SWCON201
Opensource & Software
Development Methods and Tools

Design Pattern

Department of
Software Convergence

경희대학교
KYUNG HEE UNIVERSITY

Contents

- Agenda
- Class
- Summary
- Reference
- Homework

Agenda

- Re-usability to increase Product-ability, and Reliability.
- Re-usability scope is language, standard libraries, and open sources → So code only ?
- Software Design also !!!

Software Design Pattern(s)

- A general, reusable solution to a commonly occurring problem within a given context in software design
- Not a finished design that can be transformed directly into source or machine code
- A description or template for how to solve a problem that can be used in many different situations

- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system
- Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

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

Object-Oriented Design Pattern(s)

- Typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved
- Patterns that imply mutable state may be unsuited for functional programming languages, some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages

Adapter Pattern (Goal)

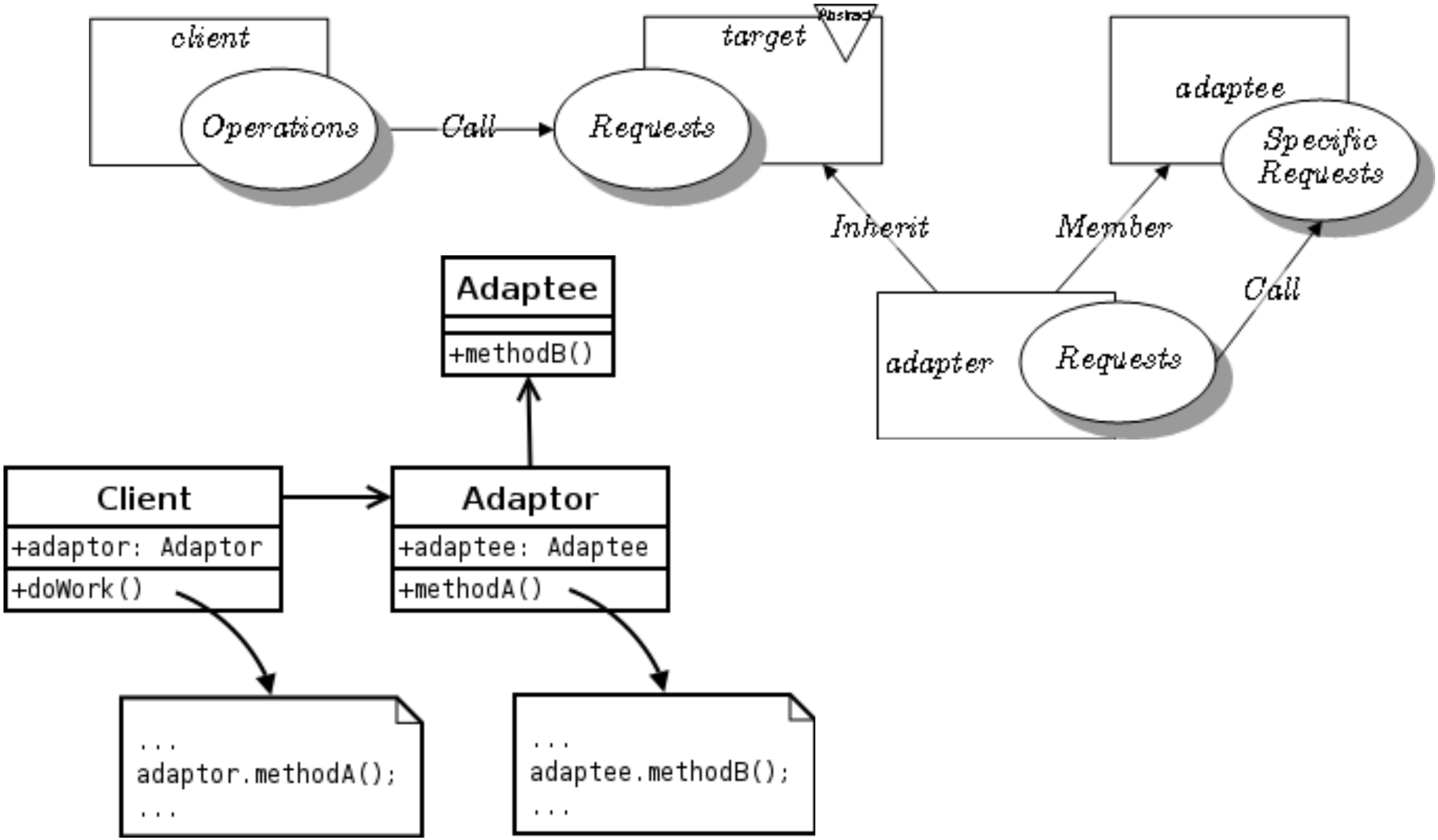
- The Adapter design pattern solves problems like:
  - How can a class be reused that does not have an interface that a client requires?
  - How can classes that have incompatible interfaces work together?
  - How can an alternative interface be provided for a class?
- lOften an (already existing) class can't be reused only because its interface doesn't conform to the interface clients require

Adapter Pattern (Detail)

- The Adapter design pattern describes how to solve such problems:
  - Define a separate Adapter class that converts the (incompatible) interface of a class (Adaptee) into another interface (Target) clients require
  - Work through an Adapter to work with (reuse) classes that do not have the required interface
- The key idea in this pattern is to work through a separate Adapter that adapts the interface of an (already existing) class without changing it
- Clients don't know whether they work with a Target class directly or through an Adapter with a class that does not have the Target interface

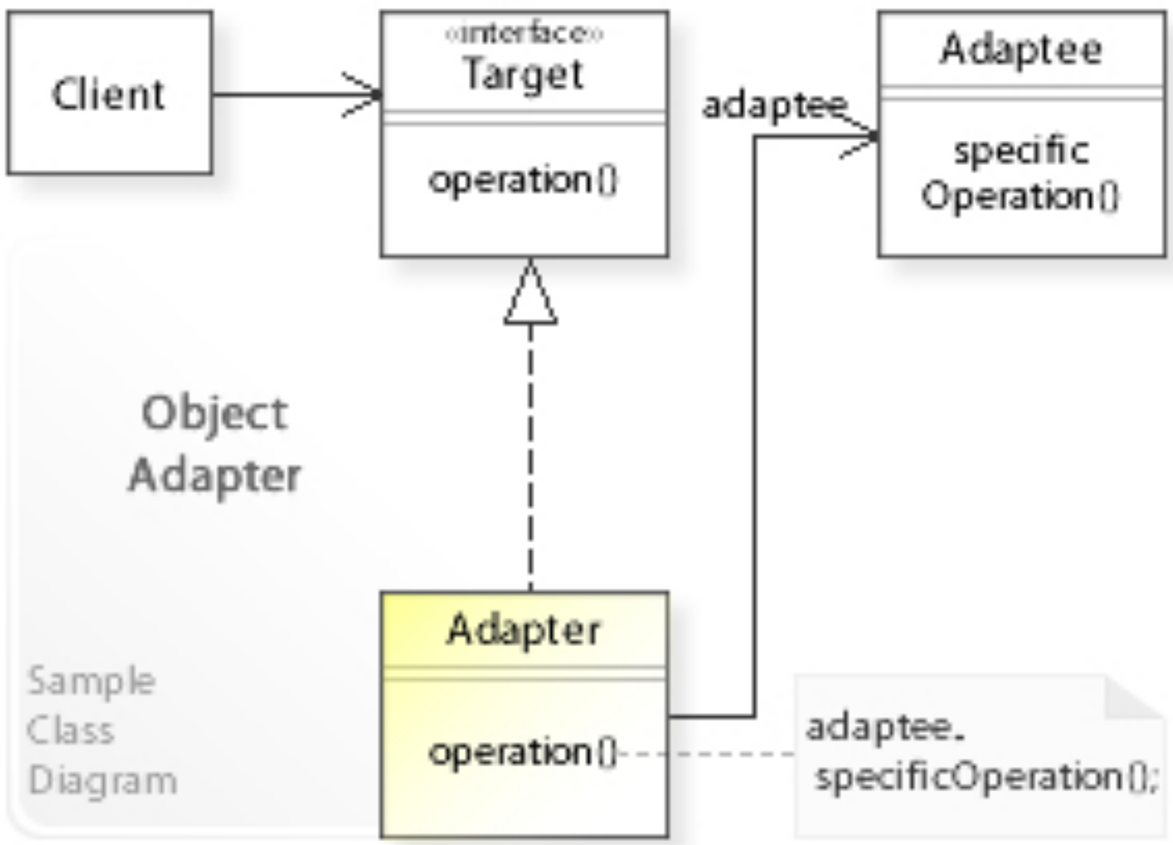
Reference: https://en.wikipedia.org/wiki/Adapter_pattern

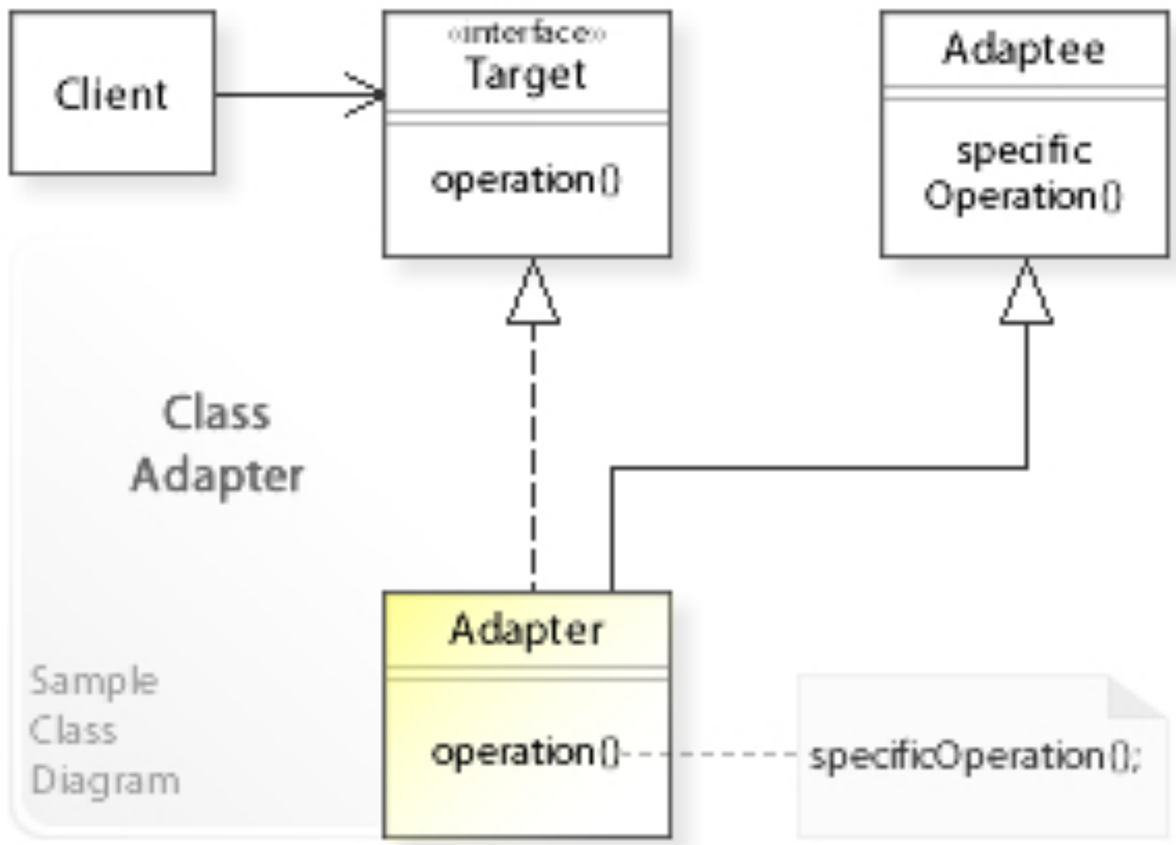
Adapter Pattern (UMLs)



Reference: https://en.wikipedia.org/wiki/Adapter_pattern

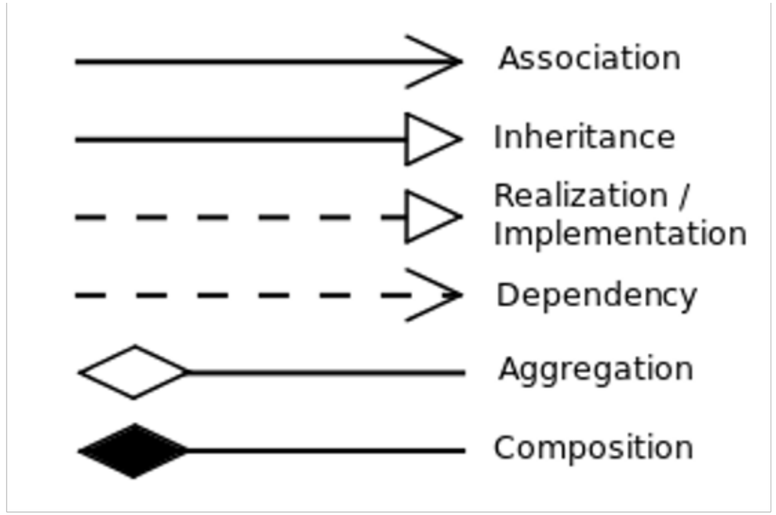
Adapter Pattern (UMLs)

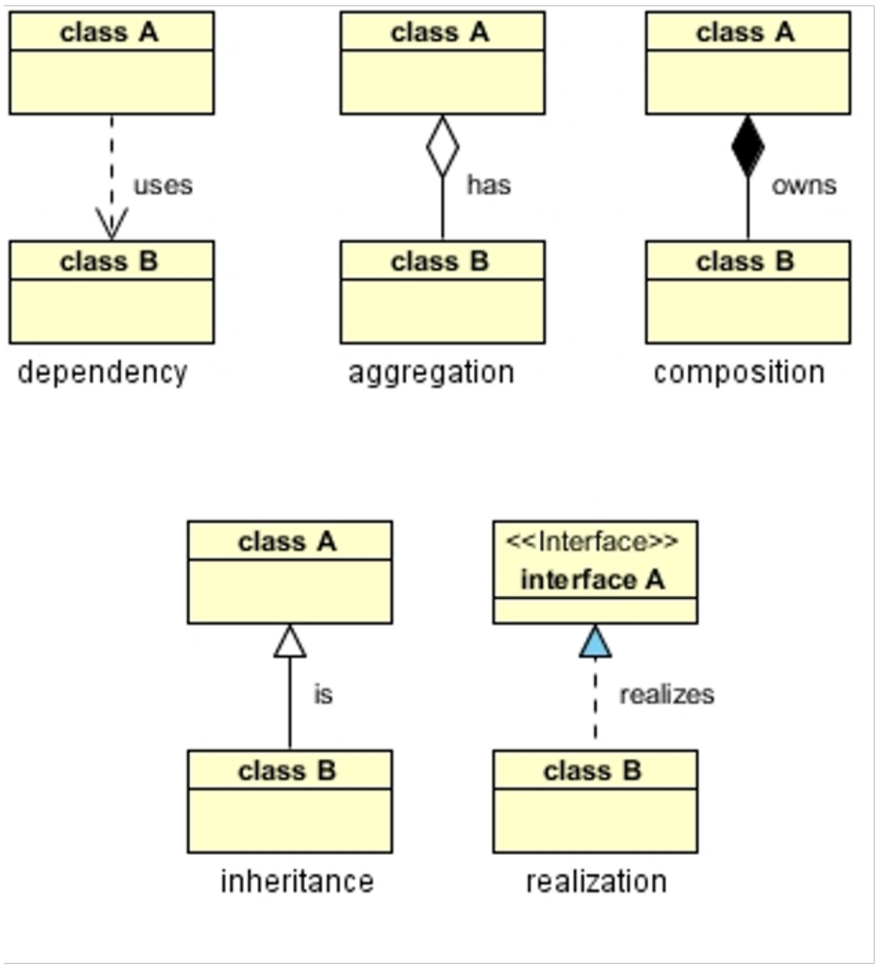


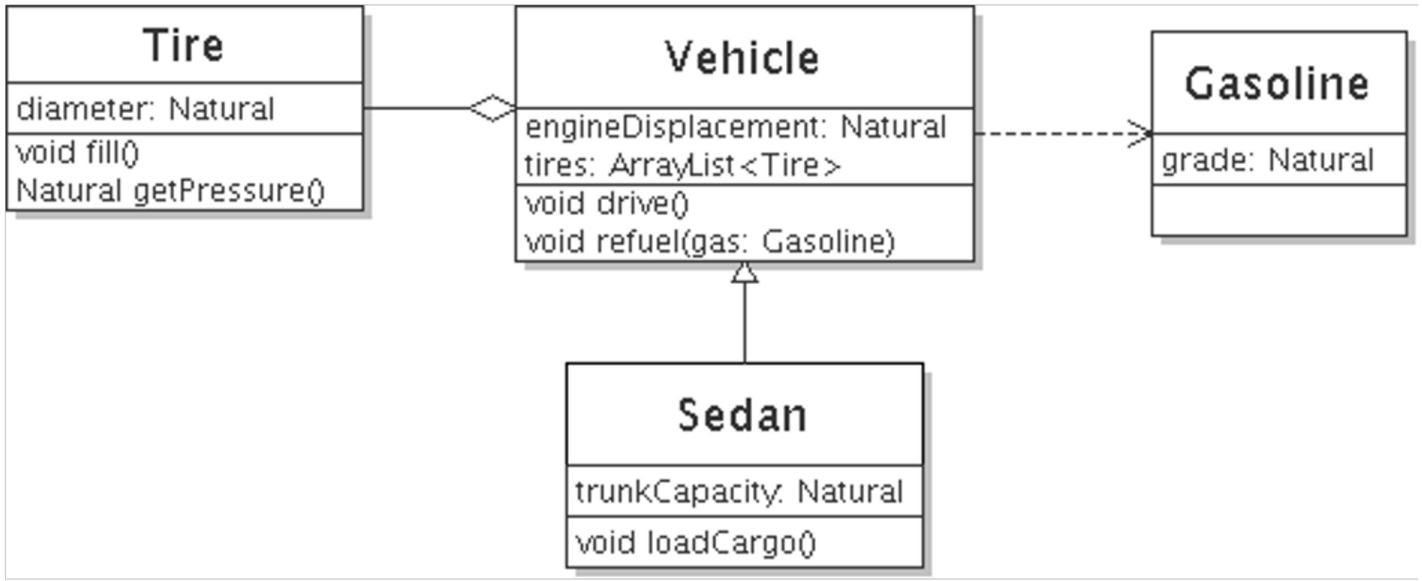


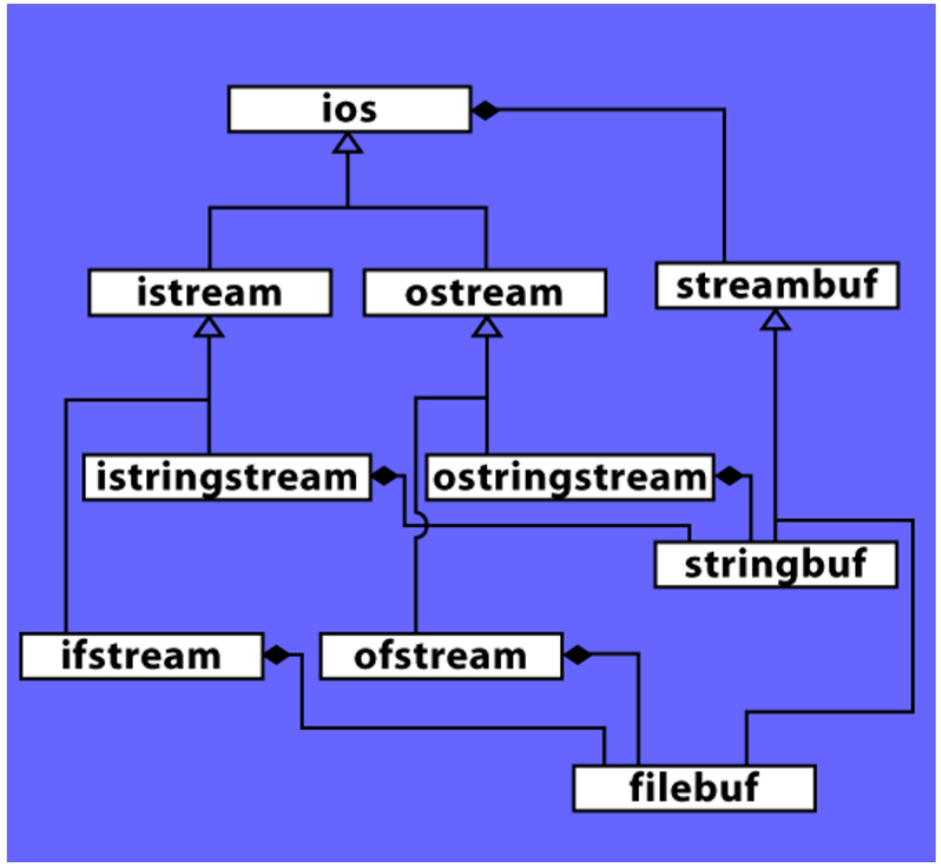
Reference: https://en.wikipedia.org/wiki/Adapter_pattern

Reference – UML (Unified Modeling Language)









Reference: http://users.csc.calpoly.edu/~jdalbey/308/Resources/ooDesignHowTo.html
http://usna86-techbits.blogspot.kr/2012/11/uml-class-diagram-relationships.html

Adapter Pattern (Example)

- Goal is reusing Recharging component/code

  https://en.wikipedia.org/wiki/Adapter_pattern

Façade Pattern (Goal)

- Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse
- The Façade design pattern solves problems like:
  - To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem
  - The dependencies on a subsystem should be minimized.

Reference: https://en.wikipedia.org/wiki/Facade

Façade?

- A facade is generally one exterior side of a building, usually, but not always, the front
- It is a foreign loan word from the French façade, which means "frontage" or "face"

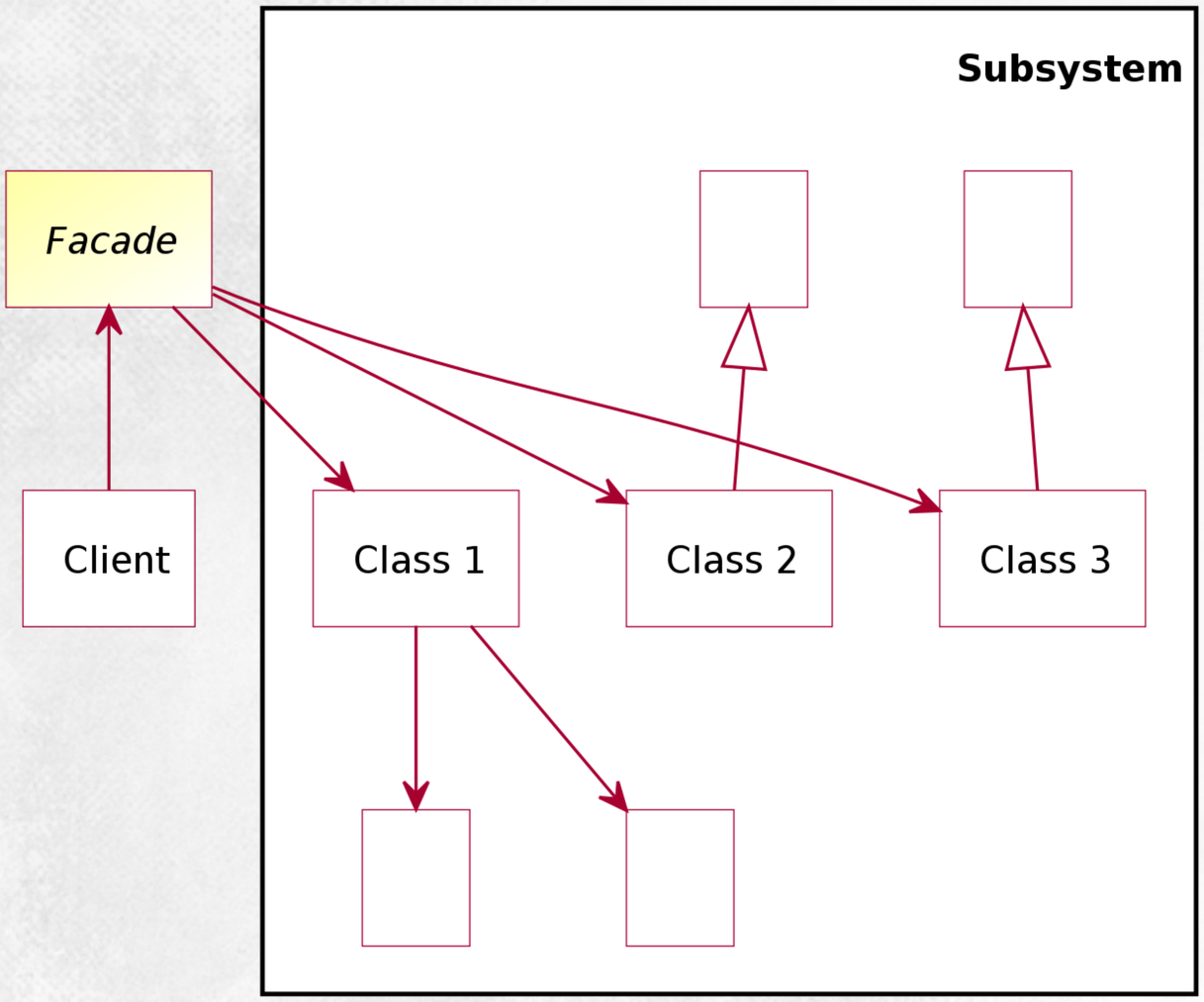
Façade Pattern (Detail)

- Commonly used with object-oriented programming. The name is an analogy to an architectural façade
- A facade is an object that provides a simplified interface to a larger body of code, such as a class library
- A facade can:
  - make a software library easier to use, understand, and test, since the facade has convenient methods for common tasks
  - make the library more readable, for the same reason
  - reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system
  - wrap a poorly-designed collection of APIs with a single well-designed API

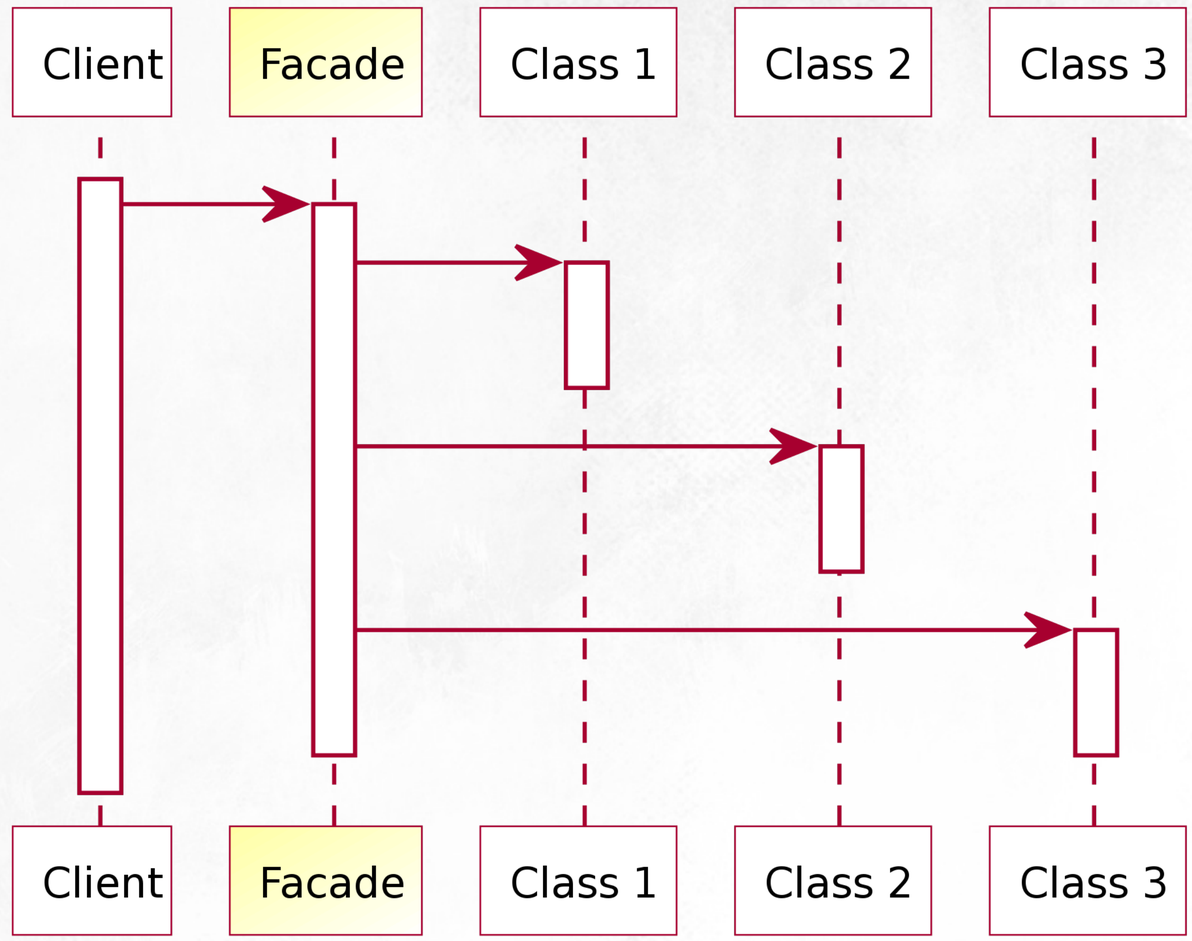
Façade Pattern (Detail)

- Implementors often use the facade design pattern when a system is very complex or difficult to understand because the system has a large number of interdependent classes or because its source code is unavailable
- This pattern hides the complexities of the larger system and provides a simpler interface to the client
- It typically involves a single wrapper class that contains a set of members required by the client
- These members access the system on behalf of the facade client and hide the implementation details

Façade Pattern (UMLs)

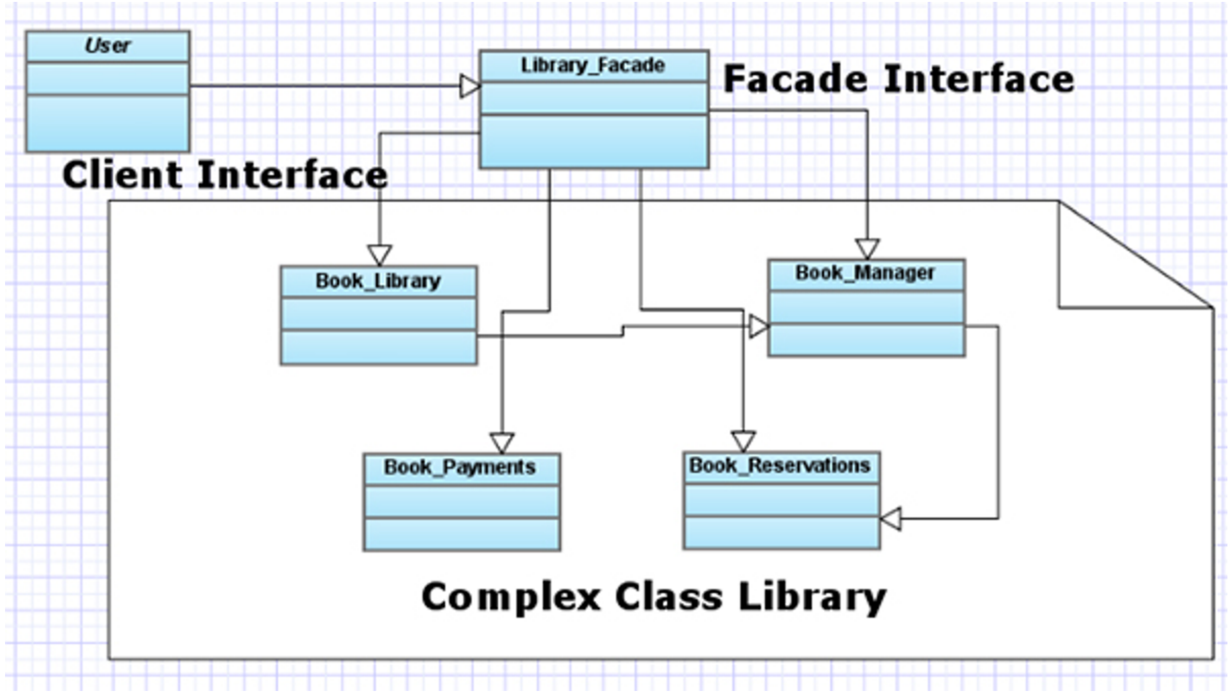


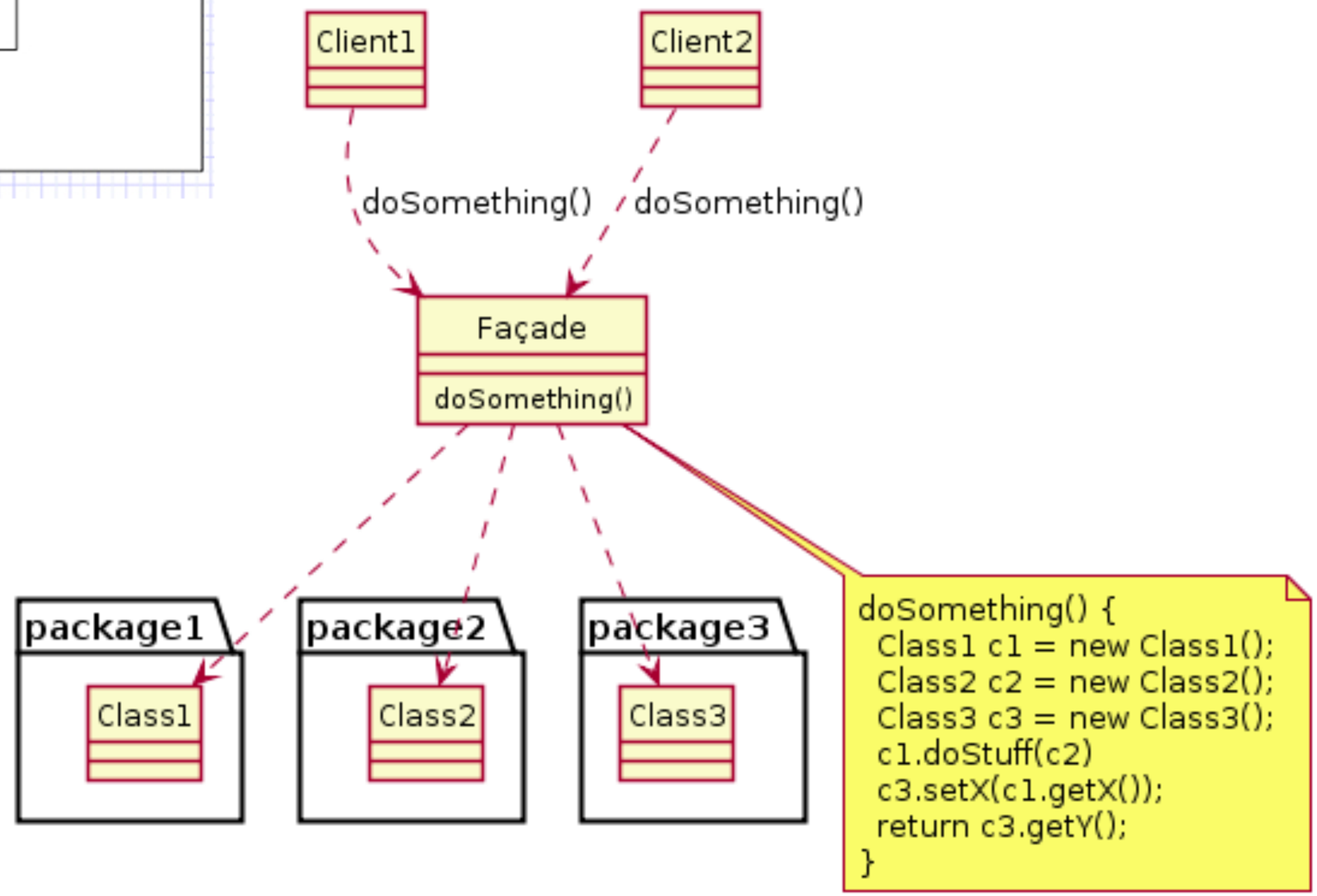
Sample class diagram



Sample sequence diagram

Façade Pattern (UMLs)





Façade Pattern (Example)

- Goal is Booting complex system using simple interface

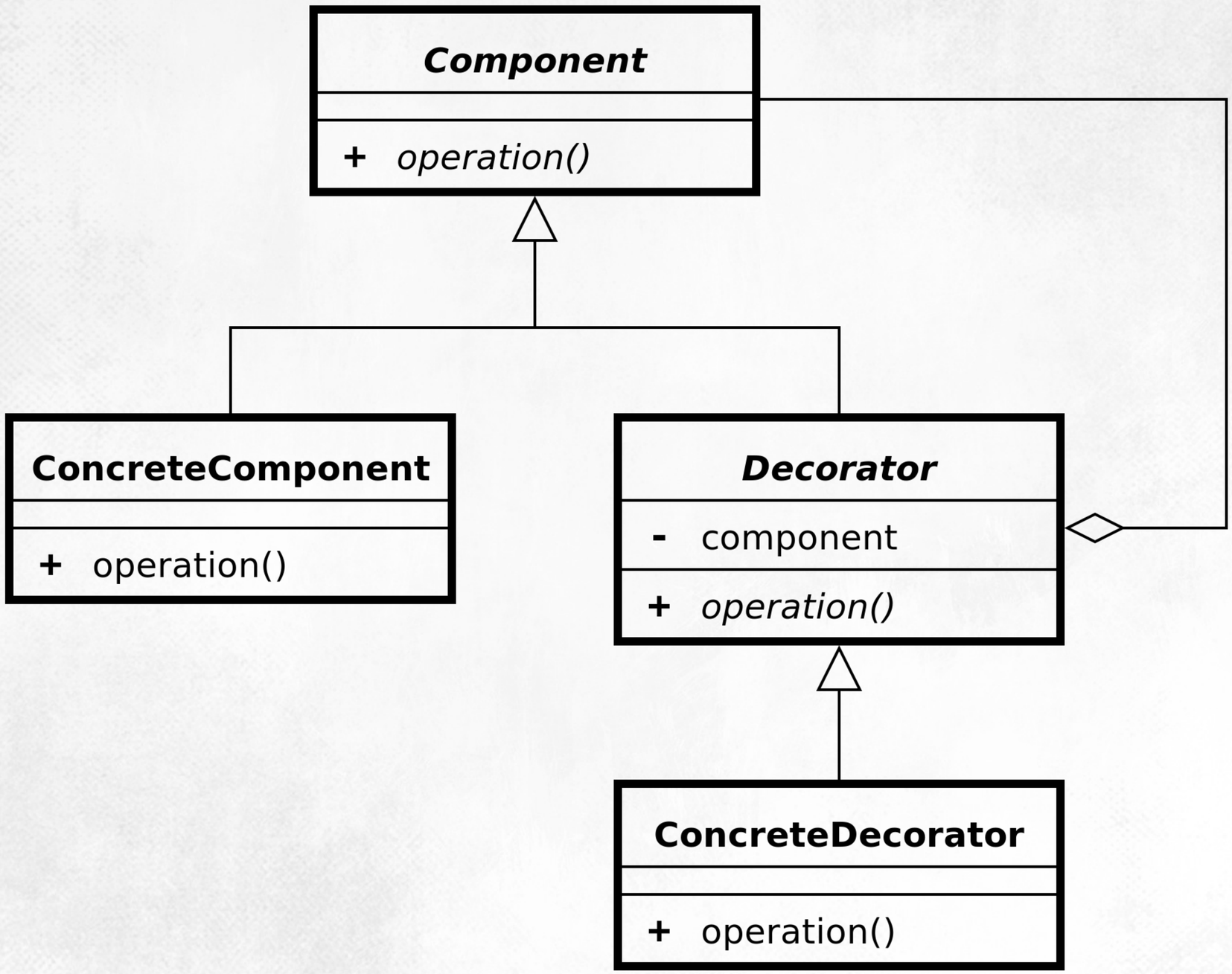
  https://en.wikipedia.org/wiki/Facade_pattern

Decorator Pattern (Goal)

- The Decorator design pattern solves problems like:
  - Responsibilities should be added to (and removed from) an object dynamically at run-time.
  - A flexible alternative to subclassing for extending functionality should be provided.
- What solution does the Decorator design pattern describe?
  - implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it
  - perform additional functionality before/after forwarding a request

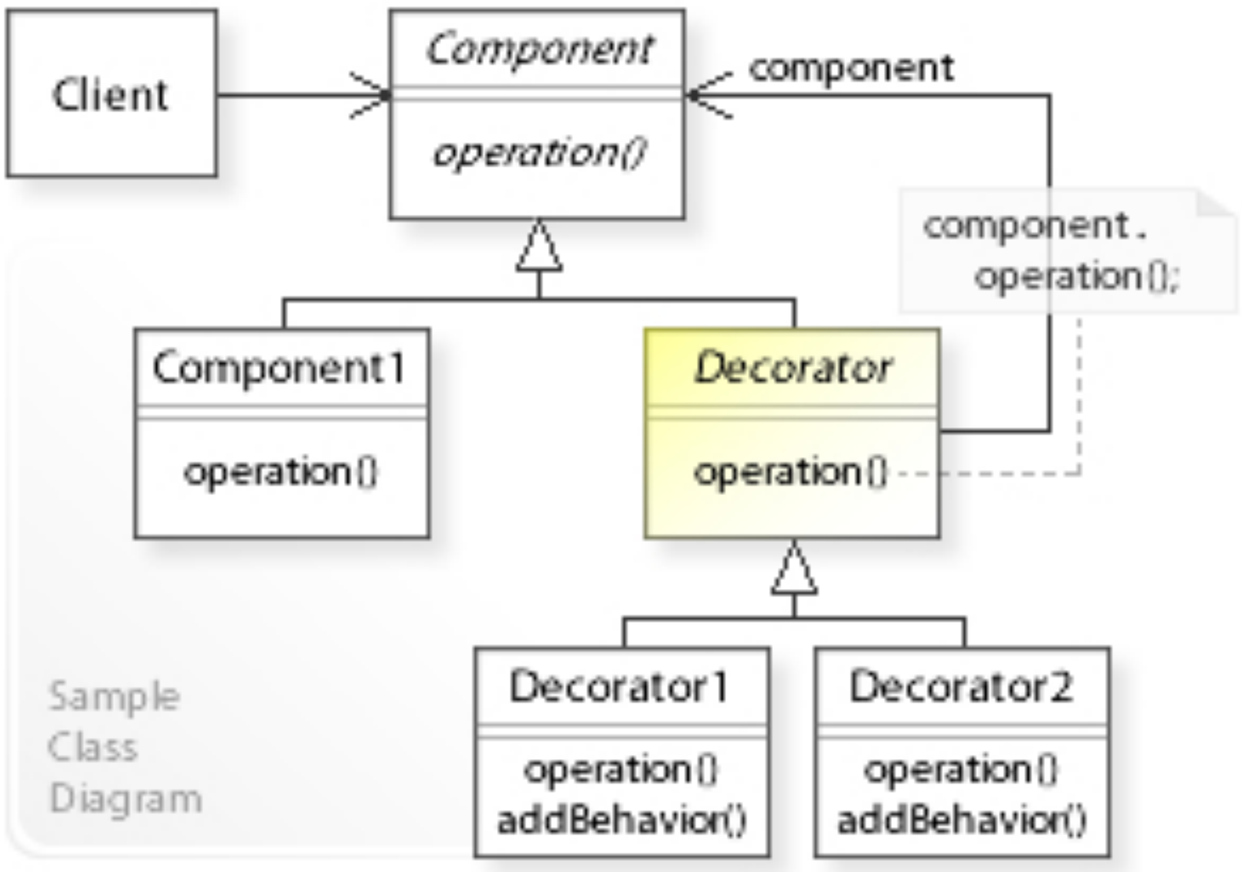
Reference: https://en.wikipedia.org/wiki/Decorator_pattern#UML_class_and_sequence_diagram

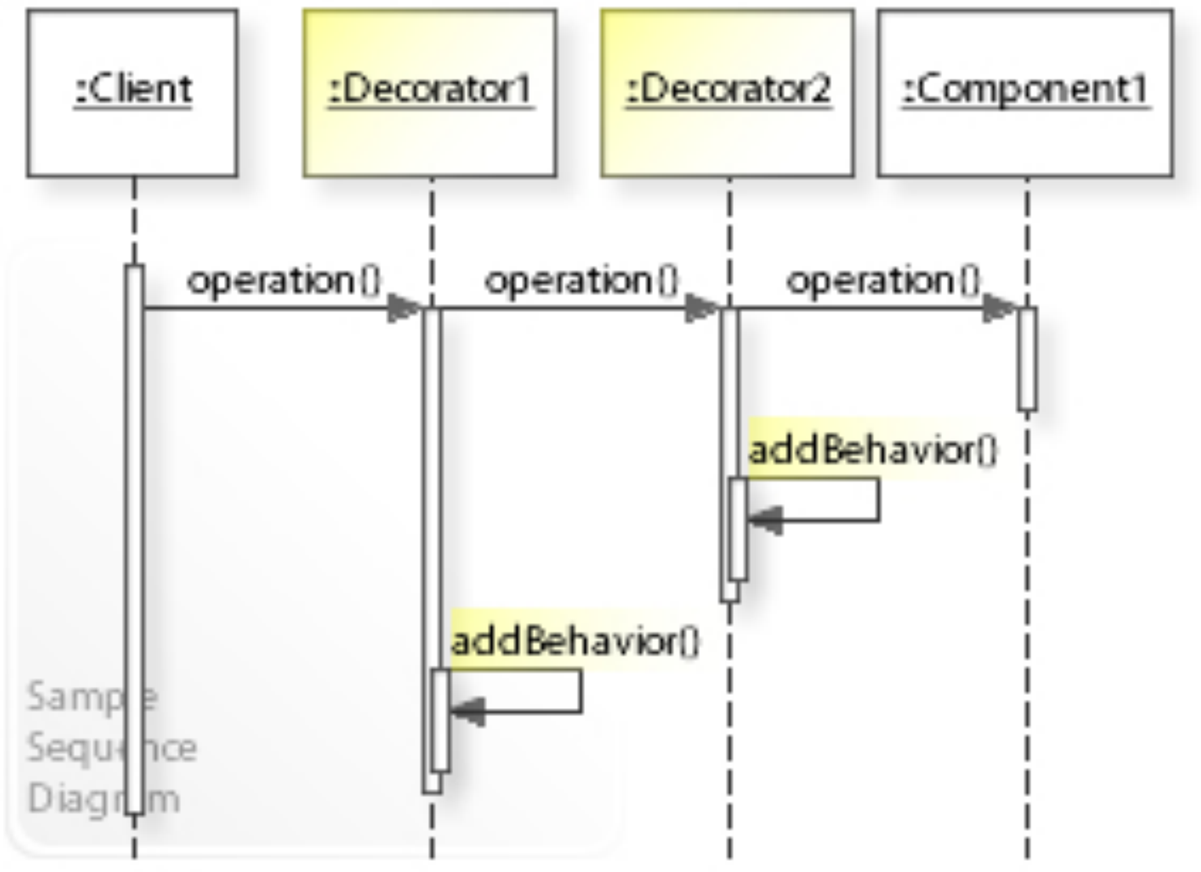
Decorator Pattern (UMLs)



Reference: https://en.wikipedia.org/wiki/Decorator_pattern#UML_class_and_sequence_diagram

Decorator Pattern (UMLs)





Reference: https://en.wikipedia.org/wiki/Decorator_pattern#UML_class_and_sequence_diagram

Decorator Pattern (Example)

- Add color to existing shape [1/2]

  https://en.wikipedia.org/wiki/Decorator_pattern#UML_class_and_sequence_diagram

```cpp
#include <iostream>
#include <string>

struct Shape {
  virtual ~Shape() = default;

  virtual std::string GetName() const = 0;
};

struct Circle : Shape {
  void Resize(float factor) { radius *= factor; }

  std::string GetName() const override {
    return std::string("A circle of radius ") + std::to_string(radius);
  }

  float radius = 10.0f;
};
```

Decorator Pattern (Example)

- Add color to existing shape [2/2]

  https://en.wikipedia.org/wiki/Decorator_pattern#UML_class_and_sequence_diagram

```
struct ColoredShape : Shape {
  ColoredShape(const std::string& color, Shape* shape)
      : color(color), shape(shape) {}

  std::string GetName() const override {
    return shape->GetName() + " which is colored " + color;
  }

  std::string color;
  Shape* shape;
};

int main() {
  Circle circle;
  ColoredShape colored_shape("red", &circle);
  std::cout << colored_shape.GetName() << std::endl;

}
```

Target to decorate

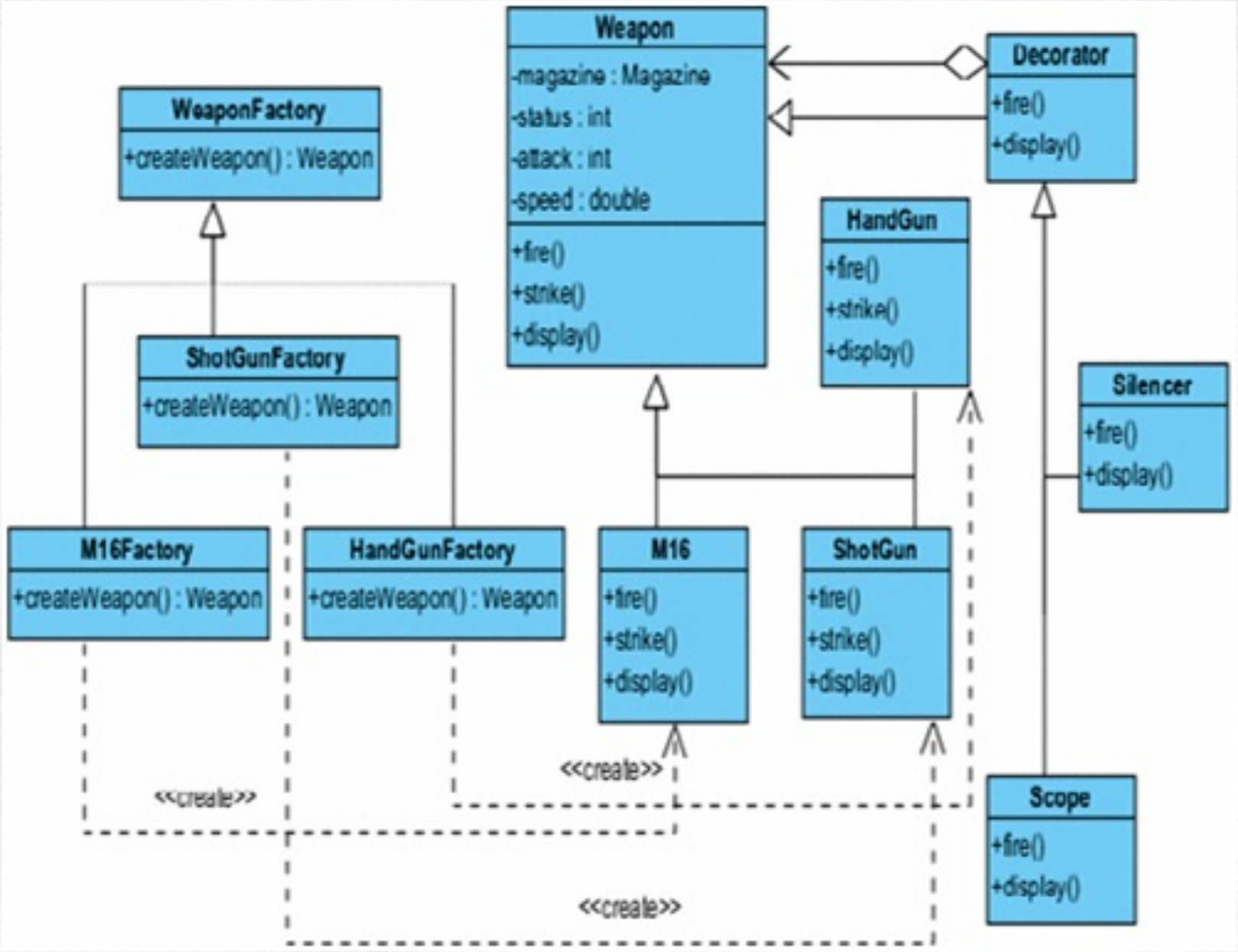
Target's original operation

Target's decorated operation

Decorator Pattern (Game)

- Design Patterns for FPS Game Weapons

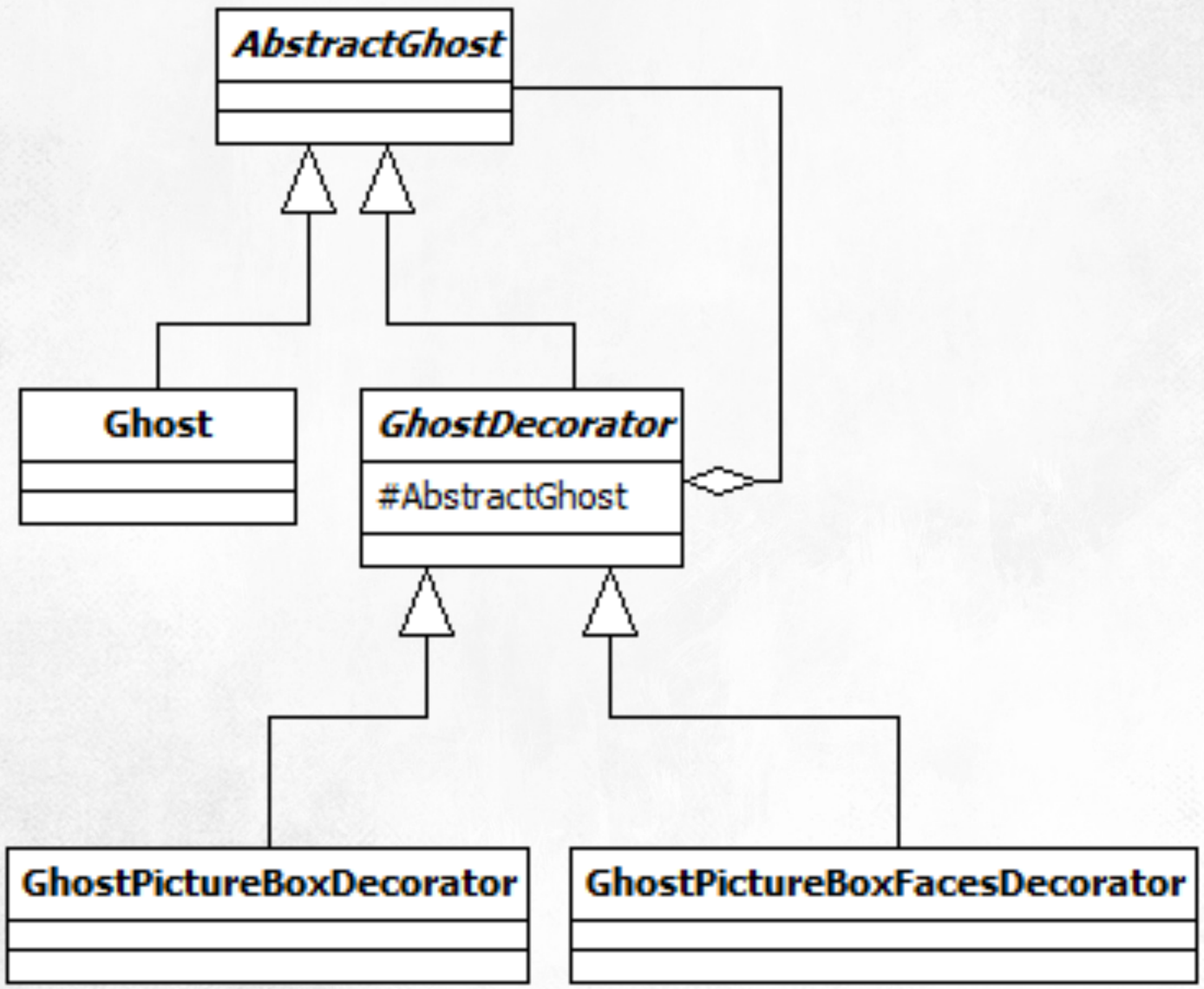
  https://dzone.com/articles/design-patterns-for-fps-game-weapons-1



Decorator Pattern (Game)

- Decorator Pattern Usage on a Simplified Pacman Game

  https://www.codeproject.com/Articles/690410/Decorator-pattern-usage-on-a-simplified-Pacman-gam



History of Design Pattern

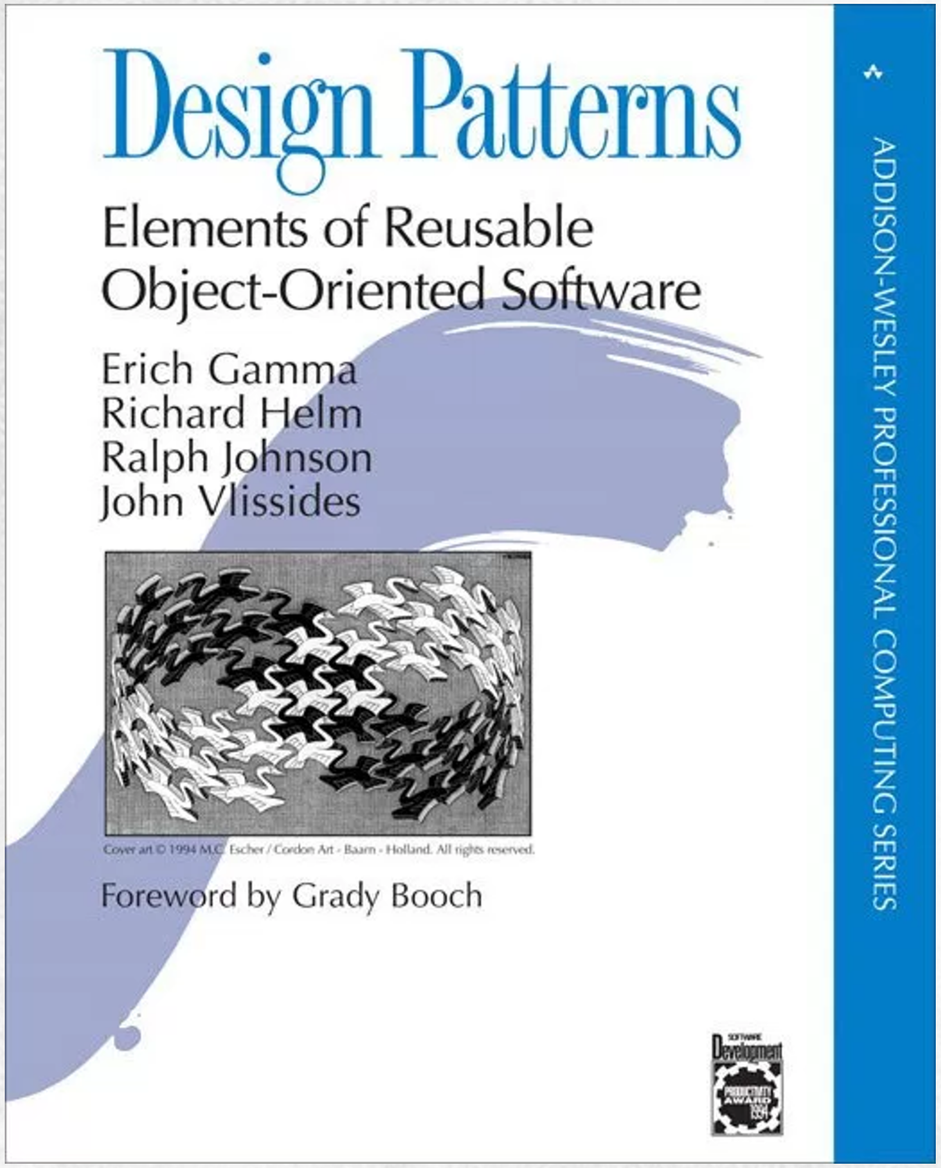
- Patterns originated as an architectural concept by Christopher Alexander (1977/79)
- In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming – specifically pattern languages – and presented their results at the OOPSLA conference that year
- Design patterns gained popularity in computer science after the book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994 by the so-called "Gang of Four" (Gamma et al.), which is frequently abbreviated as "GoF"

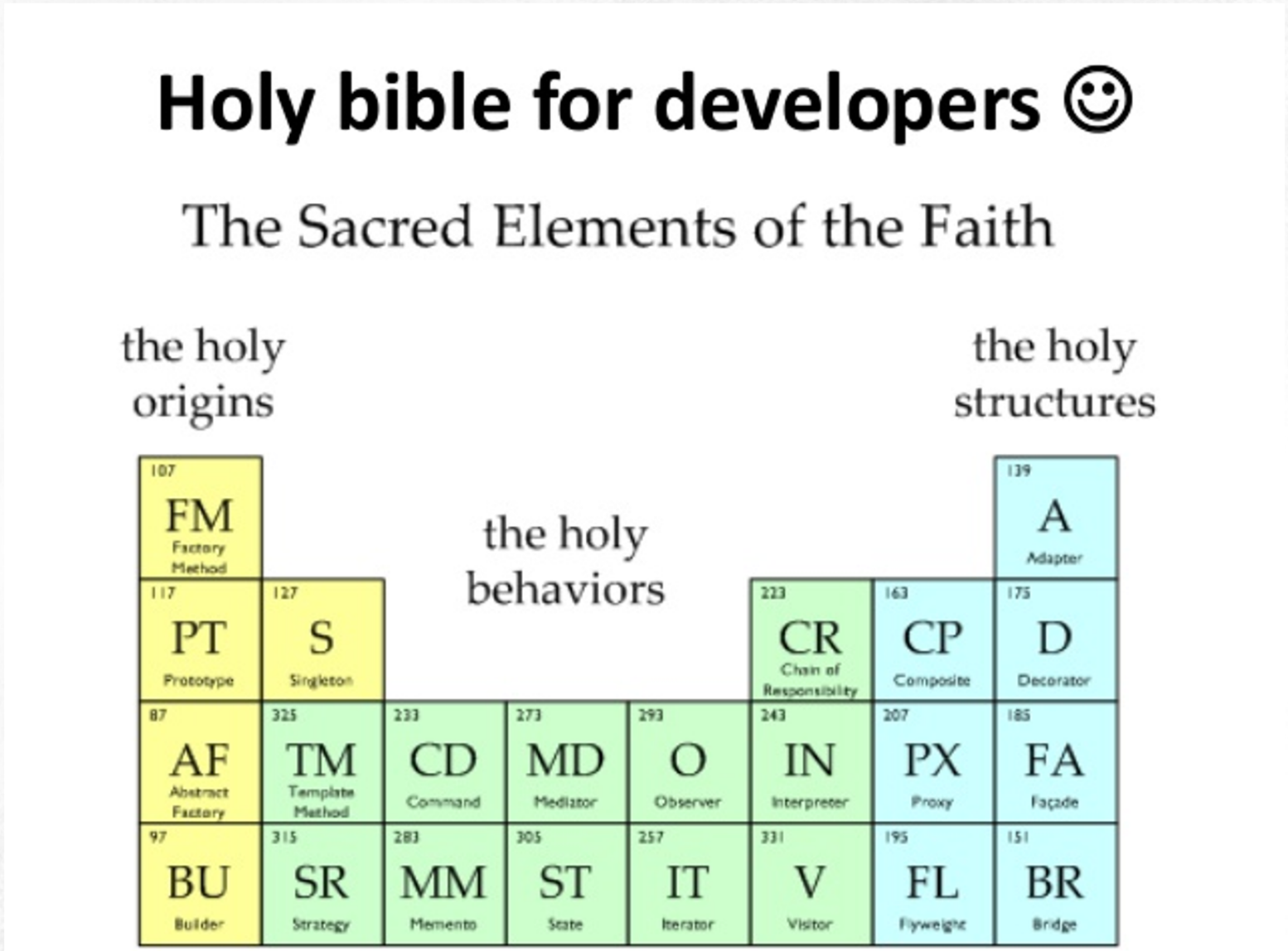
Reference: https://en.wikipedia.org/wiki/Software_design_pattern

Design Patterns by GoF (Who?)

Reference: https://www.quora.com/How-important-are-design-patterns-in-software-development

Design Patterns by GoF (the Book)



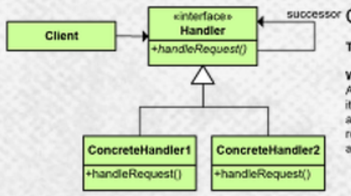


Reference: https://en.wikipedia.org/wiki/Software_design_pattern
https://www.slideshare.net/alimenkou/the-modern-view-on-implementation-of-classic-design-patterns-in-java

Design Patterns by GoF (23 patterns for Java)

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

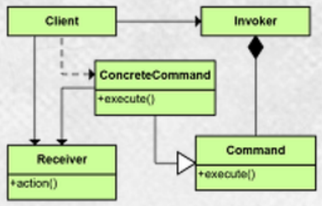
Chain of Responsibility



Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

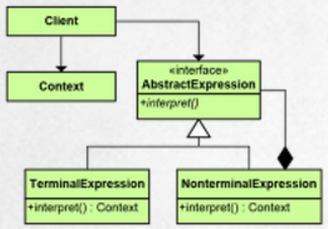
Command



Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

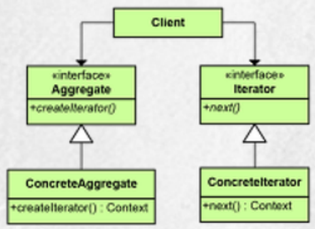
Interpreter



Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

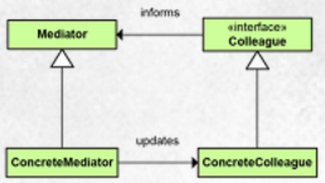
Iterator



Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator

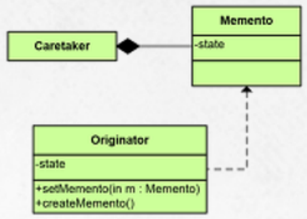


Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison Wesley Longman, Inc.

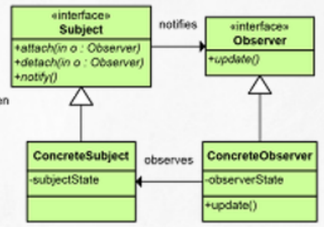
Memento



Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

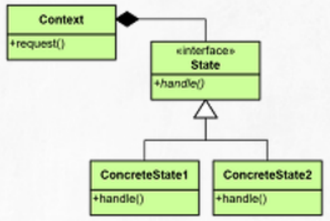
Observer



Type: Behavioral

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

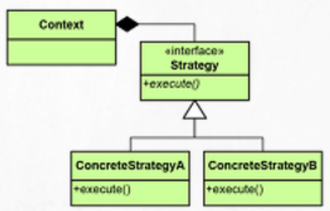
State



Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

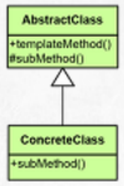
Strategy



Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

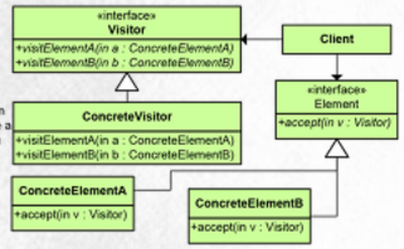
Template Method



Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

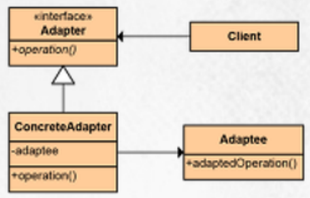
Visitor



Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

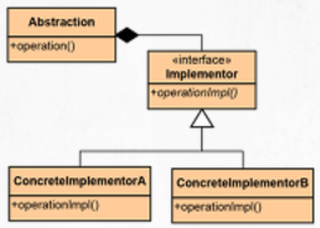
Adapter



Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

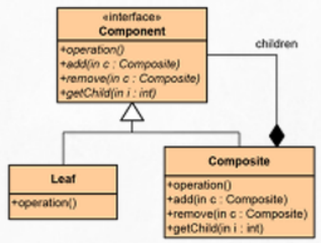
Bridge



Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

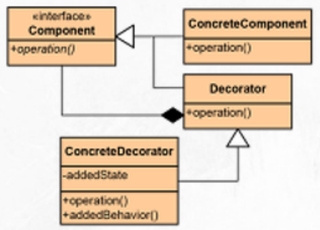
Composite



Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

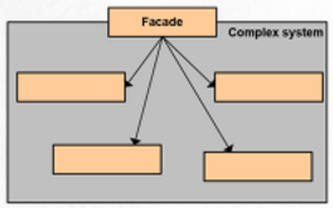
Decorator



Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

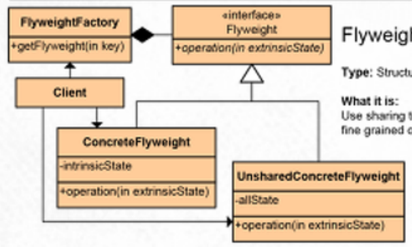
Facade



Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Flyweight

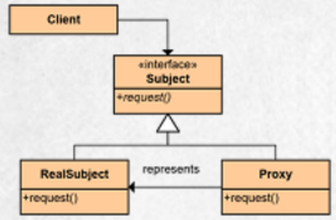


Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison Wesley Longman, Inc.

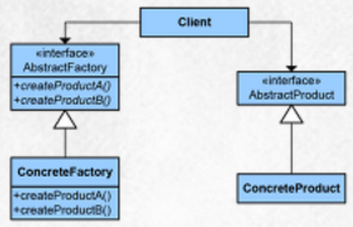
Proxy



Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

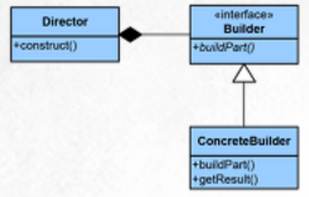
Abstract Factory



Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

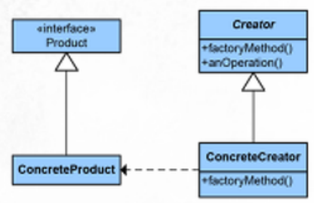
Builder



Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.

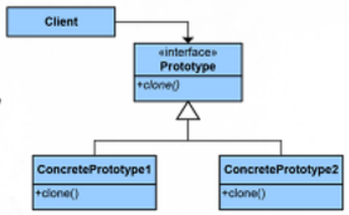
Factory Method



Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

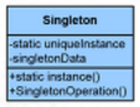
Prototype



Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton

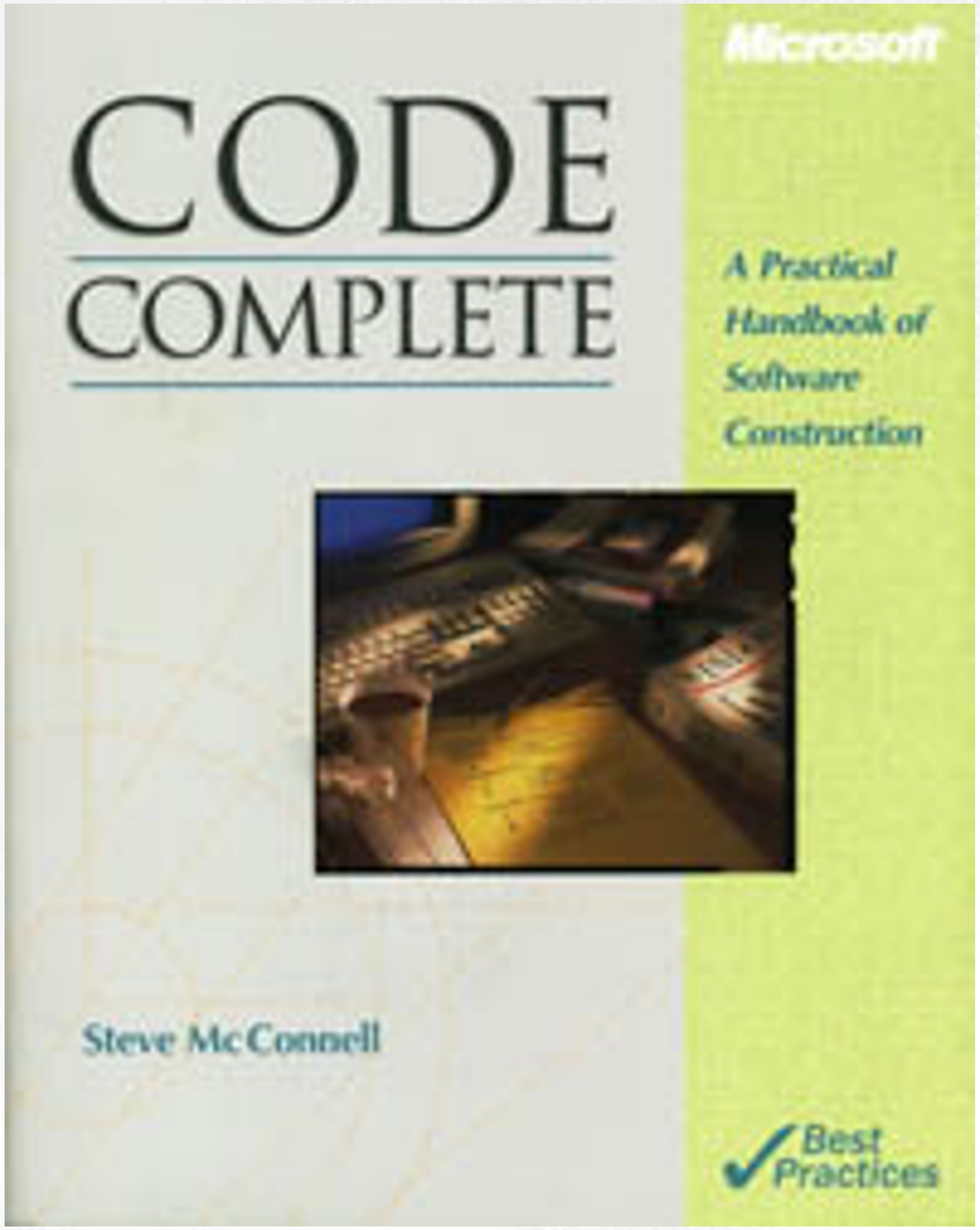


Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

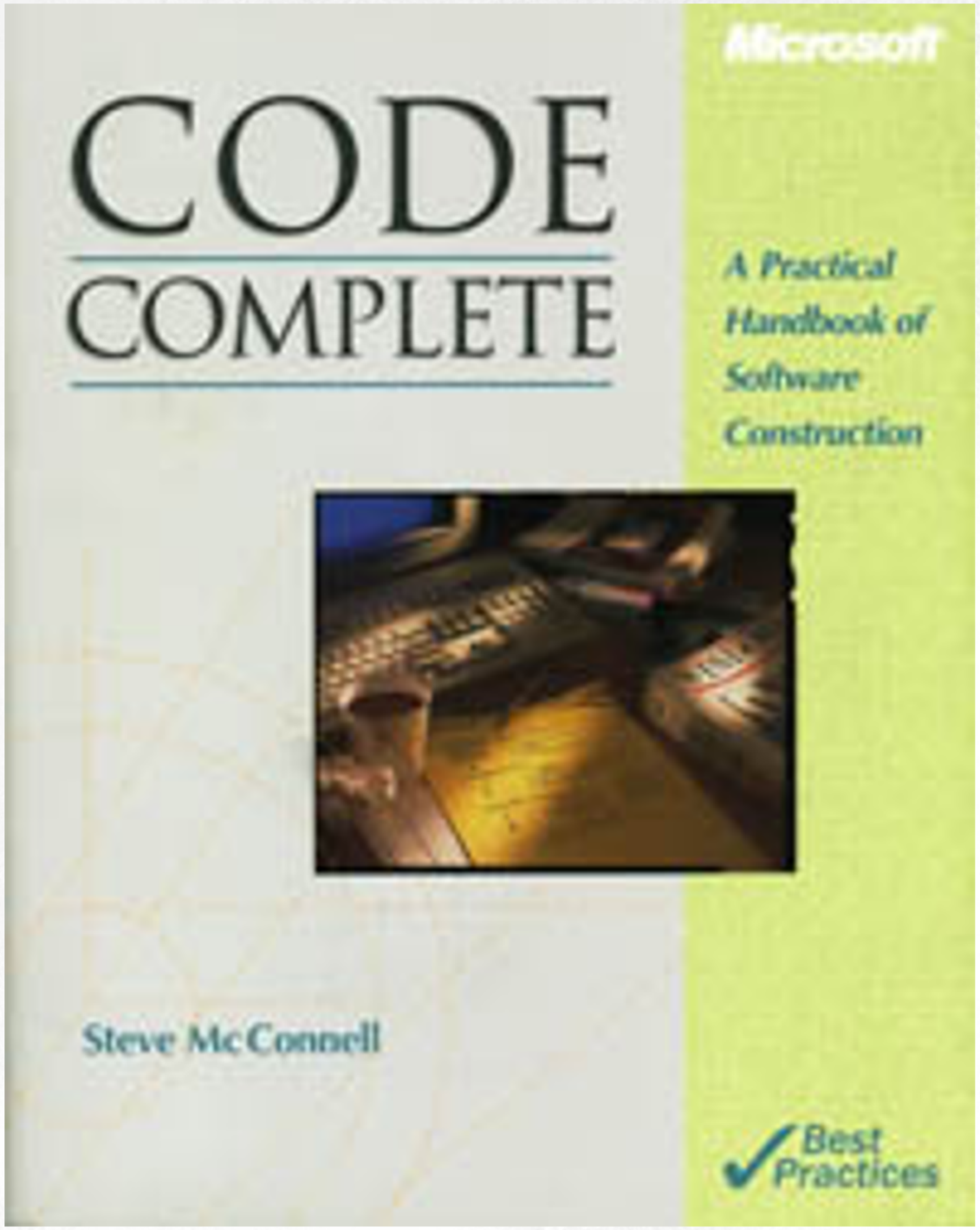
Before the Design Pattern of GoF

- Code Complete
  - is a software development book, written by Steve McConnell and published in 1993 by Microsoft Press
  - encouraging developers to continue past code-and-fix programming and the big design up front & waterfall models
  - is also a compendium of software construction techniques, which include techniques from naming variables to deciding when to write a subroutine



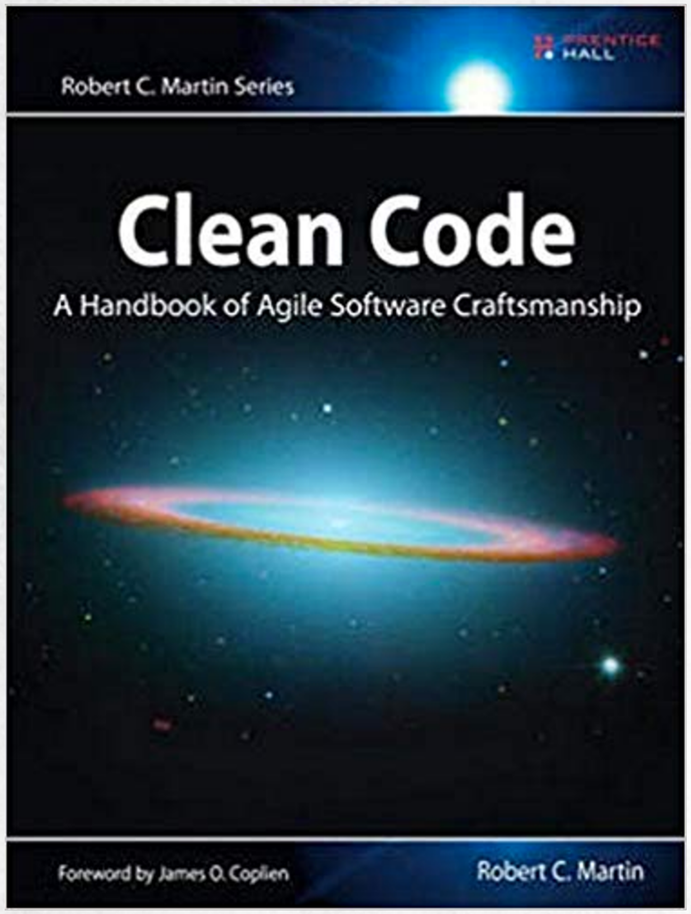
Before the Design Pattern of GoF

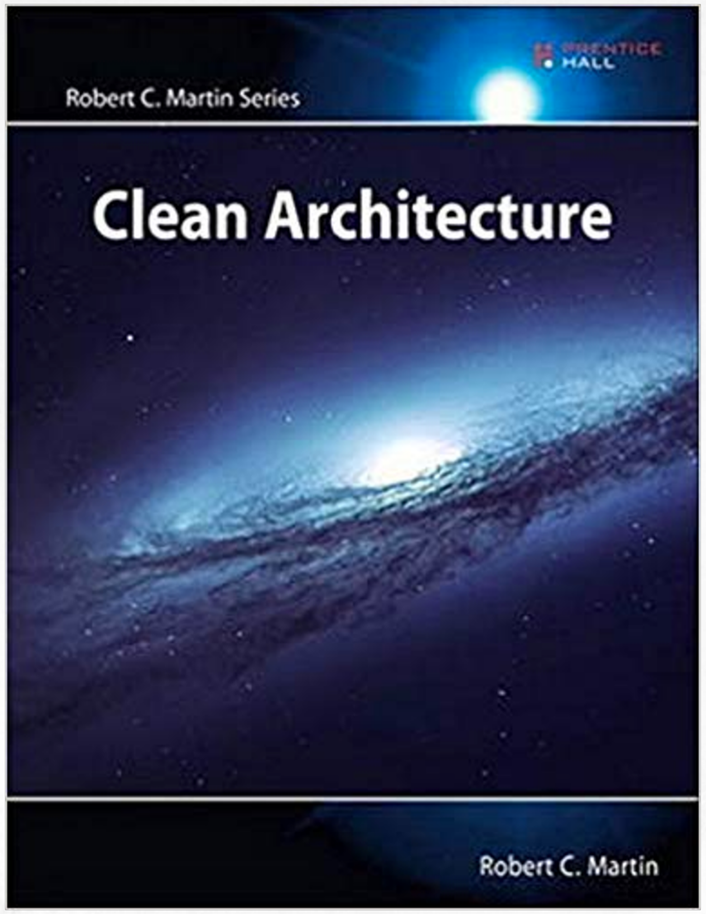
- Code Complete defines the main activities in construction as:
  - detailed design
  - construction planning
  - coding and debugging
  - unit testing
  - integration, and
  - integration testing

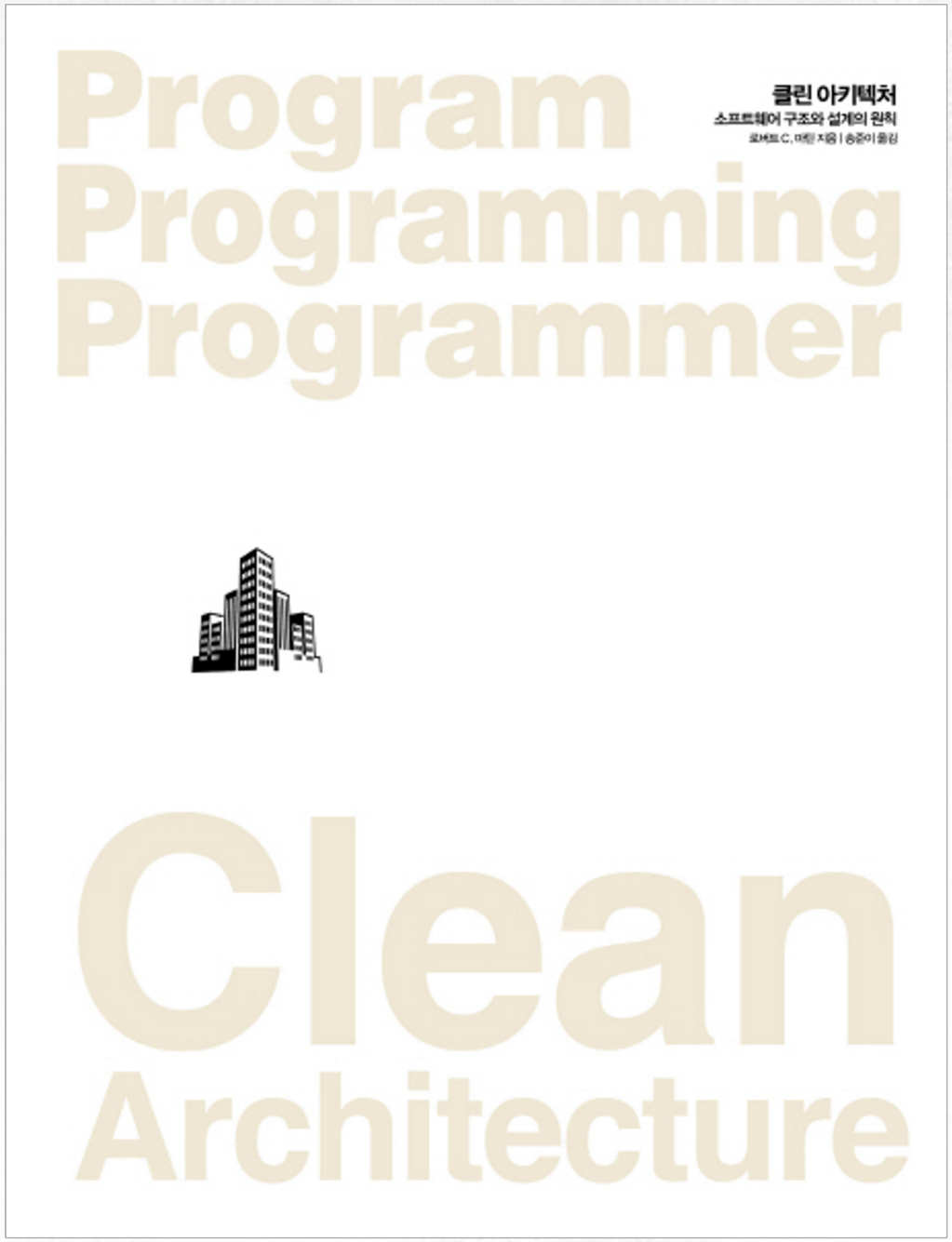


Reference: https://en.wikipedia.org/wiki/Code_Complete

Latest Books on Software Design







GoF vs. Code Complete

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

Creational patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[14]	Other
Abstract factory	Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.	Yes	Yes	—
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.	Yes	No	—
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.	No	No	—
Factory method	Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	—
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.	No	No	PoEAA[15]
Multiton	Ensure a class has only named instances, and provide a global point of access to them.	No	No	—
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	—
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.	Yes	No	—
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.	No	No	—
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	—

GoF vs. Code Complete (continued)

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

Structural patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[14]	Other
Adapter, Wrapper, or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.	Yes	Yes	—
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	—
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	—
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	—
Extension object	Adding functionality to a hierarchy without changing the hierarchy.	No	No	Agile Software Development, Principles, Patterns, and Practices[16]
Facade	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.	Yes	Yes	—
Flyweight	Use sharing to support large numbers of similar objects efficiently.	Yes	No	—
Front controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.	No	No	J2EE Patterns[17] PoEAA[18]
Marker	Empty interface to associate metadata with a class.	No	No	Effective Java[19]
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.	No	No	—
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	—
Twin[20]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.	No	No	—

GoF vs. Code Complete (continued)

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

Behavioral patterns [edit]

Name	Description	In *Design Patterns*	In *Code Complete*[14]	Other
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)	No	No	—
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	—
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.	Yes	No	—
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	—
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	—
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.	Yes	No	—
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	No	—
Null object	Avoid null references by providing a default object.	No	No	—
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.	Yes	Yes	—
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.	No	No	—
Specification	Recombinable business logic in a Boolean fashion.	No	No	—
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.	Yes	No	—
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes	Yes	—
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes	Yes	—
Visitor	Represent an operation to be performed on instances of a set of classes. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.	Yes	No	—
Fluent Interface	Design an API to be method chained so that it reads like a DSL. Each method call returns a context through which the next logical method call(s) are made available.	No	No	—

GoF vs. Code Complete (continued)

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

Concurrency patterns [edit]

Name	Description	In *POSA2*[21]	Other
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.	Yes	—
Balking	Only execute an action on an object when the object is in a particular state.	No	—
Binding properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[22]	No	—
Compute kernel	The same calculation many times in parallel, differing by integer parameters used with non-branching pointer math into shared arrays, such as GPU-optimized Matrix multiplication or Convolutional neural network.	No	—
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.	Yes	—
Event-based asynchronous	Addresses problems with the asynchronous pattern that occur in multithreaded programs.[23]	No	—
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.	No	—
Join	Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high-level programming model.	No	—
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[24]	No	PoEAA[15]
Messaging design pattern (MDP)	Allows the interchange of information (i.e. messages) between components and applications.	No	—
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.	Yes	—
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.	Yes	—
Read-write lock	Allows concurrent read access to an object, but requires exclusive access for write operations. An underlying semaphore might be used for writing, and a Copy-on-write mechanism may or may not be used.	No	—
Scheduler	Explicitly control when threads may execute single-threaded code.	No	—
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.	No	—
Thread-specific storage	Static or "global" memory local to a thread.	Yes	—
Safe Concurrency with Exclusive Ownership	Avoiding the need for runtime concurrent mechanisms, because exclusive ownership can be proven. This is a notable capability of the Rust language, but compile-time checking isn't the only means, a programmer will often manually design such patterns into code - omitting the use of locking mechanism because the programmer assesses that a given variable is never going to be concurrently accessed.	No	—
CPU atomic operation	x86 and other CPU architectures support a range of atomic instructions that guarantee memory safety for modifying and accessing primitive values (integers). For example, two threads may both increment a counter safely. These capabilities can also be used to implement the mechanisms for other concurrency patterns as above. The C# language uses the Interlocked class for these capabilities.	No	—

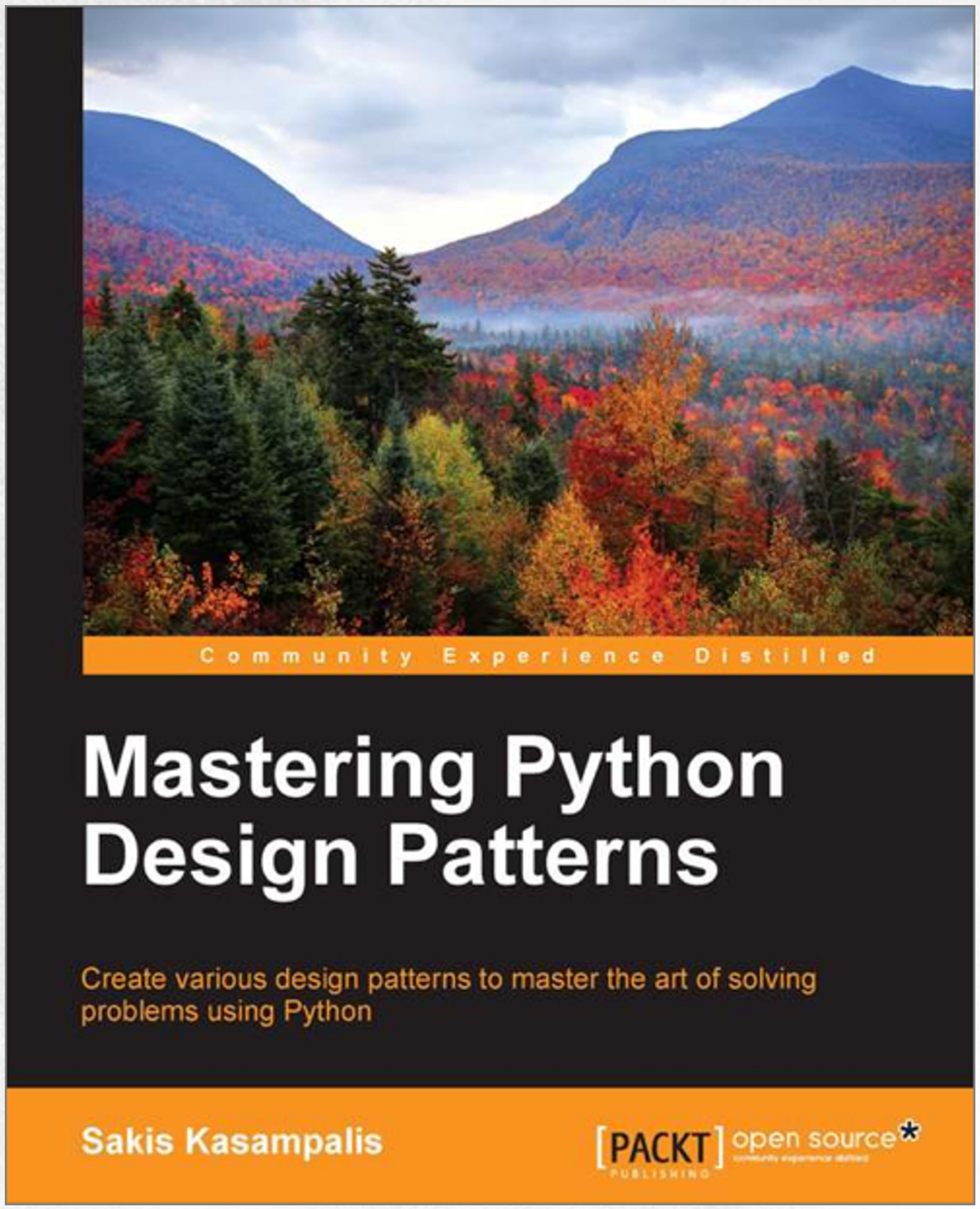
Criticism about Design Pattern

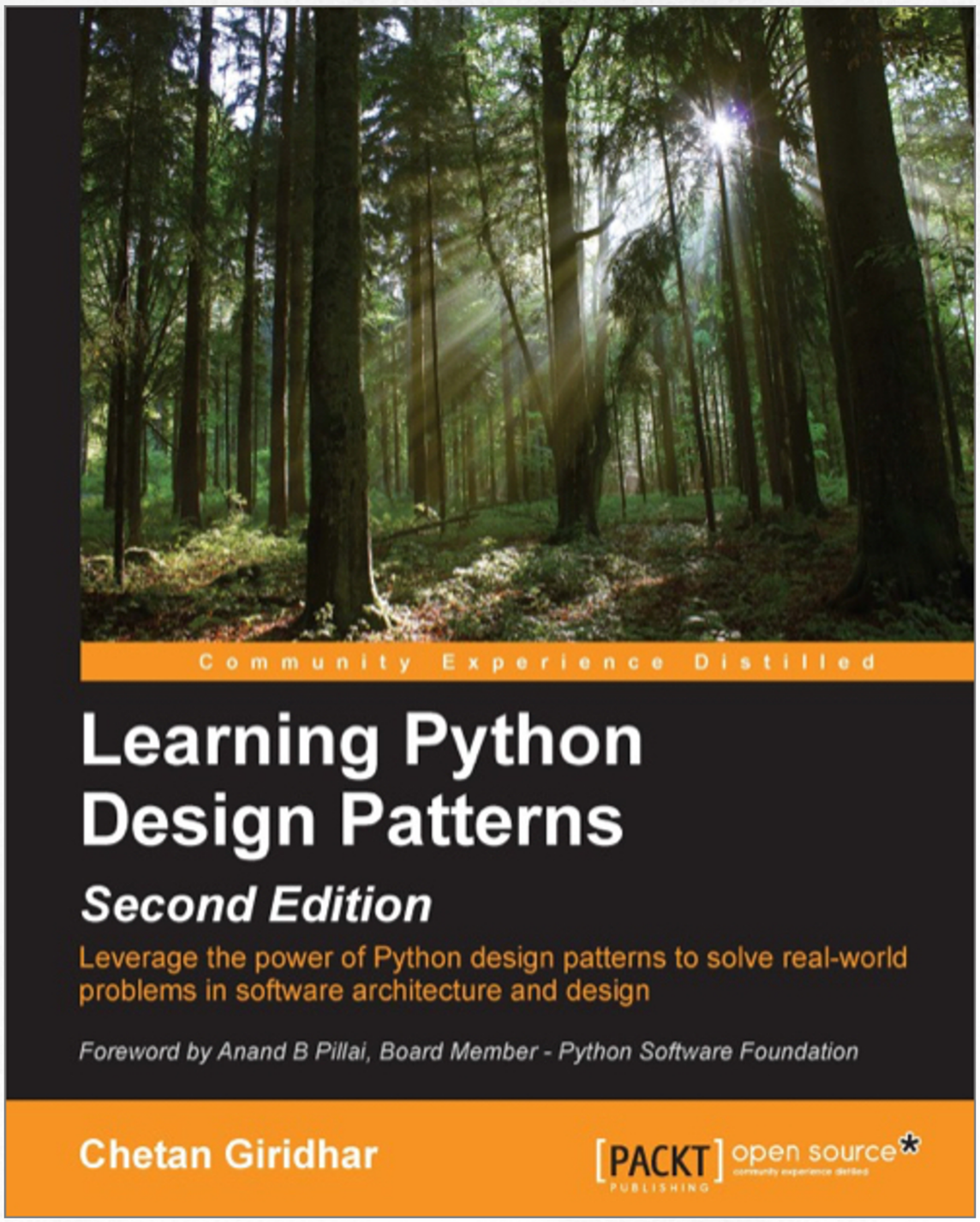
- Poor language?
  - The design patterns may just be a sign of some missing features of a given programming language (Java or C++ for instance)
  - Peter Norvig demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan
- Moreover, inappropriate use of patterns may unnecessarily increase complexity
- Another point of criticism is the lack of an updated version since the Design Patterns book was published in 1994

Summary

- Design Patterns
- Experience sharing between programmers

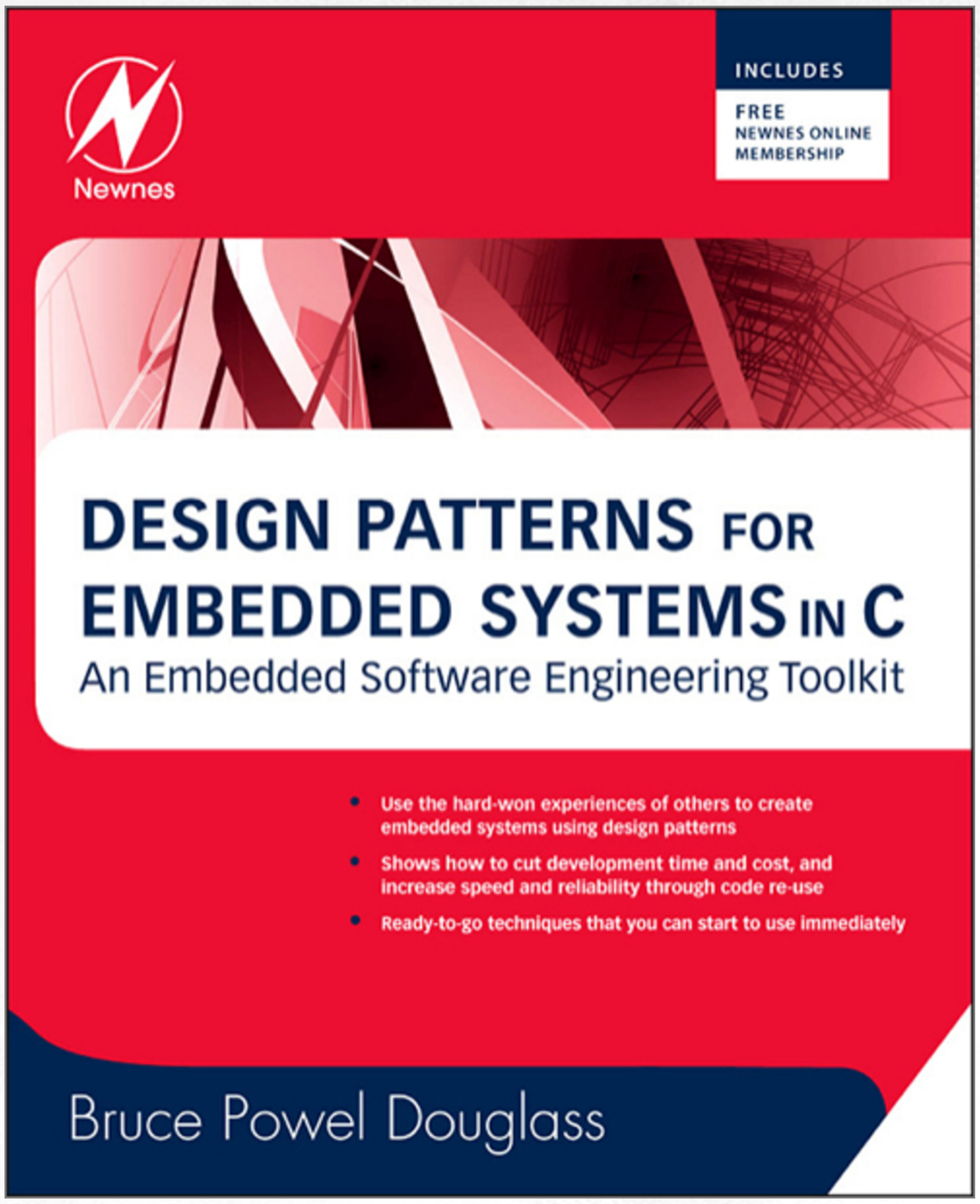
Reference


Community Experience Distilled
Mastering Python Design Patterns
Create various design patterns to master the art of solving problems using Python
Sakis Kasampalis
[PACKT] open source
PUBLISHING


Community Experience Distilled
Learning Python Design Patterns
Second Edition
Leverage the power of Python design patterns to solve real-world problems in software architecture and design
Foreword by Anand B Pillai, Board Member - Python Software Foundation
Chetan Giridhar
[PACKT] open source
PUBLISHING

Reference

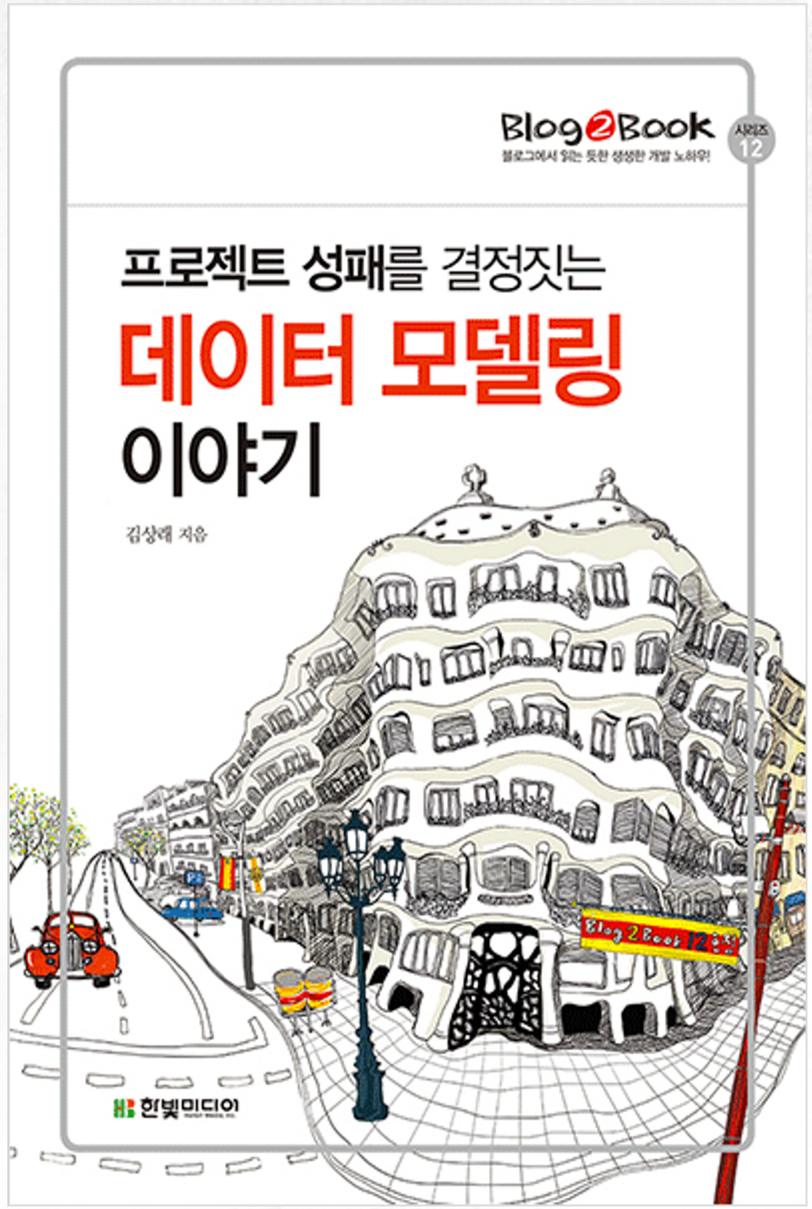

O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates


Newnes
INCLUDES
FREE
NEWNES ONLINE
MEMBERSHIP
DESIGN PATTERNS FOR
EMBEDDED SYSTEMS IN C
An Embedded Software Engineering Toolkit
Use the hard-won experiences of others to create embedded systems using design patterns
Shows how to cut development time and cost, and increase speed and reliability through code re-use
Ready-to-go techniques that you can start to use immediately
Bruce Powel Douglass

Reference


팟캐스트
나는 프로그래머다
2탄
알맹이만 쏙쏙
방송보다 더 밀도 높게
더 유용하게
아이튠즈 팟캐스트 2위
누적 100만 다운로드
개발자의 가려운 곳을 긁어주마!
IT인이라면 누구나 울고 웃으며 공감할
개발과 삶에 대한 나·프·다의 못다 한 이야기!
한빛미디어


IT 분야의 필수 자기계발서
IT 개발자, 엔지니어, 컨설턴트의 길을
걸어가는 당신을 위한 안내서
IT 전문가로 사는 법
장동인 지음
한빛미디어


Blog2Book
사이즈 12
프로젝트 성패를 결정짓는
데이터 모델링 이야기
김상래 지음
한빛미디어

Reference


GAME X PROGRAMER
게임
프로그래머로
산다는 것


개발자의
글쓰기
변수 네이밍부터
릴리스 노트,
장애 보고서,
기술 블로그까지,
프로그래머의 글쓰기
고민 끝!


Program
Programming
Programmer
실용주의 프로그래머
The
Pragmatic
Programmer

Homework

- Selects a SINGLE pattern, and surveys [goal, detail, UML, code]

Thank you

경희대학교
KYUNG HEE UNIVERSITY