CS2062

Introduction

Information Systems

- Information system: collects, processes, stores, and outputs information
- Subsystem: component of a system
- Components: hardware, software, inputs, outputs, data, people, and procedures
- Supersystem: collection of systems
- Automation boundary: separates automated part of system from manual

Systems Development Life Cycle (SDLC)

- Process of building, deploying, using, and updating an information system
- Variations of SDLC
    - Predictive: project planned entirely in advance
    - Adaptive: planning leaves room for contingencies

- Pure approaches to SDLC are rare

Traditional Predictive SDLC Approaches

- Planning, analysis, design, implementation, support

Phase	Objective
Project planning	Identify the scope, ensure feasibility
Analysis	Understand and document the business needs

Phase	Objective
Design	Design the solution system based on requirements
Implementation	Build, test, and install the system with trained users
Support	Keep the system running productively

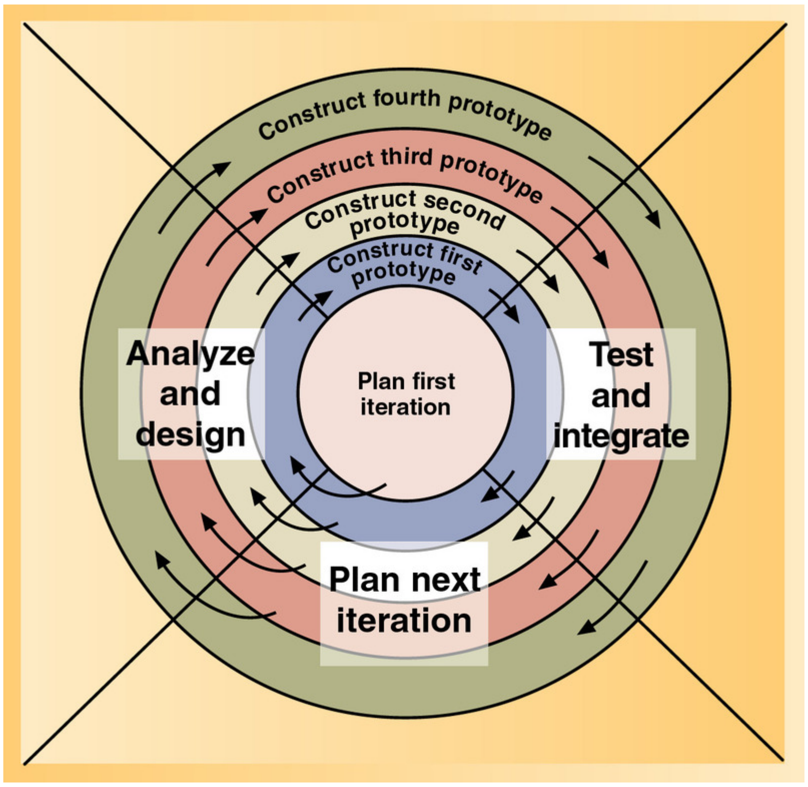
- Pure waterfall approach
  - Assumes project phases can be sequentially executed

- Modified waterfall approach
  - Tempers pure waterfall by recognizing phase overlap

Newer Adaptive Approaches

- Spiral model
  - Activities radiate from centre starting point

- Iterative problem solving
- Approaches to structuring iterations
  - Define and implement the key system functions
  - Focus on one subsystem at a time
  - Define by complexity or risk of certain components
  - Complete parts incrementally


Construct fourth prototype
Construct third prototype
Construct second prototype
Construct first prototype
Analyze and design
Plan first iteration
Test and integrate
Plan next iteration

Unified Process Life Cycle

- Includes 4 phases, consisting of iterations
- Inception: develop and refine system vision
- Elaboration: define requirements and core architecture
- Construction: continue design and implementation
- Transition: move the system into operational mode

Models

- Models abstract separate aspects of the real world
- UML: standard notation
- PERT or Gantt charts: model project itself

Techniques

- Collection of guidelines
- Proven techniques are embraced as “Best Practices”

Tools to support system development

- CASE (Computer Aided System Engineering)
- Variations on CASE
  - Visual modelling tools
  - Integrated application development tools
  - Round-trip engineering tools

Requirements

Gather detailed information

- Dialog with users of new system and similar systems
- Develop expertise in business domain

Define requirements

- Models record/communicate functional requirements
- Modelling continues while information is gathered

Prioritize requirements

- Scarcity of resources limit function implementation
- Scope creep: tendency of function list to grow
- Scope creep adversely impacts project
  - Cost overruns
  - Implementation delays

- Solution: prioritization of functions

Evaluate requirements with users

- Models built and validated as per user requirements
- Process is iterative

System requirements

- System requirements consist of capabilities and constraints
- Functional
  - Directly related to use cases
  - Documented in graphical/textual models

- Nonfunctional
  - Performance, usability, reliability, security
  - Documented in narrative descriptions to models

Models

- Models are great communicators
- UML activity diagram is a one type of model

Purpose of Models

- Modelling is a dynamic process
  - Draws together various team members and users
  - Simulates electronic execution of tasks
  - Spurs refinement and expansion of requirements
  - Promotes informal training

Types of Models

- No universal models
- Models chosen based on nature of information
- Categories
  - Mathematical
  - Descriptive
  - Graphical

Use Cases and Domain Classes

Events and Use Cases

- Use case
  - Activity the system carries out

    - Entry point into modelling process

- Identifying use cases
    - Event decomposition
    - Elementary business process (EBP)
        - A task performed by one person at a given time in response to a business event
        - Leaves the system in a stable state after completion

Event Decomposition

- Develops use cases based on system response to events
- Perceives system as black box interfacing with external environment
- Keeps focus on EBPs and business requirements

Types of Events

- External events: occur outside the system, and usually caused by external agent
- Temporal events: occurs when a system reaches a point
- State events: asynchronous events responding to system trigger

Identifying Events

- Distinguish events from prior conditions and responses
- Trace sequence of events initiated by external agent
- Identify technology dependent events
- Defer specifying technology dependent events
- Perfect technology assumption

Event Table

Event	Trigger	Source	Use Case	Response	Destination

Event

The event that causes the system to do something

Trigger

How does the system know the event occurred?

- External: data entering the system
- Temporal: definition of the point that triggers the system processing

Source

For external: the external agent is the source of the data entering the system

Use Case

What does the system do when the event occurs?

- This is what is important to define for functional requirements

Response

What output (if any) is produced by the system?

Destination

What external agent gets the output produced?

Use Case Modelling

System Processes

- Define use cases into two tiers
  - Overview level derived from event table and use case diagrams
  - Detailed level derived from combination of:
    - Use case description
    - Activity diagram
    - Sequence diagram

Use Case Diagram

Representation	Meaning
Simple stick figure	Actor
Actor’s hands	Direct system access
Oval	Use case
Connecting lines	Match actors to use cases

Use Cases and Actors

- Source
  - Person/thing initiating the business event
  - Must be external to the system

- Actor
  - Person/thing that touches the system
  - Lies outside of automation boundary

Automation boundary

- Line drawn around the entire set of use cases
- Defines interface between actors and computer system

includes Relationships

- **includes** or **uses** relationship
  - Use case calling services of common subroutine

    - Common subroutine itself becomes additional use case

- Base use case *cannot* be completed without included use case
- Notation
    - Relationship denoted by connecting line with arrow
    - Direction of the arrow indicates major/minor cases

extends Relationships

- Similar to includes, but the base use case *can* be completed without the extending use case

includes and extends

Notation: dashed arrows

- Includes: From base use case to included use case
- Extends: From extending use case to base use case

Developing a Use Case Diagram

- To identify additional use cases, divide one large use case into two, or define another use case based on a common subroutine
- Distinguish between temporal and state events

Use Case Detailed Descriptions

- Written at 3 levels of detail
    - Brief description: summary statement conjoined to activity diagram
    - Intermediate description: expands brief description with internal flow of activities
    - Fully developed description: expands intermediate description for richer view

- Fully developed use case description
    - Consists of eleven compartments
    - User, actor, stakeholder, EBP, and conditions identified

- Compartments are: Use case name, Scenario, Triggering event, Brief description, Actors, Related use cases, Stakeholders, Preconditions, Postconditions, Flow of events, Exception conditions

Activity Diagram

- Documents requirements of business process/transaction workflows
- A type of workflow diagram that describes the users, activities, and their workflows
- Workflow: Sequence of steps to process a business transaction

Problem Domain Classes

Developing initial list of things

- List nouns users mention when discussing system
- Event table as source of potential things
- Select nouns with questions concerning relevance

Drawing Class Diagrams

Class diagrams

State-chart Diagrams

State and Transition

- State is a semi-permanent condition of an object
- External event can change the current state of an object

- Object maintains its state until an event occurs, moving it to another state
- These movements are called transitions
- Transitions are considered to be short compared to states
- Transition name is the trigger/message event which caused the object to transition

Guard Condition and Action Expression

- Guard condition is a true/false condition which acts as a qualifier for the transition
- Condition should be true for the transition trigger to fire
- Action expression is a procedural expression that executes when the transition trigger fires

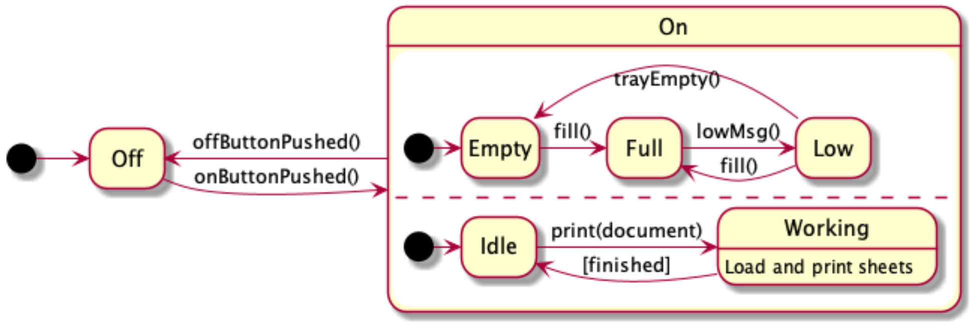
Identifying States

- Simple states reflect simple conditions, such as “On”
- Complex states labelled with gerunds or verb phrases, such as “Being shipped”
- Active states usually not labelled with nouns
- Status conditions reported to management/customers, such as “Shipped”

Concurrency

- Condition of being in more than one state at a time
- Two modes of representation
  - Use synchronization bars and concurrent paths
  - Nest low-level states inside higher-level states

- Higher-level states are also called *composite states*
  - Complex structure of sets of states and transitions
  - Represent a higher level of abstraction



Rules

1. Select the classes that will require state charts
2. List all the status conditions for selected classes
3. Specify transitions that cause object to leave the identified state
4. Sequence state-transition combinations in correct order
5. Identify concurrent paths
6. Look for additional transitions
7. Expand each transition as appropriate
8. Review and test each statechart

Multi-Tier Architectures

Single Tier Architecture

- Mainframe systems
- Everything is processed inside a single machine, and all resources are attached to it
- Accessed via dumb terminals

Advantages

- Simple
- Efficient
- Uncomplicated

Disadvantages

- Less flexibility
- Missing two-way interaction

2-Tier Architecture

- Client-server model
- Logical system components are mostly on the client (thick client)

Advantages

- More flexible
- Separation of concerns

Disadvantages

- Only a limited number of clients can be connected to the server at a given time

3-Tier Architecture

- Similar to traditional client/server architecture, but there is now an application server to handle the business logic
- Client takes in user requests and displays the GUI and data, but has no part in producing results (thin client)

Tier (Layer)

- Each tier should be able to be constructed separately
- Several tiers should be able to be joined together to make a whole system
- Each tier has a dedicated set of responsibilities
- There must also be some sort of boundary between tiers
- Each tier should not be able to operate independently without interaction with other tiers
- It should be possible to swap one tier with an alternative component which has similar characteristics

Advantages

- Scalability
- Reusability through separation of concerns
- Data integrity (only valid data passes through application server)
- Better security (client can't access data directly)
- Reduced distribution (business logic is only modified on the application server)
- Improved availability (can use redundant application/database servers)

Disadvantages

- Increased complexity

Design Patterns

Types of Design Patterns

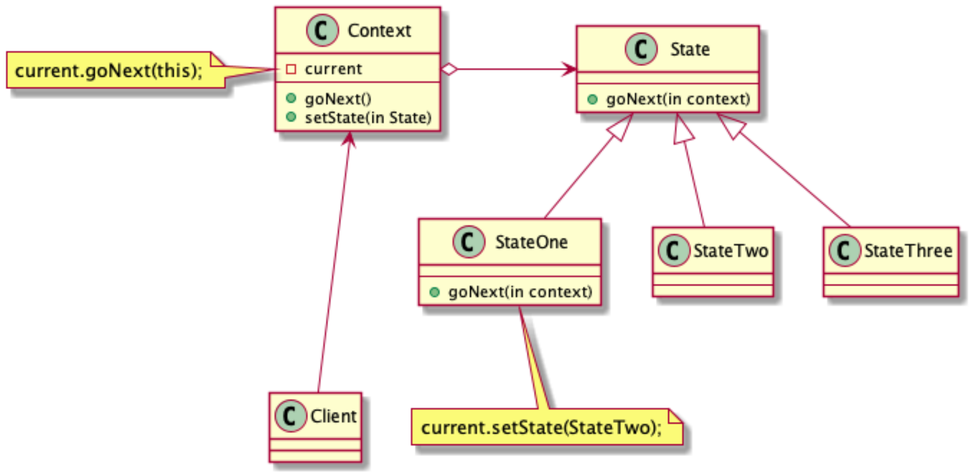
GoF Patterns

- Creational: Help assign responsibilities to classes to instantiate new objects
- Behavioural
- Structural: Concerned with how classes and objects are composed to form larger structures

Behavioural Design Patterns

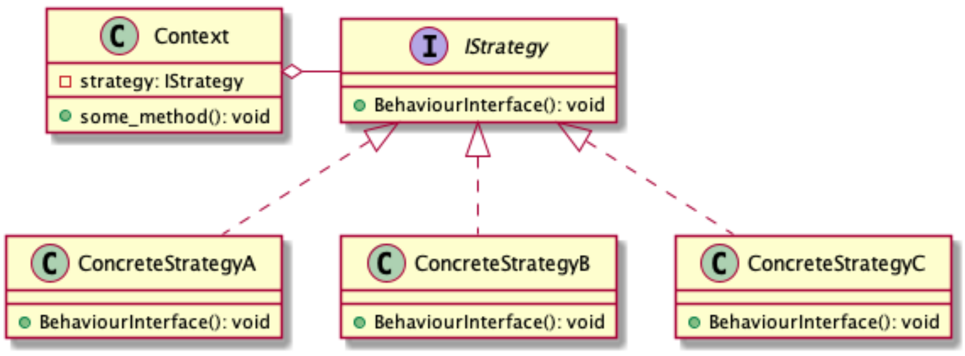
State

- Object's behaviour is a function of its state, and it must change its behaviour at run-time depending on that state
- State pattern allows an object to alter its behaviour when its internal state changes



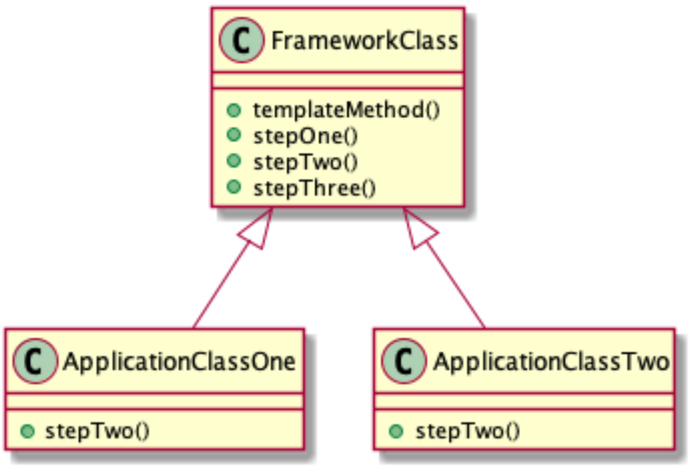
Strategy

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Client has the ability to select different algorithms at runtime



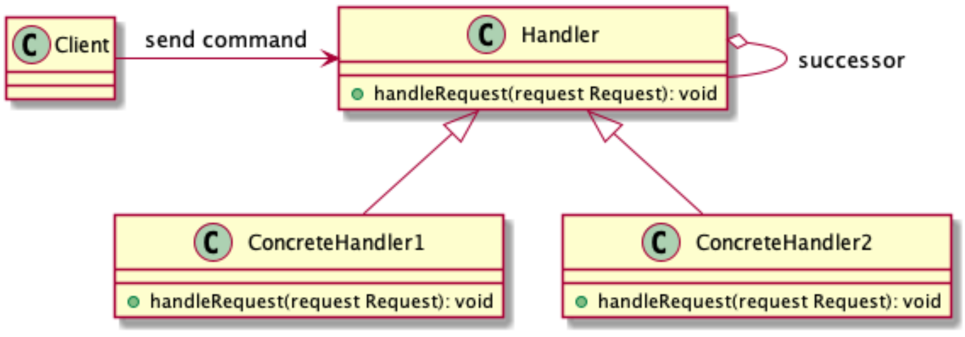
Template

- Template: a preset format, used as a starting point for a particular application, so that the format does not have to be recreated each time it is used
- This pattern defines an algorithm in a base class using abstract operations that subclasses override to provide concrete behaviour



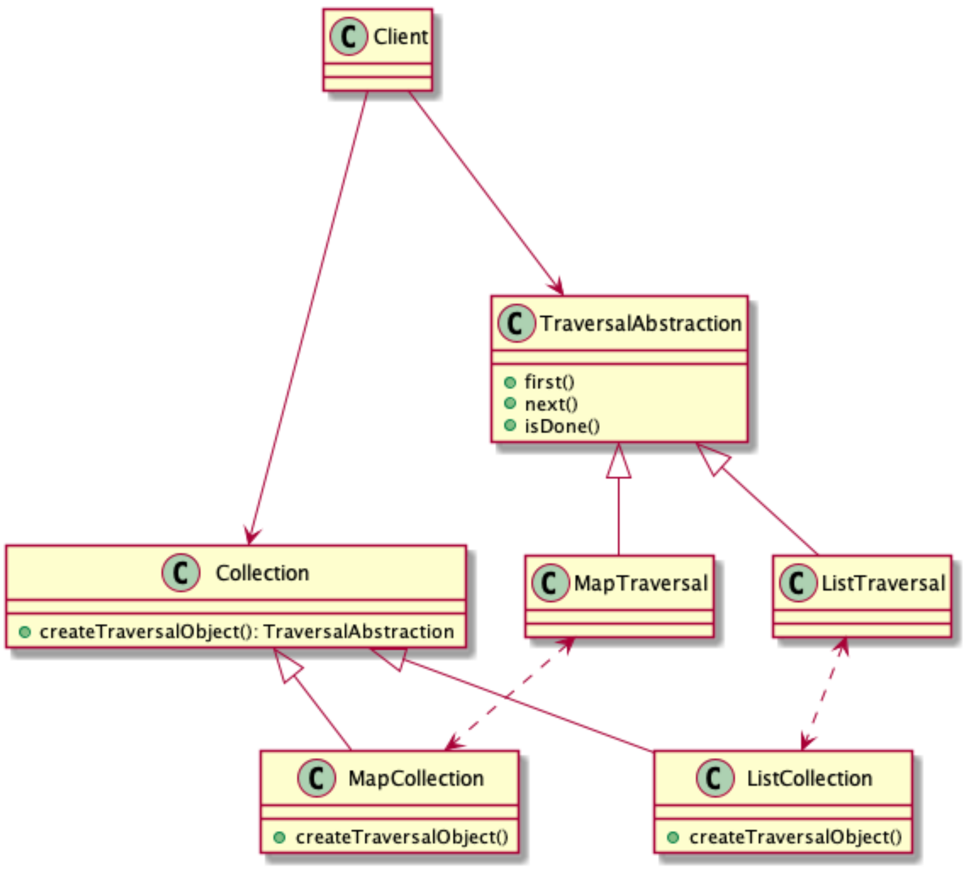
Chain of Responsibility

- Use to avoid coupling the sender of a request to its receiver
- Chains the receiving objects and passes the request along the chain until an object handles it
- Effective when
    - more than one object can handle a command
    - handler is not known in advance
    - handler should be determined automatically
    - request needs to be addressed to a group of objects without specifying its receiver
    - the group of objects that may handle the command must be specified in a dynamic way


Client
send command
Handler
handleRequest(request Request): void
successor
ConcreteHandler1
handleRequest(request Request): void
ConcreteHandler2
handleRequest(request Request): void

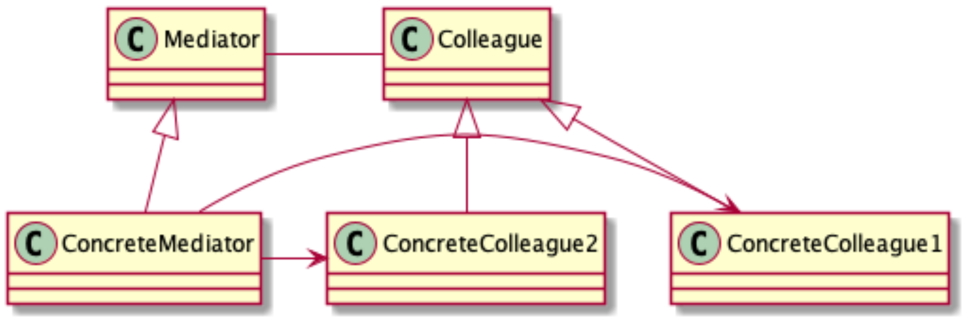
Iterator

- Use when the traversal of different data structures should be done in an abstract way
- Polymorphic traversal



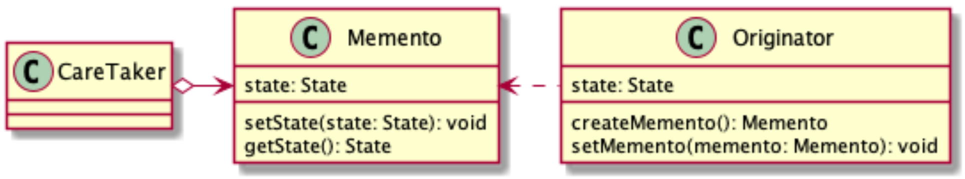
Mediator

- Define an object that encapsulates how a set of objects interact
- Promotes loose coupling by keeping objects from referring to each other explicitly
- Lets you vary their interaction independently



Memento

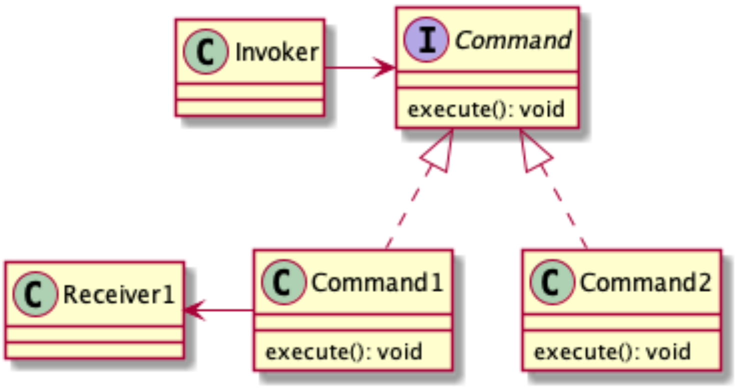
- Use to restore an object back to one of its previous states
- Capture and externalize an object's internal state so that the object can be returned to this state later
- Encapsulation is not violated



Observer

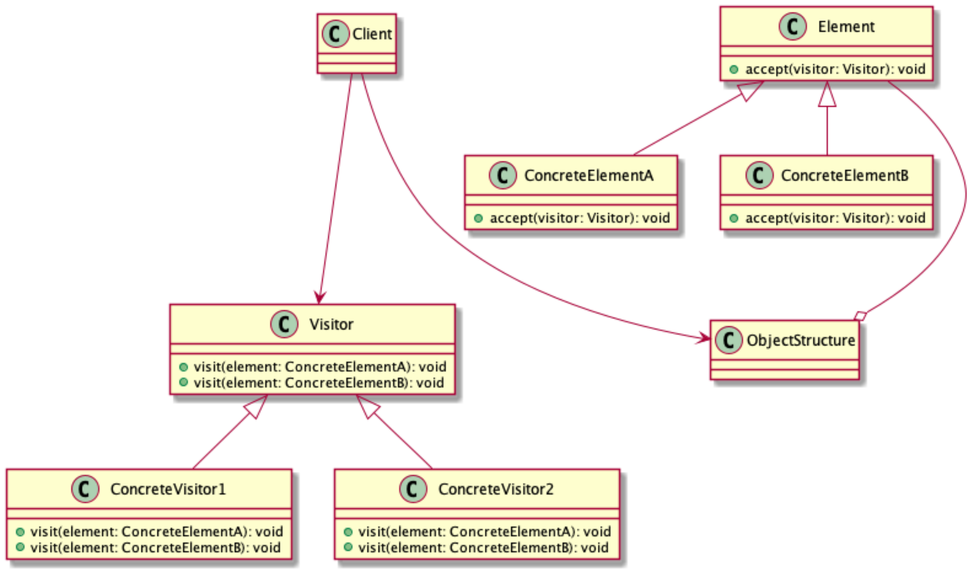
Command

- Use when you need to issue requests/commands to objects without knowing anything about the operation requested or the receiver of the request
- It decouples the object that invokes the operation from the one that knows how to perform it



Visitor

- Collections contain objects of different types, where some operations have to be performed on all the elements without knowing the type
- Visitor pattern represents an operation to be performed on the elements of an object structure

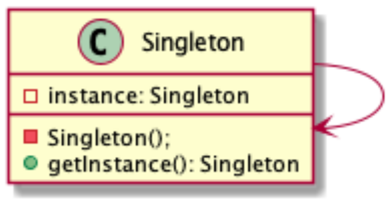

Client
Element
accept(visitor: Visitor): void
ConcreteElementA
accept(visitor: Visitor): void
ConcreteElementB
accept(visitor: Visitor): void
Visitor
visit(element: ConcreteElementA): void
visit(element: ConcreteElementB): void
ObjectStructure
ConcreteVisitor1
visit(element: ConcreteElementA): void
visit(element: ConcreteElementB): void
ConcreteVisitor2
visit(element: ConcreteElementA): void
visit(element: ConcreteElementB): void

Creational Design Patterns

Singleton

- Use when an application needs *only one* instance of a class

- Provide a global access point to it
- Initialization on first use and reuse thereafter



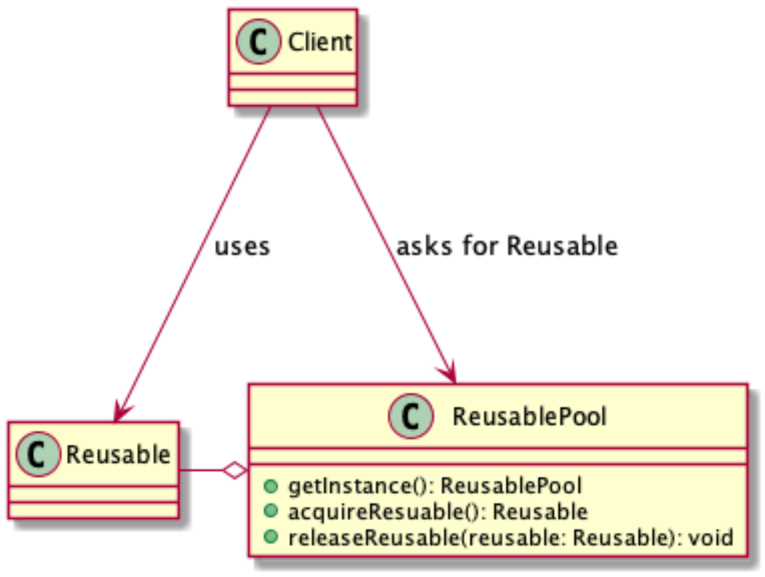
Multiton

- When a request is made for an object, a key is passed to the static method
- If the key has not been used before, a new object is instantiated, linked to the key and returned
- If the key has been used before, the object previously linked to that key is returned
- Multiton provides the functionality of a group of singletons

Object Pool

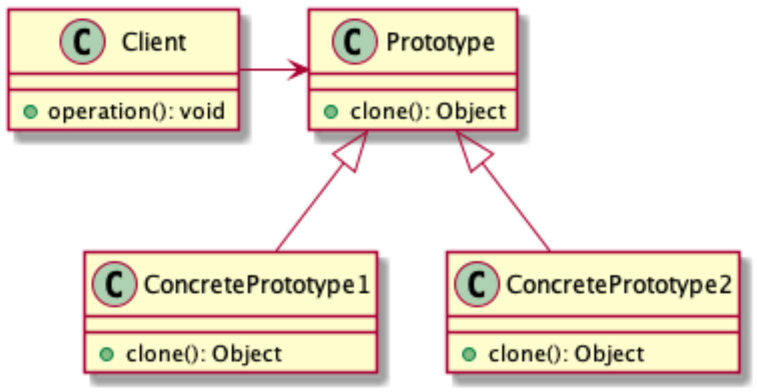
- Allows to reuse objects that are expensive to create and maintain
- When a client requests an object,
  - see if an object can be returned from the pool
  - if not, see if a new one can be created
  - if cannot, wait till one gets released

- Once used, client should mark the object as released


Client
uses
asks for Reusable
Reusable
ReusablePool
getInstance(): ReusablePool
acquireResuable(): Reusable
releaseReusable(reusable: Reusable): void

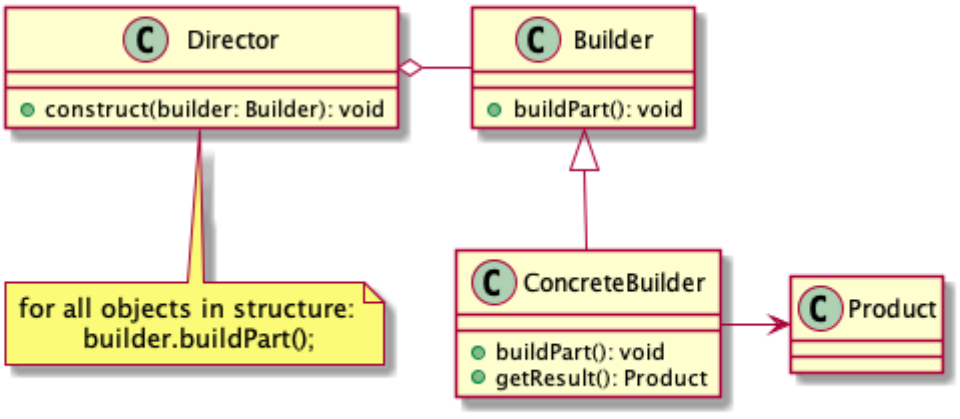
Prototype

- Use when creating an object is costly
- Rather than creating a new object, clone an existing object
- Allows creation of objects without knowing any details of how to create them



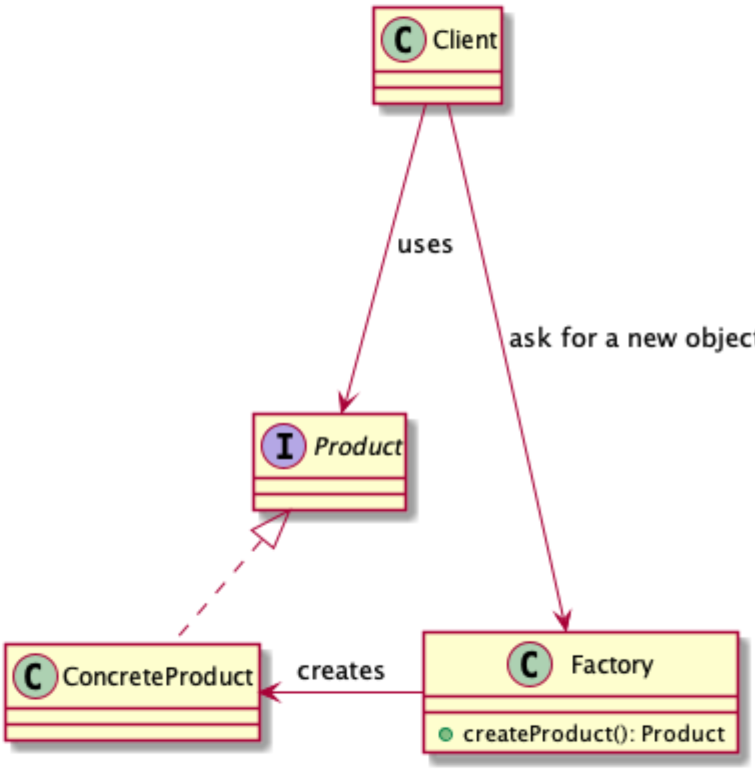
Builder

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



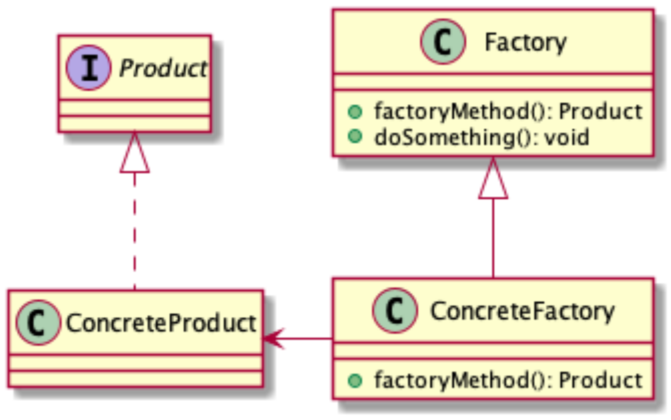
Factory

- Use when objects have to be created for the client without exposing the logic to the client
- Object is referred through a common interface



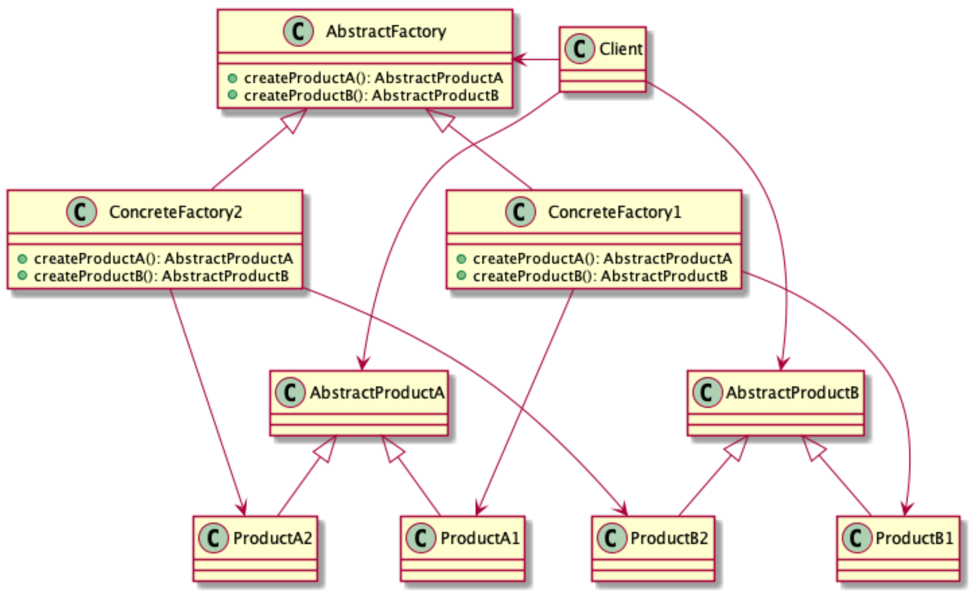
Factory Method

- AKA Virtual constructor
- Similar to factory
- Defines an interface for creating an object
- Lets subclasses decide which class to instantiate



Abstract Factory

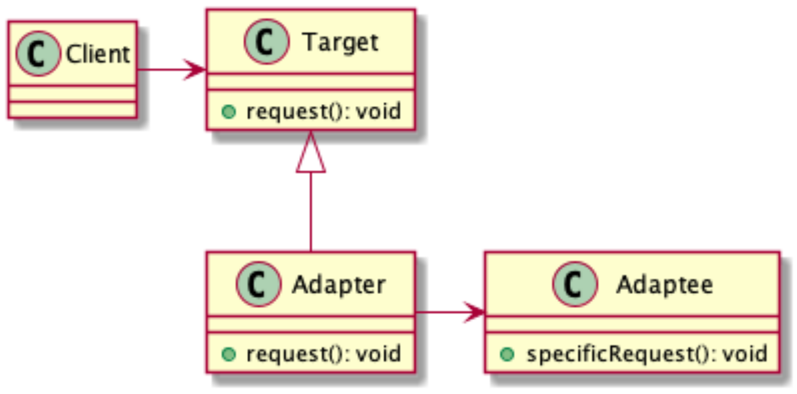
- AKA Kit
- Offers the interface for creating a family of related objects, without explicitly specifying their classes
- Looks similar to Factory method


AbstractFactory
createProductA(): AbstractProductA
createProductB(): AbstractProductB
Client
ConcreteFactory2
createProductA(): AbstractProductA
createProductB(): AbstractProductB
ConcreteFactory1
createProductA(): AbstractProductA
createProductB(): AbstractProductB
AbstractProductA
AbstractProductB
ProductA2
ProductA1
ProductB2
ProductB1

Structural Design Patterns

Adapter

- AKA Wrapper
- Use when an external component is not compatible with the current system
- Convert the interface of a class into another interface that clients expect
- Lets classes work together, that could not otherwise because of incompatible interfaces

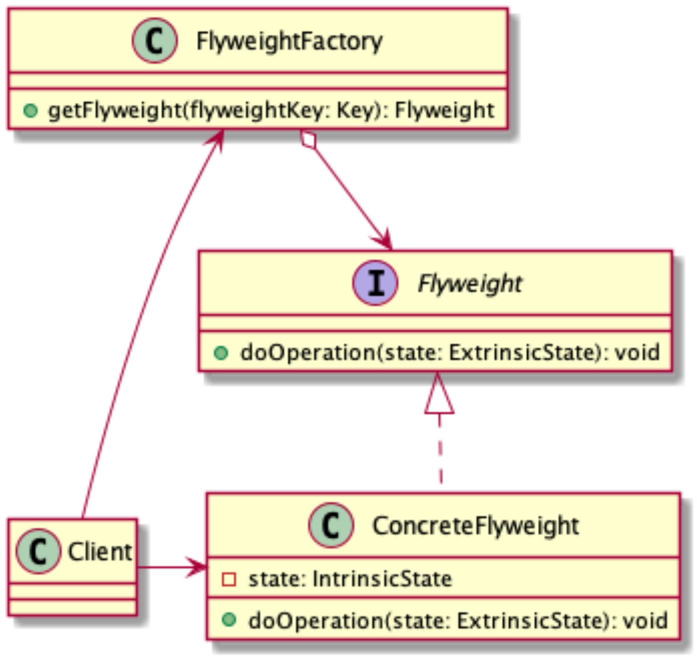


Facade

- Use when a segment of a client community needs a simplified interface to the overall functionality of a complex subsystem
- Facade defines a higher-level interface that makes the subsystem easier to use
    - Encapsulates a complex subsystem within a single interface object
    - Reduces coupling between different subsystems
    - Reduces the learning curve to use a subsystem
    - Caveat: limits the features and flexibility that "power users" may need

Flyweight

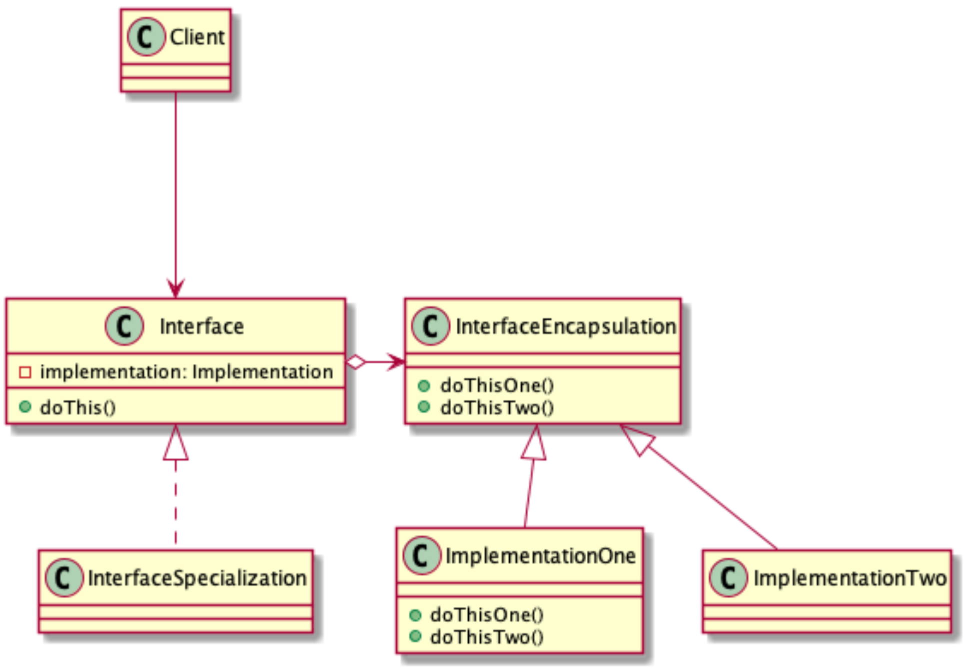
- Some programs require a large number of objects that have some shared state among them
- Maintains intrinsic state and provides methods to manipulate extrinsic state



FlyweightFactory
getFlyweight(flyweightKey: Key): Flyweight
Flyweight
doOperation(state: ExtrinsicState): void
Client
ConcreteFlyweight
state: IntrinsicState
doOperation(state: ExtrinsicState): void

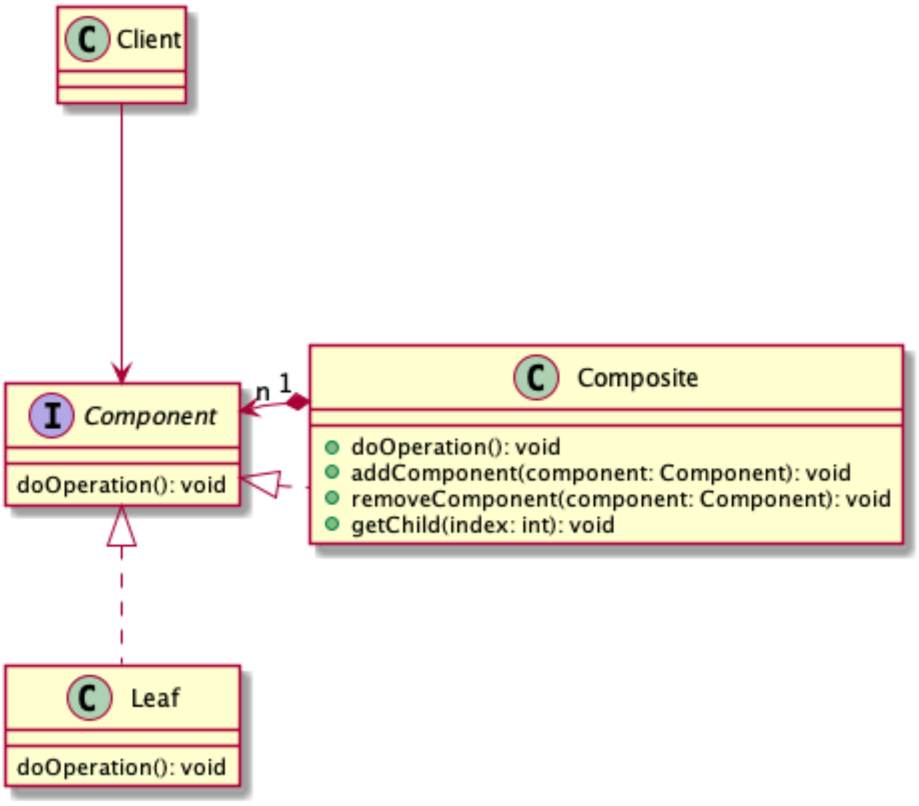
Bridge

- AKA Handle/Body
- Decouple abstraction from implementation so that the two can vary independently
- Avoid compile-time binding between interface and implementation
- Can be used
  - if two orthogonal dimensions exist
  - to have run-time binding of the implementation



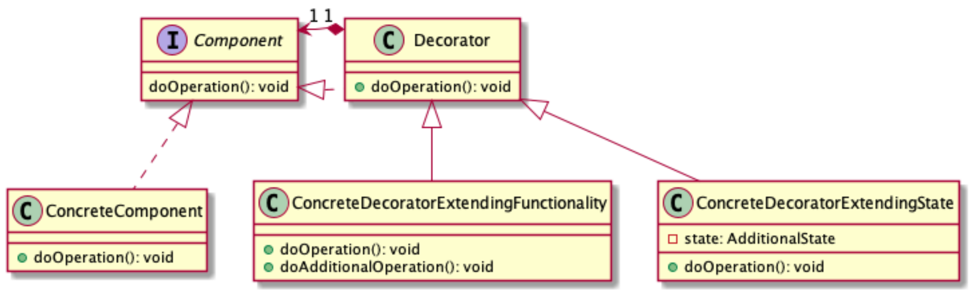
Composite

- Use when a group of objects should be treated in similar way as a single object
- Composes objects in term of a tree structure to represent whole-part hierarchy
- Lets clients treat individual objects and compositions of objects uniformly
- Recursive composition


C Client
I Component
doOperation(): void
n
1
C Composite
doOperation(): void
addComponent(component: Component): void
removeComponent(component: Component): void
getChild(index: int): void
C Leaf
doOperation(): void

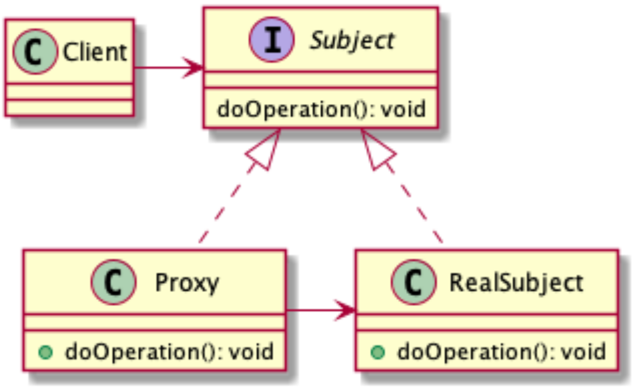
Decorator

- Similar to wrapper
- Use when new behaviour or state information should be added to objects at run time
- Wraps the original class and provides additional functionality without distorting the original



Proxy

- AKA Surrogate
- Useful to control access to an object
- Proxies are light objects exposing the same interface as the heavy objects
- They instantiate the heavy objects when needed


C Client
I Subject
doOperation(): void
C Proxy
doOperation(): void
C RealSubject
doOperation(): void