UCLA CS130 Software Engineering Fall21 Review Note: Midterm

By Patricia Xiao

UML Diagrams

Static / Structure Modeling: fixed, code-level

- Class Diagrams
- etc. (e.g. Component Diagrams)

Dynamic / Behavioral Modeling: capturing execution of the system

- Use Case Diagrams
- Sequence Diagrams
- State Chart Diagrams
- etc. (e.g. Activity Diagrams)

Class Diagrams

Models: high-level class relations
Components:

- Class (rectangle)
  - Upper section: name of the class
  - Middle section: attributes (type, visibility)
  - Bottom section: methods (type, visibility)
- Relations (links between classes): Dependency, Association, Aggregation, Composition, Generalization, Realization

Class Diagram: Visibility Symbols

Public (+)   Private (-)   Protected (#)
Package ($\tilde{}$)   Derived (/)   Static (underlined)

Class Diagram: Multiplicity Definition

Multiplicity (Cardinality)
Of a **class**: A number in the upper right corner of the component; the number of objects at runtime; usually omitted and by default > 1.
Of a **relation**: Placed near the ends of an edge, indicating the number of instances of one class linked to an instance of the other class on the other side of the edge.

Class Diagram: Multiplicity Symbols

n	exactly n	$m..n$	at least m, at most n
$*$	many	$1..*$	at least one, could be more
$0..1$	zero or one	$0..0$	must be empty

Class Diagram: Relations

From weak to strong, from general to specific:

- Dependency (uses) — A uses B (dashed line pointing from A to B)
- Association (has-a) — A has a field of B object (solid line pointing from A to B)
- Aggregation (owns) — satisfies iff
  - A has a field that is a list of B objects

  (solid line pointing to B with an **unfilled** diamond at the A end / association end)
- Composition (part-of) — satisfies iff
  - A has a field that is a list of B objects
  - B object can't live outside A

  (solid line pointing to B with an **filled** diamond at the A end / association end)
- Generalization (is-a) — B extends A / subclassing (close-headed solid line pointing to A)
- Realization — B implements A / sub-typing (close-headed dashed line pointing to A)

Use Case Diagram

Specify: Actors, System (scenario), Goals
Models: high-level **interactions**
Components:

- Actors (stick figures) – role (one user can have multiple roles)
- Use Cases (ovals) – scenario
- Relations (edges): association, inclusion, extension, generalization

Actors are **not** directly interacting with each other.

Use Case Diagram: Relations

Association

- actor – case (undirected solid line)
- case – case (dashed line with arrow)
  - inclusion (e.g. *ride* ≪ include ≫ *push button*, arrow pointing to *push button*)
  - extension – exceptional variation (e.g. *derail* is an ≪ exception ≫ of *ride*, arrow pointing to *ride*)

Generalization/Specialization (close-headed arrow pointing to more general one); e.g. *Synchronize Data* generalize *Synchronize Data Wirelessly*

Sequence Diagram

Models: **communication** between elements
Belongs to **Interaction** Diagrams (include: Sequence diagrams, Communication diagrams, Interaction overview diagrams, Timing diagrams)
Components:

- Class Roles / Participants (top-row) / Actors
  - instance_name : Class_Type
  - not necessarily an object in the system, e.g. can be human actors.
- Activation or Execution Occurrence (dispatch: solid black dot, destroy ≪ destroy ≫)
- Messages (horizontal arrows)
  - Method Invocation (solid line with arrow)
  - e.g. a:A point to b:B with text execute(0), then it means a (of class A) calls b.execute(0), b is of class B.
  - Return value via dashed line pointing back
- Lifelines (dashed vertical lines)
  - Invocation Lifetime: vertical rectangles
  - can be nested across actors, and threads within a single actor
- **Loop** (while / for, [condition]) / **Alt** (if-then-else, [if-condition] – horizontal dashed line – [else]) / **Opt** (if-then, [if-condition]) / Par / Region; All shown as wrapped in a rectangle.

Seq Diagram: Invocation Lifetime v.s. Lifetime

When a:A create an instance of b:B at run time, we draw the rectangle with text content b:B **at the height** where a:A invokes it.

Then it starts to live. When a:A create an instance of B named b, we depict it by letting a:A pointing to a newly-created b:B column via dashed line and text: create(params); where params are the parameters needed for instantiate an object of class B.

Invocation Lifetime is not **Lifetime**.

Lifetime is represented by the dashed line, invocation lifetime is represented by the thin vertical rectangle along the dashed line.

Seq Diagram: Class Name and Type

If name of an object of class A is unknown, it is okay to leave it blank, e.g. : A.

State Chart Diagram

Models: high-level **state behaviors** of objects
Components:

- Initial State (black filled circle) – start
- Transition (solid arrow)
  - *trigger [guard] / effect*
  - *trigger* if *guard*, make *effect*
  - e.g. Somewhere is a Door's State Machine: *use key [door locked] / [door → unlock]*
- State (rounded rectangle) – of object
- Fork (rounded solid rectangular bar) – 1 incoming arrow, n outgoing arrows; represent splitting into concurrent states.
- Join (rounded solid rectangular bar) – n incoming arrows from the joining states, m outgoing arrow towards the common goal states; multiple states concurrently converge into one on the occurrence of an event or events.
- Self transition (solid line w. arrow pointing back to itself) – the state of the object does not change upon the occurrence of an event
- Composite State (rounded rectangle) – wrapping around a lot of other states
- Final state (black filled circle within a circle) – the final state in a state machine diagram

UML Diagram: Translations

format	Class	UC	Seq	State	Code
Class	N/A	✗	✗	✗	✓
UC	✗	N/A	✗	✗	✗
Seq	✗	✗	N/A	✗	✓
State	✗	●	✗	N/A	●
Code	✓	●	✓	✓	N/A

UC represents Use Case Diagram, Seq represents Sequence Diagram. Code refers to Java-style pseudo code. The meaning of the marks are listed below:

✓	sufficient (for the row) to transform to (the column)
●	transformation (from row to column) is doable but needs some extra clarification
✗	very unlikely to directly transform (the row) to (the column)

Software Design Principles

- **Information Hiding** (IH)
- Low Coupling (LC): Reduce the dependencies between modules (classes, packages, etc)
- High Cohesion (HC): A module contain functions that logically belong together.
- Separation of Concerns (SoC): a single concern is easily separated from the rest of concerns.
- etc. (e.g. Law of Demeter (LoD), Abstraction, Liskov Substitution Principle, ...)

There are many different principles. In this class we focus on information hiding.

Modularization

- Decomposition of a software system into multiple independent modules.
- Easy to interpret & maintain & code-reuse, etc.

Parnas' Information Hiding (IH) Principle

- A principle for breaking program into **modules**.
- API should (1) only contain design decisions unlikely to change (2) do not reveal any volatile information.
- Makes anticipated changes affect modules in an isolated and independent way.

Information Hiding (IH) Principle: Conclusion

Information hiding principle is:

- an analysis of how changes will affect existing code
- and assessment of changeability.

Modularization: Practice

Identify the Modules': **name**, **role**, **input**, **output**.
Changeability Assessment: for **different scenarios**, which module / which module's API(s) need to be changed.
Code Critique:

1. What information is hidden (by XXX Module)?
2. Changes you anticipate? (any new features you may want for the system)
3. Readability and comprehensibility? (e.g. consistent arguments, self-explanatory coding, etc.)
4. Capability to support independent work assignment? (low coupling)

Modularization: Different Ways to Achieve

Functional decomposition (Flowchart approach)

- Each module corresponds to each step in a flow chart.

Information Hiding (IH)

- Each module corresponds to a design decision that are likely to change and that must be hidden from other modules.
- Interfaces definitions were chosen to reveal as little as possible.

Design Patterns: Categories

Creational Design Pattern

- **Factory Method**: defines an interface for creating an object but lets subclasses decide which class to instantiate; lets a class defer instantiation to subclasses.
- **Abstract Factory**: provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Singleton**: ensures a single object creation.
- etc. (e.g. Prototype)

Structural Design Pattern

- **Adaptor**: adapts legacy code to a target interface.
- **Façade**: simplifies complex interfaces of multiple subsystems.
- **Flyweight**: share common resources by separating usage contexts from used objects.
- etc. (e.g. Composite)

Behavioral Design Pattern

- **Strategy**: defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime; lets the algorithm vary independently from clients that use it.
- **Observer**: defines one-to-many dependency between objects, when the subject changes state, all of its observers are notified and updated.
- **Mediator**: defines an object that encapsulates how a set of objects interact, encapsulates many to many dependencies between objects, centralizing control logic, reduces the variety of messages.
- **Command**: decouples a receiver object's actions from invokers.
- **Template Method**: set a common workflow where sub steps may vary at subclass.
- **State**: encode complex state transitions.
- etc. (e.g. Interpreter)

Design Patterns: References

- Book: Head First Design Patterns
- SourceMaking:
  https://sourcemaking.com/design_patterns/
- ReactiveProgramming:
  https://reactiveprogramming.io/blog/en/design-patterns/factory-method
- Refactoring.Guru:
  https://refactoring.guru/design-patterns

Factory Method Pattern
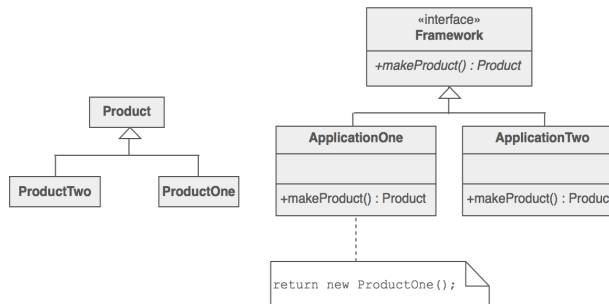
- Factory / Creator: include a factory method
- Concrete Factories / Concrete Creators: implement factory method
- Product
- Concrete Products

Factory Method: Class Diagram Draft

Factory Method – Class diagram
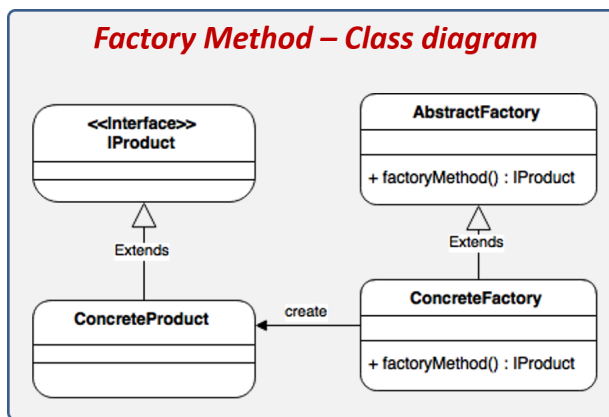
Factory Method: Sequence Diagram Draft

Factory Method – Diagram of sequence
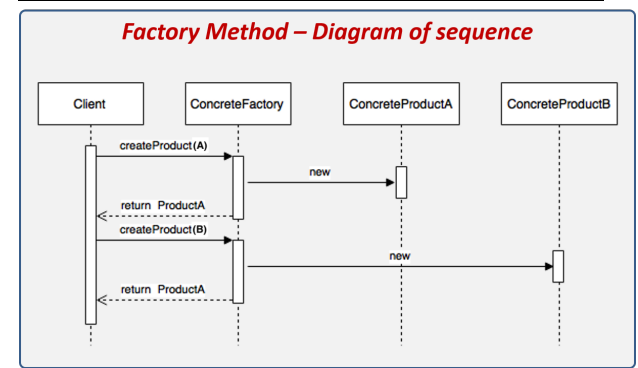
- **Not** an accurate Sequence Diagram.

Abstract Factory Pattern
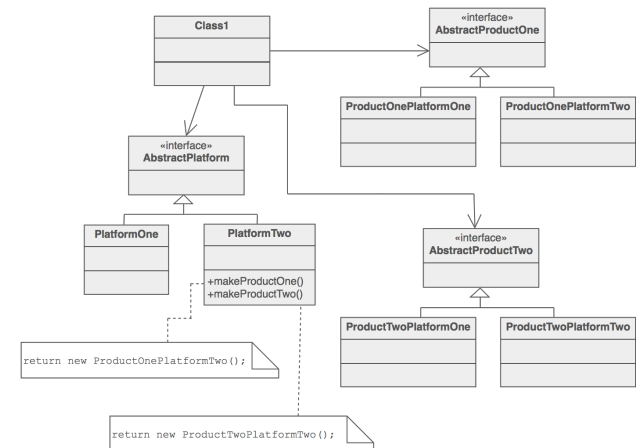
- Abstract Factory / Abstract Creator: include makeProductOne, makeProductTwo, etc.
- Concrete Factories / Concrete Creators: implement factory method
- ProductOne, ProductTwo, etc.
- Concrete ProductOneA, Concrete ProductOneB; Concrete ProductTwoA, etc.

When adding new products to the abstract factory, the interface has to be changed.
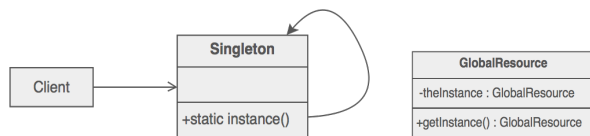
Abstract Factory: Class Diagram Draft

Abstract Factory – Class diagram
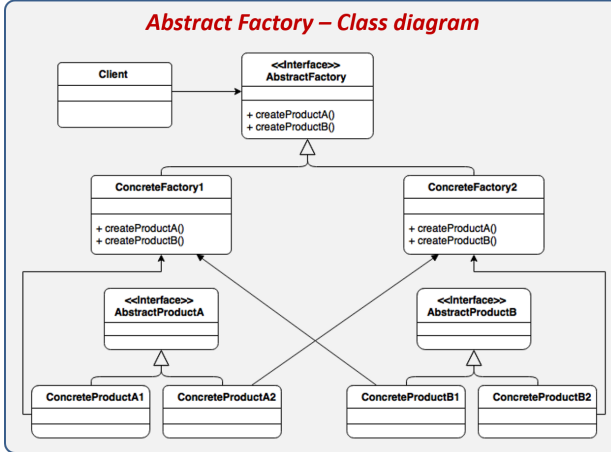
Abstract Factory: Sequence Diagram Draft

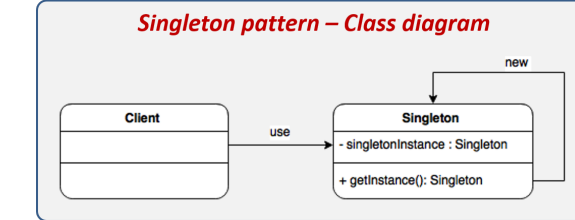*Abstract Factory – Diagram of sequence*
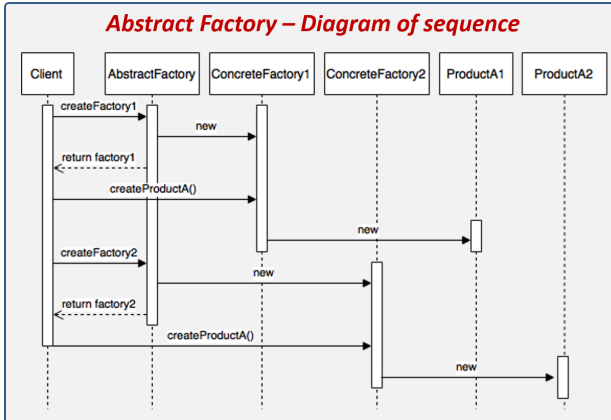
- **Not** an accurate Sequence Diagram.
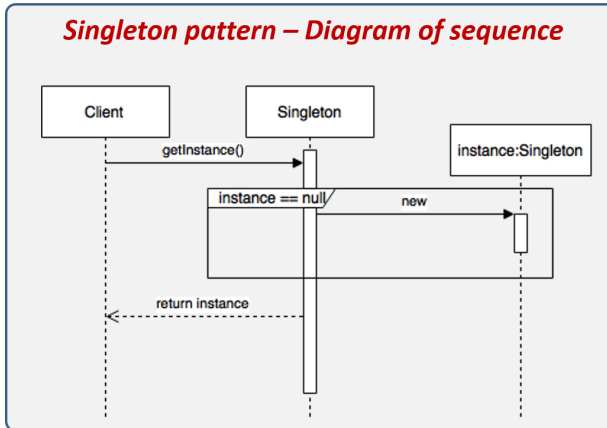
Singleton Pattern

- The class of the single instance is responsible for access and "initialization on first use". The single instance is a private static attribute, accessed via a public static method.
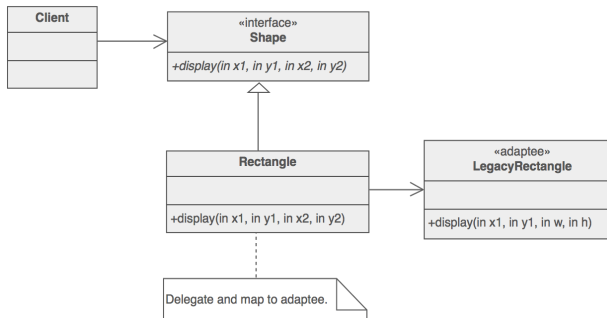
Singleton: Class Diagram Draft

Singleton pattern – Class diagram

Singleton: Sequence Diagram Draft

Singleton pattern – Diagram of sequence

- **Not** an accurate Sequence Diagram.

Adapter Pattern

- Adapter: represents the implementation of the Target, hide details of Adaptee; e.g. Rectangle
- Adaptee: represents the class with the incompatible interface; e.g. LegacyRectangle
- Target: e.g. Shape

Adapter: Class Diagram Draft

Adapter pattern – Class diagram

Adapter: Sequence Diagram Draft
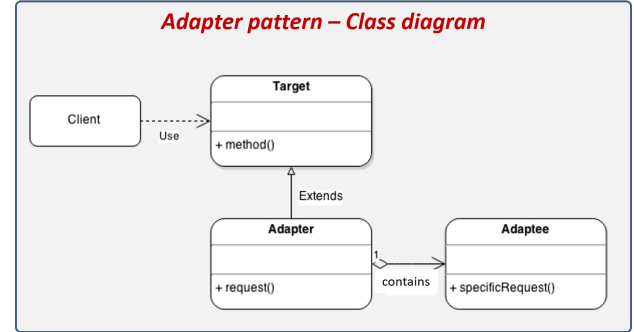
Adapter pattern – Sequence diagram
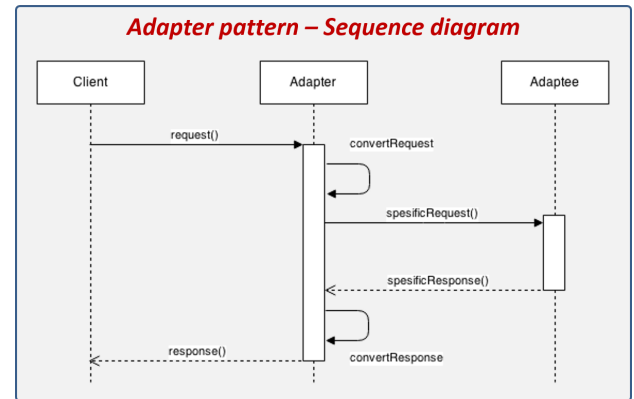
- **Not** an accurate Sequence Diagram.

Adapter: a.k.a. Wrapper
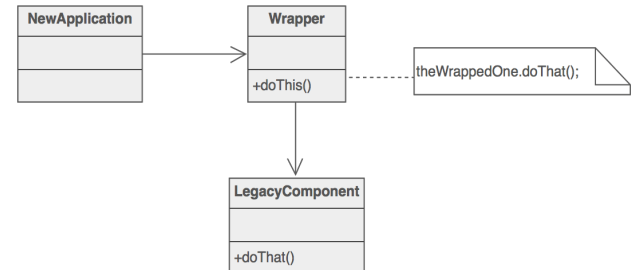
Façade Pattern
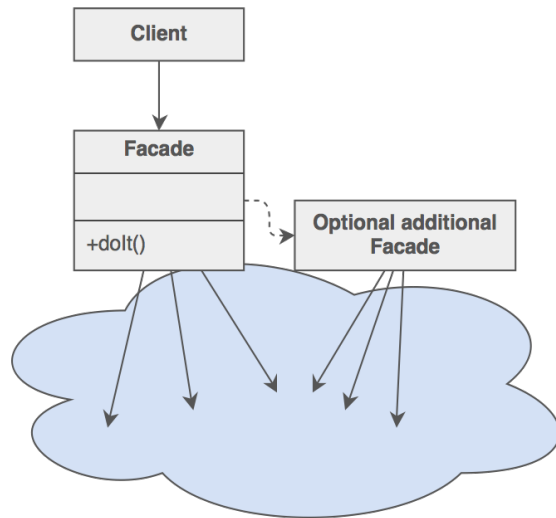
- The Façade defines a unified, higher level interface to a subsystem that makes it easier to use.
- IFacade: high-level interface, hiding the complexity of interacting with multiple systems.
- DefaultFacadeImpl: implementation of IFacade, in charge of communicating with all the subsystems.
- Subsystems: represents all the modules or subsystems with interfaces for communication.
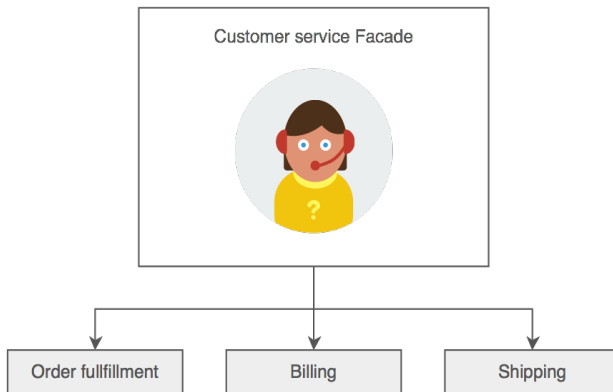
- As an example, the customer-service system could be incredibly complex without Façade.

Façade: Class Diagram Draft
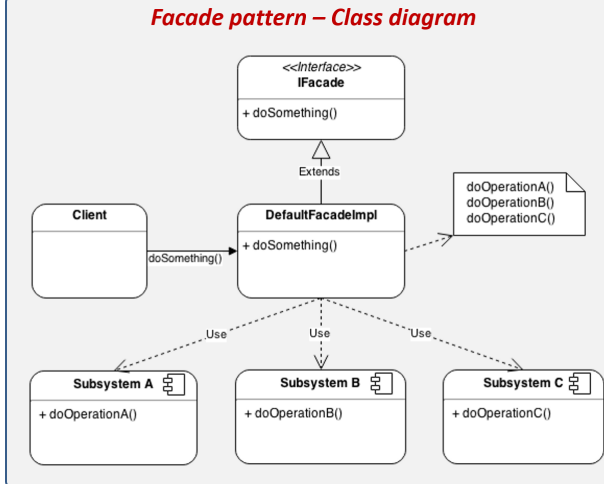
Façade: Sequence Diagram Draft
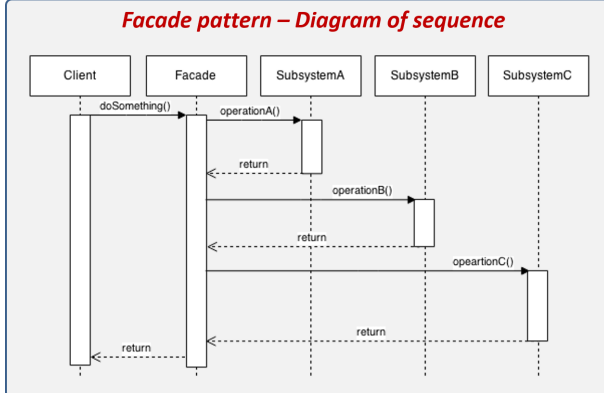
- **Not** an accurate Sequence Diagram.

Flyweight Pattern
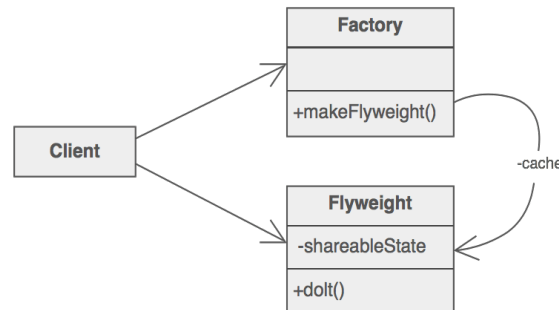
Flyweight Pattern

- FlyweightFactory: factory class for building the Flyweight objects.
- Flyweight: the objects we want to reuse in order to create lighter objects.

Flyweight: Class Diagram Draft
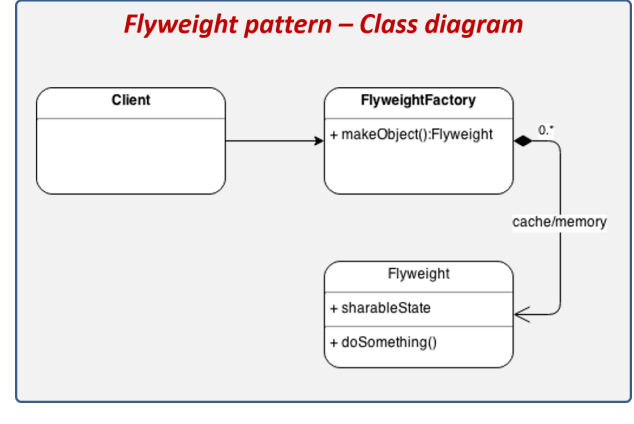
Flyweight: Sequence Diagram Draft
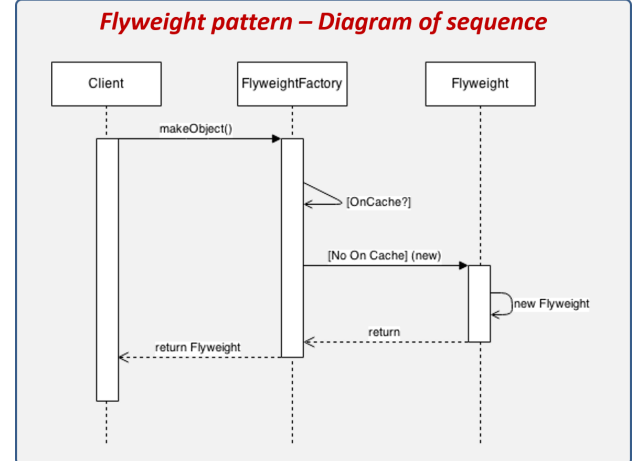
- **Not** an accurate Sequence Diagram.

Strategy Pattern
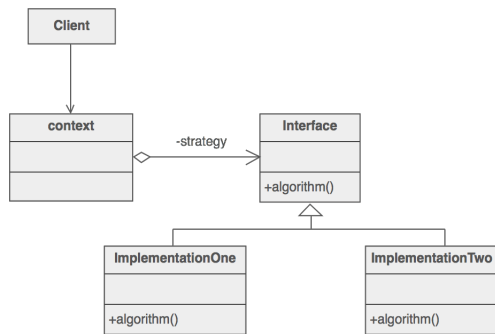
- Strategy Interface: define the common interface of all strategies that must implement.
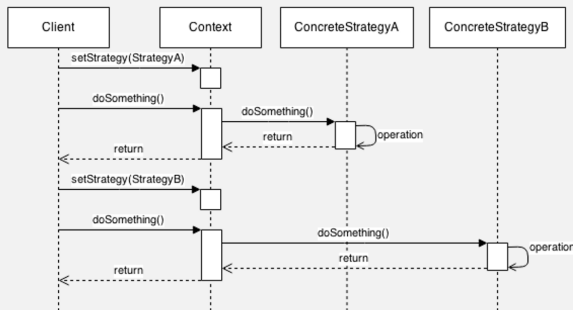- Concrete Strategy: inherit from Strategy Interface, they implement concrete strategies.

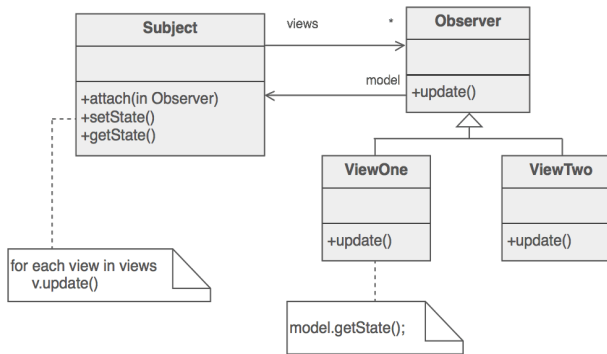## Strategy: Class Diagram Draft

Strategy pattern – Class diagram
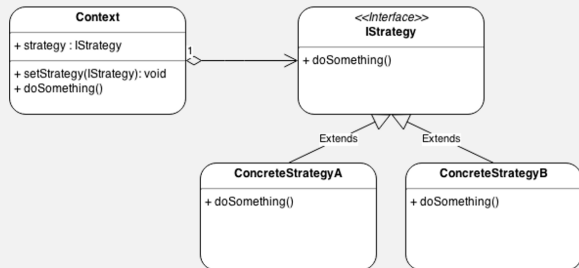
Strategy: Sequence Diagram Draft

Strategy pattern – Diagram of sequence

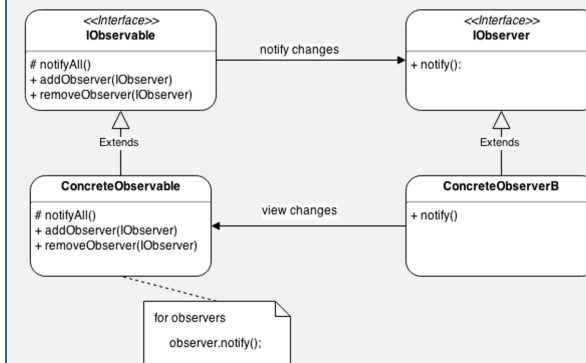- **Not** an accurate Sequence Diagram.

Observer Pattern

- Subject: interface of all observable subject classes, in it, methods that (1) keep track of observers listening to itself (2) notify the observers when change happens, are defined.
- Concrete Subject: the observable class; it implements all methods defined in Subject interface.
- Observer: interface observing the changes on Subject.
- Concrete Observer: Concrete class watching the changes on Subject, inherits from Observer, implements its methods.

It defines a one-to-many dependency between objects so that when one object (a concrete observable subject) changes state, all of its dependents (corresponding concrete observers) are notified and updated automatically.
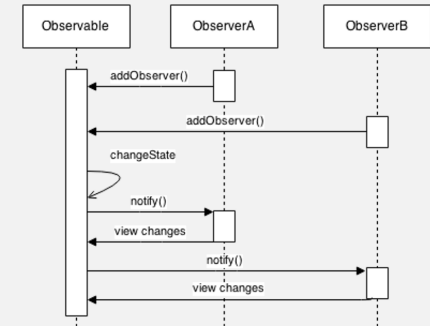
Observer: Class Diagram Draft

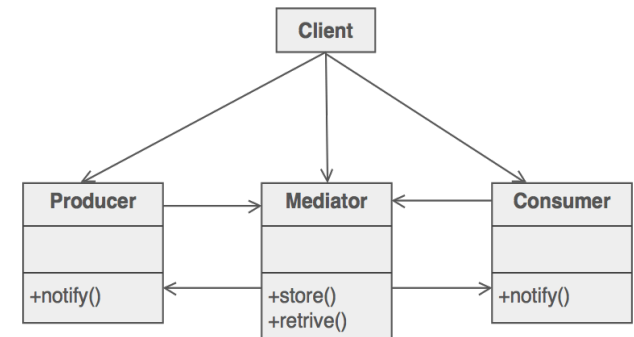
Observer pattern – Class diagram

Observer: Sequence Diagram Draft

Observer pattern – Diagram of sequence

- **Not** an accurate Sequence Diagram.
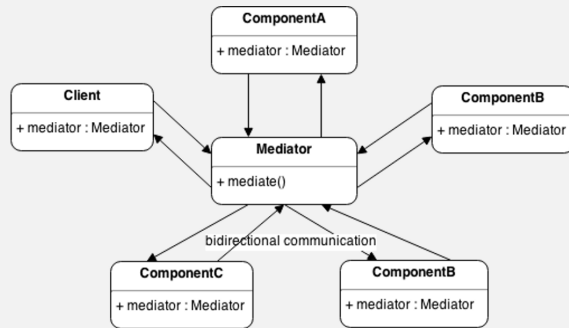
Mediator Pattern

- Mediator: defines the interface for communication between colleague objects.
- Concrete Mediator: implements the mediator interface and coordinates communication between colleague objects.
- Colleague (Peer): defines the interface for communication with other colleagues
- Concrete Colleague: implements the colleague interface and communicates with other colleagues through its mediator only; e.g. Producer, Consumer in the figure.

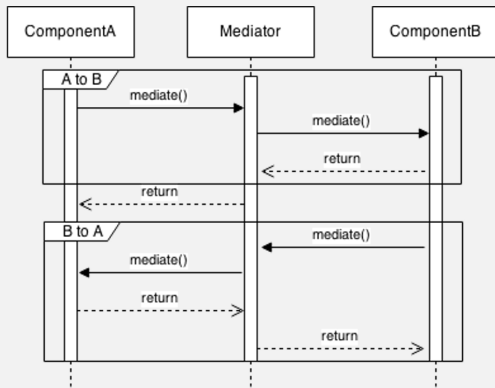Centralize many-to-many complex communications and control between related objects (colleagues).

Mediator: Class Diagram Draft

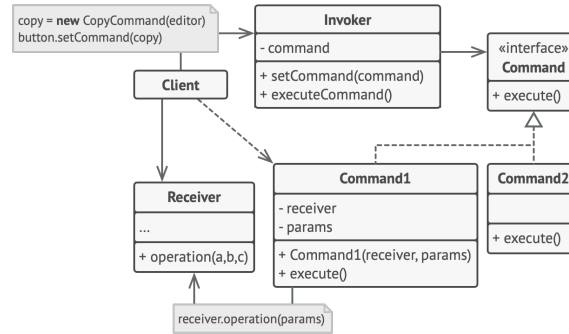

Mediator: Sequence Diagram Draft

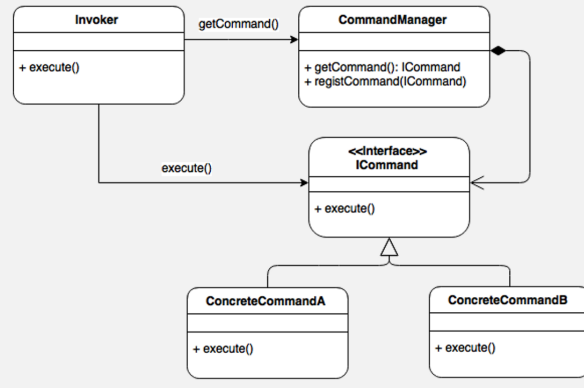- **Not** an accurate Sequence Diagram.

Command Pattern

- Command: interface describing the structure of the commands, defining the generic execution method for all of them (e.g. execute, undo).
- Concrete Command: inheriting from Command, each of these classes represents a command that can be executed independently.
- Receiver: informed by the Concrete Command and take actions.
- Invoker: the action triggering one of the commands, hold a command and at some point execute it.
- (optional) Command Manager: manage all the commands available at runtime, from here we create / request commands.

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.
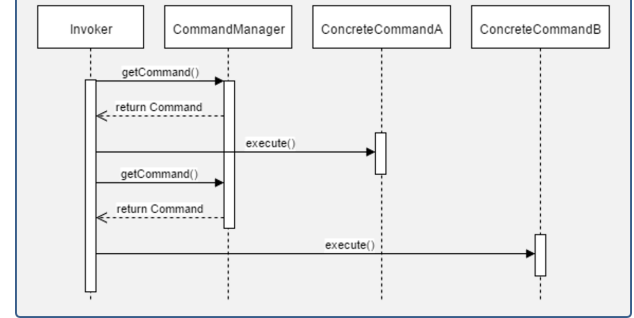
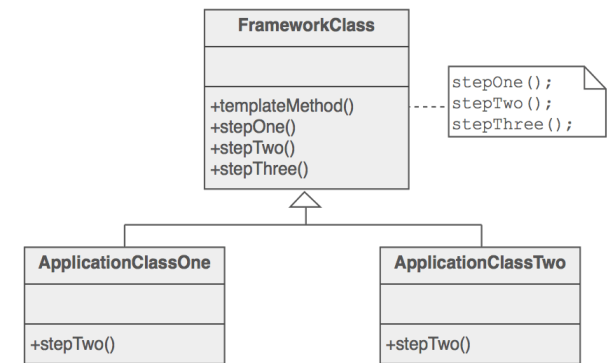## Command: Class Diagram Draft

Command: Sequence Diagram Draft

- **Not** an accurate Sequence Diagram.
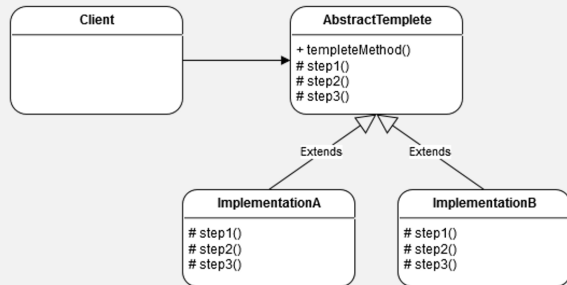
Template Method Pattern

- Abstract Template: an abstract class including a series of operations which define the necessary steps for carrying out the execution of the algorithm; e.g. Framework Class in the figure.
- Implementation: the class inherits from Abstract Template and implements its methods to complete the algorithm; e.g. Application Class One / Two in the figure.

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses; subclasses may redefine certain steps of an algorithm without changing its overall structure.
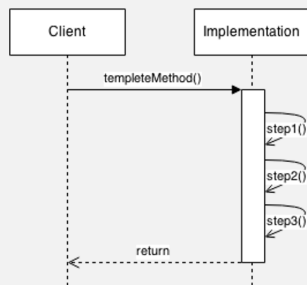
Template Method: Class Diagram Draft
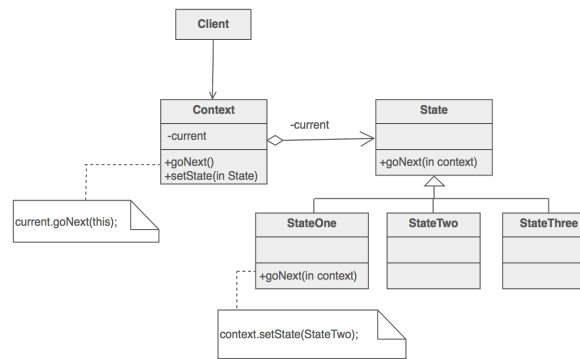
Templete Method – Class diagram

Template Method: Sequence Diagram Draft

Templete Method pattern – Diagram of sequence
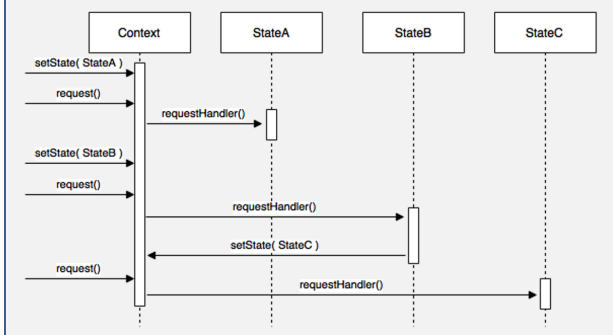

- **Not** an accurate Sequence Diagram.

State Pattern

- Context: the component subject to changing states, it has its current state as one of its properties; e.g. in a vending machine example, this would represent the machine.
- State: abstract base class used for generating different states, usually works better as an abstract class, instead of as an interface, because it allows us to set default behaviors.
- Concrete State: inherit from State, each one of these represent a possible state the application could go through during its execution.

State: Class Diagram Draft

State pattern – Class diagram

State: Sequence Diagram Draft

State pattern – Diagram of sequence

- **Not** an accurate Sequence Diagram.