Software Design Pattern Process for KUPE

컴퓨터공학과
조원: 임담섭, 박문학, 추로요, 낭룽아웅

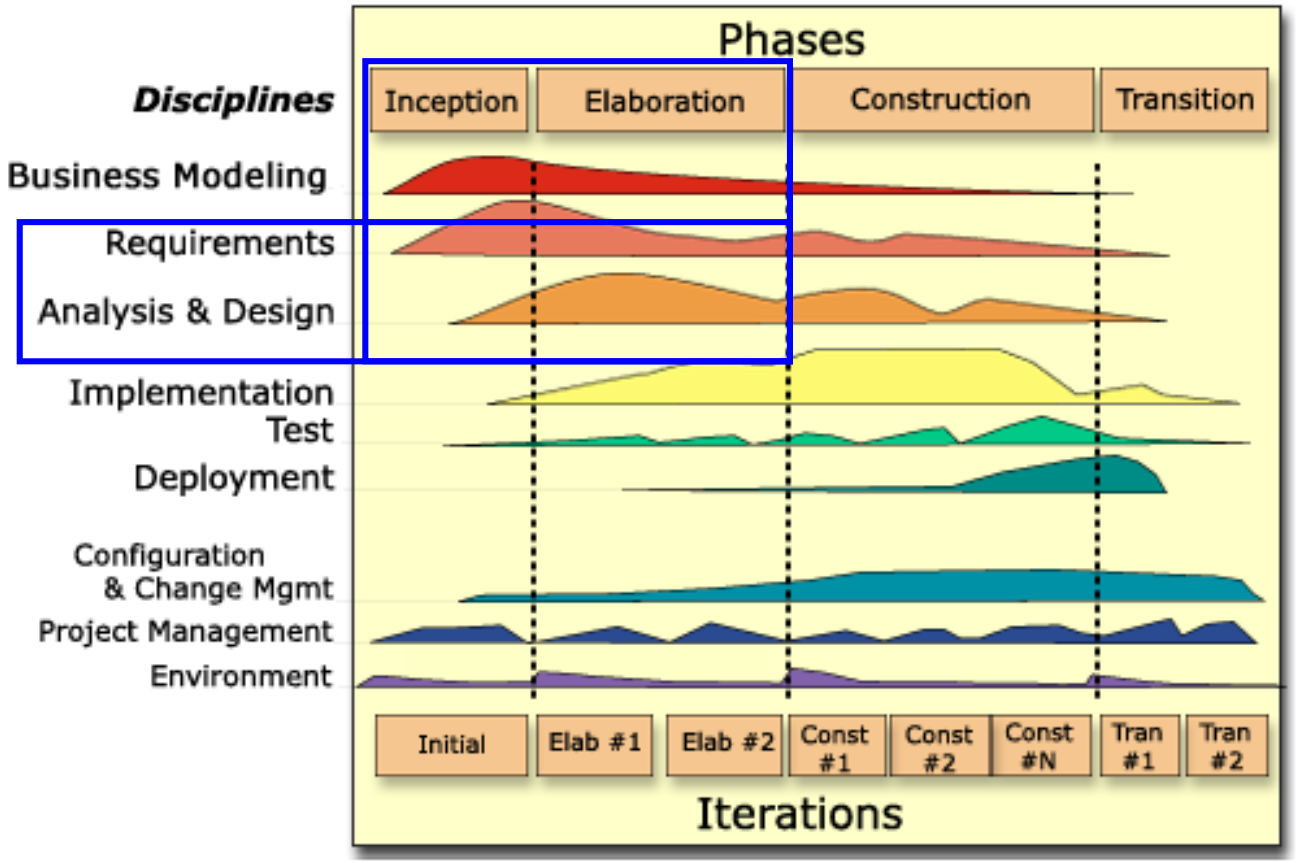
Software Design Pattern

- In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Ref. https://en.wikipedia.org/wiki/Software_design_pattern

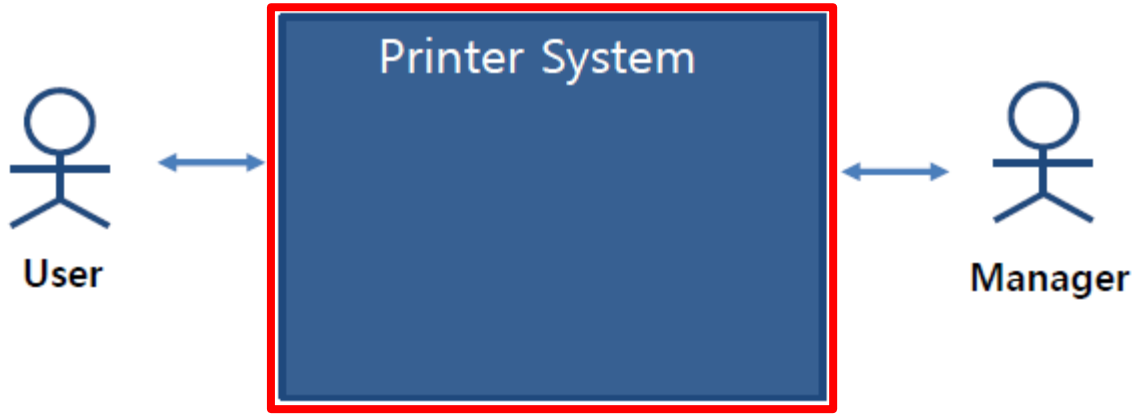
Software Design Pattern with RUP

- Software Design Pattern는 RUP 단계 중 주로 Requirements와 Analysis & Design에서 수행됨



Apply software design patterns to KUPE

- Software Design Pattern을 KUPE에 적용하는 목적은 Quality Requirement를 만족시키기 위함이라 생각함
  - Quality Requirement을 달성할 경우 제품의 차별성을 가질 수 있음
- 따라서, Quality Requirement를 추출하는 방법을 소개하고, KUPE의 Case Study의 Printer System에 간단히 적용해 보려고 함



Apply SDP to KUPE

- Non-functional Requirements를 추출하기위해서 소프트웨어 품질에 대한 국제표준인 ISO/IEC 9126에 따라 품질(Quality)을 적용하려고함

Activity 1003. Define Requirements

- Functional requirements
  - A requirement that specifies a function that a system or system component must be able to perform
  - Analyzed and Realized in Use-Case model
- Non-functional requirements
  - Constraints on the services or functions offered by the system as timing constraints, constraints on the development process, standards, etc.
  - Portability, Reliability, Usability, Efficiency(Space, Performance)
  - Delivery, Implementation, Standards
  - Ethical, Interoperability, Legislative(Safety, Privacy)
- Recommended reference : IEEE Std. 830-1998

Activity 1003. Define Requirements

- Steps
  1. Gather all kinds of useful documents
  2. Write an overview statement (objective and name of the system, etc.)
  3. Determine customers who use the product
  4. Write goals of the project
  5. Identify system functions
     - Functional requirements
     - Add function references(such as R1.1, ...) into the identified functions
     - Categorize identified functions into Event, Hidden, and Frill
  6. Identify system attributes
     - Non-functional requirements
  7. Identify other requirements (Optional)
     - Assumptions, Risks, Glossary, etc.

Categorization	
Event	should perform / visible to users
Hidden	should performs / invisible to users
Frill	optional

ISO/IEC 9126

ISO/IEC 9126 품질모델

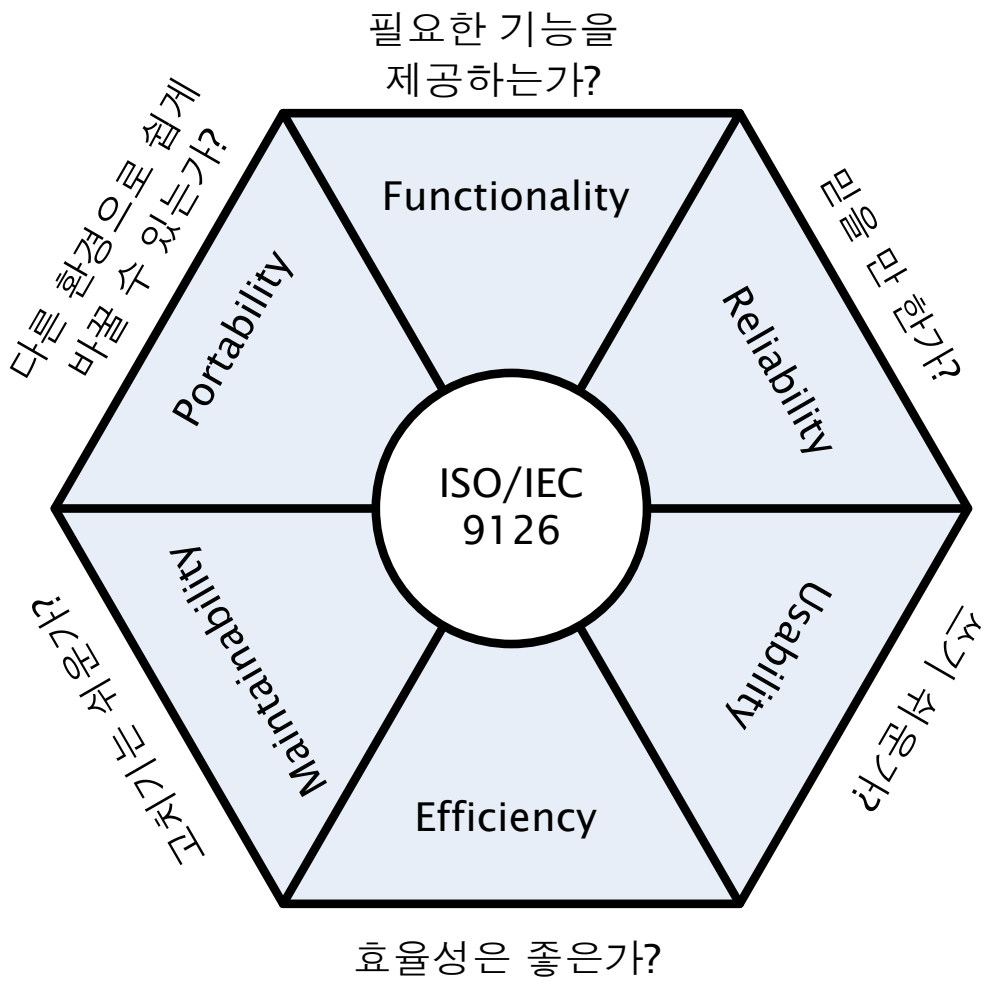
- 1998년도에 ISO와 IEC에 의해 제안된 품질 모델로서 소프트웨어 품질 인증을 위한 국제 표준
- 이후 **ISO/IEC 25010:2011 로 대체됨.**

- 품질 모델
  - 9126품질모델에서는 품질을 외부와 내부로 분리
  - 외부와 내부에 대해 각각 6개의 특징(Characteristics)들을 명시
  - 각 특징(Characteristics)들은 부 특성들(Sub Characteristics)로 나누어진다.
  - 각 특성들은 내부와 외부 Metrics에 의해서 측정

Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

품질모델

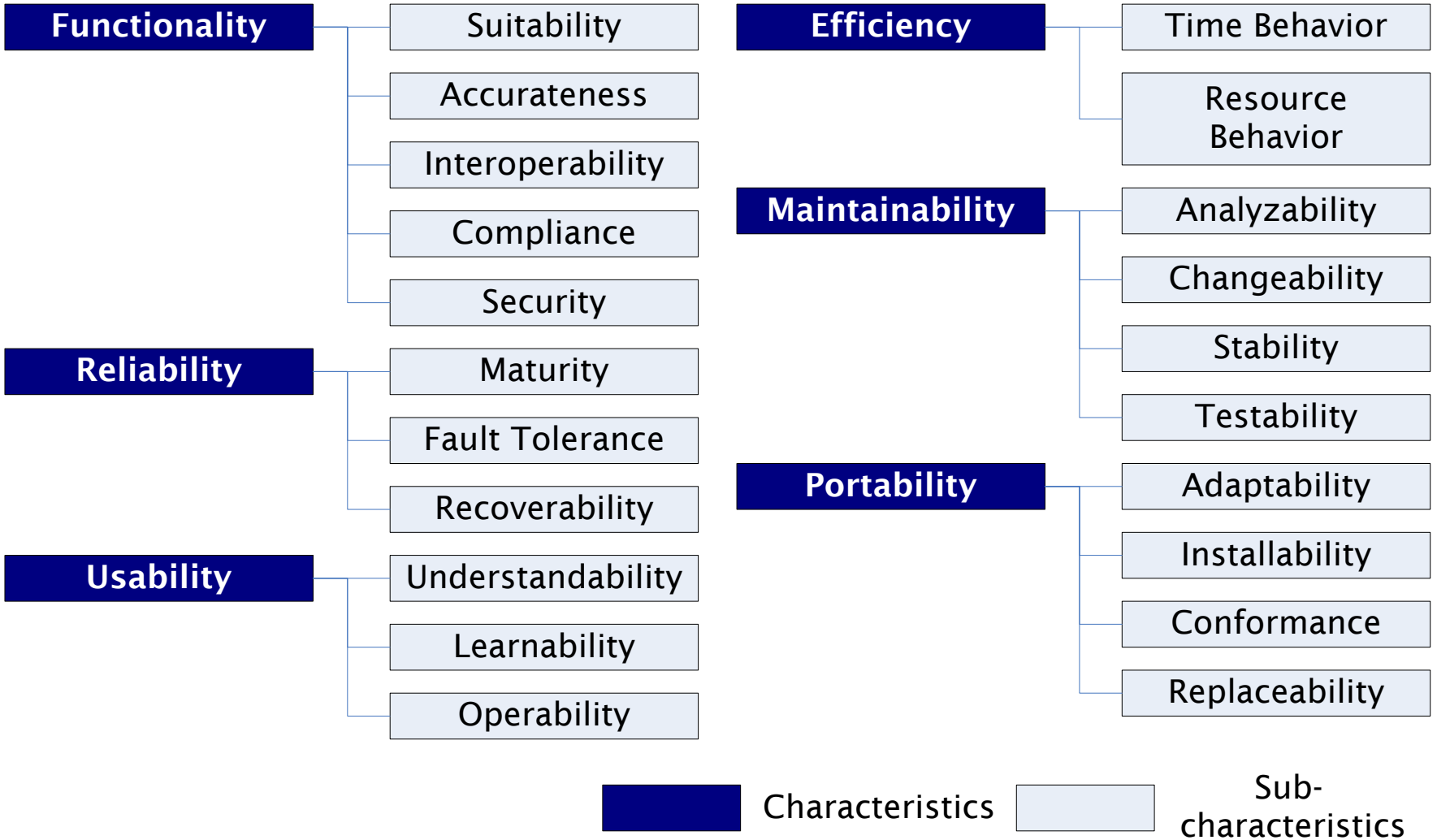
- ISO/IEC 9126



Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

품질모델(ISO/IEC 9126)

- Product's View of Quality



Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

품질모델 (Quality Model) - 1

- ISO/IEC 9126

기능성 Functionality	소프트웨어가 특정 조건에서 요구를 충족시키는 기능을 제공할 수 있는 능력	**적합성 Suitability**	특정 작업을 처리하는 기능의 적합성과 존재에 영향을 미치는 속성
		정밀성 Accurateness	합의한 결과나 효과 또는 명문화한 권리 조항에 영향을 미치는 속성
		상호운영성 Interoperability	정해진 소프트웨어와 상호 작용하는 능력에 영향을 미치는 속성
		준거성 Compliance	표준, 관습, 법률, 규격, 협정 같은 정해진 규칙을 준수하는 속성
		보안성 Security	소프트웨어의 프로그램이나 데이터에 사고든 고이든 적법하지 않게 접근하는 것을 막을 수 있는 능력에 영향을 주는 속성
신뢰성 Reliability	소프트웨어가 특정 상황에서 특정 기간 동안 일정 수준 이상으로 동작할 수 있는 능력	**성숙도 Maturity**	해결하지 못하고 남아있는 장애 때문에 생기는 장애 빈도에 영향을 주는 속성
		오류 허용성 Fault tolerance	소프트웨어에 오류가 생기거나 정해진 상호 작용 방식에 문제가 생겼을 때도 최소 정해 놓은 수준으로 동작하는 능력에 영향을 미치는 속성
		회복성 Recoverability	장애로 직접 영향을 받는 데이터를 복구하고 일정 수준 이상으로 다시 동작하는 능력과 이때 필요한 시간과 노력에 영향을 미치는 속성

Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

품질모델 (Quality Model) - 2

- ISO/IEC 9126

사용성 Usability	특정 조건에서 소프트웨어를 쓰고 배우고 이해하기 쉽게 만드는 능력	**이해성 Understandability**	소프트웨어의 개념과 활용 방법을 이해하는 데 필요한 사용자의 노력에 영향을 미치는 속성
		습득성 Learnability	소프트웨어의 활용 방법을 배우고 익히는 데 필요한 사용자의 노력에 영향을 미치는 속성
		운영성 Operability	소프트웨어를 운영하고 관리하는 데 필요한 사용자 노력에 영향을 미치는 속성
효율성 Efficiency	특정 조건에서 적절한 자원을 소모하면서 적절한 성능을 제공할 수 있는 능력	**실행 효율성 Time behavior**	소프트웨어가 기능을 수행할 때 적절한 응답 시간, 처리효율, 처리시간을 제공할 수 있는 능력에 영향을 미치는 속성
		자원 효율성 Resource Behavior	소프트웨어가 기능을 수행할 때 적절 양의 자원을 소모하는 능력에 영향을 미치는 속성

Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

품질모델 (Quality Model) - 3

- ISO/IEC 9126

유지보수성 Maintainability	소프트웨어를 얼마나 쉽게 수정할 수 있는지 측정한 능력. 수정에는 정정, 개선, 환경변화 수용, 요구사항 변화 수용, 기능 추가 같은 것들이 있다.	**분석성 Analyzability**	장애 원인과 겸함을 분석하고 수정해야 할 부분을 찾아내는 데 필요한 노력에 영향을 미치는 속성
		변경성 Changeability	소프트웨어를 수정할 때, 장애를 제거할 때, 소프트웨어 동작 환경을 변경할 때 드는 노력에 영향을 미치는 속성
		안정성 Stability	수정으로 예상하지 못한 문제가 발생할 위험에 영향을 미치는 속성
		시험성 Testability	수정된 소프트웨어를 검증하는 데 필요한 노력에 영향을 미치는 속성
이식성 Portability	소프트웨어가 다른 환경으로 이식될 수 있는 능력.	**적응성 Adaptability**	원래 필요한 작업이나 방법만 써서 소프트웨어가 동작하는 환경을 바꿀 수 있는 가능성에 영향을 미치는 속성
		설치성 Installability	특정 환경에 소프트웨어를 설치할 때 필요한 노력에 영향을 미치는 속성
		일치성 Conformance	소프트웨어가 이식성과 관련된 표준이나 관례를 잘 따르는지 나타내는 속성
		대체성 Replaceability	대상 소프트웨어를 특정한 다른 소프트웨어로 바꿔서 쓸 수 있는지 나타내는 속성

Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

Apply to Printer System of Case Study(1)

Phase: Plan and Elaboration

Non-Functional Requirements 추가

- KUPE의 Activity 1001에 아래와 같은 Non-Functional Requirements를 추가함

Activity 1001. Define Draft Plan

- Non-Functional Requirements
  - 인쇄 품질은 충분히 좋아야 한다.
  - 인쇄 가격은 적당해야 한다.
  - 프린터는 소음이 적어야 한다.
- Resource
  - Human Efforts : 1-0.5 M/M
  - Cost : 250,000

추가

- 장애 발생시 0.1초 내로 프린터 시스템을 멈춰야한다.
  - 멈추지 않으면 프린터 부품에 손상이 갈수 있기 때문임
- 다양한 브랜드(Samsung, HP등)의 프린터를 추가 및 확장할 수 있어야한다.
  - 프린터를 제조하는 회사가 다양하며, 동일 회사에서도 다양한 모델이 존재하기때문임

Quality Requirements with ISO/IEC 9126

- Quality Requirements Description
  - QR 1: Reliability – Maturity(성숙도)
    - 해결하지 못하고 남아있는 장애 때문에 생기는 장애 빈도에 영향을 주는 속성

Ref. #	Description
QR 1.1	장애 발생시 0.1초 내로 프린터 시스템을 멈춰야한다.

  - QR 2: Maintainability – Changeability(변경가능성)
    - 소프트웨어를 수정할 때, 장애를 제거할 때, 소프트웨어 동작 환경을 변경할 때 드는 노력에 영향을 미치는 속성
      - Printer System에는 다양한 브랜드(Samsung, HP등)의 프린터를 추가 및 확장할 수 있어야 서로 다른 회사 및 모델으로 설치할 경우 최소한 변경만으로도 시스템에 적용가능해야함

Ref. #	Description
QR 2.1	프린터를 관리하는 부분은 최소한 변경으로도 작동할 수 있도록 유연한 구조로 만들어야 한다.

Define Requirements 추가

- KUPE의 Activity 1003을 Functional Requirements(Activity 10031)와 Quality Requirements(Activity10032)로 나누어서 아래와 같이 추가함

Activity 1003. Define Requirements

Activity 10031. Functional Requirements

Ref. #	Function	Category
R 1.1	System Access	Event
R 1.2	Make Account	Event
R 1.3	Identify Balance	Event
R 1.4	Recharge Balance	Event
R 2.1	Request Print	Event
R 2.2	Check Balance	Hidden
R 3.1	Identify Paper	Event
R 3.2	Recharge Paper	Event
R 3.3	Identify User	Event
R 3.4	Identify Money	Event

Activity 1003. Define Requirements

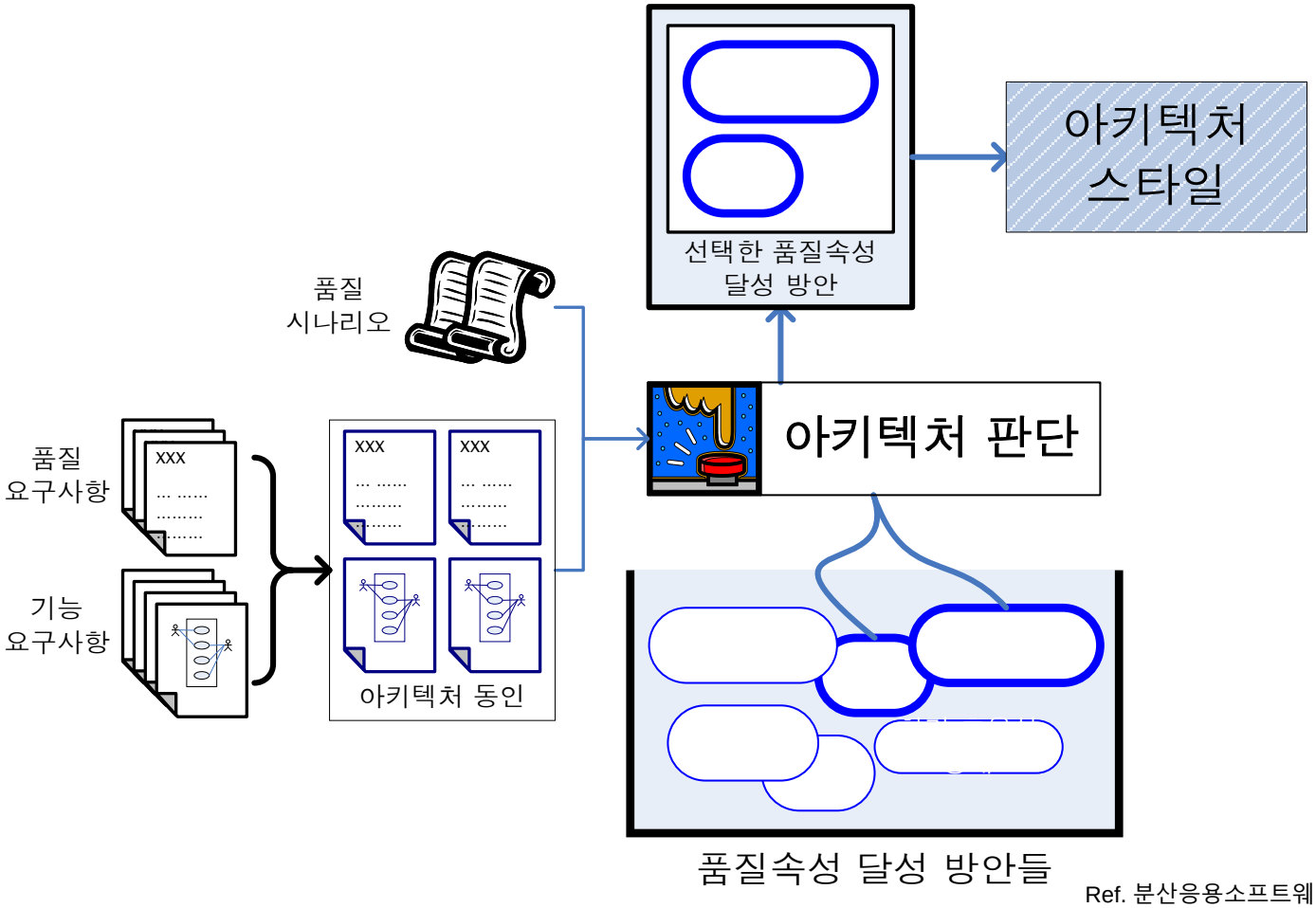
Activity 10032. Quality Requirements

Ref. #	Description
QR 1.1	장애 발생시 0.1초 내로 프린터 시스템을 멈춰야한다.
QR 2.1	프린터를 관리하는 부분은 최소한 변경으로도 작동할 수 있도록 유연한 구조로 만들어야 한다.

Architecture Analysis & Design

Architecture Style(Pattern) 선택

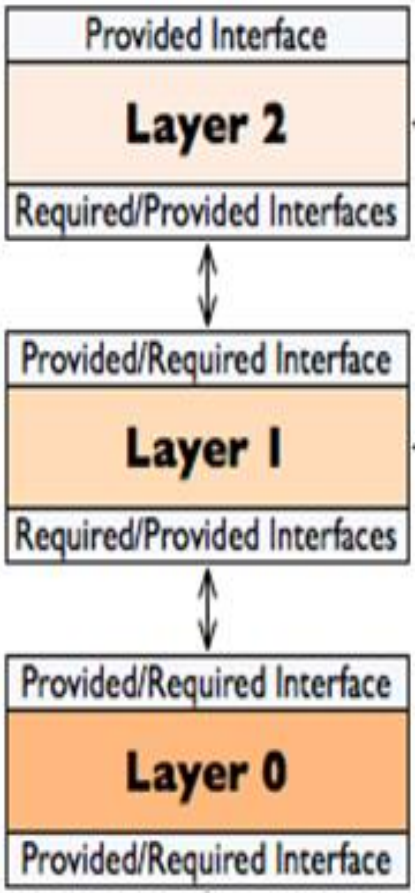
- 요구사항에 충족할 수 있는 품질속성 달성 방안을 선택하고 실현할 수 있는 아키텍처 스타일(아키텍처 패턴)을 선택한다.

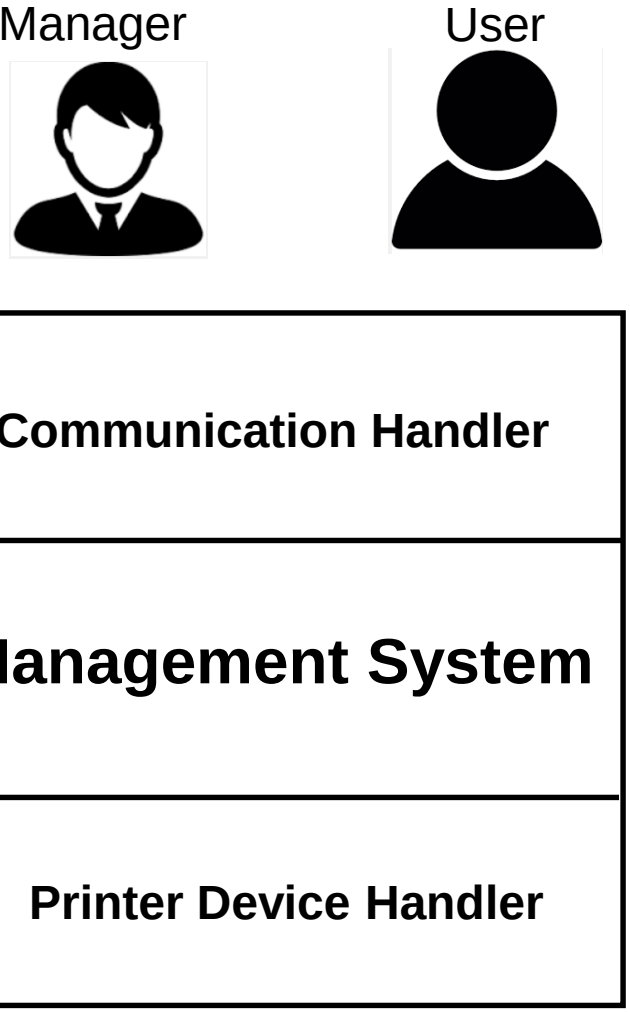


Ref. 분산응용소프트웨어개발 – 민덕기 수업자료

Architecture Style 선택

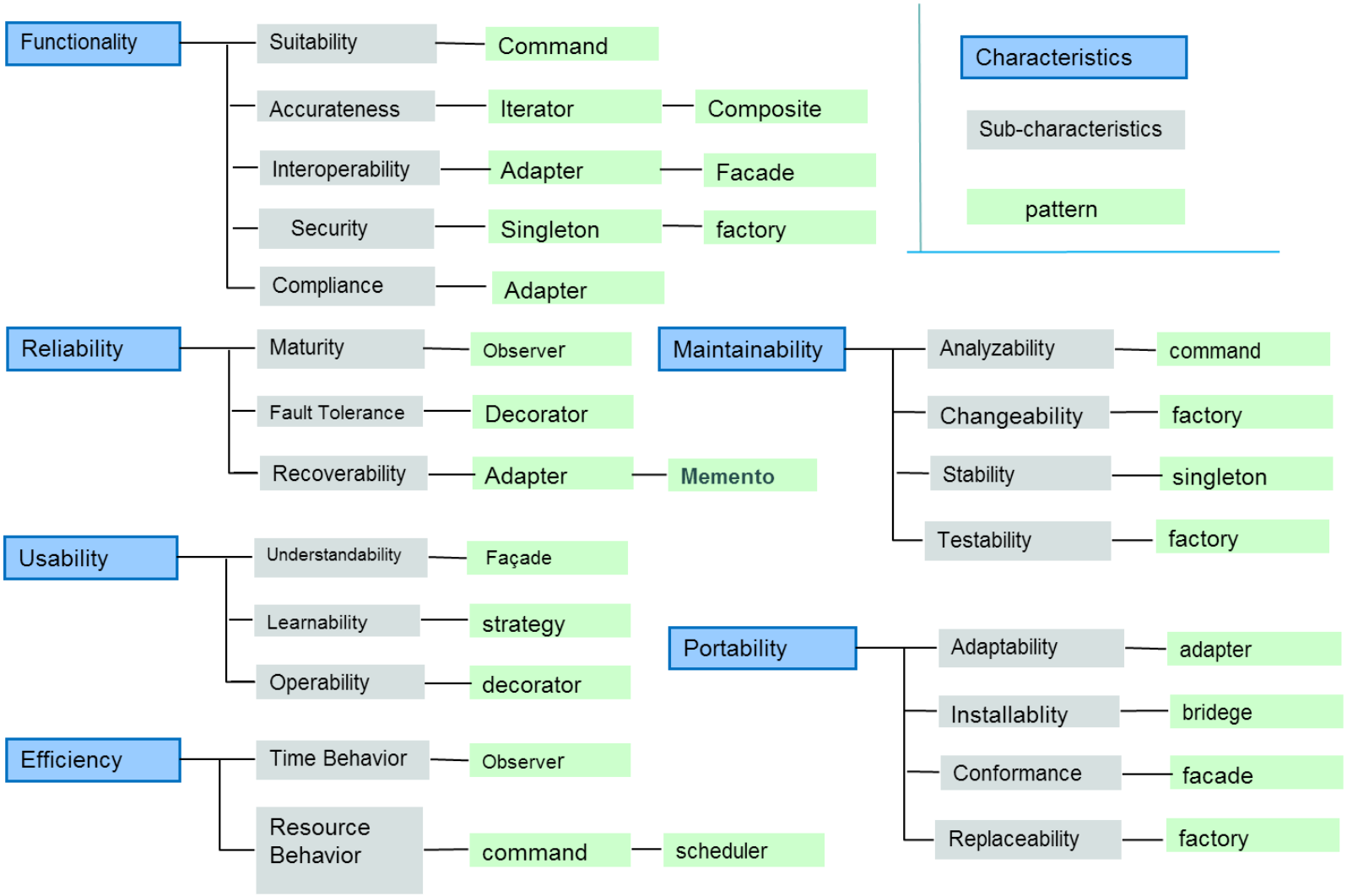
- Layered Architecture Style





Architecture Design 확인

- 품질 요구사항을 만족을 위한 Design Guidelines을 확인



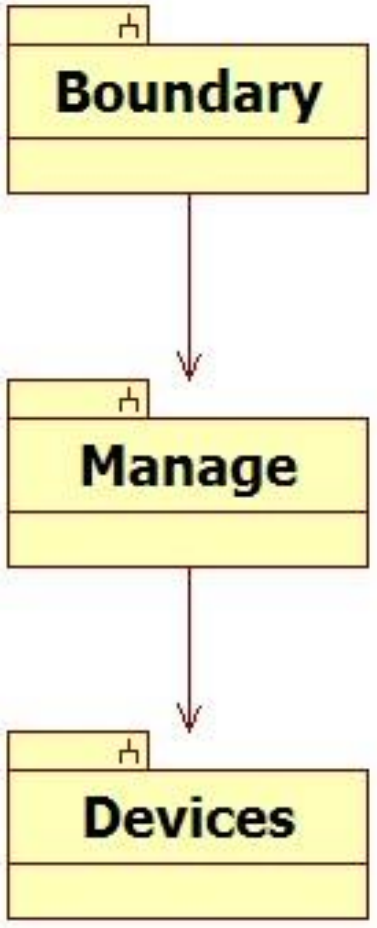
Apply to Printer System of Case Study(2)

Phase: Analysis & Design

Architecture Analysis

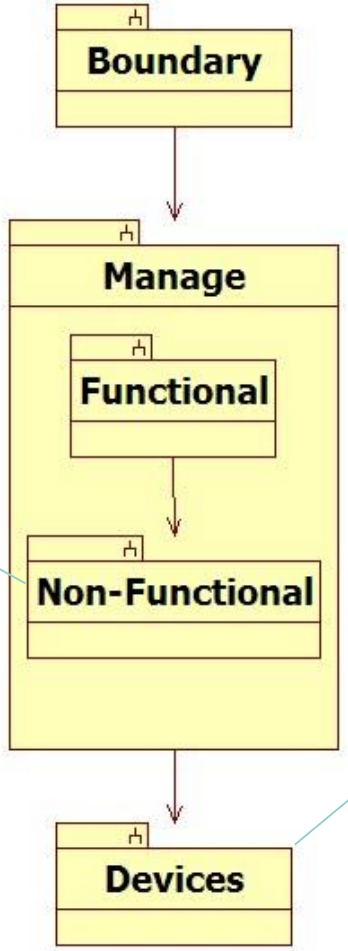
- Printer System의 Subsystem을 High-Level 구조를 정의

- Subsystem Type Categorization 종류
  - Boundary Module
    - 외부시스템과의 인터페이스를 담당
  - Manage Module
    - Boundary Module을 통해 받은 요청을 처리 담당
  - Device Module
    - 장치와 인터페이스를 담당하는 모듈



Architecture Analysis

- Printer System의 Subsystem을 High-Level 구조를 정의
  - 기능요구사항과 품질요구사항에 따라 Manage 모듈을 두개의 서브 모듈로 분리
    - Functional
    - Non-Functional



QR1.1(장애 발생시 0.1초 내로 프린터 시스템을 멈춰야한다.)을 만족해야 하는 서브시스템 타입

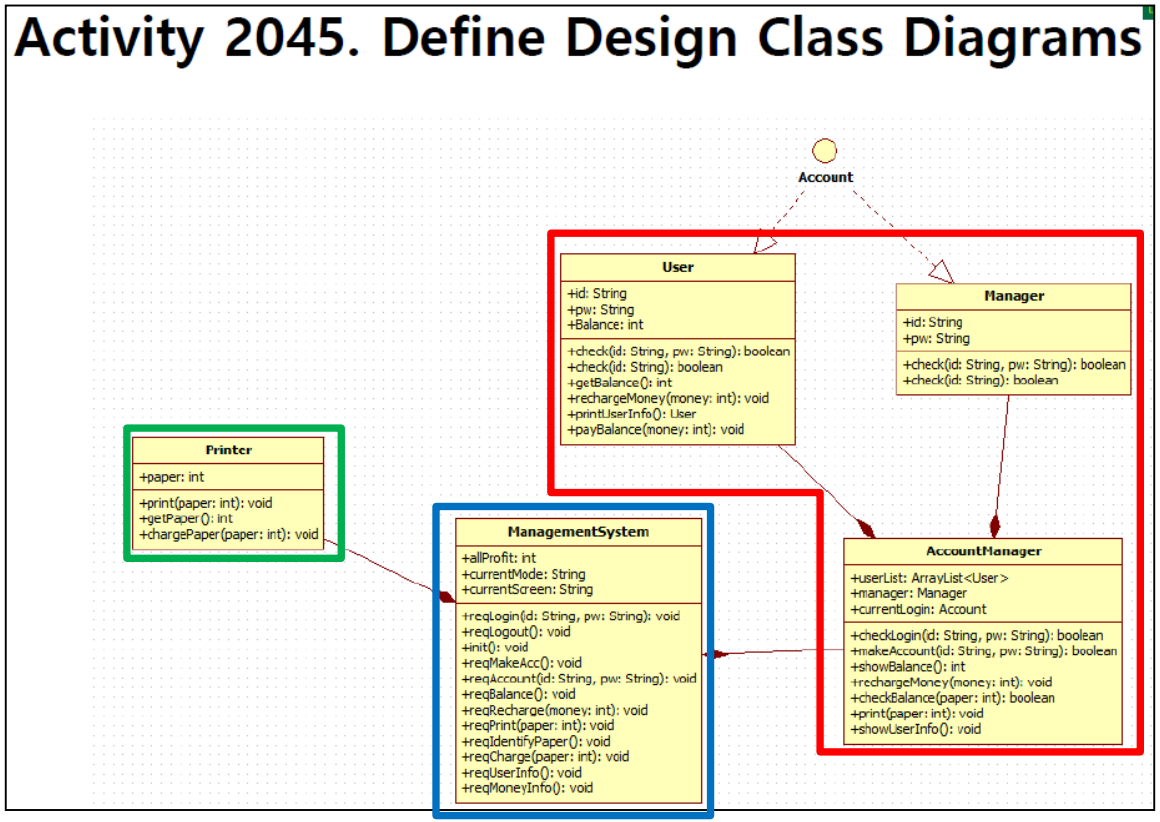
QR2.1(프린터를 관리하는 부분은 최소한 변경으로도 작동할 수 있도록 유연한 구조로 만들어야한다.)를 만족해야 하는 서브시스템 타입

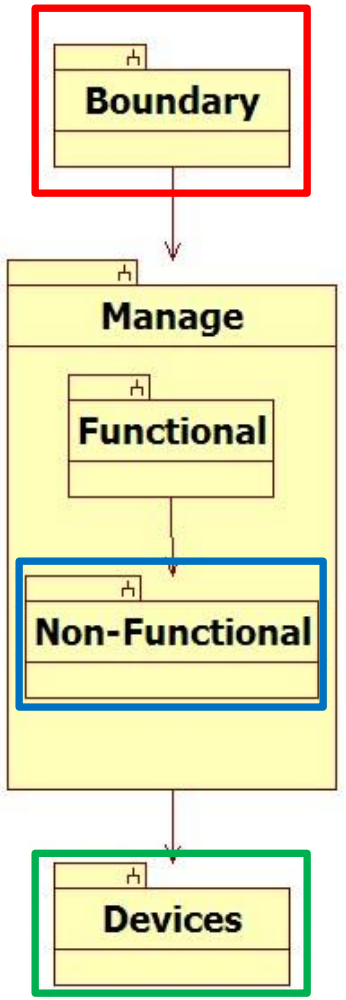
Architecture Design 확인

- 품질 요구사항을 만족을 위한 Design Guidelines을 확인

Architecture Design 적용

- KUPE의 Activity 2045에 분석한 결과를 적용한다.
  - ManagementSystem에
    - Q 1.1을 만족하기위해 System을 진단하는 메소드 필요
    - Q 2.1을 만족하기위해 다양한 Printer를 지원할 수 있도록 Factory Pattern을 사용해야함

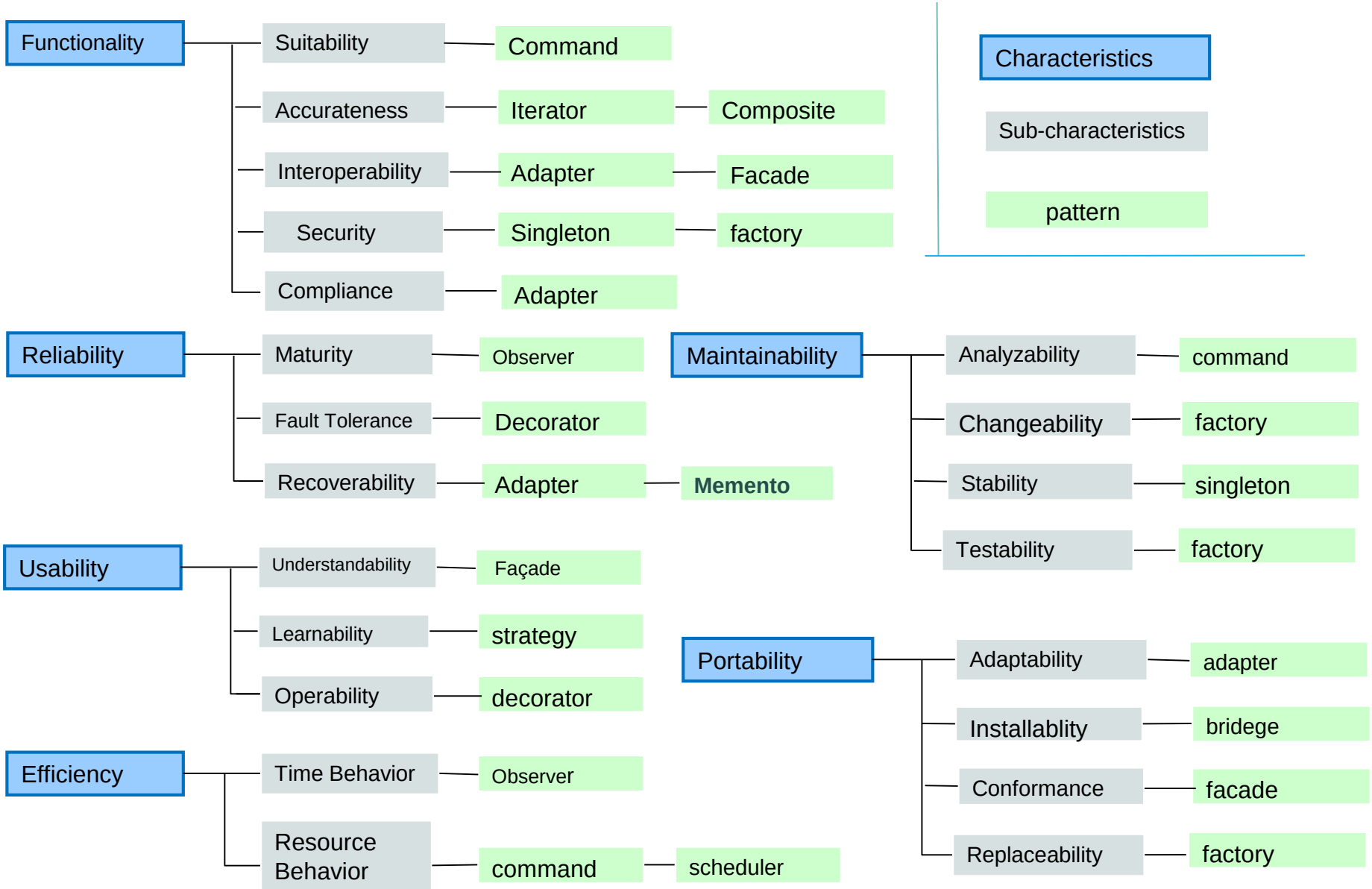




Q & A

Design Pattern Details

추가자료


Functionality
Suitability
Command
Accurateness
Iterator
Composite
Interoperability
Adapter
Facade
Security
Singleton
factory
Compliance
Adapter
Characteristics
Sub-characteristics
pattern
Reliability
Maturity
Observer
Fault Tolerance
Decorator
Recoverability
Adapter
Memento
Maintainability
Analyzability
command
Changeability
factory
Stability
singleton
Testability
factory
Usability
Understandability
Façade
Learnability
strategy
Operability
decorator
Portability
Adaptability
adapter
Installablity
bridege
Conformance
facade
Replaceability
factory
Efficiency
Time Behavior
Observer
Resource Behavior
command
scheduler

Command Design Pattern

definition

- The Command Pattern **encapsulates a request as an object**, there by letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

- Details

•**Command** - declares an interface for executing an operation;

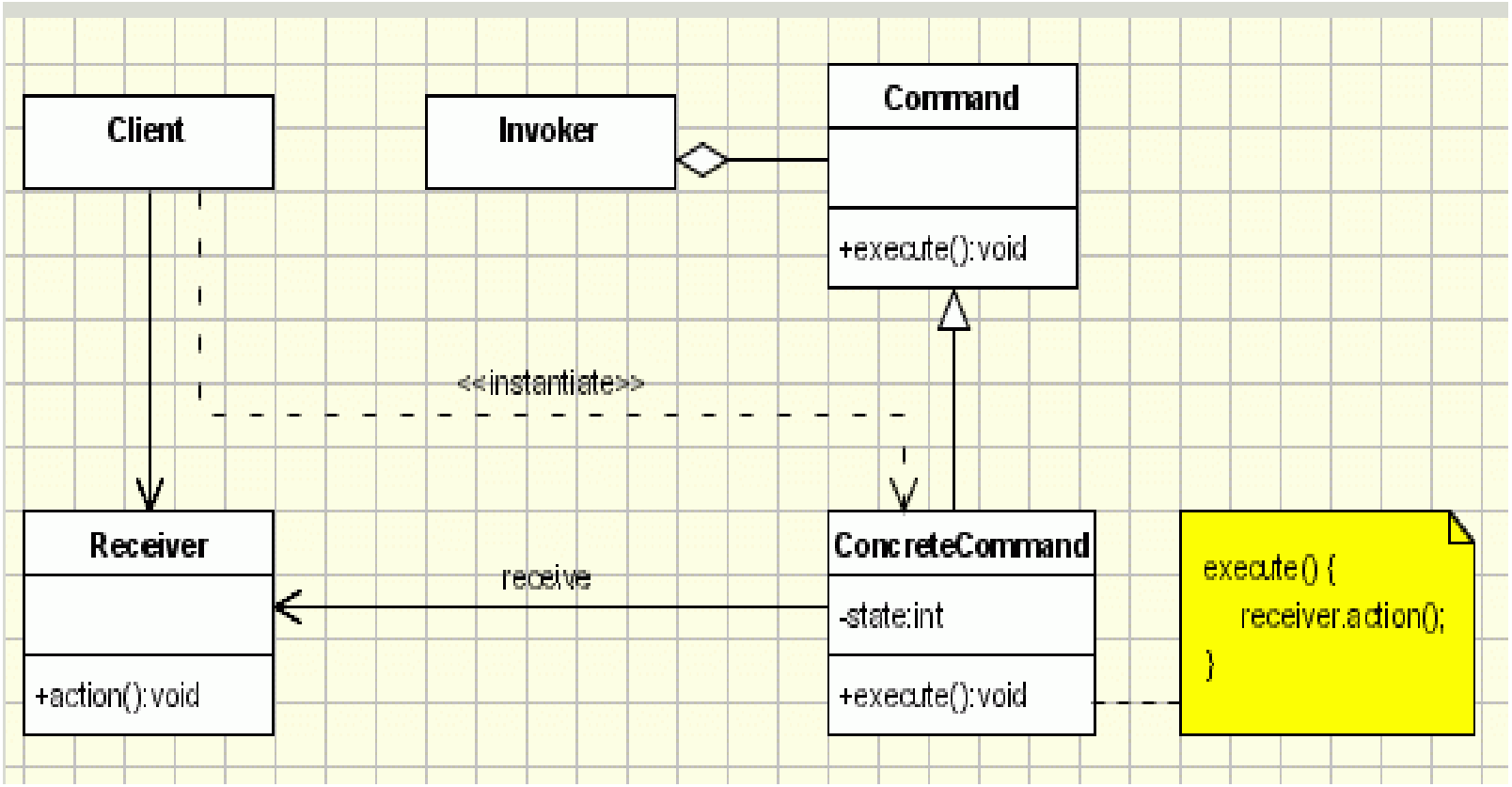
•**ConcreteCommand**
  - Extends the Command interface
  - It defines **a link** between the Receiver and the action.

• **Client** - creates a ConcreteCommand object and sets its receiver;

•**Invoker** - asks the command to carry out the request;

•**Receiver** - knows how to perform the operations;

Command Design Pattern



The Client **asks for a command** to be executed. The Invoker takes the command, **encapsulates it** and places it in a queue, in case there is something else to do first, and the ConcreteCommand that is **in charge of the requested command**, sending its result to the Receiver.

Summary

Advantage

- A command object could be sent across a network to be executed elsewhere or it could be **saved as a log of operations**

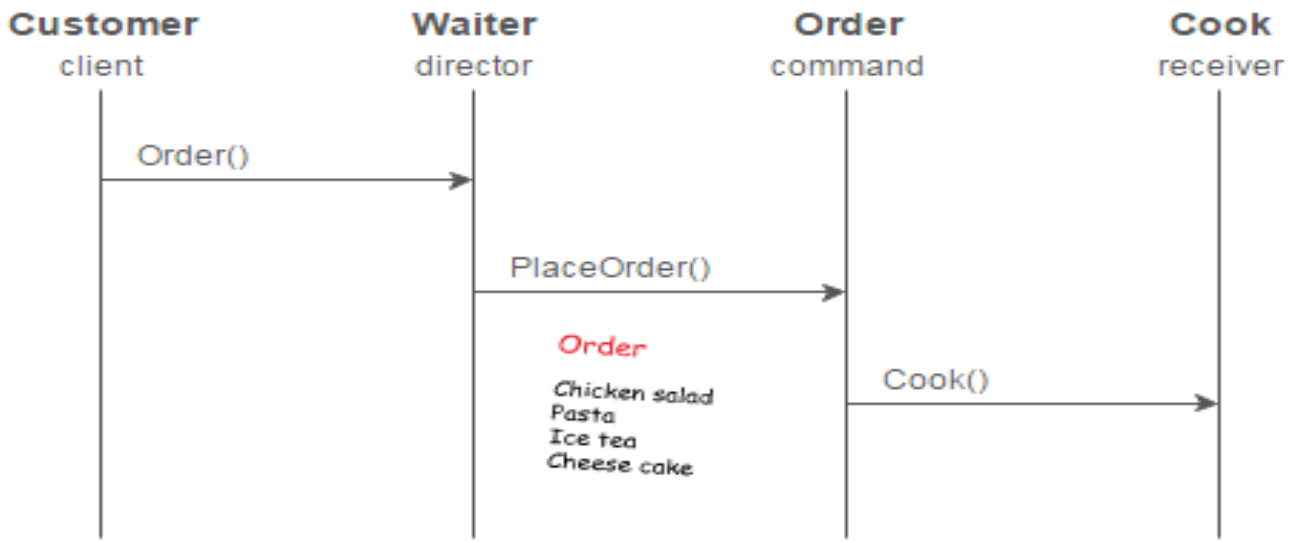
Disadvantage

- Need to issue requests to objects **without knowing anything** about the operation being requested or the receiver of the request.

Example

- The waiter or waitress **takes an order or command** from a customer and encapsulates that order by writing it on the check.

- The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter **is not dependent on the menu**, and therefore they can support commands to cook many different items.



Observer Pattern

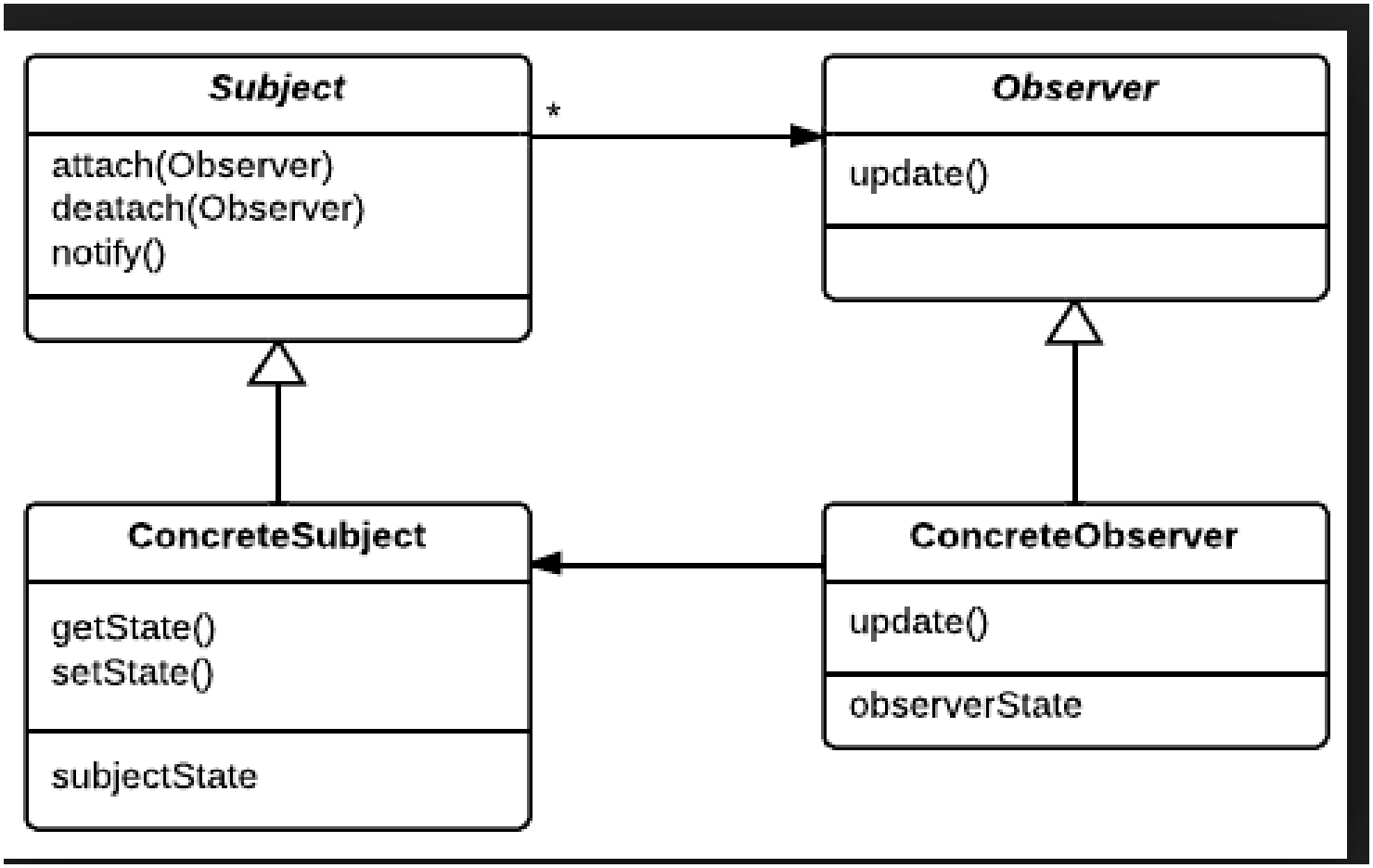
Definition

•Define a **one-to-many dependency** between objects so that when one object **changes state**, all its dependents are notified and **updated automatically**.

Details

.
- Subject
  - has a list of observers
  - Interfaces for attaching/detaching an observer
- Observer
  - An updating interface for objects that **gets notified of changes in a subject**
- ConcreteSubject
  - **Stores "state** of interest" to observers
  - **Sends notification** when state changes
- ConcreteObserver
  - Implements updating interface

Observer Pattern



Observer Pattern - Consequences

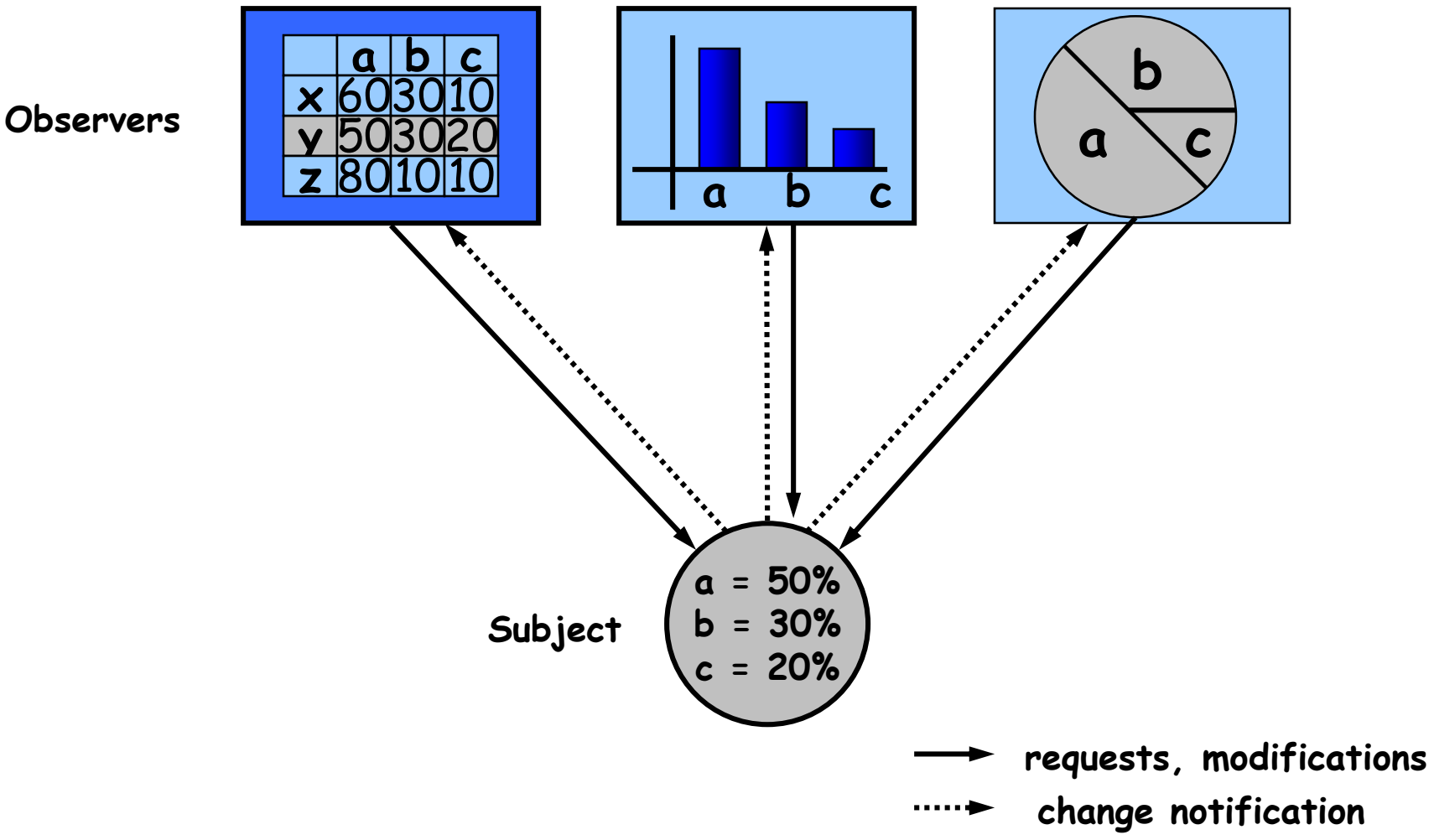
Advantage

- Loosely Coupled
  - Reuse subjects without reusing their observers, and vice versa
  - Add observers without modifying the subject or other observers

Disadvantage

- A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Observer Pattern - Example



Decorator Pattern

Definition

- the **decorator pattern** allows behavior to be added to an individual object. **without affecting the behavior** of other objects from the same class.

Details

Component

- Interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent

- Defines an object to which **additional responsibilities** can be added.

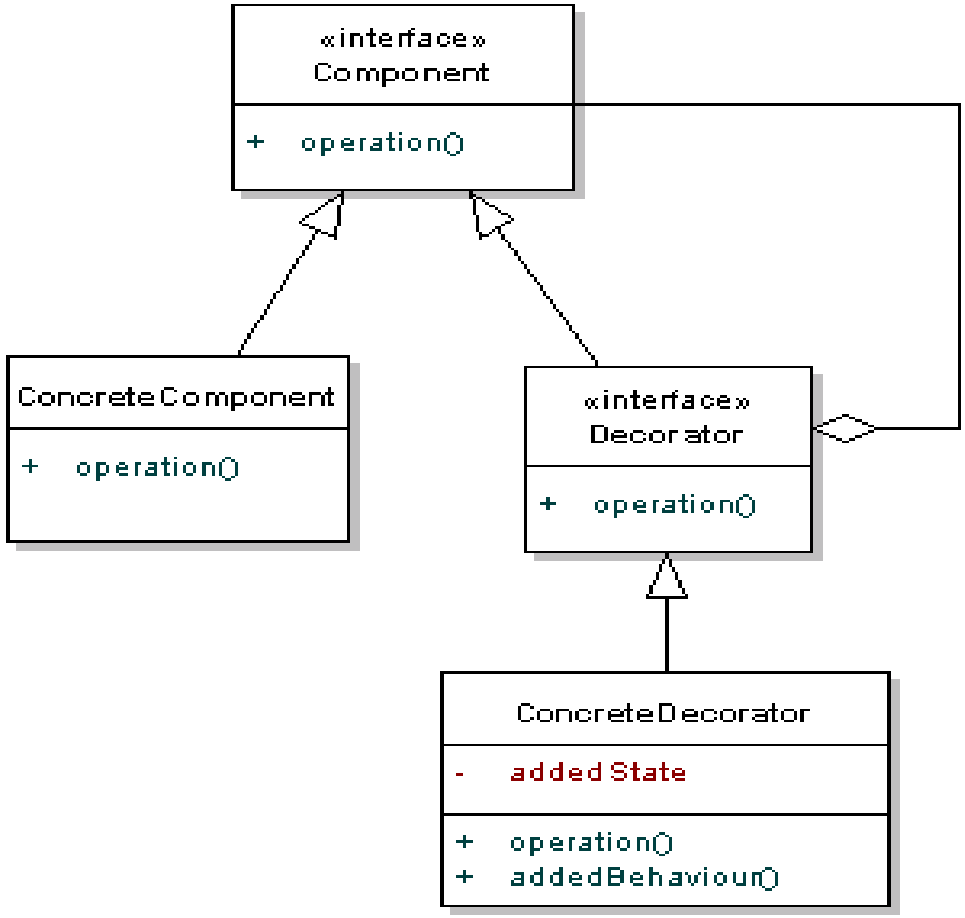
Decorator

- Maintains a reference to a Component object and defines an interface that conforms to Component's interface.

•**Concrete Decorators**

- Concrete Decorators extend the functionality of the component by **adding state or adding behavior**.

Decorator pattern



summary

- Advantage
- **Attach additional responsibilities to an object dynamically**. Decorators provide a flexible alternative to subclassing for extending functionality.

- Disadvantage

You want to add behavior or state to individual objects at run-time. **Inheritance is not feasible** because it is static and applies to an entire class.

Adapter pattern

Definition

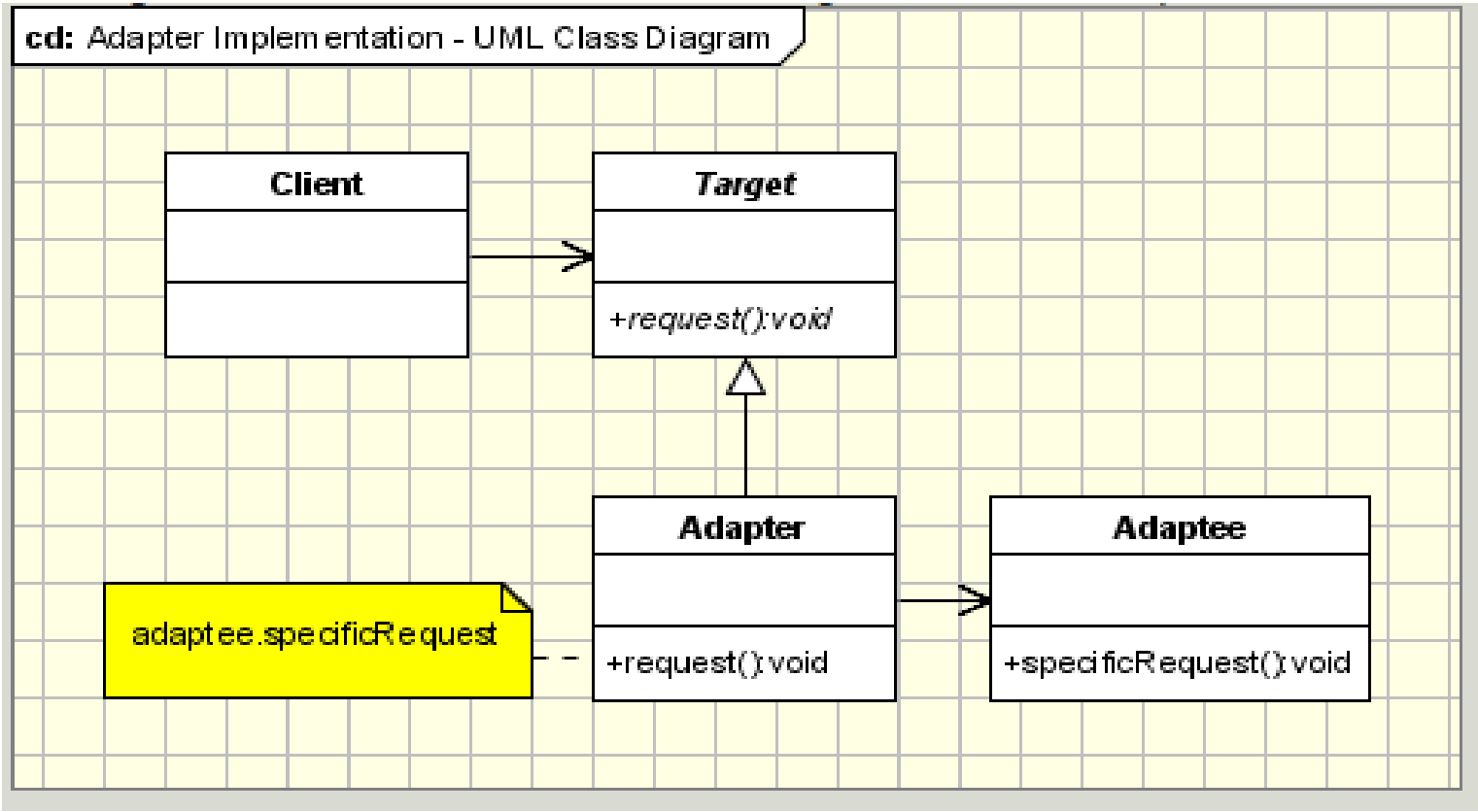
Adapter pattern allows the interface of an existing class to be used as **another interface**.

It is often used to make existing classes work with others **without modifying their source code**.

Details

- **Target** - defines the domain-specific interface that Client uses.
- **Adapter** - adapts the interface Adaptee to the Target interface.
- **Adaptee** - defines an existing interface that needs adapting.
- **Client** - collaborates with objects conforming to the Target interface.

Adapter pattern



summary

Advantage

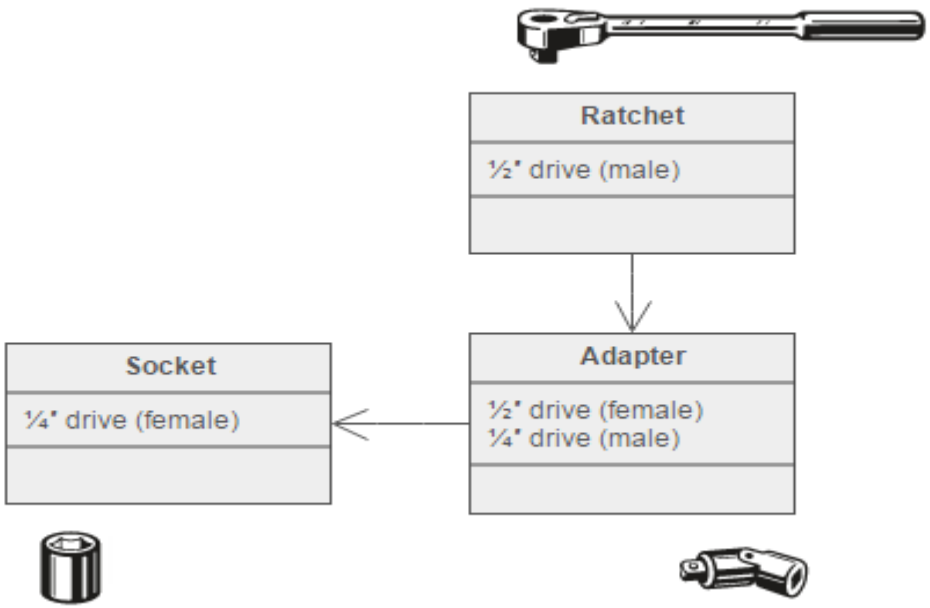
- Convert the interface of a class into another interface clients expect. Adapter lets **classes work together** that couldn't otherwise because of incompatible interfaces.

Disadvantage

- **It unnecessarily increases the size of the code as class inheritance is less used and lot of code is needlessly duplicated between classes.**

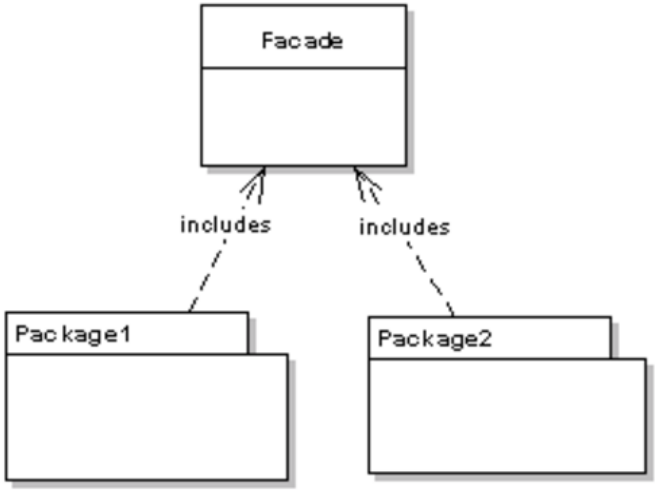
Example

- Socket wrenches provide an example of the Adapter.
- Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used.
- A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



Facade Pattern

- Definition
  - Facade pattern is a software design pattern commonly used with object-oriented programming
- Details
  - The diagram definition of the Facade pattern is quite simple - all you're really doing is insulating client from the subsystem



summary

Advantage

• **Provide a unified interface** to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use**.**

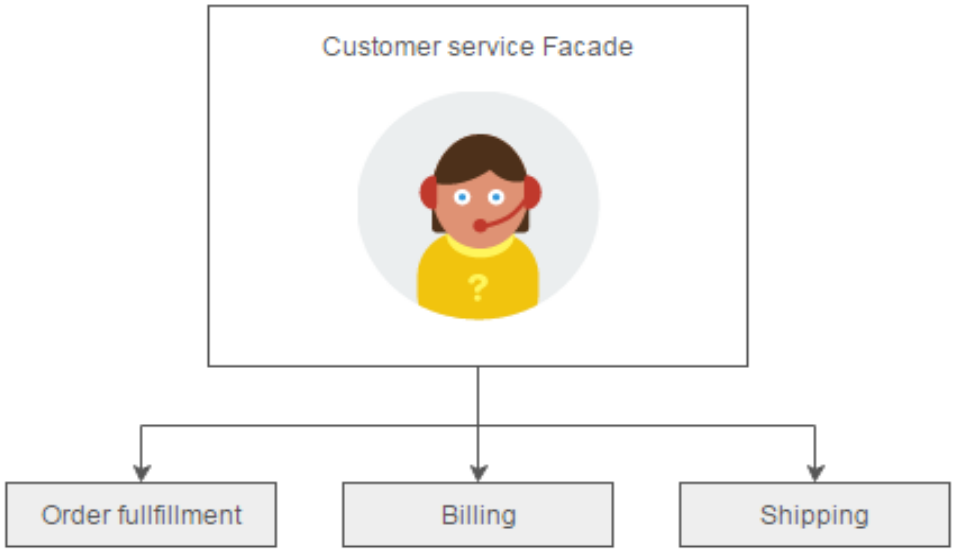
Disadvantage

- It does not prevent sophisticated clients from accessing the underlying classes
- Note that Facade **does not add any functionality,** it just simplifies interfaces

Example

- The consumer calls one number and speaks with a **customer service representative.**

- The customer service representative acts as a **Facade**, providing an interface to the order fulfillment department, the billing department, and the shipping department.



Head First Design Patterns

Memento pattern

Memento pattern-definition

Definition

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state (unto via rollback).

Details

•**Memento**

- **Stores** internal state of the Originator object. The state can include any number of state variables.

•**Originator**

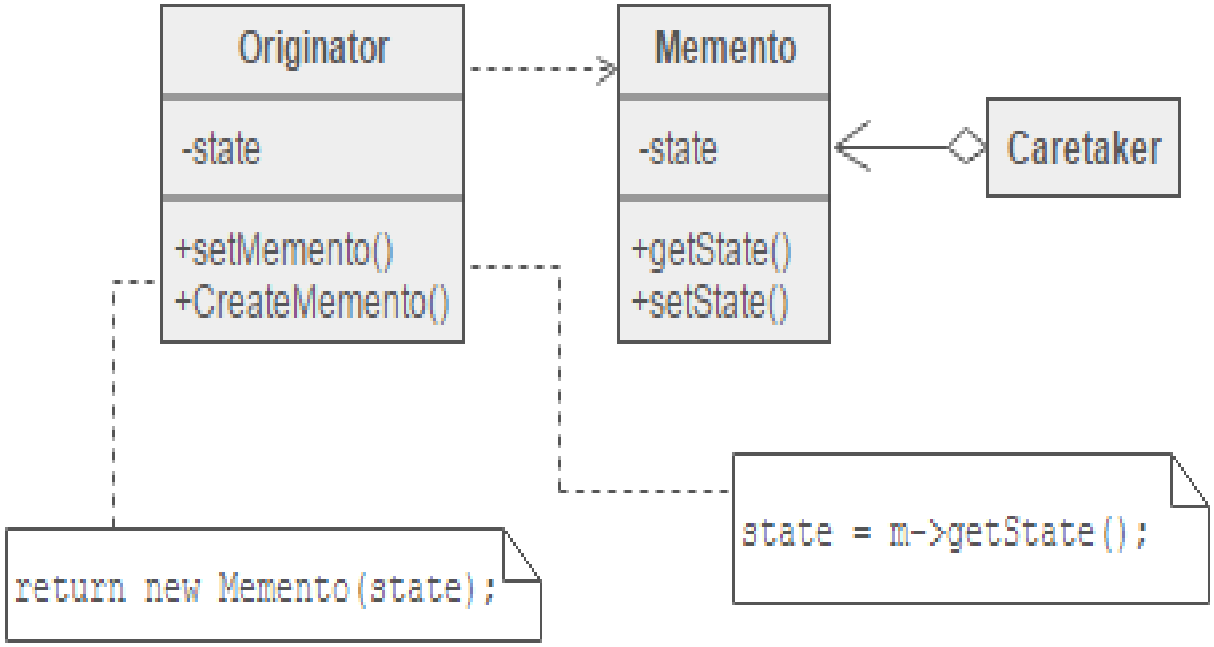
- Creates a memento object capturing the originators internal state.
- Use the memento object to **restore its previous state**.

•**Caretaker**

- Responsible for keeping the memento.
- The memento is opaque to the caretaker, and the caretaker must not operate on it.

Memento pattern-details

Structure



SUMMARY

- Advantage

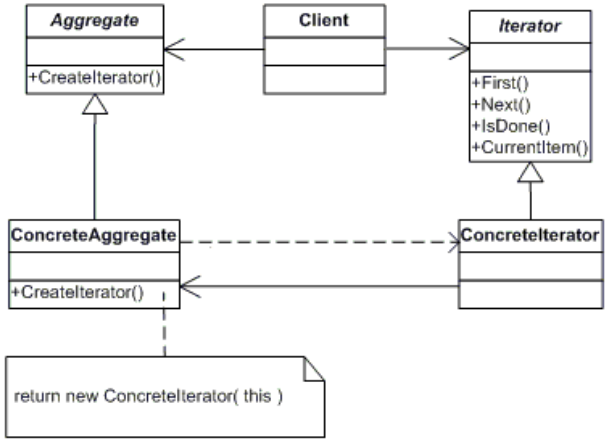
Promote undo or rollback to full object status.

- Disadvantage

Resource consumption is too large, if the class member variables too much, it will **take up a lot of memory**

Iterator Pattern

- Definition
  - Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements
- Details
  - The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.



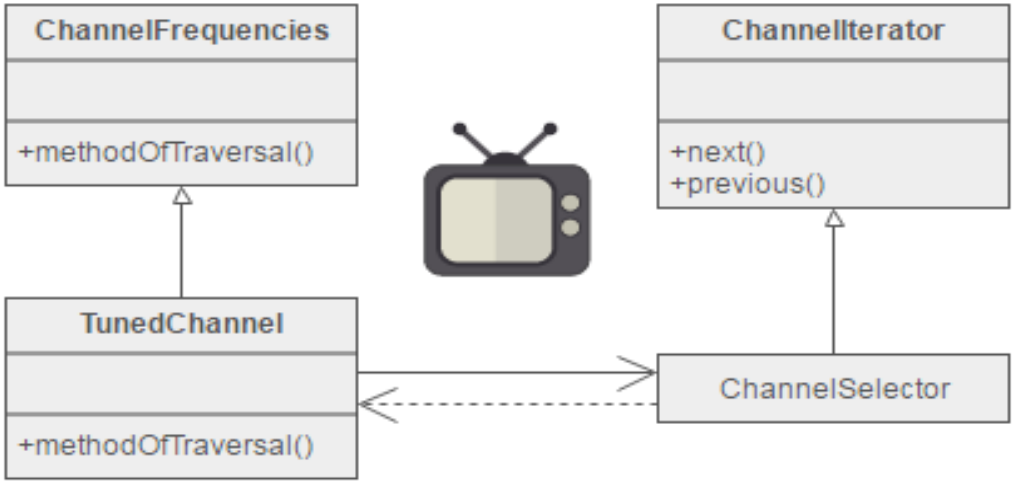
- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Iterator Pattern

- Advantage
  - It is easier to implement different algorithms to reuse the same iterators on different aggregates and to subclass the iterator in order to change its behavior
- Disadvantage
  - The **main disadvantage** is that the iterator will have to access internal members of the aggregate

Iterator Pattern

- Example
  - Consider **watching television in a hotel room in a strange city**
    - When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Composite Pattern

- Definition
  - Composite pattern is a partitioning design pattern
- Details
  - The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object
  - The **intent** of a composite is to "compose" objects into tree structures to represent part-whole hierarchies

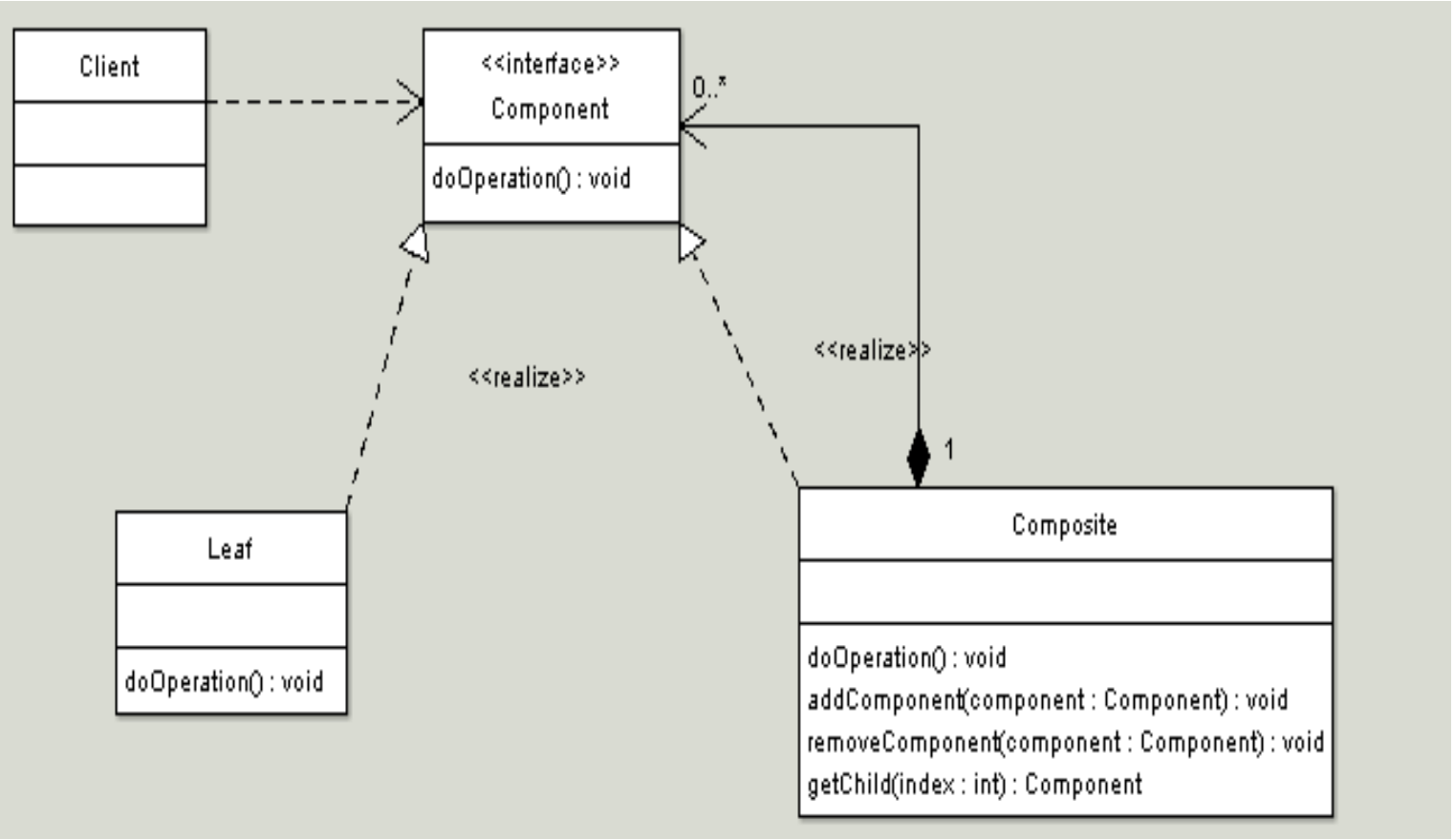
•**Component** - Component is the **abstraction for leafs and composites**. It defines the interface that must be implemented by the objects in the composition.

•**Leaf** - Leafs are objects that have no children. They implement services described by the Component interface

•**Composite** - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components.

•**Client** - The client manipulates objects in the hierarchy using the component interface.

Composite Pattern



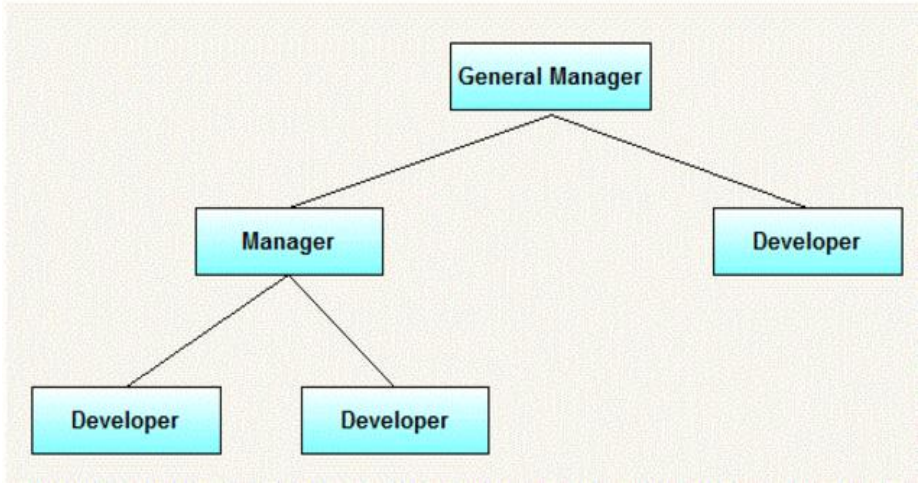
Composite Pattern

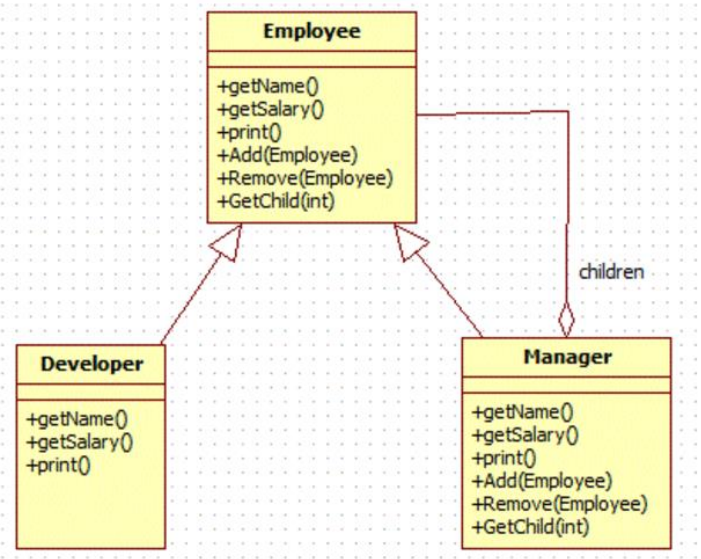
- Advantage
  - The Composite pattern makes the client simple:
    - Clients use the Component class interface to interact with objects in the composite structure
    - If call is made to a Leaf, the request is handled directly
- Disadvantage
  - Once tree structure is defined, the composite design makes the tree overly general
  - **In specific cases**, it is difficult to restrict the components of the tree to only particular types

Composite Pattern

- Example
  - In a small organization, there are 5 employees
    - At top position, there is 1 general manager
    - Under general manager, there are two employees,
      - One is manager and other is developer and further manager has two developers working under him
      - We want to print name and salary of all employees from top to bottom

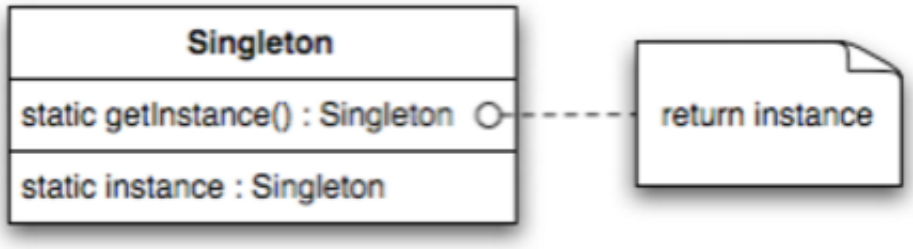
Tree structure for example:





Singleton Pattern

- Definition
  - Singleton pattern is a design pattern that restricts the instantiation of a class to one object
- Details
  - Ensures that a class has only one instance and provides a global point of access to it
  - It's important for some classes to have exactly one instance



- The **instance** class variable holds our one and only instance of Singleton
- The getInstance() method is static and public
  - Return a instance
  - You can conveniently access this method from anywhere in your code using Singleton.getInstance()

Singleton Pattern

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

- Encapsulated "just-in-time initialization" or "initialization on first use".

- Disadvantage
  - **Unit testing is more difficult** (because it introduces a global state into an application).

Singleton Pattern

- Example
  - There can be many printers in a system but there should only be one printer spooler