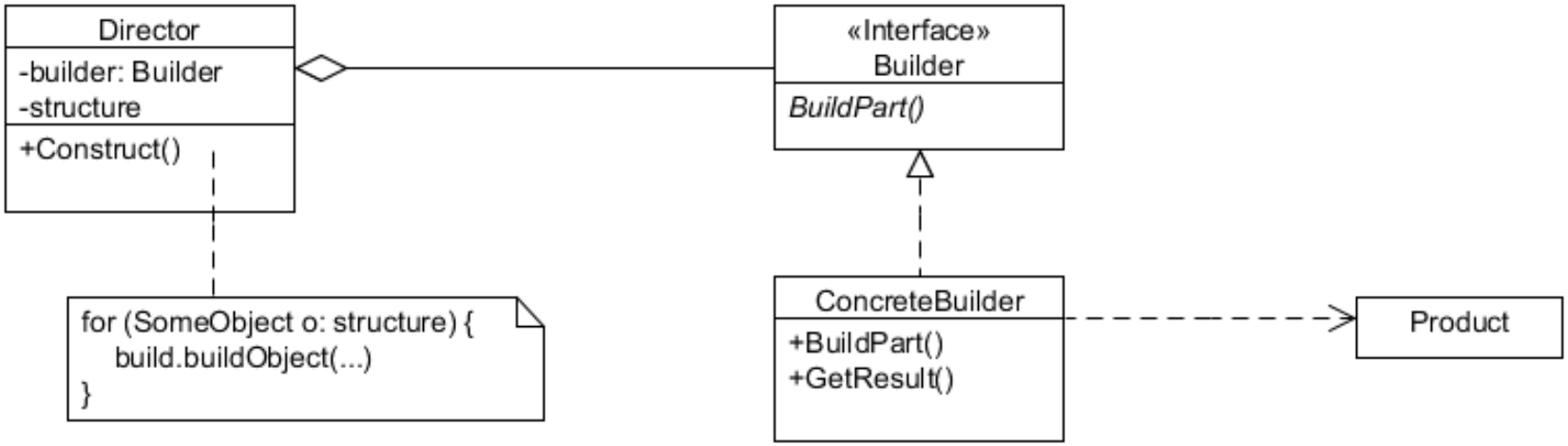
Creational Pattern

Builder

Builder Motivation

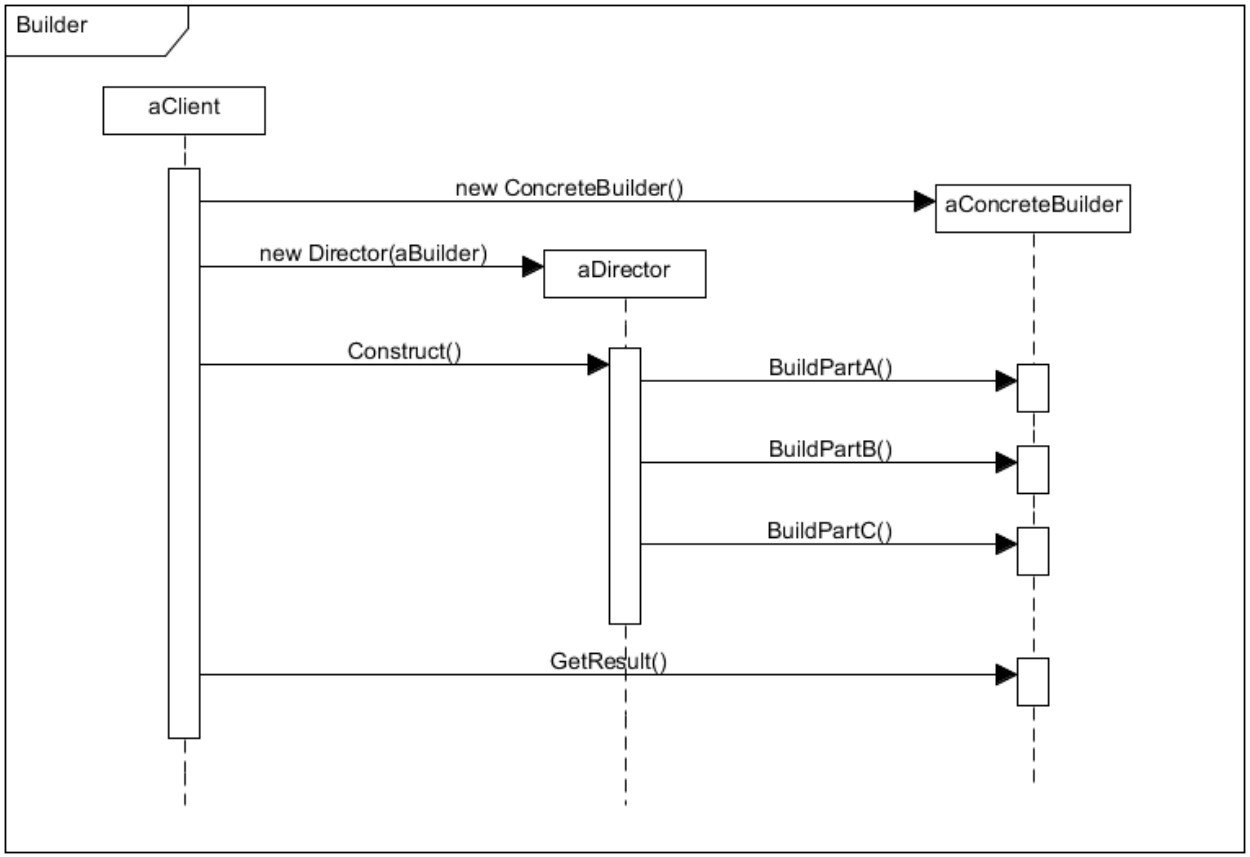
- Separate construction of a complex object from its representation.
  - Allow the same construction process to create different representations.
- Example:
  - Read a Markdown file
  - Construct: ASCII, TeX, HTML based on same Markdown data
- Example: You want to separate parsing of an input file from construction of specific translation/representation of the data.

Builder Structure



Director is responsible for processing input. Builder is responsible for building the product.

Builder Collaboration



Builder
Consequences and Implementation

- Builder Pattern
  - lets you vary a product's internal representation
  - isolates code for construction and representation
  - gives you finer control over the construction process
- Builder interface must be general enough to allow for construction using all concrete builders
- No abstract class for Product?
  - In general, products are different enough that there is little to gain from a common parent.
- Is Builder an interface or an abstract class?
  - Abstract class has benefit that it can have empty default methods.

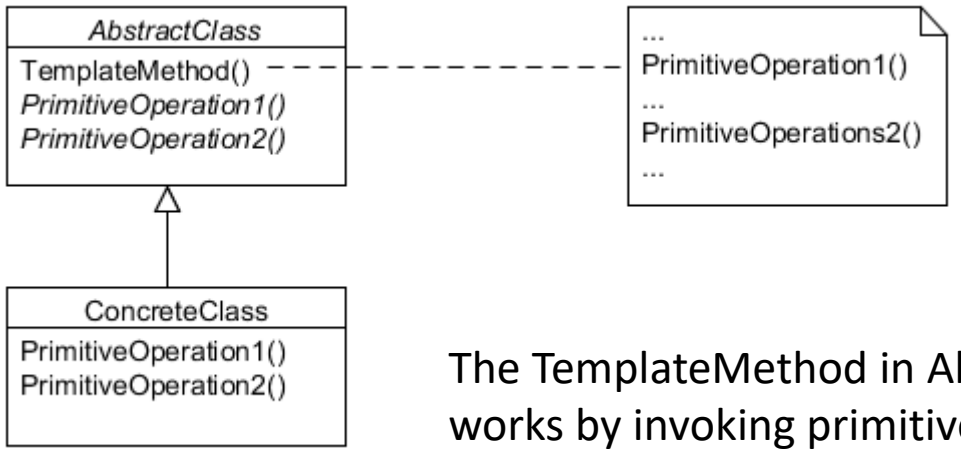
Behavioral Pattern

Template Method

Template Method Motivation

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- For example: Consider an application that creates, edits, updates documents (text, spreadsheets, drawings, …).
  - I have to have methods for displaying documents, saving documents, opening documents, …
    - These all depend on the specifics of the document
  - I need a window with Open, Save, etc. menu items
    - These are all independent of the specifics of the document
- What I want is an abstract Application class that provides a skeleton application with subclasses providing the particulars for each kind of document.

Template Method Structure



The TemplateMethod in AbstractClass works by invoking primitive operations that are defined in ConcreteClass.

Template Method Example

- Input a number (a double say) compute a function, output the result.
  - doFunction is a Template Method, compute is a Primitive Operation

```
abstract class AbstractFn {

  void doFunction(Scanner s) {
    double x = s.nextFloat();
    double y = compute(x);
    System.out.println(y);
  }

  abstract double
      compute(double x);
}
```

```
class SinFn extends AbstractFn {
  @Override
  double compute(double x) {
    return Math.sin(x);
  }
}

class SqrtFn extends AbstractFn {
  @Override
  double compute(double x) {
    return Math.sqrt(x);
  }
}
```

Template Method Applicability

- Implement invariant part of an algorithm once and leave varying behavior to subclasses.
  - Eliminate code duplication
- Factor common behavior in subclasses into the parent.
- To provide a structure for subclass extensions.

Template Method Consequences and Implementation

- Template methods use the Hollywood Principle: “Don’t call us, we’ll call you.”
  - Parent class calls operations on child class, not vice versa.
- Template methods may call:
  - Primitive operations – abstract operations
  - Concrete AbstractClass operations – operations that are generally useful to subclasses
  - Hook operations – provide default behavior, frequently doing nothing, but may be overridden
- When writing a class with template methods, you should provide documentation about:
  - Which methods are abstract – user must provide an implementation
  - Which methods are hooks – user may override
  - Which methods should not be overridden

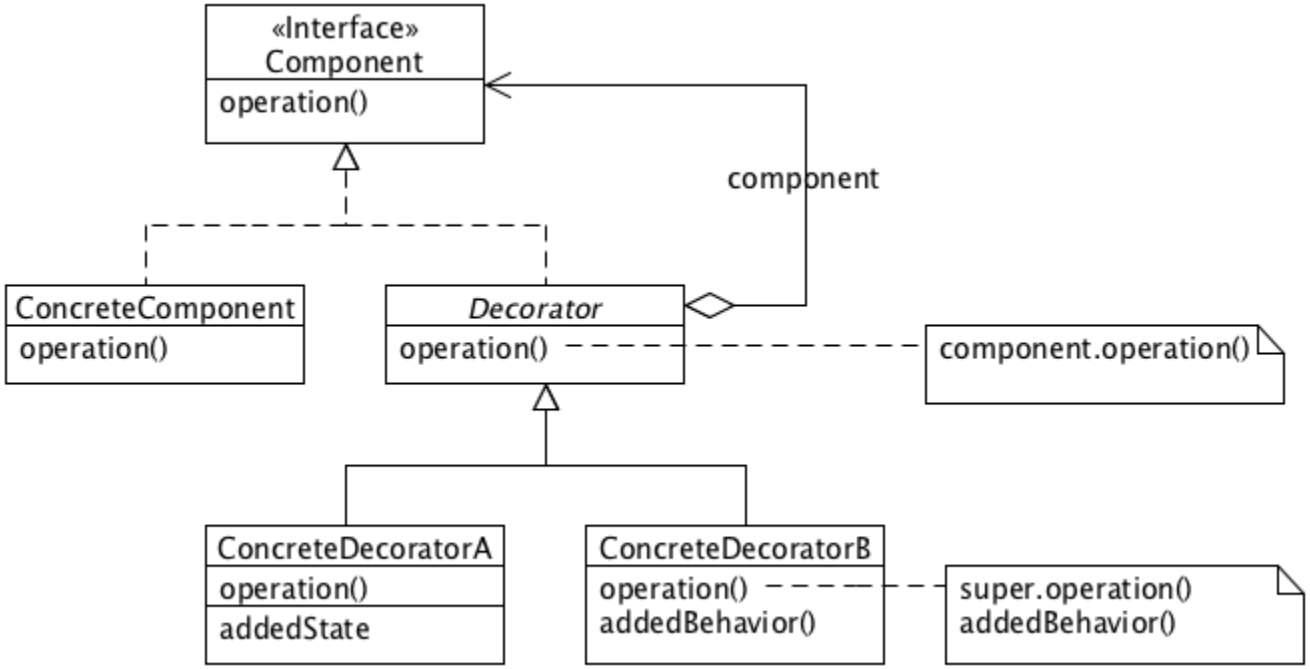
Structural Pattern

Decorator

Decorator
Motivation and Applicability

- Add responsibilities to individual objects
  - But, not to an entire class
- Allow added responsibilities to be withdrawn
- Avoid heavy classes (lots of behavior) high in the class hierarchy
- Minimize size of inheritance hierarchy
- When subclassing is impractical

Decorator Structure



Decorator
Consequences and Implementation

- Provides more flexibility than static inheritance (good)
- A decorator and its component aren't identical
  - Don't rely on object identity (Java == for objects) when using decorators
- Potential problem: Lots of little objects
- Decorator and ConcreteComponent must implement a common interface
  - This common interface needs to be lightweight – avoids having heavyweight Decorators
- Compare to strategy:
  - Decorator changes the behavior, strategy changes the implementation

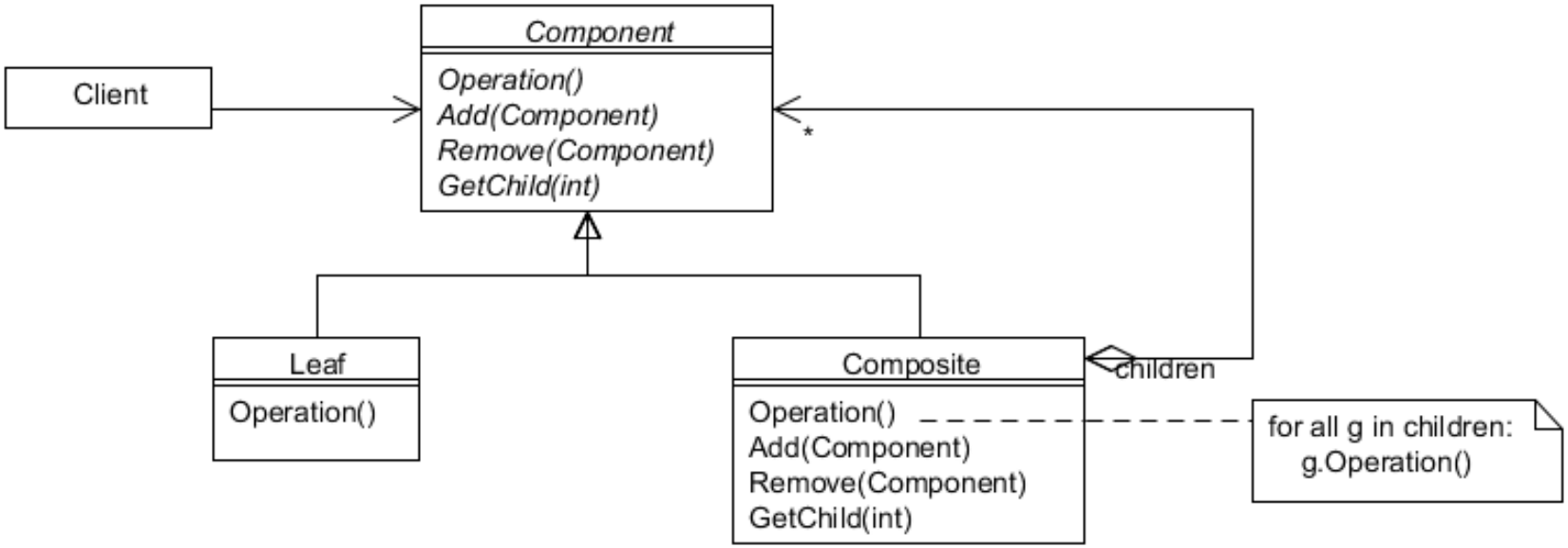
Structural Pattern

Composite

Composite Motivation

- Assemble complex objects out of simple components.
- Allow user code to treat simple and complex objects the same.
- Example:
  - A diagram can be simple (a line), or complex (lines, boxes, text, …)
  - In my code I want to be able to treat simple and complex diagrams the same.

Composite Structure



Composite Consequences

- Simplifies client code
- Makes it easy to add new types of components
- Can make your design overly general
  - You may want to restrict which types of components are children of a specific type of composite. The type system won't help you do this.

Composite Implementation

- Does child have reference to parent?
  - Can ease traversal and editing of component tree
- Sharing components
  - Useful, but can't do this with parent references
- Where to declare the child management operations?
  - These don't make sense for Leafs
  - Transparency: Declare them at the root and provide default implementations which are used by Leafs.
    - Allows you to treat all components uniformly
  - Safety: Only declare them for composites.
    - Type checking will ensure that you don't try to perform child management on a Leaf.

Composite Example

In JavaFX:

- Node is the superclass for all components in the scene graph.
  - Everything you seen on the display is represented by a Node.
  - Node is the *Component* in the design pattern.
- Parent is the superclass for all components that contain other components.
  - Parent is the *Composite* in the design pattern.

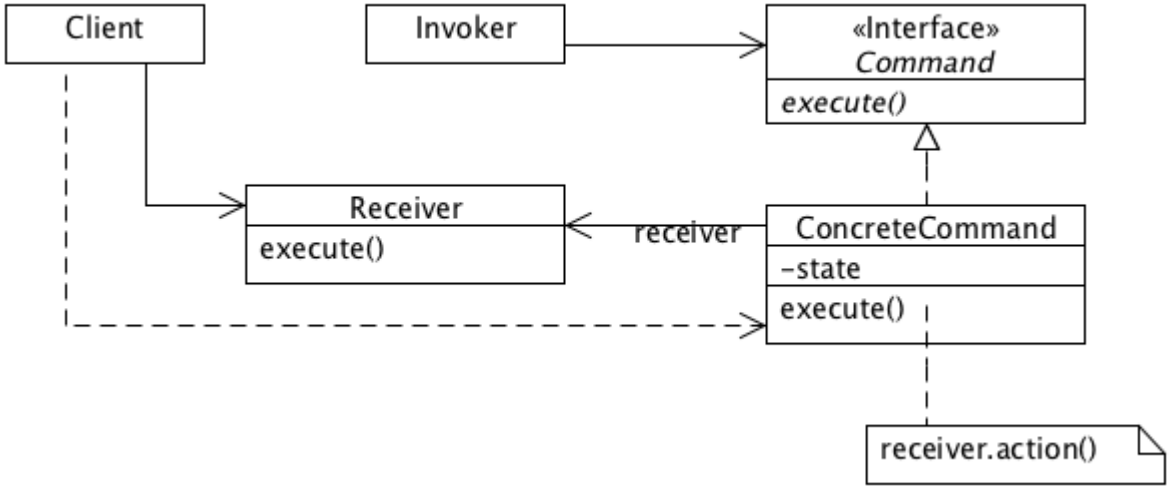
Behavioral Pattern

Command

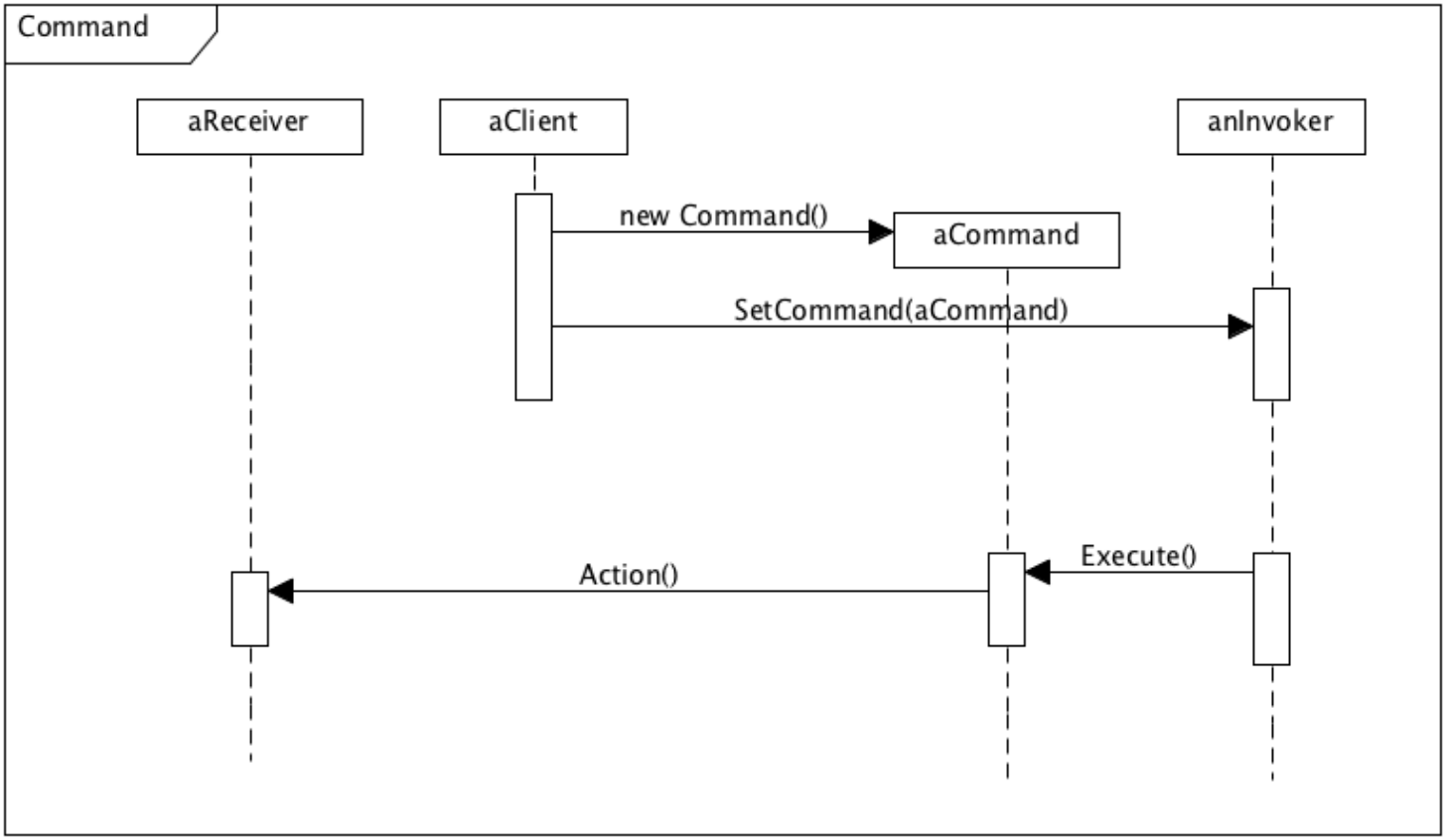
Command Motivation

- Issue commands without knowing the operation being requested or the receiver of the request
  - Common problem in GUI frameworks
    - Menus invoke operations without knowing/understanding the operation
    - Undo/Redo of previous commands

Command Structure



Command Collaboration



Command Applicability

- Parameterize an object by an action to perform
- Specify, queue, and execute requests at different times
  - Command object can have a lifetime that is independent of the original request
- Support undo/redo
  - Execute operation stores state for executing undo and/or redo
  - Command interface must have undo and redo operations
- Support logging operations
  - Logging changes can be used to recreate a state after a crash

Command
Consequences and Implementation

- Decouple object invoking operation from one that knows how to perform it
- Assemble commands into a composite (macro)
- Make it easy to add new commands
- How intelligent should a command be?
  - Minimal: Command invokes operation on receiver, receiver does everything else
  - Maximal: Command implements everything without delegating to receiver

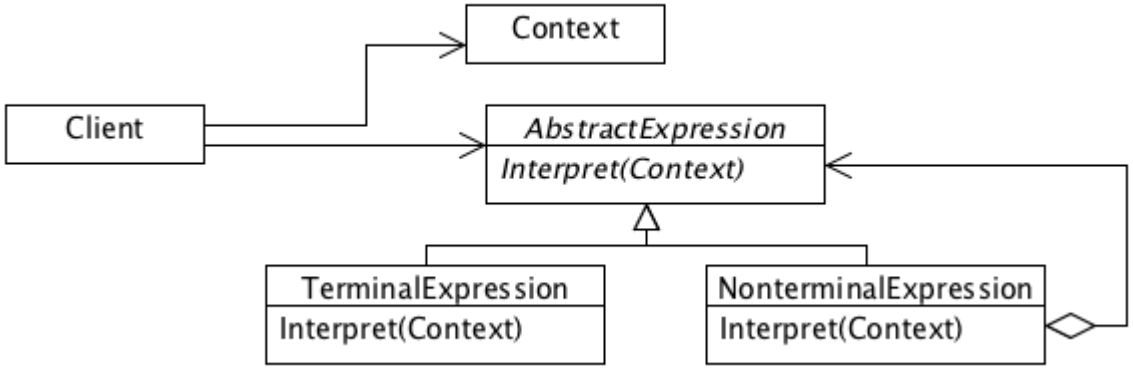
Behavioral Pattern

Interpreter

Interpreter
Motivation and Applicability

- You need to provide a scripting language for your application
- A scripting language requires
  - A grammar – how to form valid sentences/programs/statements in the language
  - A parser – a program that convert a string representation for a sentence into an Abstract Syntax Tree (AST)
  - An interpreter – a program that interprets an AST within a context

Interpreter Structure



Interpreter Implementation

- Generally works best when
  - Grammar is simple
  - Efficiency is not important
- Pattern does not explain how to create the AST
- Variants
  - Embed a standard scripting language into your application.
  - Extend a standard scripting language with your application.