Design Defects & Restructuring

Week 1: 27 Aug 2022

Rahim Hasnani

Introduction

- Instructor Intro
- Style: More of facilitator
- Motivation to teach this course
- Students' introduction & expectations

Course Outline & Policies

- Outline
- Whether this course is right for you
- Effort expectation
- Assignment copying/similarity policy
- Late submission policy
- Projects Discussion
- Home works

Recap OOP

- Class, Object, Attributes, Behaviors
- Inheritance, encapsulation, overloading vs overriding
- Virtual functions/methods
- Static members and Static functions
- Abstract class
- Interfaces
- Abstract class vs interface
- Static Class
- Inner Classes
- Struct vs Class

Recap Elementary OOA&D

- Basic relationships
  - Is-A
  - Has-A
  - Association
- Nouns and Verbs
- Basic Class Design

Exercise:

- Provide definition and implementation of a class called IntList. The class should use a fixed sized array to store numbers and should implement the following.

a. Allow user to add an element (a number) to the list
b. Allow user to remove a number from the list by specifying its position (index)
c. Allow user to clear the list
d. Allow user to find out the number of elements in the list
e. Throw an exception if the list has no capacity during the add operation
f. Provide arithmetic mean of the list.
g. Provide range of the list (e.g., Range of { 3, 6, 7, 10, 4, 2 } is 9).

Exercise

- Design classes for the following scenario:

Each Item has a code, name and a price. A Food Item additionally captures the expiry date and a flag identifying whether it is suitable for vegetarians. Provide class definitions identifying data members, accessors (member functions to get/set properties), inheritance relationships (if any), operations and access specifiers.

Design Defects & Restructuring

Week 2: 02 Sep 22

Rahim Hasnani

Agenda

- Last Week Left Overs
  - Points Distribution
- Home works
- Any questions on last week
- OOA & D Exercise
- Cohesion & Coupling
- SOLID
- GOF Patterns – Introduction
- Expect an assignment today!

Exercise

- *GoldSoft has recently won a project for development of an inventory system. The inventory system is to be developed for a big distributor with over 1,000 products. The distributor has warehouses where products are stored. The distributor currently carries products of five principals (manufacturers) and this is likely to increase with time. [The main business of a distributor is to order and carry products from principals and distribute them to wholesalers]. The major transactions in the inventory system are 1) Receipt of products from principal 2) Distribution of products to wholesaler 3) Transfer of stock from one warehouse to another 4) Stock adjustment as a result of stock count. Each of these transactions involves a number of products in different quantities (i.e., a receipt usually involves tens or even hundreds of different products in different quantities). Products have different unit of measures (some products are distributed as individual units like keyboard, some are distributed in meters like network wire, some are distributed in Kgs like Sugar, etc).*

Cohesion & Coupling

- Please see the other ppt

SOLID Principles

- Please read about this…

Cohesion and Coupling

CS 4311

Frank Tsui, Orland Karam, and Barbara Bernal, *Essential of Software Engineering*, 3rd edition, Jones & Bartett Learning. Section 8.3.

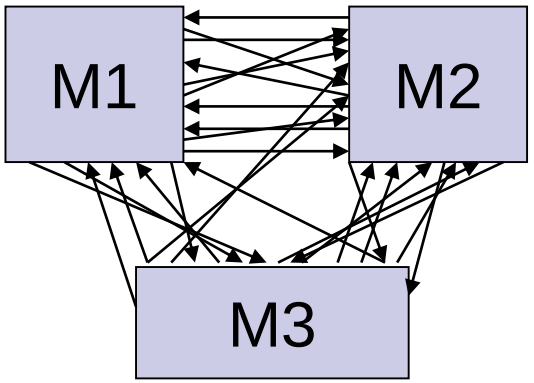
Hans van Vliet, *Software Engineering, Principles and Practice*, 3rd edition, John Wiley & Sons, 2008. (Section 12.1)

Outline

- Cohesion
- Coupling

Characteristics of Good Design

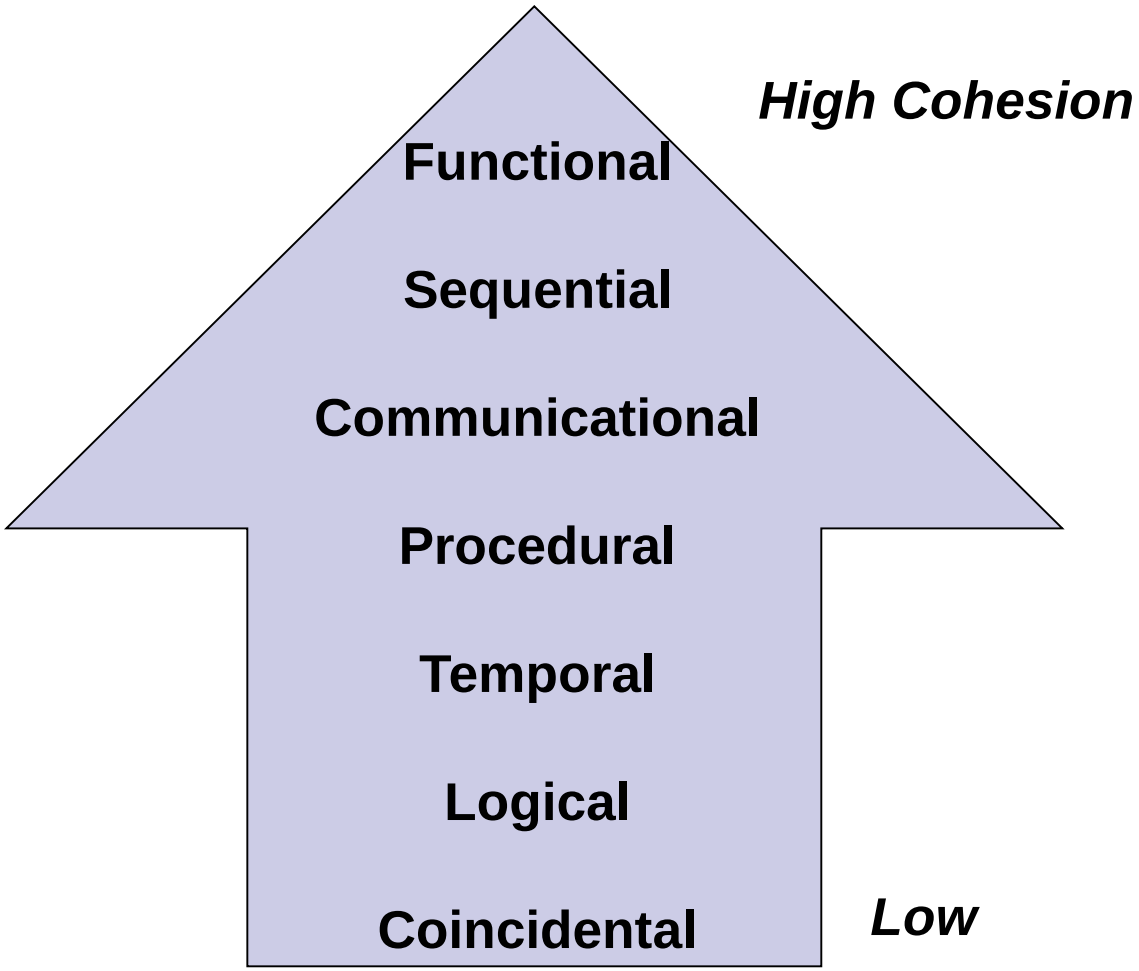
- Component independence
  - High cohesion
  - Low coupling
- Exception identification and handling
- Fault prevention and fault tolerance
- Design for change



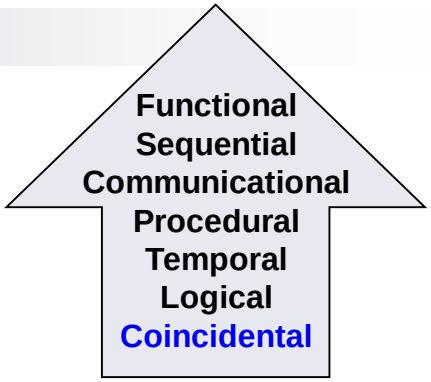
Cohesion

- Definition
  - The degree to which all elements of a component are directed towards a single task.
  - The degree to which all elements directed towards a task are contained in a single component.
  - The degree to which all responsibilities of a single class are related.
- Internal glue with which component is constructed
- All elements of component are directed toward and essential for performing the same task.

Type of Cohesion



Coincidental Cohesion

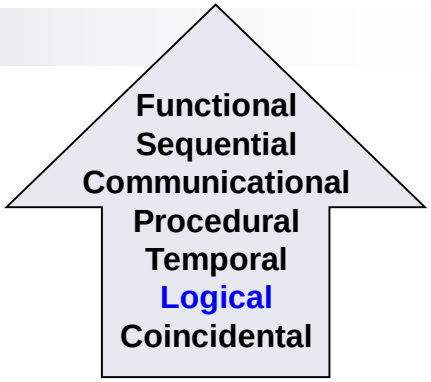


- Def: Parts of the component are unrelated (unrelated functions, processes, or data)
- Parts of the component are only related by their location in source code.
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

Example

1. Print next line
2. Reverse string of characters in second argument
3. Add 7 to 5th argument
4. Convert 4th argument to float

Logical Cohesion



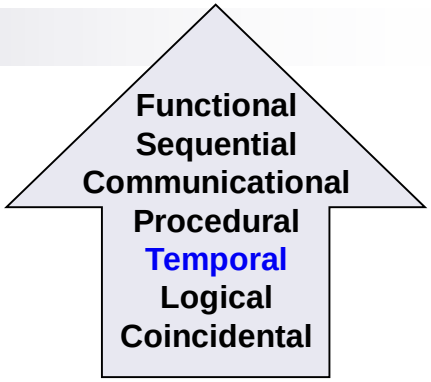
- Def: Elements of component are related logically and not functionally.
- Several logically related elements are in the same component and one of the elements is selected by the client component.

Example

- A component reads inputs from tape, disk, and network.
- All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

Improvement?

Temporal Cohesion



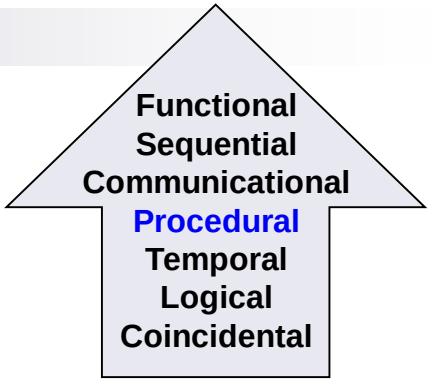
- Def: Elements are related by timing involved
- Elements are grouped by when they are processed.
- Example: An exception handler that
  - Closes all open files
  - Creates an error log
  - Notifies user
  - Lots of different activities occur, all at same time

Example

- A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

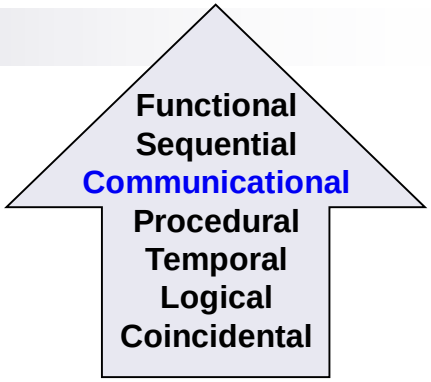
  Improvement?

Procedural Cohesion



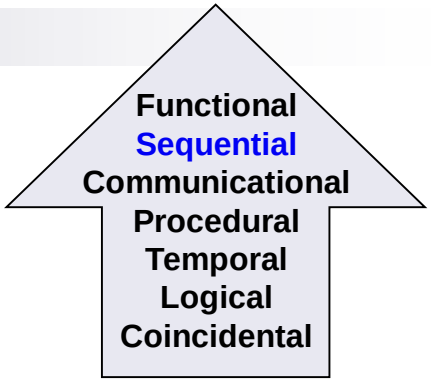
- Def: Elements of a component are related only to ensure a particular order of execution.
- Actions are still weakly connected and unlikely to be reusable.
- Example:
  - ...
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record
  - ...

Communicational Cohesion



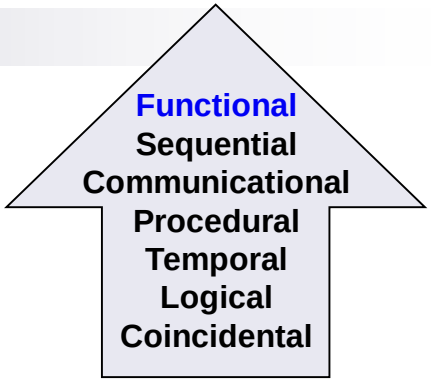
- Def: Functions performed on the same data or to produce the same data.
- Examples:
  - Update record in data base and send it to the printer
    - Update a record on a database
    - Print the record
  - Fetch unrelated data at the same time.
    - To minimize disk access

Sequential Cohesion



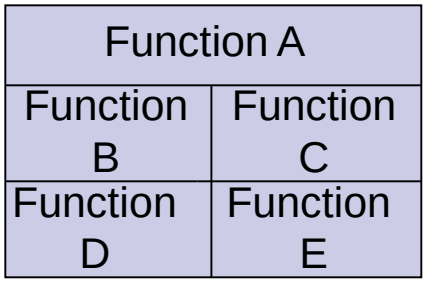
- Def: The output of one part is the input to another.
- *Data flows* between parts (different from procedural cohesion)
- Occurs naturally in functional programming languages
- Good situation

Functional Cohesion

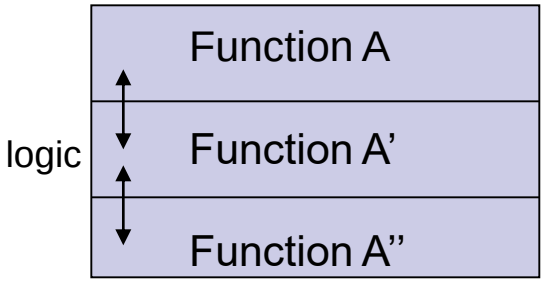


- Def: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed but
  - it performs only that function and nothing else.

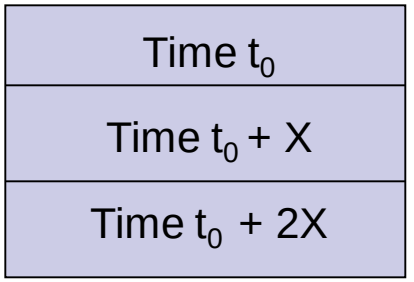
Examples of Cohesion



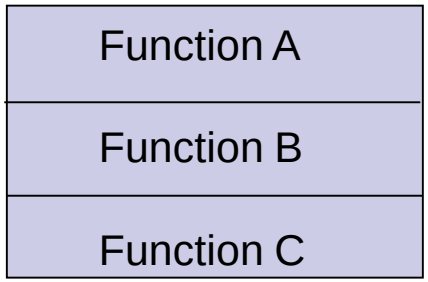
Coincidental
Parts unrelated



Logical
Similar functions

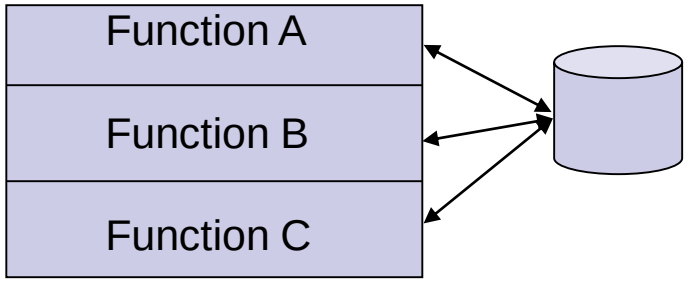


Temporal
Related by time

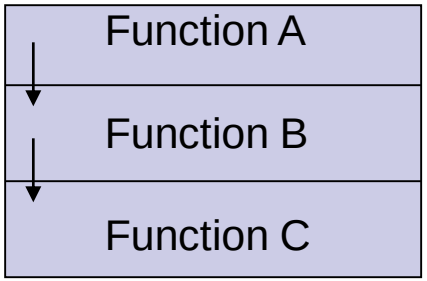


Procedural
Related by order of functions

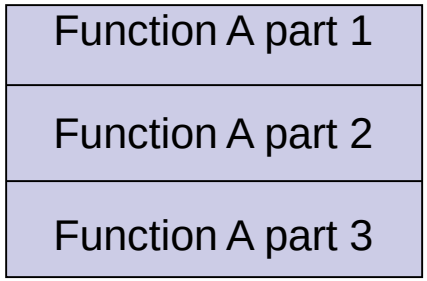
Examples of Cohesion (Cont.)



Communicational
Access same data



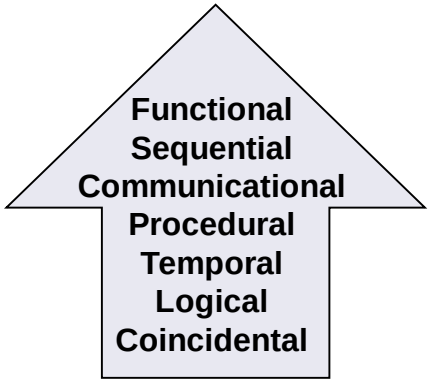
Sequential
Output of one is input to another



Functional
Sequential with complete, related functions

Exercise: Cohesion for Each Module?

- Compute average daily temperatures at various sites
- Initialize sums and open files
- Create new temperature record
- Store temperature record
- Close files and print average temperatures
- Read in site, time, and temperature
- Store record for specific site
- Edit site, time, or temperature field

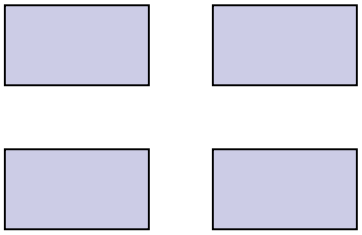


Outline

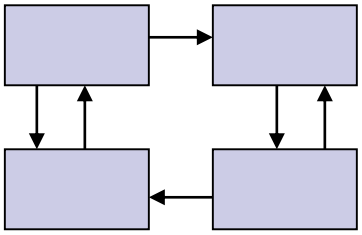
- ✓ Cohesion
- ■ Coupling

Coupling

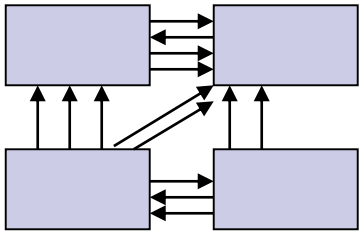
- The degree of dependence such as the amount of interactions among components



No dependencies



Loosely coupled
some dependencies



Highly coupled
many dependencies

Coupling

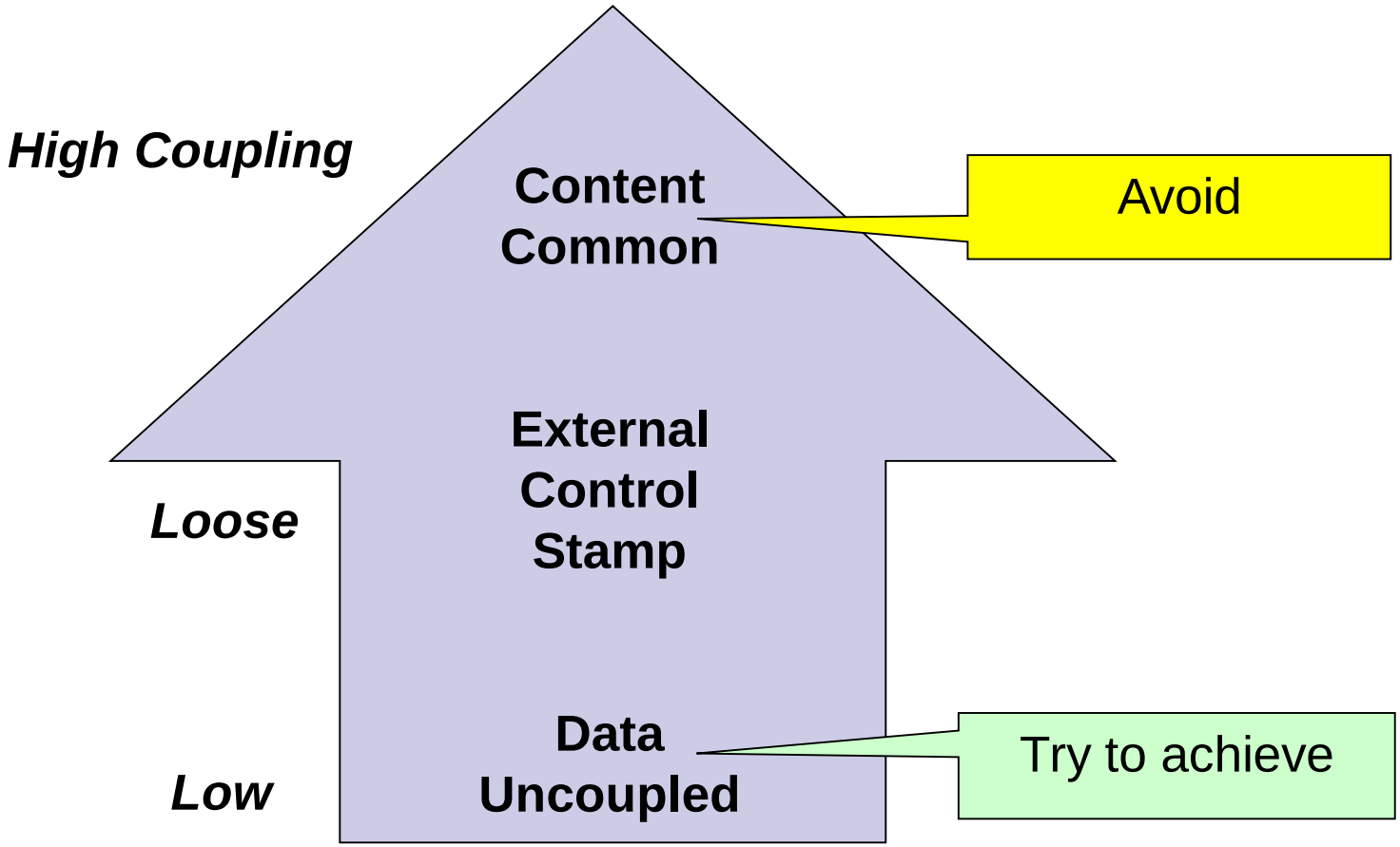
- The degree of dependence such as the amount of interactions among components
- How can you tell if two components are coupled?

- (In pairs, 2 minutes)

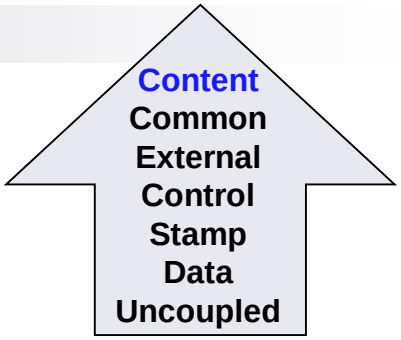
Indications of Coupling

- ?

Type of Coupling



Content Coupling



- Def: One component modifies another.
- Example:
  - Component directly modifies another's data
  - Component modifies another's code, e.g., jumps (goto) into the middle of a routine
- Question
  - Language features allowing this?

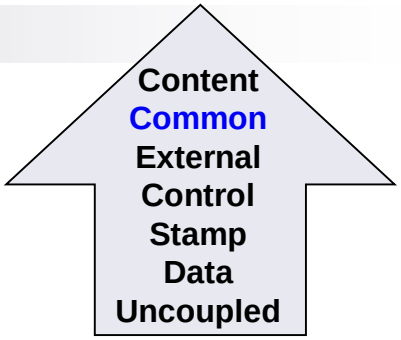
Example

Part of a program handles lookup for customer.

When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Improvement?

Common Coupling



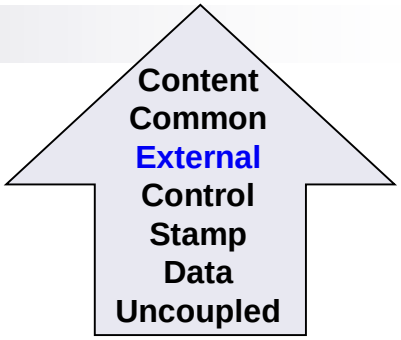
- Def: More than one component share data such as global data structures
- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses

Example

Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks. Each source process writes directly to global data store. Each sink process reads directly from global data store.

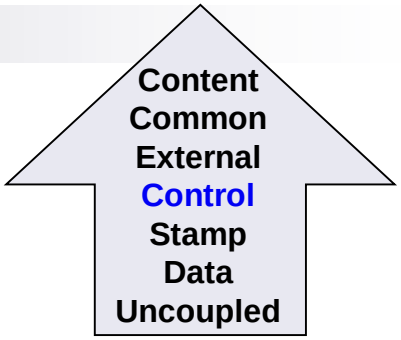
Improvement?

External Coupling



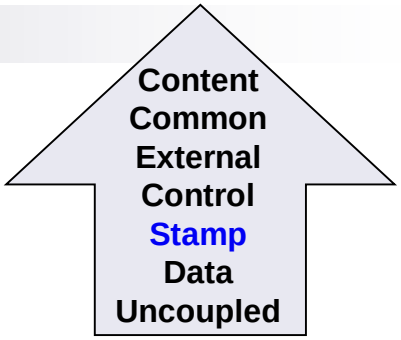
- Def: Two components share something externally imposed, e.g.,
  - External file
  - Device interface
  - Protocol
  - Data format
- Improvement?

Control Coupling



- Def: Component passes control parameters to coupled components.
- May be either good or bad, depending on situation.
  - Bad if parameters indicate completely different behavior
  - Good if parameters allow factoring and reuse of functionality
- Good example: sort that takes a comparison function as an argument.
  - The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.

Stamp Coupling



- Def: Component passes a data structure to another component that does not have access to the entire structure.
- Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation).
- The second has access to more information that it needs.
- May be necessary due to efficiency factors: this is a choice made by insightful designer, not lazy programmer.

Example

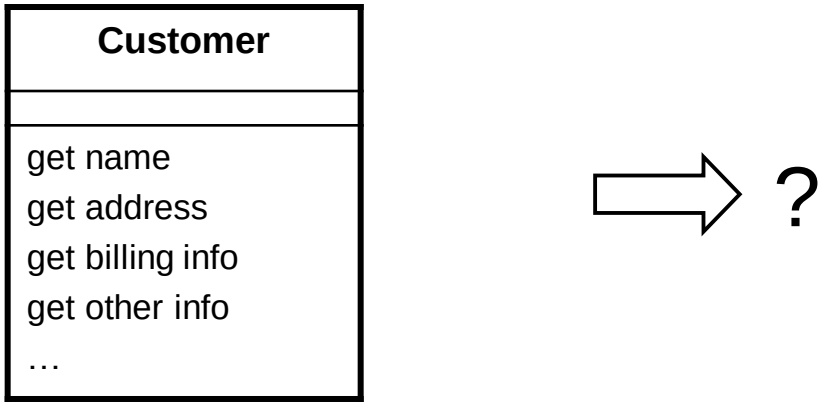
Customer Billing System

The print routine of the customer billing accepts customer data structure as an argument, parses it, and prints the name, address, and billing information.

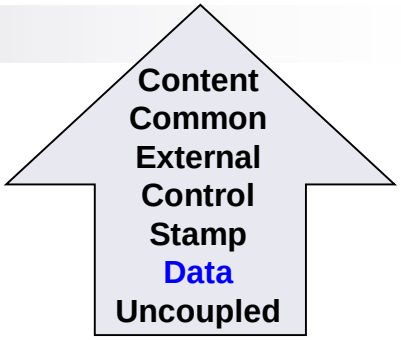
Improvement?

Improvement --- OO Solution

- Use an interface to limit access from clients



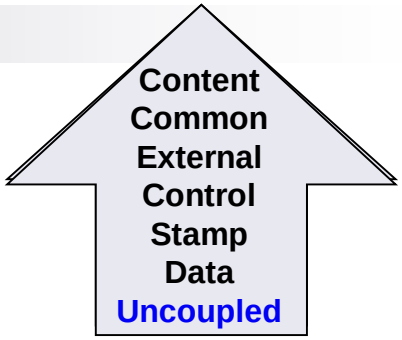
void print (Customer c) { … }



Data Coupling

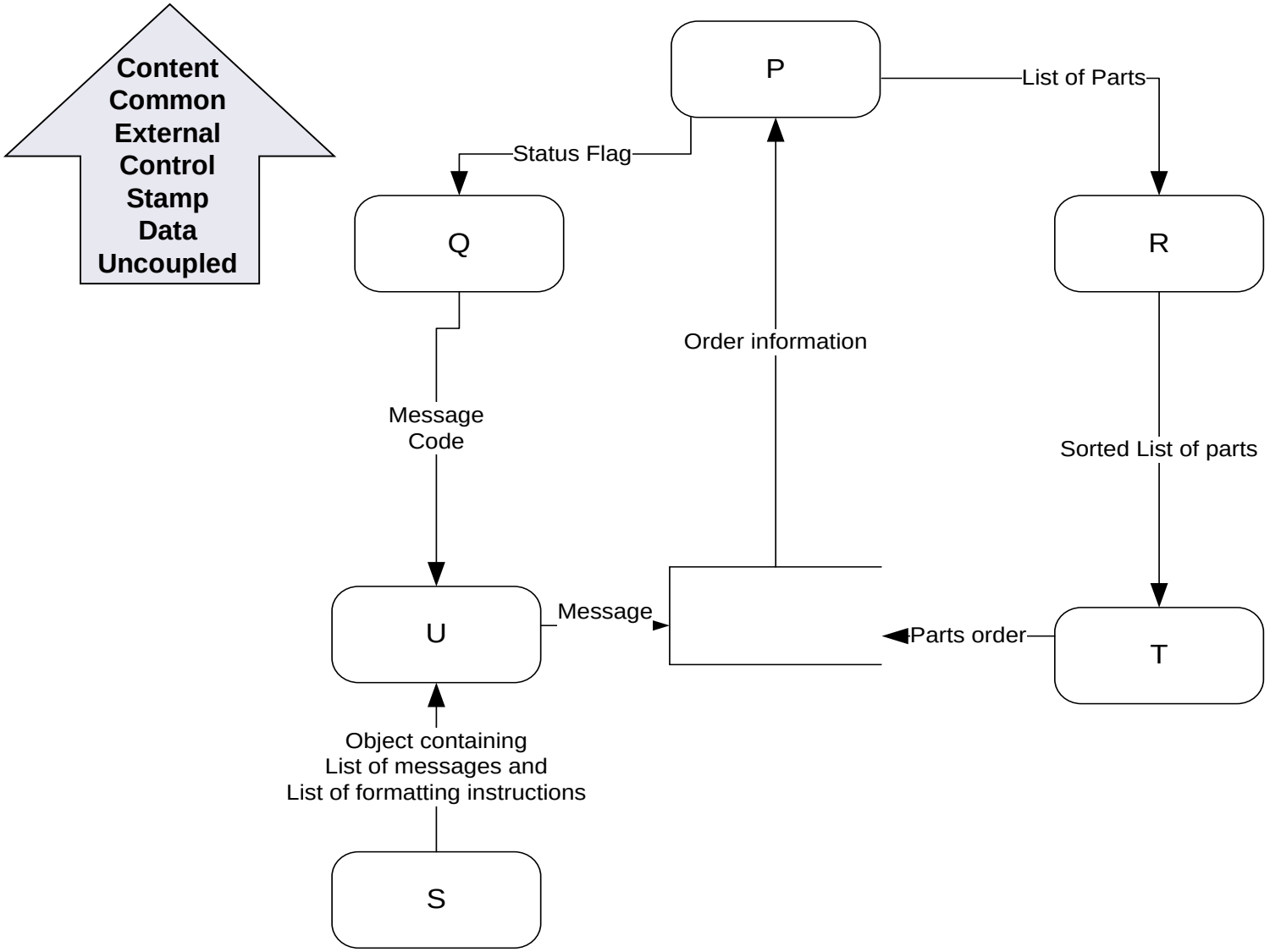
- Def: Component passes data (not data structures) to another component.
- Every argument is simple argument or data structure in which all elements are used
- Good, if it can be achieved.
- Example: Customer billing system
  - The print routine takes the customer name, address, and billing information as arguments.

Uncoupled



- Completely uncoupled components are not systems.
- Systems are made of interacting components.

Exercise: Define Coupling between Pairs of Modules



Coupling between Pairs of Modules

	Q	R	S	T	U
P					
Q					
R					
S					
T					

Consequences of Coupling

- Why does coupling matter? What are the costs and benefits of coupling?

- (pairs, 3 minutes)

Consequences of Coupling

- High coupling
  - Components are difficult to understand in isolation
  - Changes in component ripple to others
  - Components are difficult to reuse
    - Need to include all coupled components
    - Difficult to understand
- Low coupling
  - May incur performance cost
  - Generally faster to build systems with low coupling

In Class

Groups of 2 or 3:

- P1: What is the effect of cohesion on maintenance?
- P2: What is the effect of coupling on maintenance?

S.O.L.I.D Design Principles

CSCI 2300

S.O.L.I.D Decoded

- S – Single Responsibility Principle
- O – Open/Closed Principle
- L – Liskov Substitution Principle
- I – Interface Segregation Principle
- D – Dependency Inversion Principle

Single Responsibility Principle (SRP)

- Each class has one responsibility (and one reason to change)

SRP Violated (from Tic Tac Toe)

```
class Board{
   public void reset(){…}
   public boolean setPiece(…){…}
   public boolean hasWinner(…){…}
   public boolean hasOpenPosition(){…}
   public void display(){…}
}
```

Board is responsible for too much

How can we fix it:

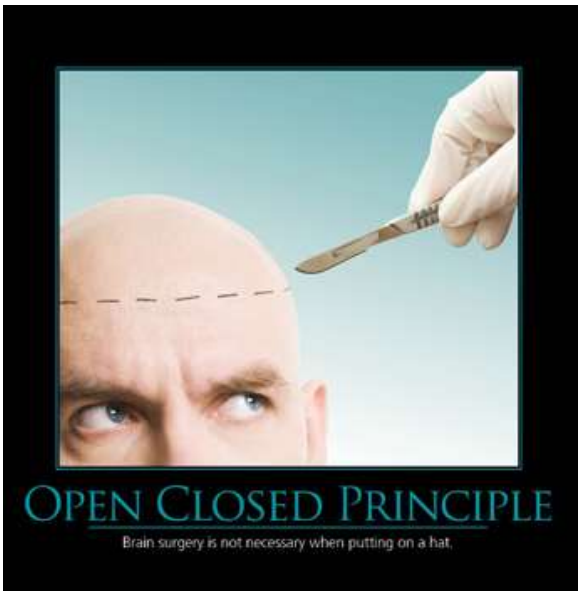
- Separate responsibilities into multiple classes:
  - Board
  - BoardView (or BoardDisplay)
- Use interfaces:
  - IBoard – interface for board
  - IBoardDisplay – interface for displaying a board
- Use composition:
  - Board – the model of the board
  - BoardDisplay has an instance of Board

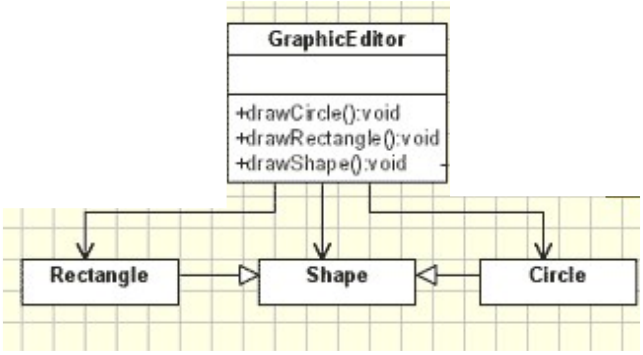
```
class Person {
    protected String firstName; //get and set methods
    protected String lastName;  //get and set methods
    protected Gender gender;    //get and set methods
    protected DateTime dateOfBirth;
    public string Format(string formatType) {
        switch(formatType) {
            case "XML":
               return xmlFormattedString; break;
            case "FirstAndLastName":
              return firstAndLastNameString; break;
            default:
              // implementation of default formatting
              return defaultFormattedString;
        }
    }
}
```

A. This class violates SRP because it encapsulates multiple attributes of a person
B. This class violates SRP because it does not have a constructor
C. This class violates SRP because it is responsible for encapsulating "person" attributes and formatting them
D. This class does not violate SRP

Open/Closed Principle (OCP)

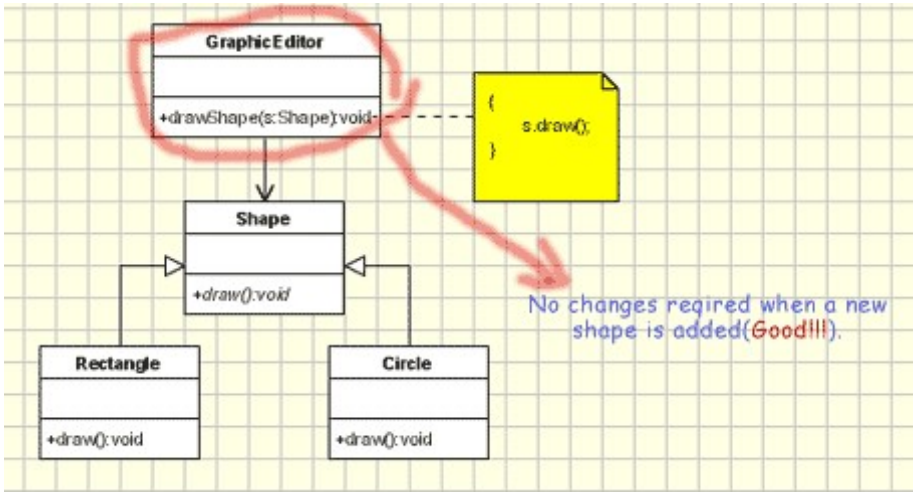
- Classes/methods/modules should be open for extension but closed for modification
- Create classes/methods/modules that whose behavior can change without recompiling the code
- If we can extend software to satisfy new requirements, without modifying existing code, the design satisfies OCP
- Simple example: pass parameters to methods instead of hard coding values.





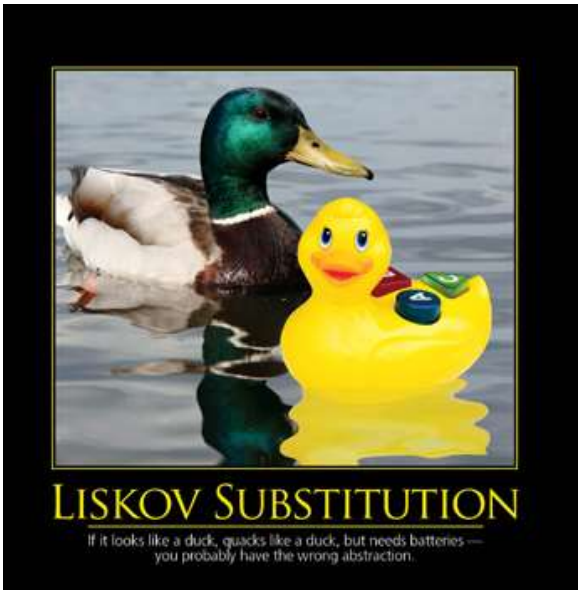
A. This design violates OCP because it has drawShape() method and shape is abstract
B. This design violates OCP because we'll need to change Graphic Editor if we add another shape.
C. This design does not violate OCP because if we add another shape, we don't have to change GraphicEditor: we can create a sub-class of GraphicEditor and add the necessary draw function there.
D. None of the above are true

Improved Design



Liskov Substitution Principle (LSP)

- Subtypes can be substituted for their base types
- Subclass IS-SUBSTITUTABLE-FOR base class



Violation of LSP

- Class “Rectangle”
- Class “Square” extends “Rectangle”
- “Square” enforces that height and width are equal
- Suppose the following code:

```
public double area(Rectangle r)
{
    r.setHeight(10);
    r.setWidth(5);
    return r.getArea();
}
```

```
public double calculate (Bill bill)
{
   if (bill instanceOf LargeGroupBill)
   {
      // add up the bill items and add 15% gratuity
   }
   else{
      // add up the bill items
   }
}
```

A. This code violates LSP because we are checking the sub-type of Bill to determine the logic
B. This code violates LSP because the calculation should be done in the Bill class
C. This code violates LSP because LargeGroupBill is not defined
D. This code does not violate LSP

```
public class Vehicle{
    public void drive(int miles){
        if (miles > 0 && miles < 300){…}
    }
}
 public class Scooter extends Vehicle{
     public void drive(int miles){
         if (miles > 0 && miles < 50){
            super.drive(miles);
         }
     }
 }
```

A. This code does not violate LSP.
B. This code violates LSP because it restricts the behavior of Vehicle.
C. This code violates LSP because the drive() method of Scooter calls parent's drive() method.

```
class ToyCar extends Vehicle{
    public void drive(int miles) {
      // Show flashy lights, make random sounds
    }
    public void fillUpWithFuel() {
       // silly lights and noises
    public int fuelRemaining { return 0;}
}
```

A. This code does not violate LSP
B. This code violates LSP because it completely changes the behavior of drive() and fillUpWithFuel()
C. This code violates LSP because fuelRemaining() returns 0
D. B and C.

Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use
- Clients should not implement methods if those methods are unused.

ISP Violation

```
public interface IMembership
{
  boolean Login(string username, string password);
  void Logout(string username);
  Guid Register(string username, string
                password, string email);
  void ForgotPassword(string username);
}
```

Improved Design

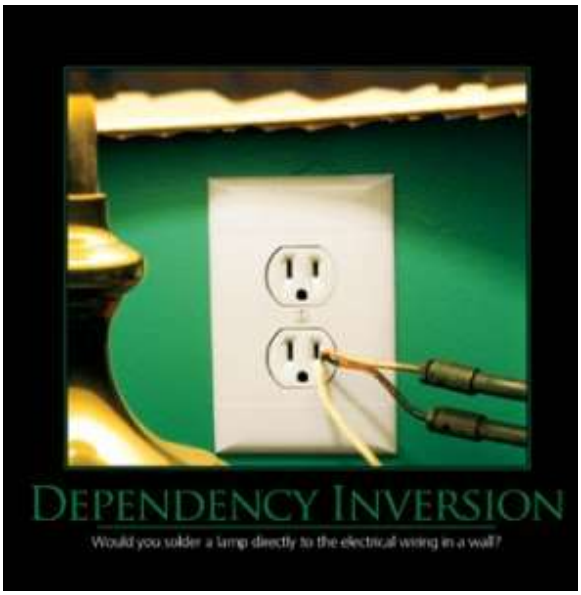
```
public interface ILogin
{
  boolean Login(String username, String password);
  void Logout(String username);
}

public interface IMembership extends ILogin
{
  Guid Register(String username, String password,
                String email);
  void ForgotPassword(string username);
}
```

```
public interface Movable{
   public void move();
   public void setSpeed(int speed);
}
public class Picture implements Movable{
   public void move(){// code for moving a
picture across the screen}
   public void setSpeed(int speed){return;}
}
```

A. This code violates ISP because Picture class has a "dummy" implementation of setSpeed()
B. This code violates ISP because Movable interface has more than one method
C. This code violates ISP because we should be able to control the speed at which Picture moves across the screen
D. This code does not violate ISP

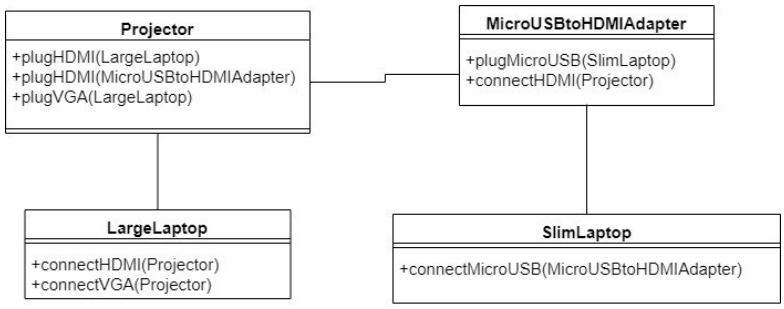
Dependency Inversion Principle (DIP)

- High level modules should not depend on low level modules
- Both should depend on abstraction
- Abstractions should not depend on details
- Details should depend upon abstraction



Dependency Inversion Principle Example

- There is a tight coupling between classes
- Dependency on low level modules
- If we add another device, we'll need to adjust existing code and add another dependency to Projector



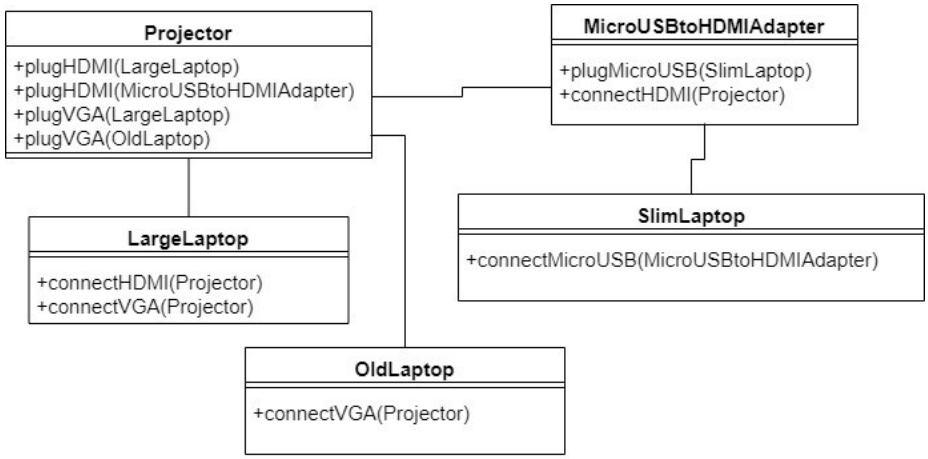
Suppose we have the following objects:

```
Projector projector
SlimLaptop slimLaptop
LargeLaptop largeLaptop
MicroUSBtoHDMIAdapter adapter
```

What operation(s) is/are possible?

A. projector.plug(slimLaptop)
B. projector.plug(largeLaptop)
C. projector.plug(adapter)
D. B and C
E. All of the above

Added a new device: Old Laptop



Consider the redesigned interfaces from the handout. What changes are needed to get the following code to work (assuming `projector` and `oldLaptop` have been instantiated).

```
projector.plug(oldLaptop)
```

A. OldLaptop implements HDMIPort
B. OldLaptop implements VGAPort
C. OldLaptop implements HDMIPlug
D. OldLaptop implements VGAPlug
E. Projector implements OldLaptop

Design Defects & Restructuring

Week 3: 16 Sep 22

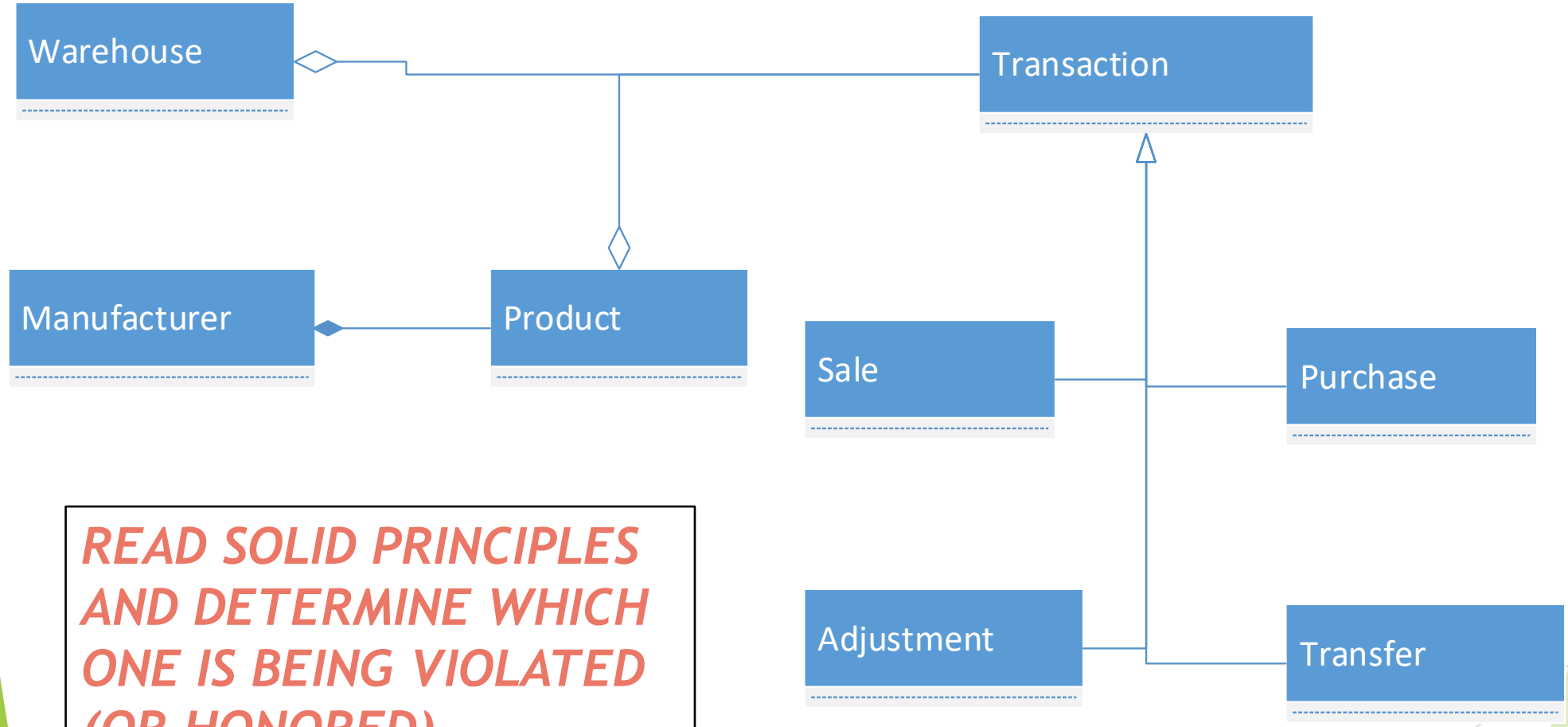
Rahim Hasnani

Agenda

- Home works
- Any questions on last week
- Last week's question
- SOLID Principles
- First Chapter of Head First Design Patterns
- GOF Design Patterns - Introduction

Exercise

- *GoldSoft has recently won a project for development of an inventory system. The inventory system is to be developed for a big distributor with over 1,000 products. The distributor has warehouses where products are stored. The distributor currently carries products of five principals (manufacturers) and this is likely to increase with time. [The main business of a distributor is to order and carry products from principals and distribute them to wholesalers]. The major transactions in the inventory system are 1) Receipt of products from principal 2) Distribution of products to wholesaler 3) Transfer of stock from one warehouse to another 4) Stock adjustment as a result of stock count. Each of these transactions involves a number of products in different quantities (i.e., a receipt usually involves tens or even hundreds of different products in different quantities). Products have different unit of measures (some products are distributed as individual units like keyboard, some are distributed in meters like network wire, some are distributed in Kgs like Sugar, etc).*

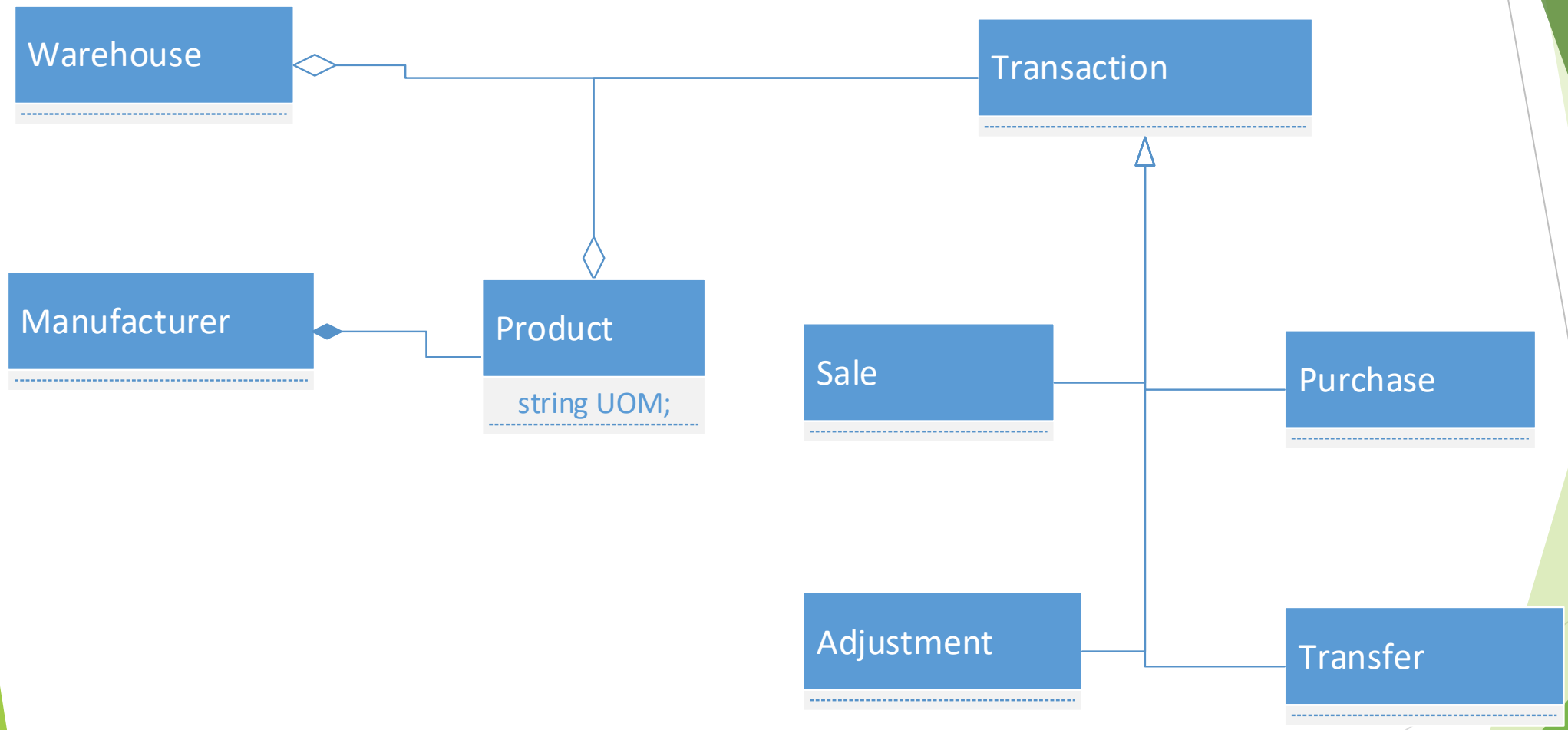


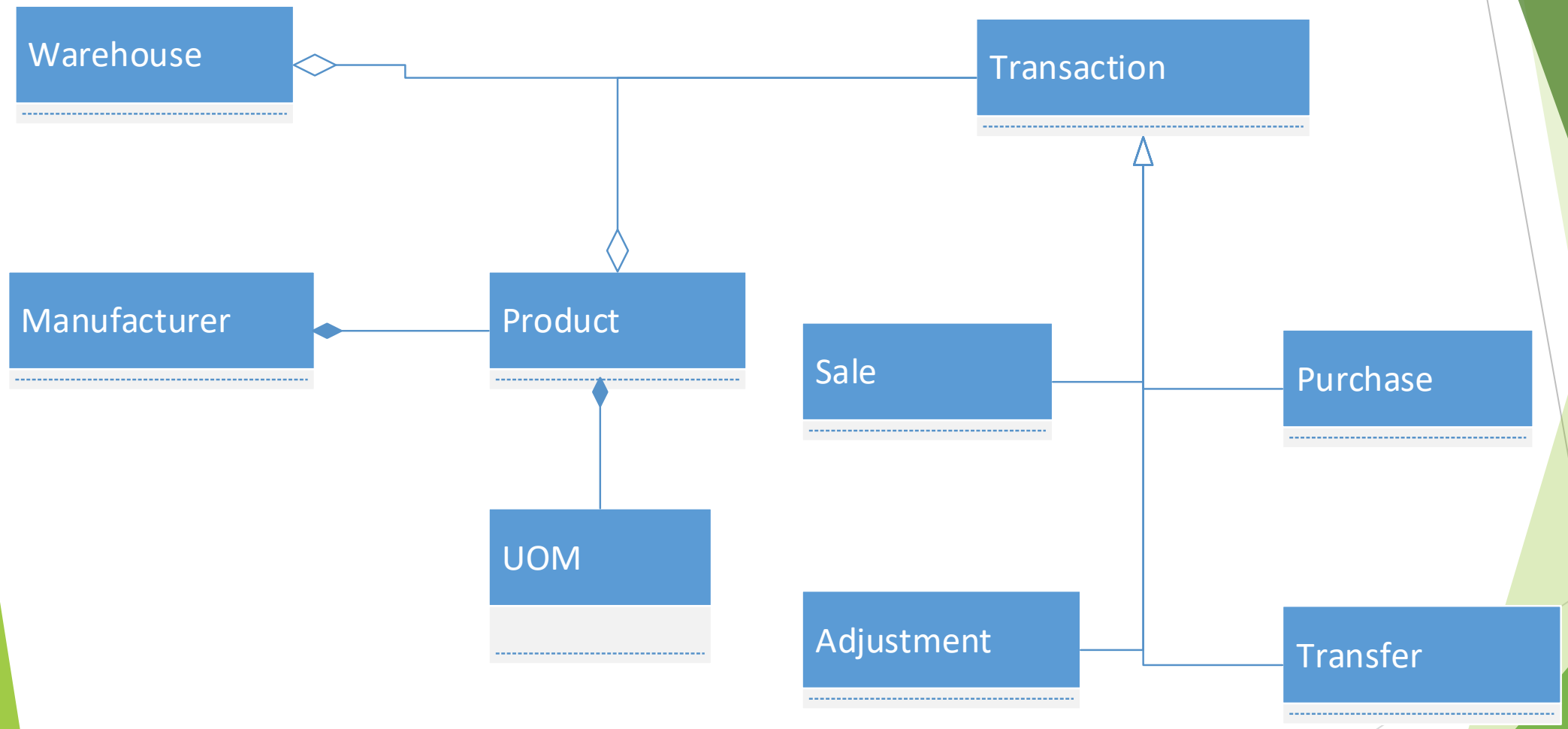
READ SOLID PRINCIPLES AND DETERMINE WHICH ONE IS BEING VIOLATED (OR HONORED)

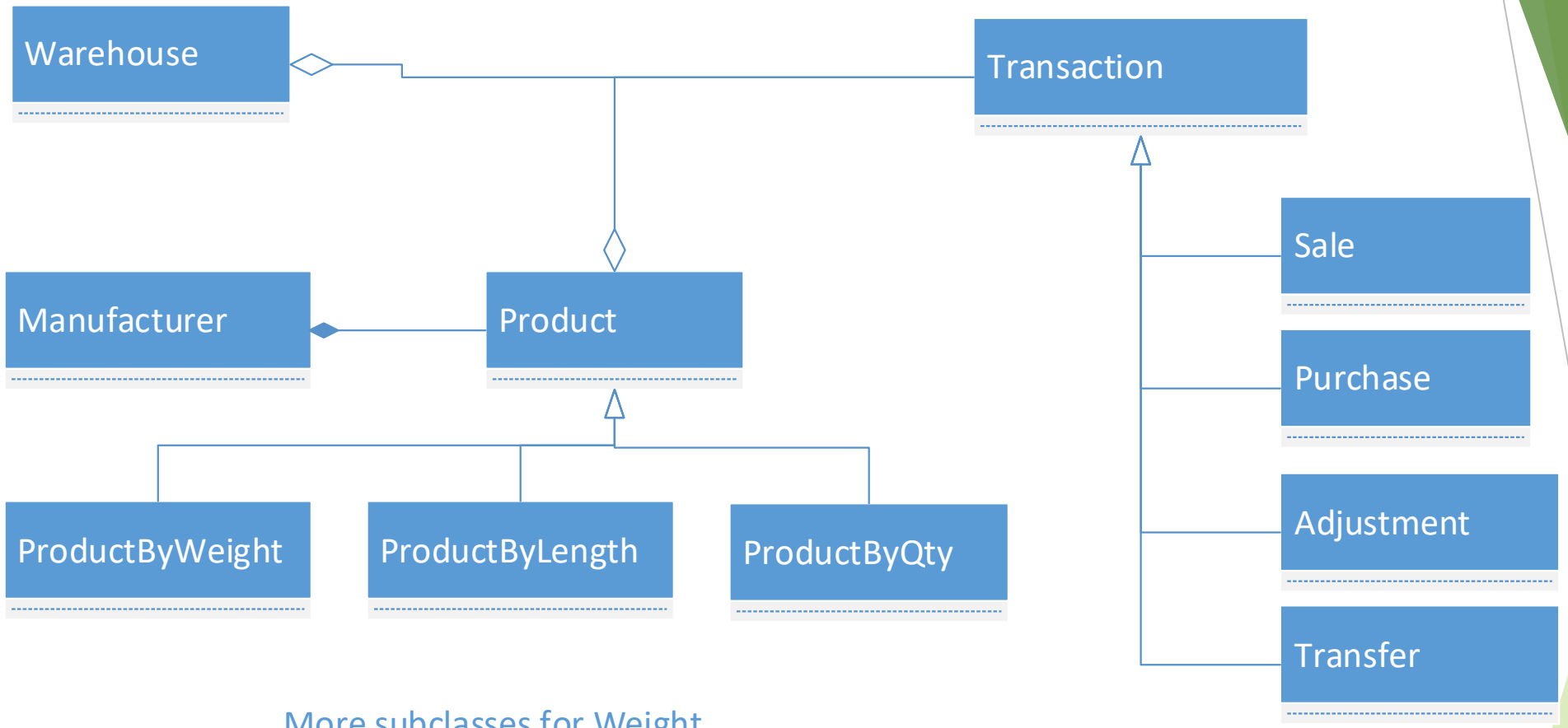
Where is UOM??

Alternatives:

1. Do nothing; UOM is just an attribute
2. Make it a composition: Product has UOM
3. Make it a hierarchy


Warehouse
Transaction
Manufacturer
Product
string UOM;
Sale
Purchase
Adjustment
Transfer


Warehouse
Transaction
Manufacturer
Product
Sale
Purchase
UOM
Adjustment
Transfer



More subclasses for Weight (Kg, Pound, Stones, Tonnes), Length (Meter, Feet, Yard, Inches) and Qty (Singles, 10s, Dozens, Gross)

Where is the Stock

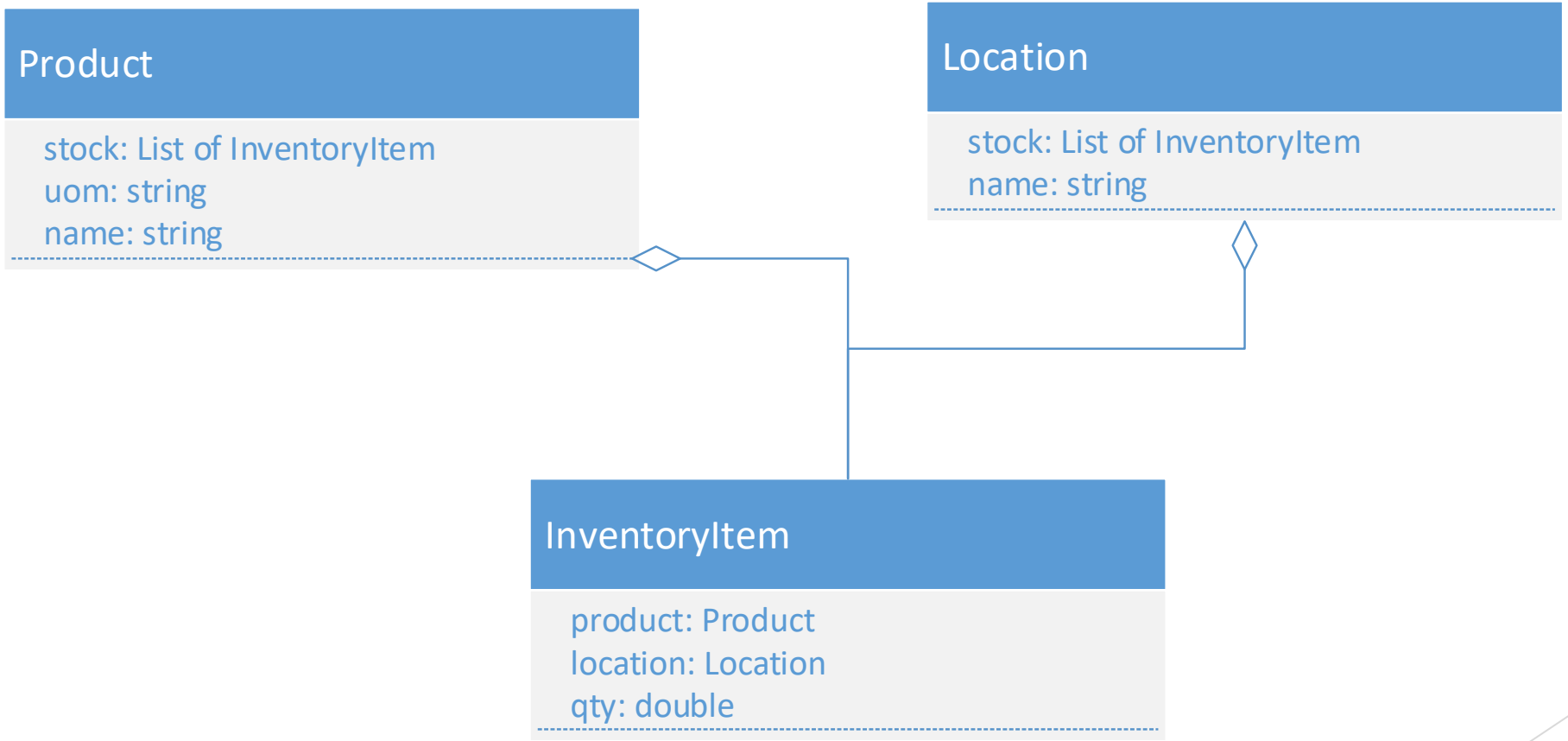
Its an inventory system!

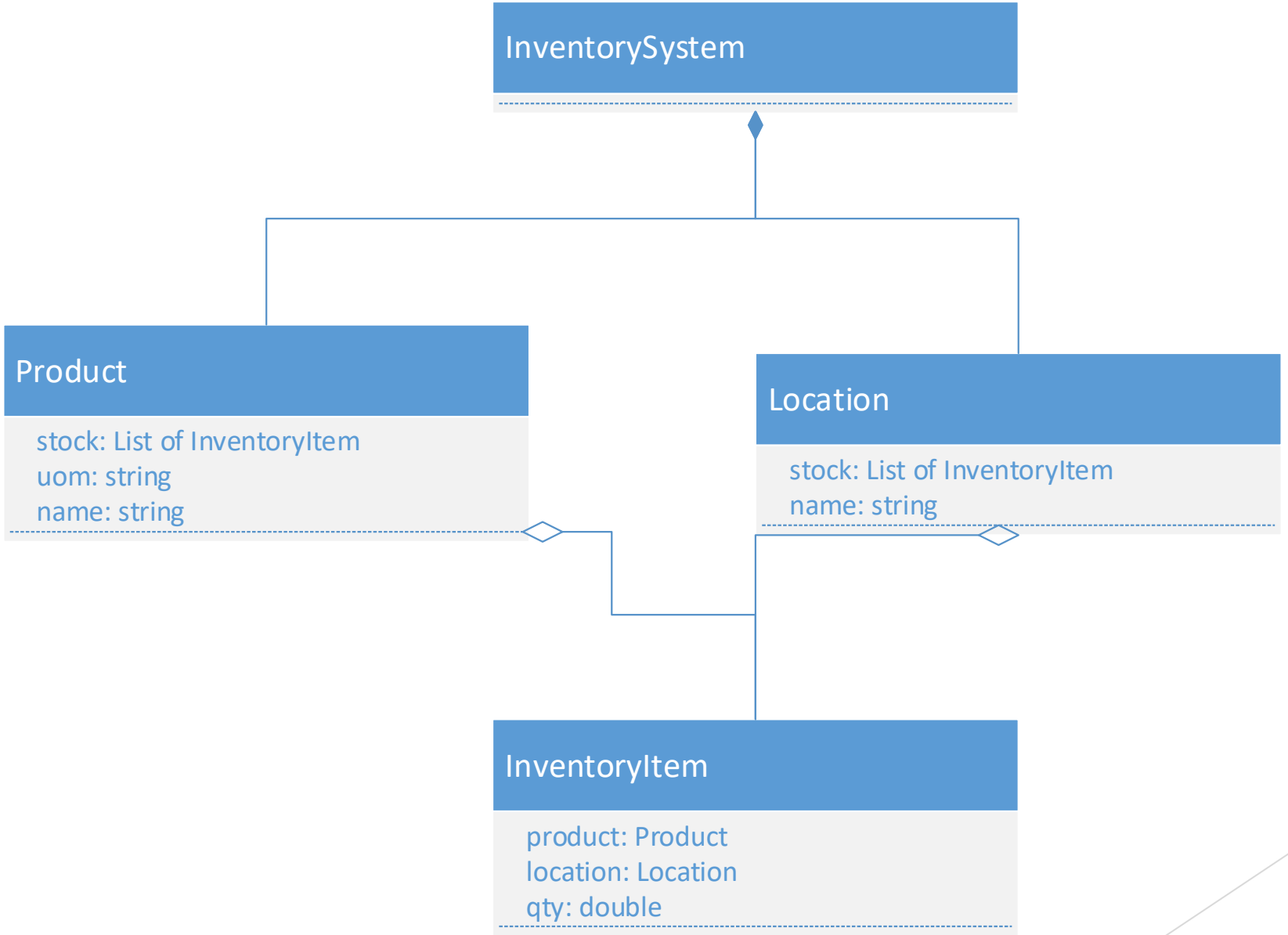
Whole purpose to keep ***track*** of stock, which is:

For a particular product, where it is and in what quantity

For a particular location, what is currently held

So, where is the stock information??


Product
stock: List of InventoryItem
uom: string
name: string
Location
stock: List of InventoryItem
name: string
InventoryItem
product: Product
location: Location
qty: double


InventorySystem
Product
stock: List of InventoryItem
uom: string
name: string
Location
stock: List of InventoryItem
name: string
InventoryItem
product: Product
location: Location
qty: double

What’s wrong with last two designs

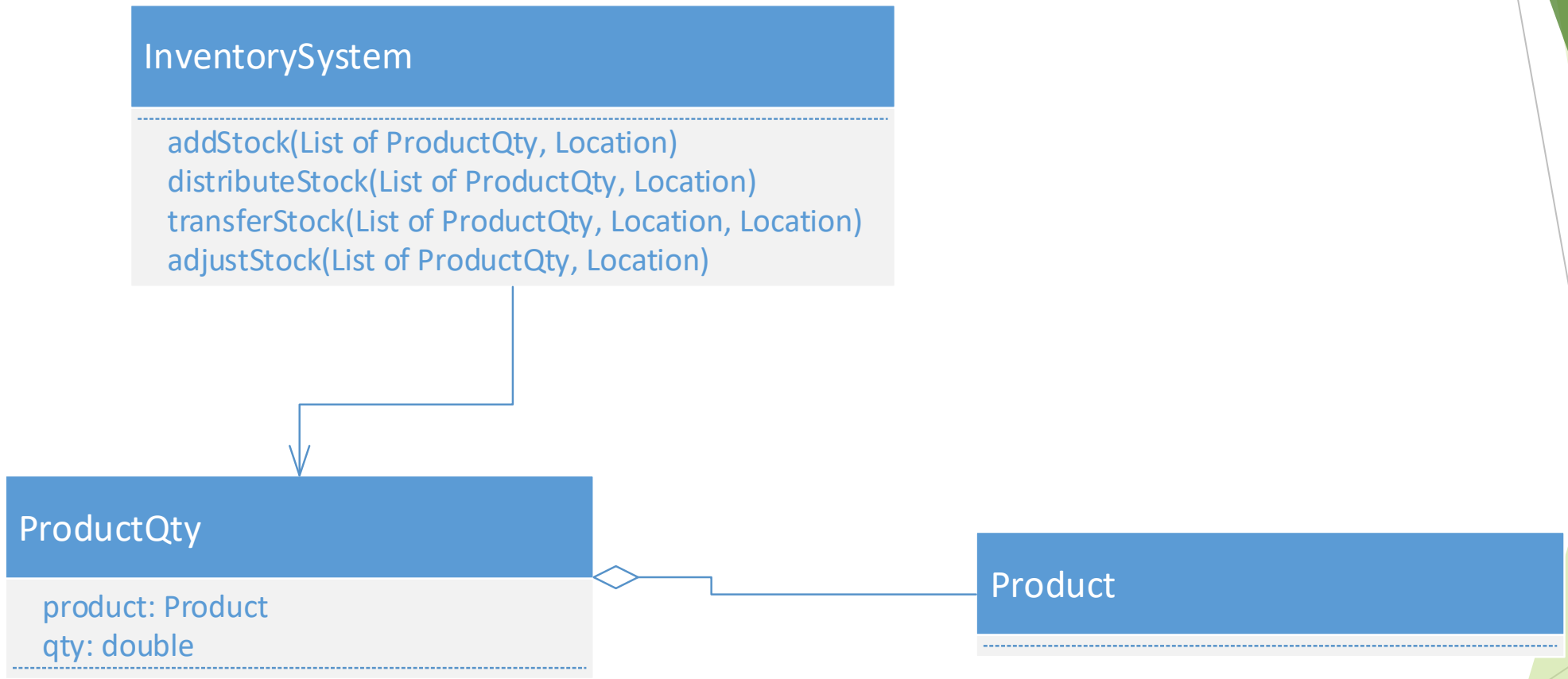
1. Nothing wrong
2. Why are we trying to replicate a database schema in our class diagram?
3. Don’t know; We certainly do not want our class design to look like a database but we still need to carry data, no?

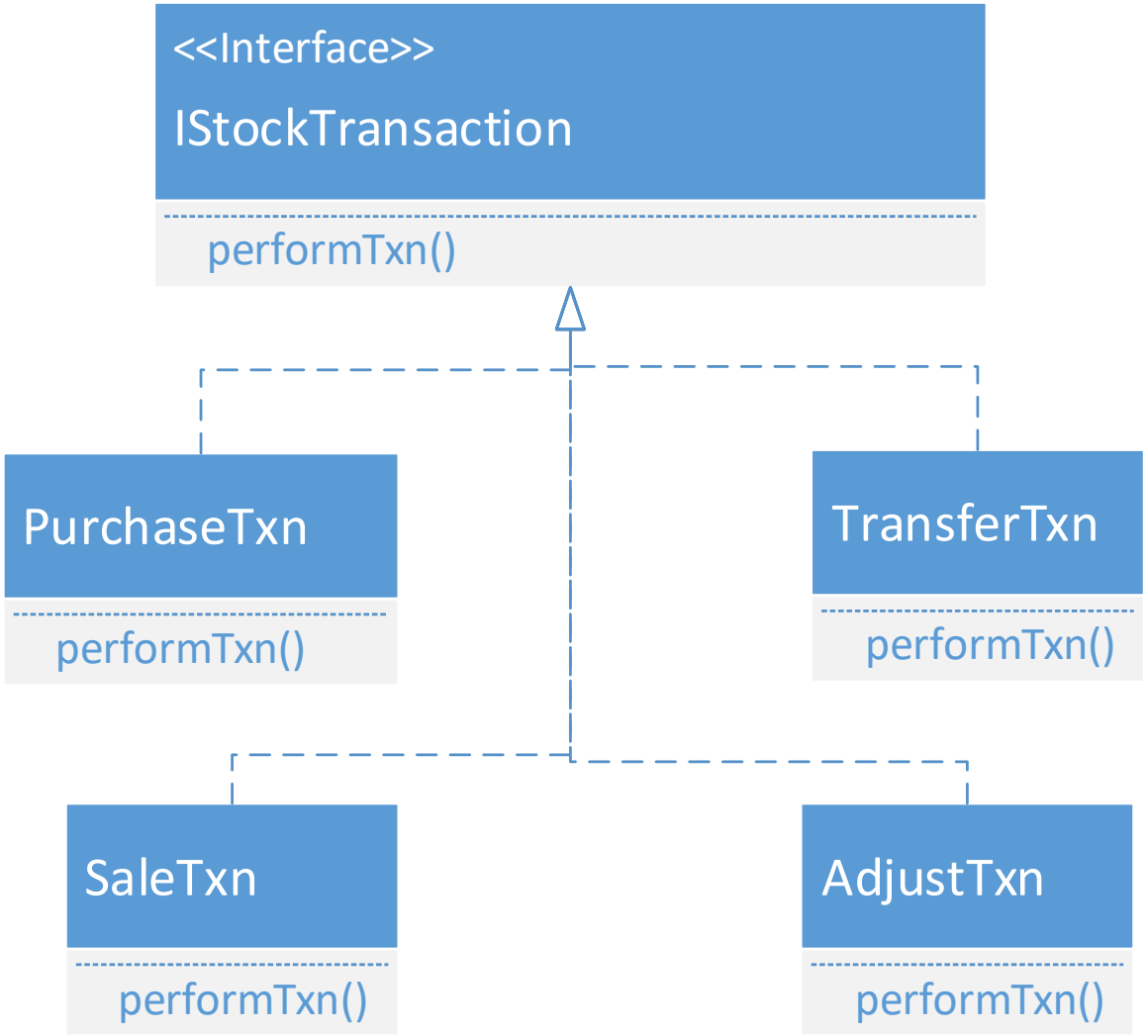
Let’s discuss Transactions

1. Four transactions are described (for now):
    1. Purchase: increases quantities of one or more products in a particular warehouse/location; Comes from ONE manufacturer
    2. Sale: decreases quantities of one or more products in a particular warehouse/location; products could be from multiple manufacturers
    3. Transfer: moves (increases at to end and decreases from from end) quantities of one of more products from one warehouse/location to another warehouse/location; products could be from One or more manufacturer
    4. Adjustment: increases or decreases quantities of one or more products in a single warehouse/location; products could be from multiple manufacturers
2. Is this something that should be abstracted?
3. Can we add more transactions later?
4. Can a non-transfer transaction involves more than one location?

Alternatives:

- Keep transactions as behaviors
- Each Transaction ***is a*** transaction
- Interfaces??


InventorySystem
addStock(List of ProductQty, Location)
distributeStock(List of ProductQty, Location)
transferStock(List of ProductQty, Location, Location)
adjustStock(List of ProductQty, Location)
ProductQty
product: Product
qty: double
Product



WOULD THIS WORK??

GOF Patterns

- Walk-through of First chapter of head first design patterns book

Design Defects & Restructuring

Week 4: 24 Sep 22

Rahim Hasnani

Agenda

- Any questions from last week!
- Midterm
- First Chapter of Head First Design Patterns
- GOF Design Patterns – Introduction
- Architecture Patterns
- MVC
- SoC, Repository and Dependency Injection

New Homework

- Subclass vs Subtype
- Their relationship with public vs private inheritance
- How subtyping is implemented/enforced/achieved in C++, Java, C#? What about Python?

GOF Patterns

- Walk-through of First chapter of head first design patterns book

GOF Design Patterns - Introduction

- Basic Idea
  - Flexibility
  - Re-use
  - Extensibility
- Pattern Categorization by Purpose
  - Creational: object creation
  - Behavioral: Ways in which classes or objects interact and distribute responsibility
  - Structural: Composition of classes or objects
- Pattern Categorization by Scope
  - Class
  - Object

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Pattern Structure

- Name
  - Increases your design vocabulary
  - Design at a higher level of abstraction
- Problem
  - When to apply the pattern
  - (sometimes,) list of conditions that must be met
- Solution
  - Elements that make up the design
  - Template, not concrete
- Consequences
  - Results & trade-offs of applying the pattern

Rules of Object Oriented Design

- Program to an interface, not an implementation
  - Don’t declare variables to be instances of particular concrete class
  - Abstract the process of object creation
- Favor object composition over class inheritance
  - Reusing by inheritance is white-box re-use
  - Reusing by composition is black-box re-use
- Others
  - Inheritance vs parameterized types (templates/generics)

Architectural Patterns

- Database driven
- N-tier
- Client Server
- Master Slave
- MVC
- Micro-Services??

MVC Pattern

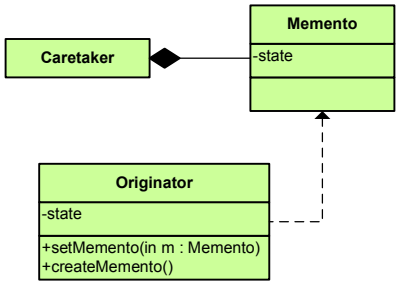
- Introduction
- SoC
- What goes into controller, model and views

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

Memento

Type: Behavioral

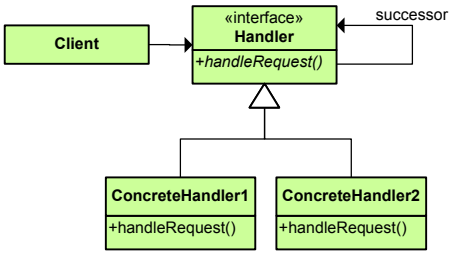
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

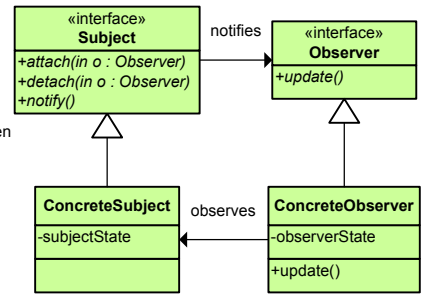
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

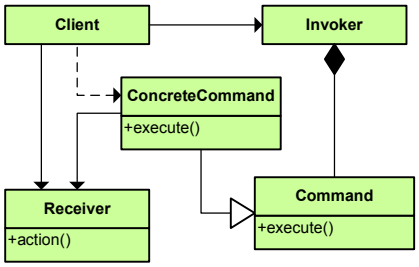
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

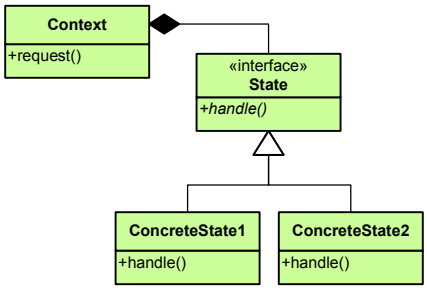
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

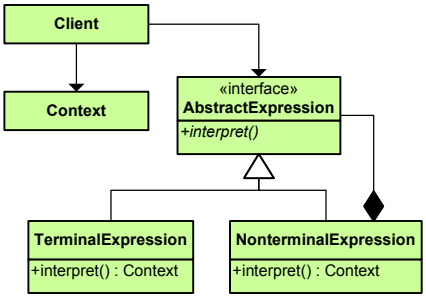
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

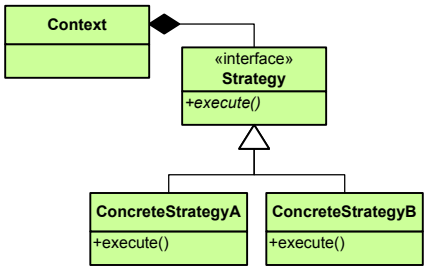
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

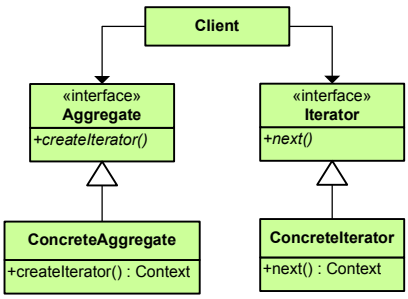
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

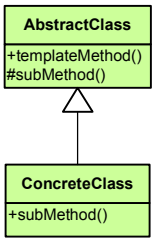
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

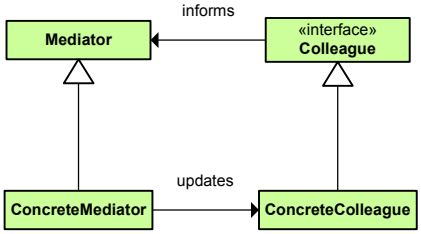
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

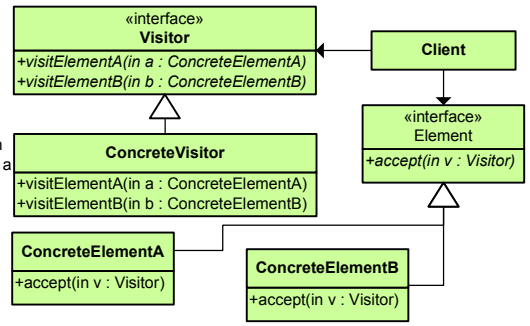
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



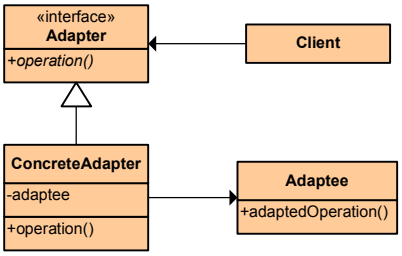
Visitor

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.



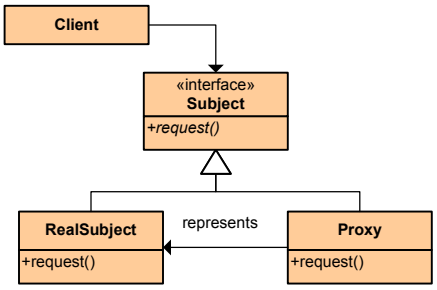
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..



Adapter

Type: Structural

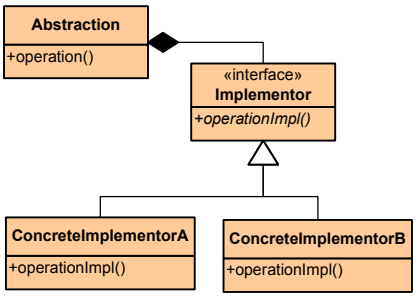
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Proxy

Type: Structural

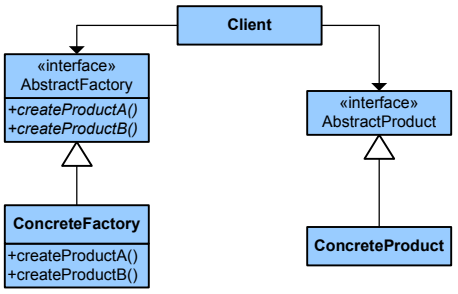
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Bridge

Type: Structural

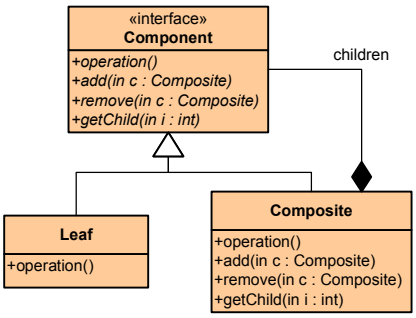
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Abstract Factory

Type: Creational

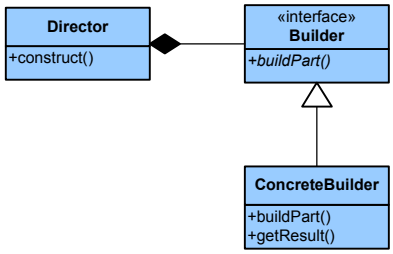
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Composite

Type: Structural

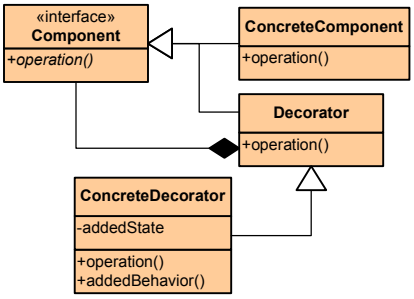
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Builder

Type: Creational

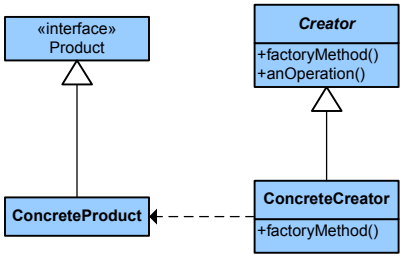
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Decorator

Type: Structural

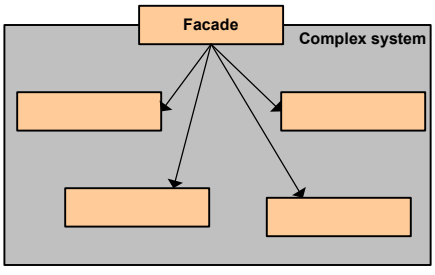
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Factory Method

Type: Creational

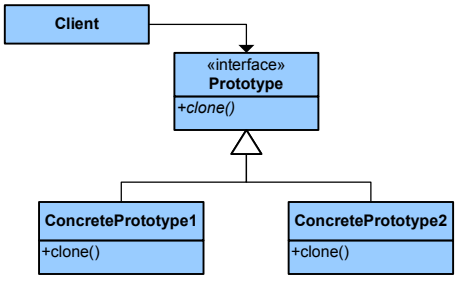
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Facade

Type: Structural

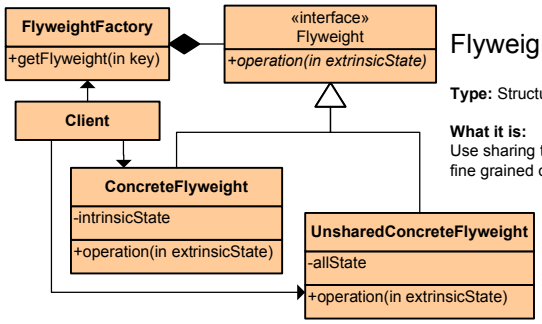
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Prototype

Type: Creational

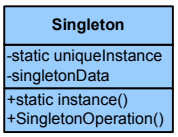
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Flyweight

Type: Structural

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



Singleton

Type: Creational

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

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

Design Defects & Restructuring

Week 5: 01 Oct 22

Rahim Hasnani

Agenda

- Dependency Injection
- Factory Method
- Introduction to Refactoring

MVC Continued – Dependency Injection

- Dependency Injection is a design pattern that implements IoC (Inversion of Control)
- Separates out the concerns of creating vs using the object, leading to loose coupling.
- Class A uses Class B => Class A depends on services of Class B
- Implementation alternatives:
  - Class A composes Class B and creates instance of class B
    - That's tightly coupled; Class A knows a LOT about class B
  - Create an interface/abstract-class to encapsulate class B services; Let class A create the concrete class and then user the interface
    - Somewhat better; still Class A is creating the concrete class

Dependency Injection Continued

Taken from https://munirhassan.com/2021/03/28/dependency-inversion-principle-and-the-dependency-injection-pattern/

```
public class Email
{
    public void SendEmail()
    {
        // code
    }
}
public class Notification
{
    private Email _email;
    public Notification()
    {
        _email = new Email();
    }
    public void PromotionalNotification()
    {
        _email.SendEmail();
    }
}
```

Dependency Injection Continued

```
public interface IMessageService
{
    void SendMessage();
}
public class Email : IMessageService
{
    public void SendMessage()
    {
        // code
    }
}
```

```
public class Notification
{
    private IMessageService
_iMessageService;

    public Notification()
    {
        _iMessageService = new Email();
    }
    public void PromotionalNotification()
    {
        _iMessageService.SendMessage();
    }
}
```

IDEAS?

Constructor Injection

```
public class Notification
{
    private IMessageService _iMessageService;

    public Notification(IMessageService _messageService)
    {
        this._iMessageService = _messageService;
    }
    public void PromotionalNotification()
    {
        _iMessageService.SendMessage();
    }
}
```

Property Injection

```
public class Notification
{
    public IMessageService MessageService
    {
        get;
        set;
    }
    public void PromotionalNotification()
    {

        if (MessageService == null)
        {
            // some error message
        }
        else
        {
            MessageService.SendMessage();

        }
    }
}
```

Method Injection

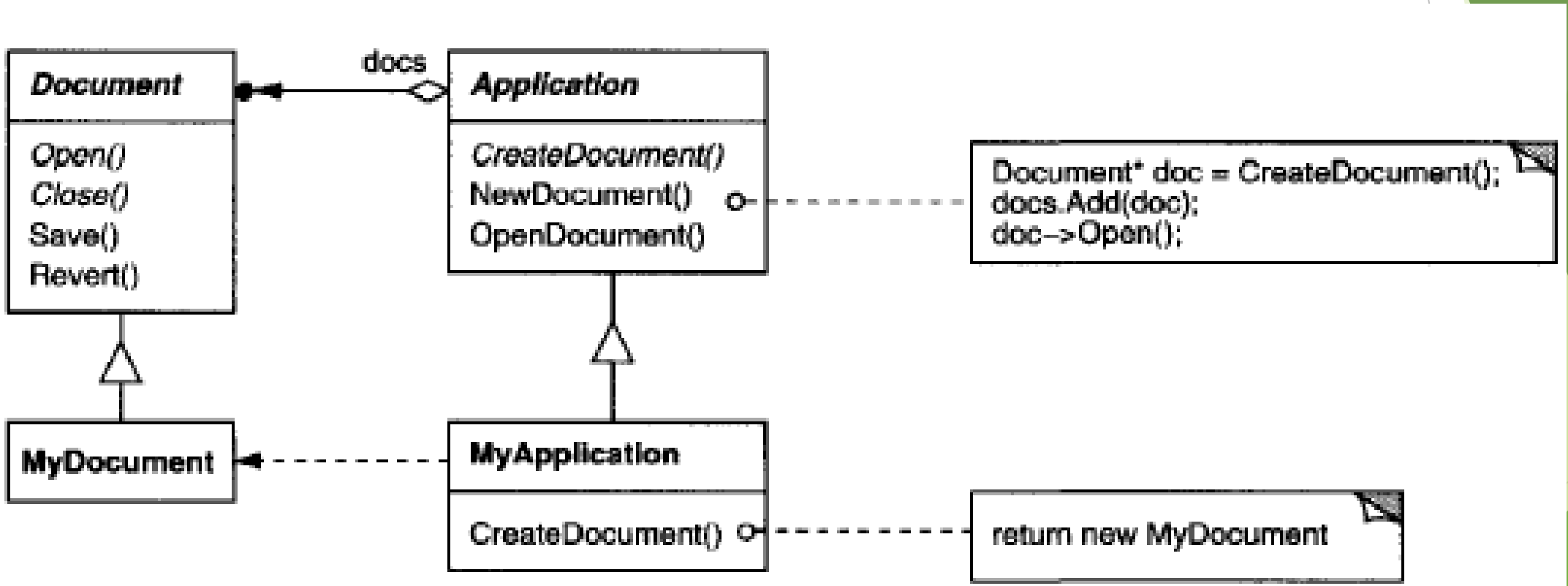
```
public class Email : IMessageService
{
    public void SendMessage()
    {
        // code for the mail send
    }
}
```

```
public class SMS : IMessageService
{
    public void SendMessage()
    {
        // code for the sms send
    }
}
```

```
public class Notification
{
    public void PromotionalNotification(IMessageService _messageService)
    {
        _messageService.SendMessage();
    }
}
```

Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class toinstantiate. Factory Method lets a class defer instantiation to subclasses.
- Also Known As
  - Virtual Constructor
- Motivation
  - Consider a design framework where abstract classes Class A and Class D have a aggregation relation whereby Class A creates instances of Class D.
  - Since both classes A and D are abstract, they will be subclassed to provide implementation.
  - Since the knowledge about which subclass of D to instantiate is specific to subclass of A, at an abstract level A cannot predict which subclass of D to create.
  - The Factory method offers a solution here


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

Design Defects & Restructuring

Week 6: 08 Oct 22

Rahim Hasnani

Observer Pattern

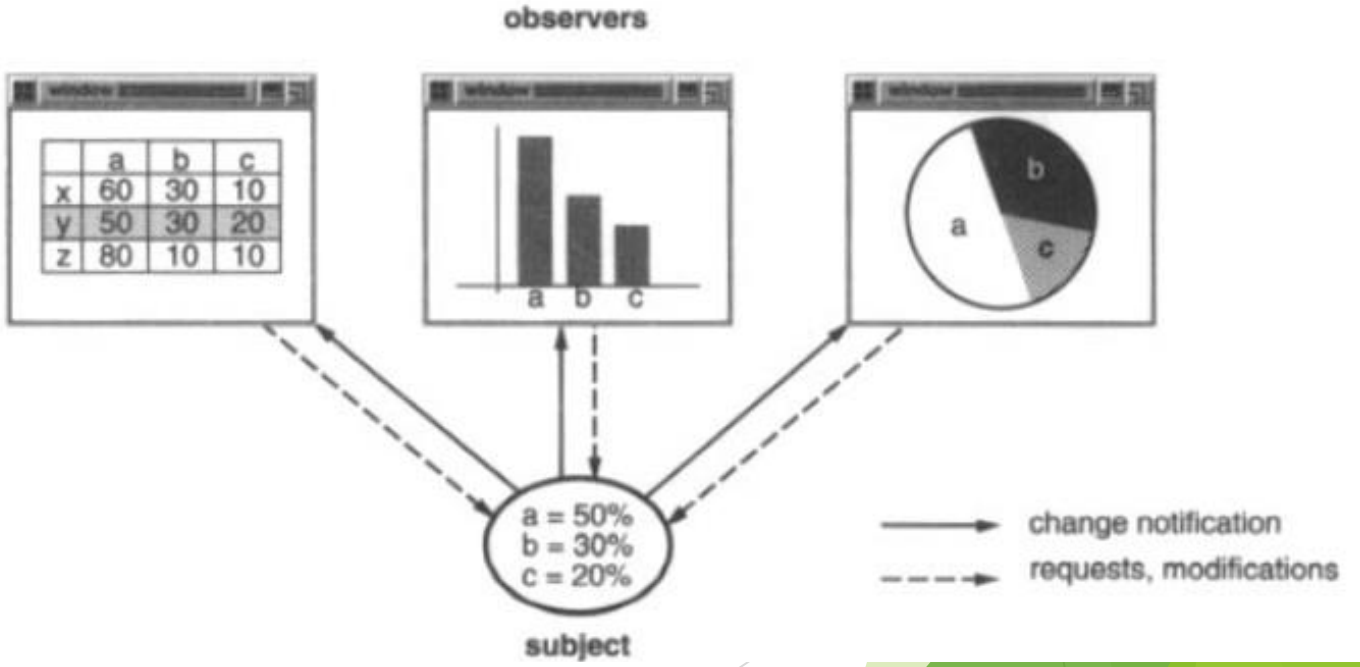
- Walk through of Chapter 2 of Head First Design Patterns

Observer Pattern from GOF

- Intent
  - Define a one-to-many dependency between objects so that when one objectchanges state, all its dependents are notified and updated automatically.
- Also Known As
  - Dependents, Publish-Subscribe
- Motivation
  - A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
  - For example in a GUI based system, classes defining application data and presentations ca n be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately and vice versa.

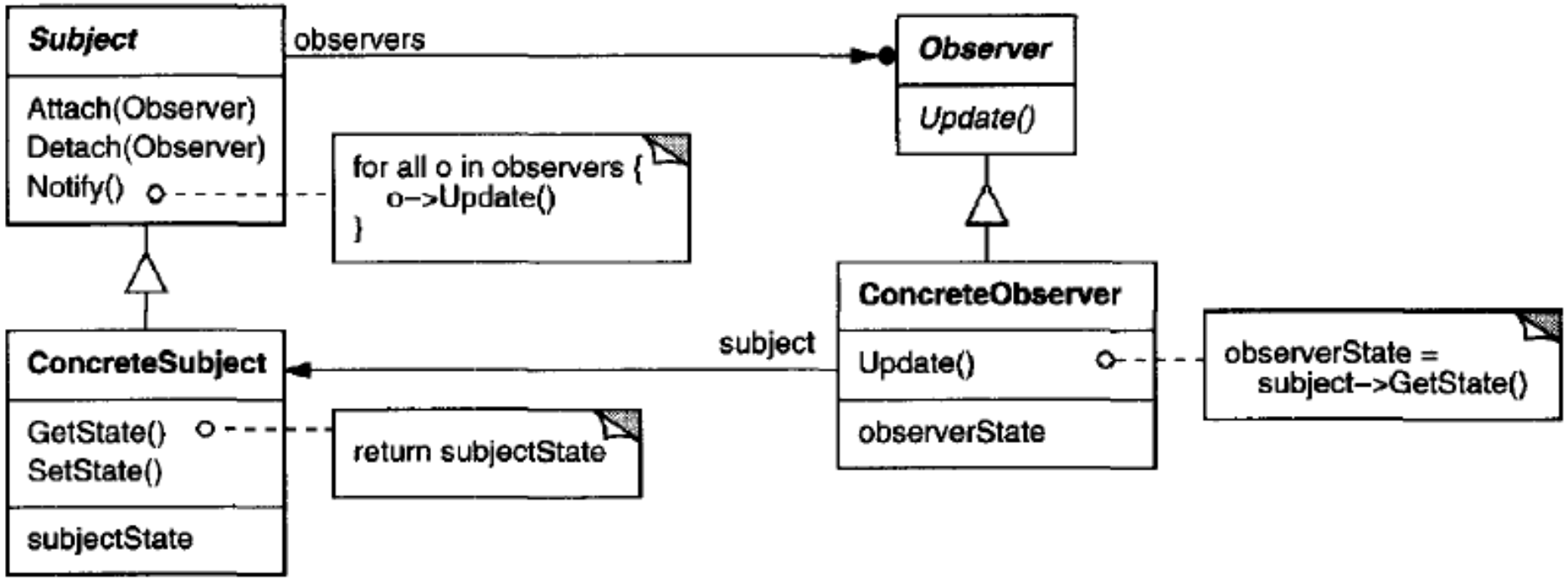
Observer Pattern from GOF

- The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observe rs. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as publish-subscribe. The subject is the publisher o f notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.



Observer Pattern from GOF

Structure



Exercise

- Maze Code is shared
- Provide a better design with an additional design goal: flexibility to add more ways to solve the maze, rather than limiting itself to back-tracking only…

Design Defects & Restructuring

Week 7: 22 Oct 22

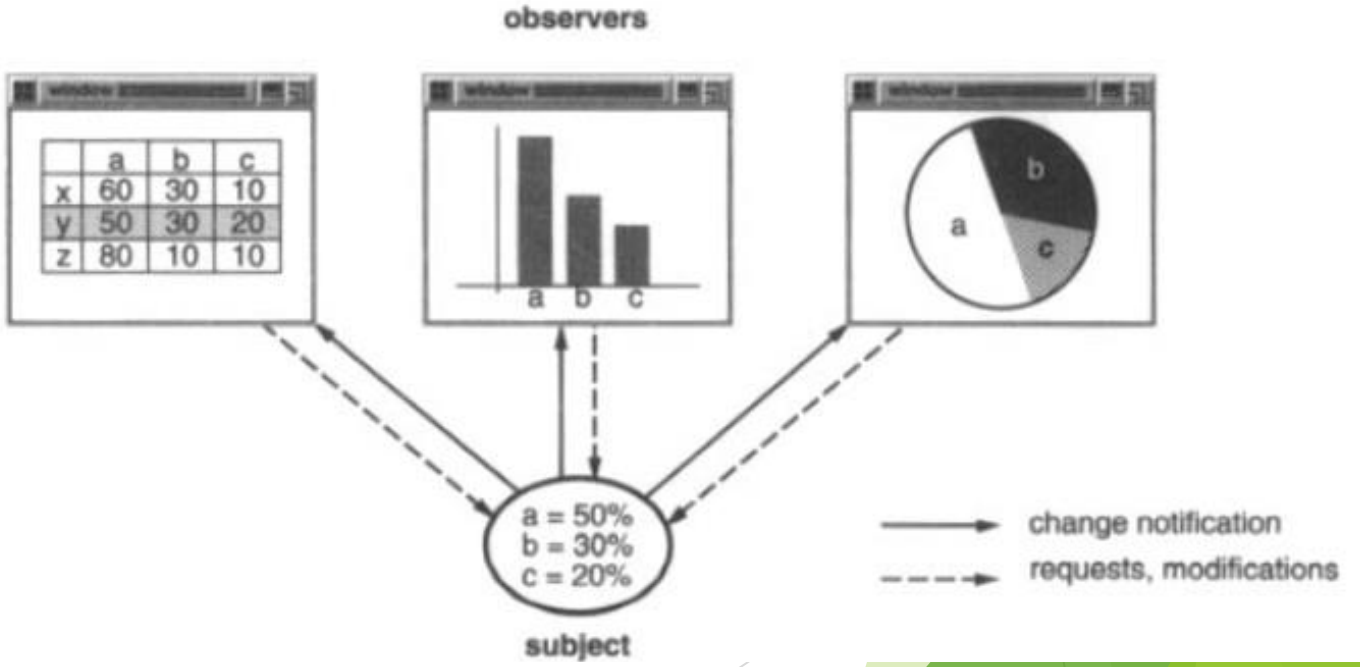
Rahim Hasnani

Observer Pattern from GOF

- Intent
  - Define a one-to-many dependency between objects so that when one objectchanges state, all its dependents are notified and updated automatically.
- Also Known As
  - Dependents, Publish-Subscribe
- Motivation
  - A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
  - For example in a GUI based system, classes defining application data and presentations ca n be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately and vice versa.

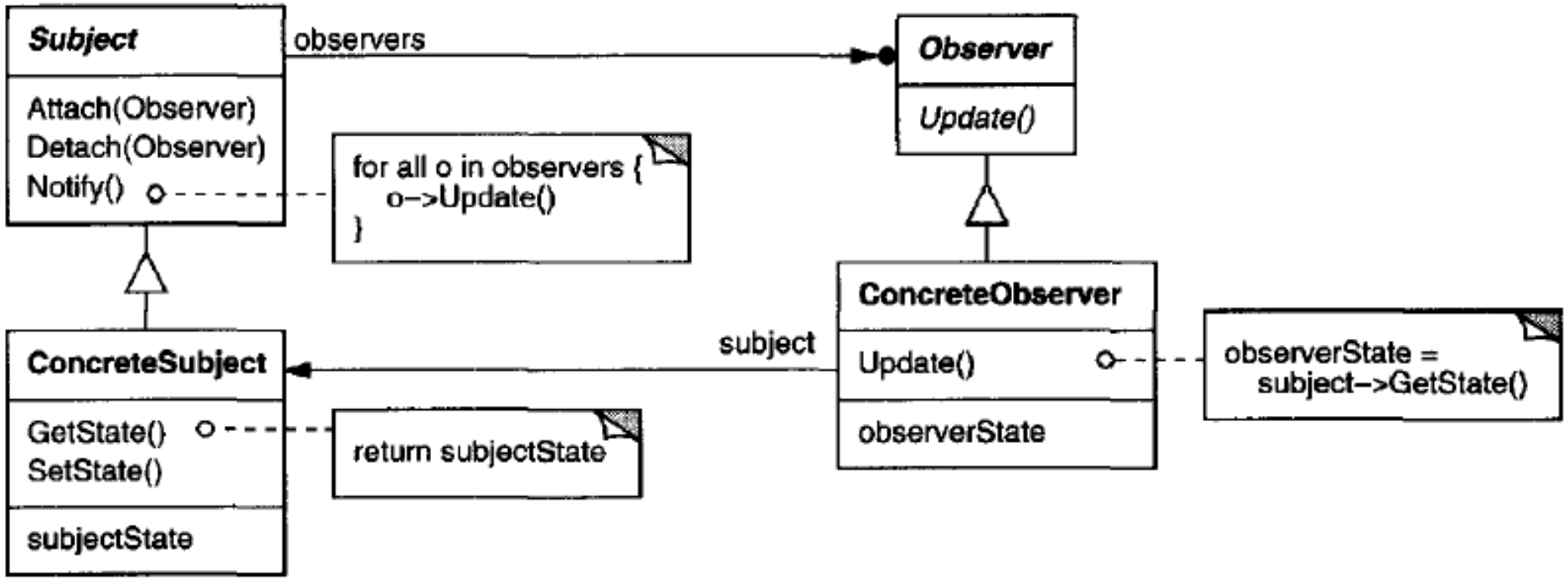
Observer Pattern from GOF

- The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observe rs. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as publish-subscribe. The subject is the publisher o f notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.



Observer Pattern from GOF

Structure



Refactoring - Definition

- As a noun:
  - “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.”
  - Example: Extract Function
- As a verb:
  - “to restructure software by applying a series of refactorings without changing its observable behavior.”
  - Example: ‘I might spend a couple of hours refactoring, during which I would apply a few dozen individual refactorings.’
- Key concepts:
  - Small changes
  - Changes/steps are behavior-preserving

Why Refactor

- Refactoring improves the design of software
  - As we add more code, it tends to decay – so need to periodically refactor
- Refactoring makes software easier to understand
  - Refactored code is more readable – easy to understand by future programmer
- Refactoring helps me find bugs
  - When carrying out refactoring, deep understanding of code is sought, which helps find bugs
- Refactoring helps me program faster (!!)
  - How? At first writing code without refactoring would be faster, but as we put more code into existing code base, it becomes harder to do it efficiently without refactoring first

When to Refactor

- Preparatory Refactoring – Making it easier to Add a feature
- Comprehension Refactoring: Making code easier to understand
- Litter-Pickup Refactoring
- Planned and Opportunistic Refactoring
- Long-Term Refactoring
- Refactoring in a Code Review

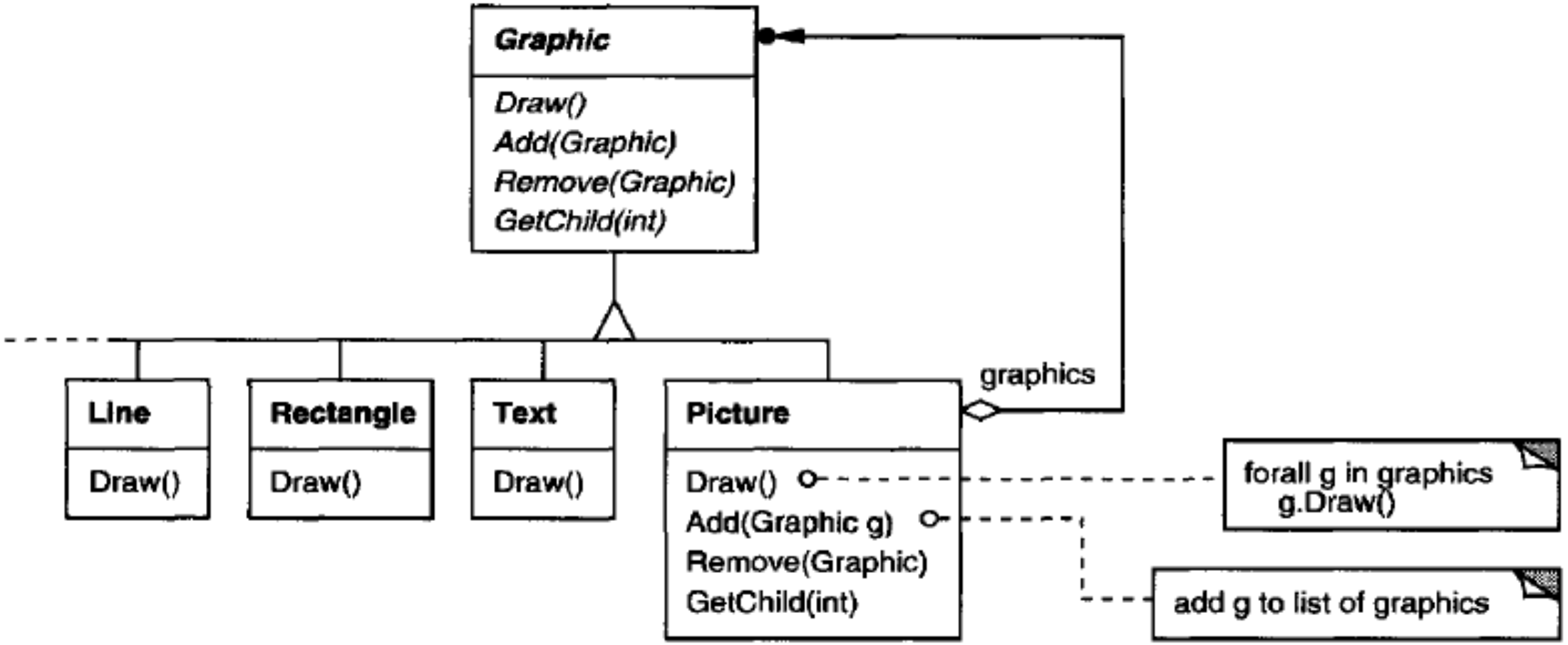
- When to Not Refactor

Problems with Refactoring

- Slowing Down New Features
- Code Ownership
- Branches
- Testing
- Legacy Code
- Databases

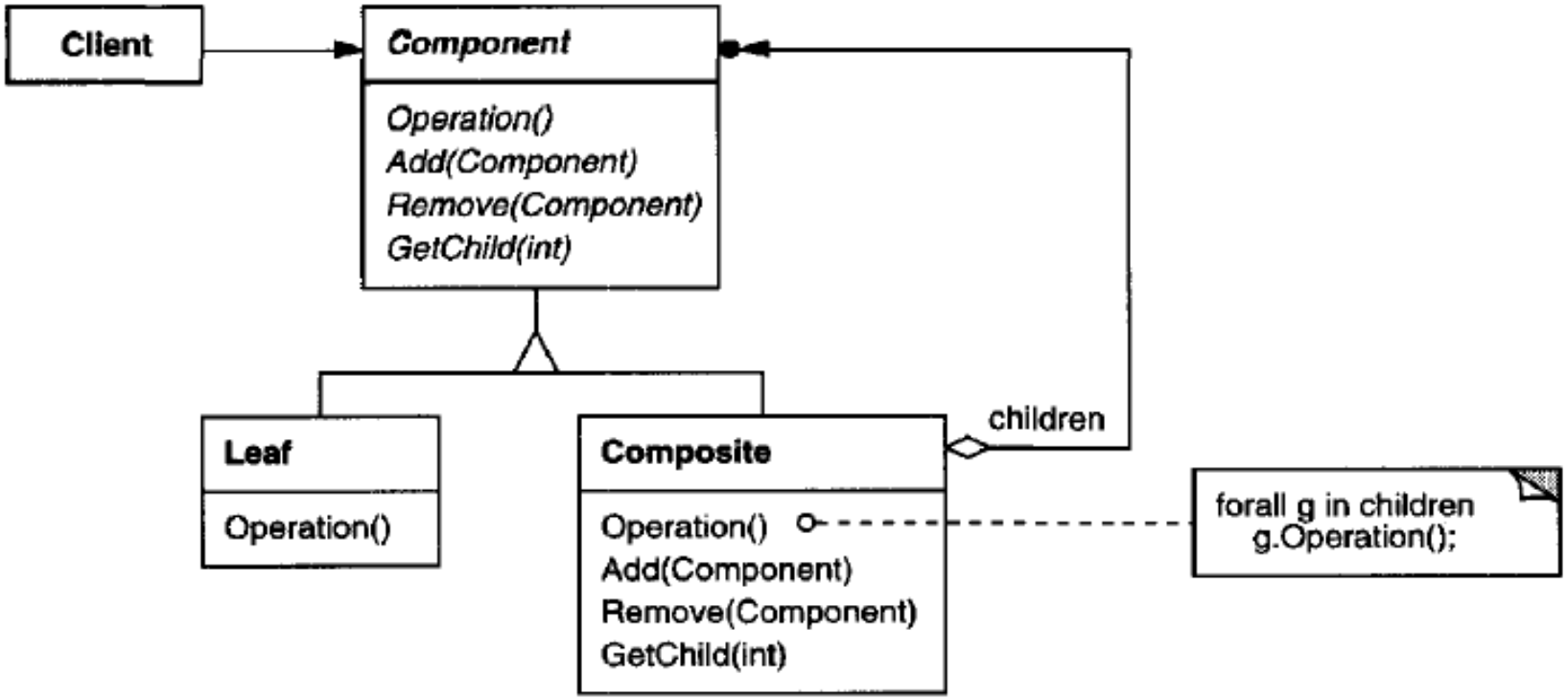
Design Pattern: Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Motivation
  - Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
  - But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

Design Pattern: Composite

- Applicability
  - Use the Composite pattern when
    - you want to represent part-whole hierarchies of objects
    - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- Structure



Design Pattern: Composite

- Participants
  - Component (Graphic)
    - declares th e interfa ce for obje cts in the composition.
    - implements default behavior for the interface common to all cl asses, as appropriate.
    - declares an interface for accessing and managing its child components.
    - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
  - Leaf (Rectangle, Line, Text, etc.)
    - represents leaf objects in the composition. A leaf has no children.
    - defines behavior for primitive objects in the composition.
  - Composite (Picture)
    - defines behavior for components having children.
    - stores child components.
    - implements child-related operations in the Component interface.
  - Client
    - manipulates objects in the composition through the Component interface.

Design Pattern: Composite

- Collaborations
  - Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
- Consequences

  The composite pattern
  - defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
  - makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition
  - makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
  - can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Exercise

- **Problem:** Create a program that reads new records from the database based on certain filter criteria and generates a file in a specified fixed length format on a location.
- **First Version:**
  - A console based application is made
  - The following is set through configuration (what is configuration??)
    - Database connection string
    - Query that provides the new records
    - Output file folder location
  - A single file code that carries out following:
    - Establishes connection with database. Throws error and exits if it cant connect
    - Runs the query and stores the result in memory. Throws error and exits if there is error
    - Verifies that query returns the correct number of column, otherwise reports error and exits.
    - Iterates over each row of the result and carries out validation (certain fields need to be present) and ignores those rows that do not fulfill the validation criteria. If all is well with a row, it converts the row into the required fixed format record for writing to file.
    - Opens file for writing and write all the converted records

Question 1:

- What are the design goals of client?
- What should be the design goals of the architect/designer to create a robust program?

Question 2

- Provide a design

Design Defects & Restructuring

Week 8: 29 Oct 22

Rahim Hasnani

Midterm 2 Syllabus

- Mid is on Thursday 3rd Nov 2022 at 10 am in Main Campus
- Syllabus
  - Refactoring Book Chapter 2
  - Assignment 2
  - Design Patterns/Principles/Concepts:
    - Dependency Injection
    - Factory Method
    - Observer Pattern
    - Composite Pattern
  - Exercises covered in Class
  - Today's lecture – (Decorator Pattern & Basic refactoring methods)

Exercise

- **Problem:** Create a program that reads new records from the database based on certain filter criteria and generates a file in a specified fixed length format on a location.
- **First Version:**
  - A console based application is made
  - The following is set through configuration (what is configuration??)
    - Database connection string
    - Query that provides the new records
    - Output file folder location
  - A single file code that carries out following:
    - Establishes connection with database. Throws error and exits if it cant connect
    - Runs the query and stores the result in memory. Throws error and exits if there is error
    - Verifies that query returns the correct number of column, otherwise reports error and exits.
    - Iterates over each row of the result and carries out validation (certain fields need to be present) and ignores those rows that do not fulfill the validation criteria. If all is well with a row, it converts the row into the required fixed format record for writing to file.
    - Opens file for writing and write all the converted records

Question 1:

- What are the design goals of client?
- What should be the design goals of the architect/designer to create a robust program?

Question 2

- Provide a design

Project

- Make Groups of 3
- Make Groups Now!
- Choose a topic
  - Design Pattern (other than GOF and other than covered in the class)
  - Refactoring method (other than mentioned in the text book)
- Deliverables
  - Make a Poster
  - Make a Video (could be a presentation and voiceover converted to video)
  - Make a Git submission available publicly
- All class work exercises would now be done in project groups

Decorator Pattern

- From GOF book
- Example from Headfirst book

Basic Refactorings

- Extract Function
- Inline Function
- Extract Variable
- Inline Variable

Design Defects & Restructuring

Week 9: 05 Nov 2022

Rahim Hasnani

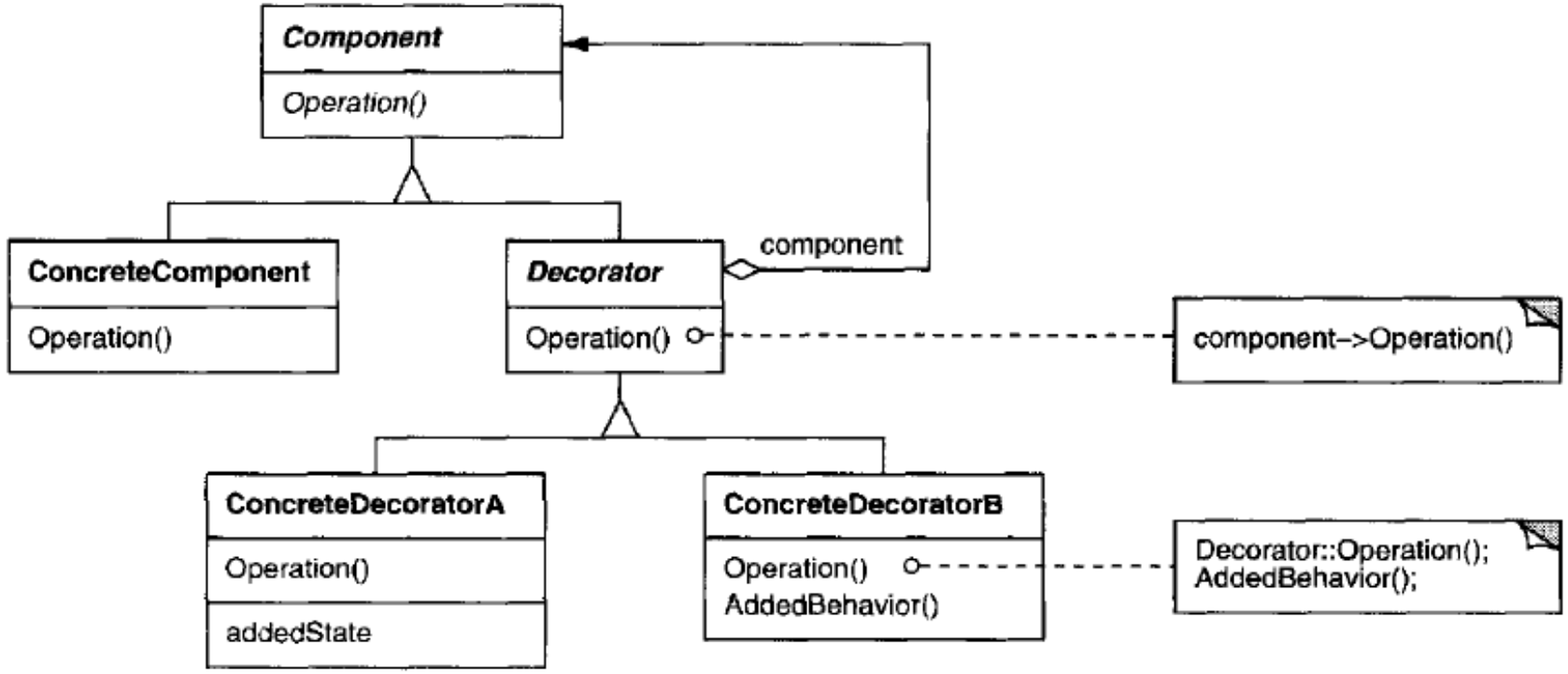
Project

- Decide on a topic NOW
- Make a git-hub repository for the project NOW and make me a member/collaborator (rhasnani@yahoo.com)

Decorator Pattern - Recap

Add responsibility to an individual object and not to an entire class

Why both aggregation and inheritance below?



Read the known uses section

Command Pattern

Encapsulating Method Invocation

Greetings!

I recently received a demo and briefing from Johnny Hurricane, CEO of Weather-O-Rama, on their new expandable weather station. I have to say, I was so impressed with the software architecture that I'd like to ask you to design the API for our new Home Automation Remote Control. In return for your services we'd be happy to handsomely reward you with stock options in Home Automation or Bust, Inc.

I'm enclosing a prototype of our ground-breaking remote control for your perusal. The remote control features seven programmable slots (each can be assigned to a different household device) along with corresponding on/off buttons for each. The remote also has a global undo button.

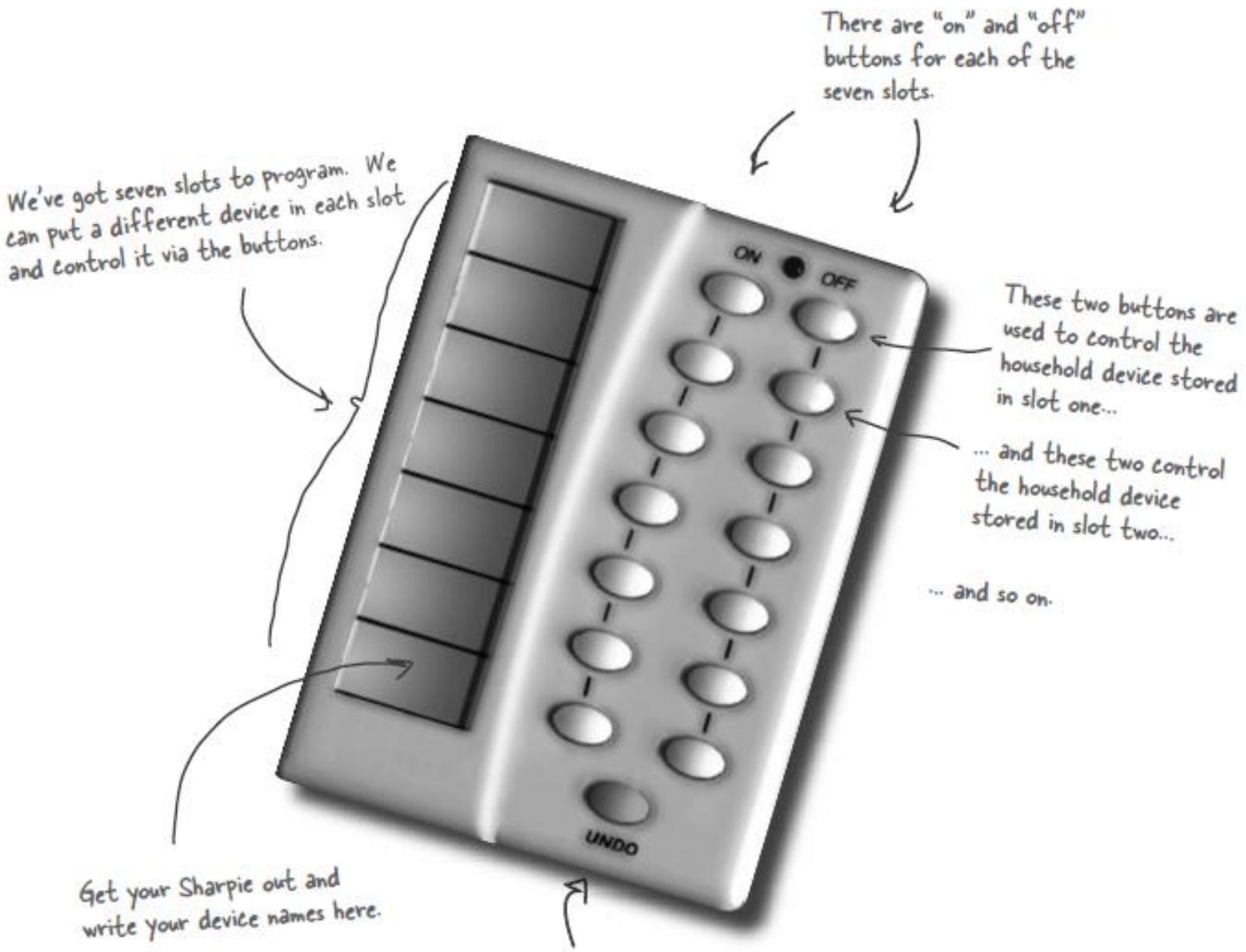
I'm also enclosing a set of Java classes on CD-R that were created by various vendors to control home automation devices such as lights, fans, hot tubs, audio equipment, and other similar controllable appliances.

We'd like you to create an API for programming the remote so that each slot can be assigned to control a device or set of devices. Note that it is important that we be able to control the current devices on the disc, and also any future devices that the vendors may supply.

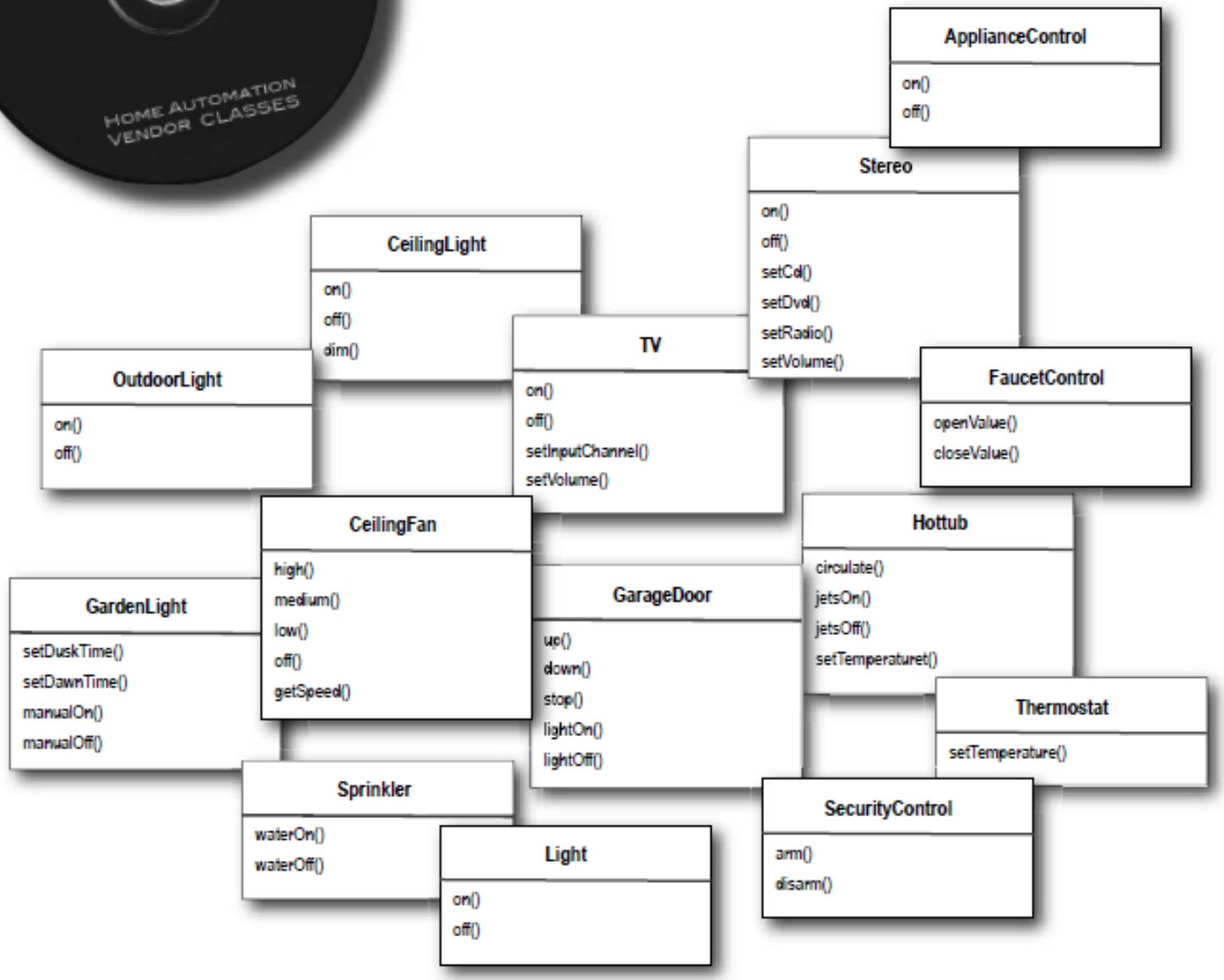
Given the work you did on the Weather-O-Rama weather station, we know you'll do a great job on our remote control!

We look forward to seeing your design.

Sincerely,


There are "on" and "off" buttons for each of the seven slots.
We've got seven slots to program. We can put a different device in each slot and control it via the buttons.
ON
OFF
These two buttons are used to control the household device stored in slot one...
... and these two control the household device stored in slot two...
... and so on.
UNDO
Get your Sharpie out and write your device names here.
Here's the global "undo" button that undoes the last button pressed.

Check out the vendor classes on the CD-R. These should give you some idea of the interfaces of the objects we need to control from the remote.



Command Pattern…

```
public interface Command {
    public void execute();
}
```

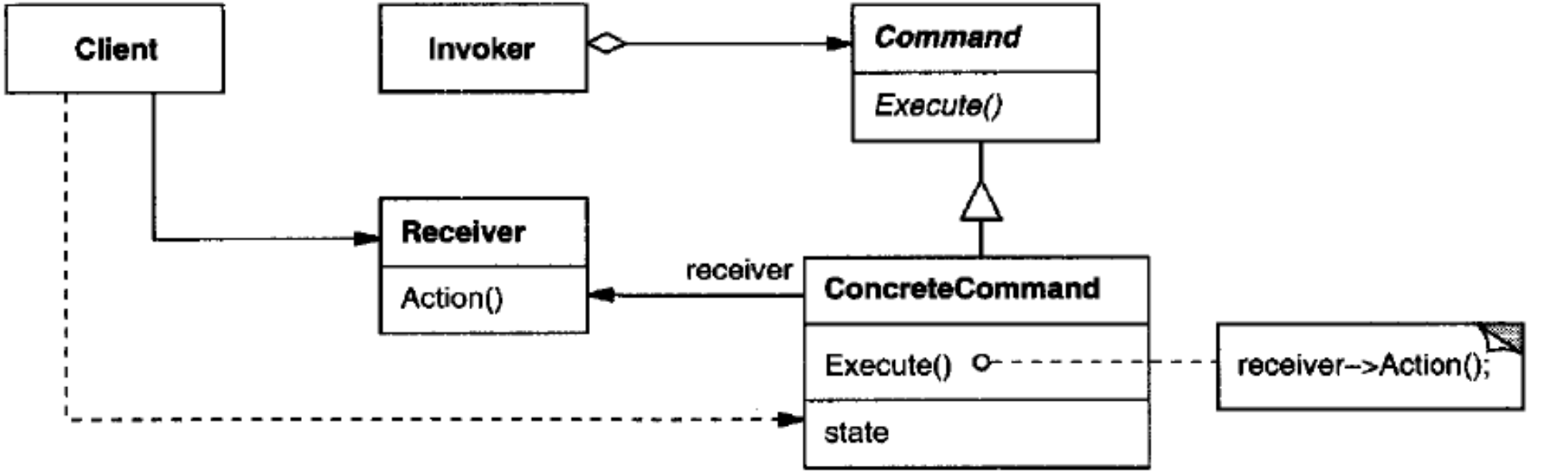
```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command)
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);


        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

Command Pattern - Structure



Adapter Pattern – Let's start from Applicability…

Applicability

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Refactoring Methods: Change Function Declaration

- Change Function Declaration
  - Name of function
  - Name & type of parameters
  - Add/Remove parameters
- Simple mechanics vs Migration mechanics
- Simple Mechanics Example: Rename Function
  - Change the method declaration to the desired declaration.
  - Find all references to the old method declaration, update them to the new one.
  - Test.

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

```
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Refactoring Methods:

- Migration Mechanics Example: Rename function
  - Use Extract Function on the function body to create the new function.
  - Test.
  - Apply Inline Function to the old functions.
  - Test.

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

```
function circum(radius) {
  return circumference(radius);
}
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Design Defects & Restructuring

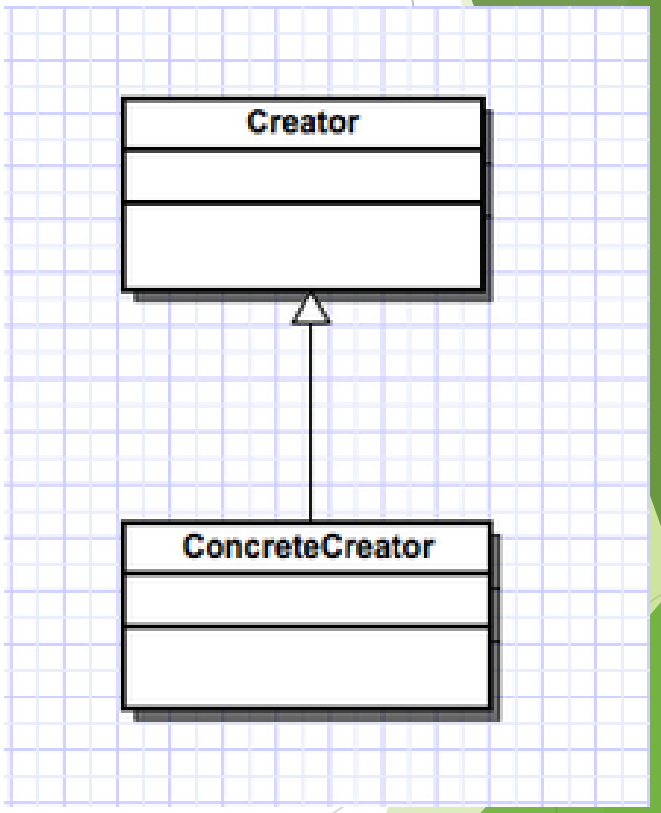
Week 10: 12 Nov 2022

Rahim Hasnani

Creational Patterns

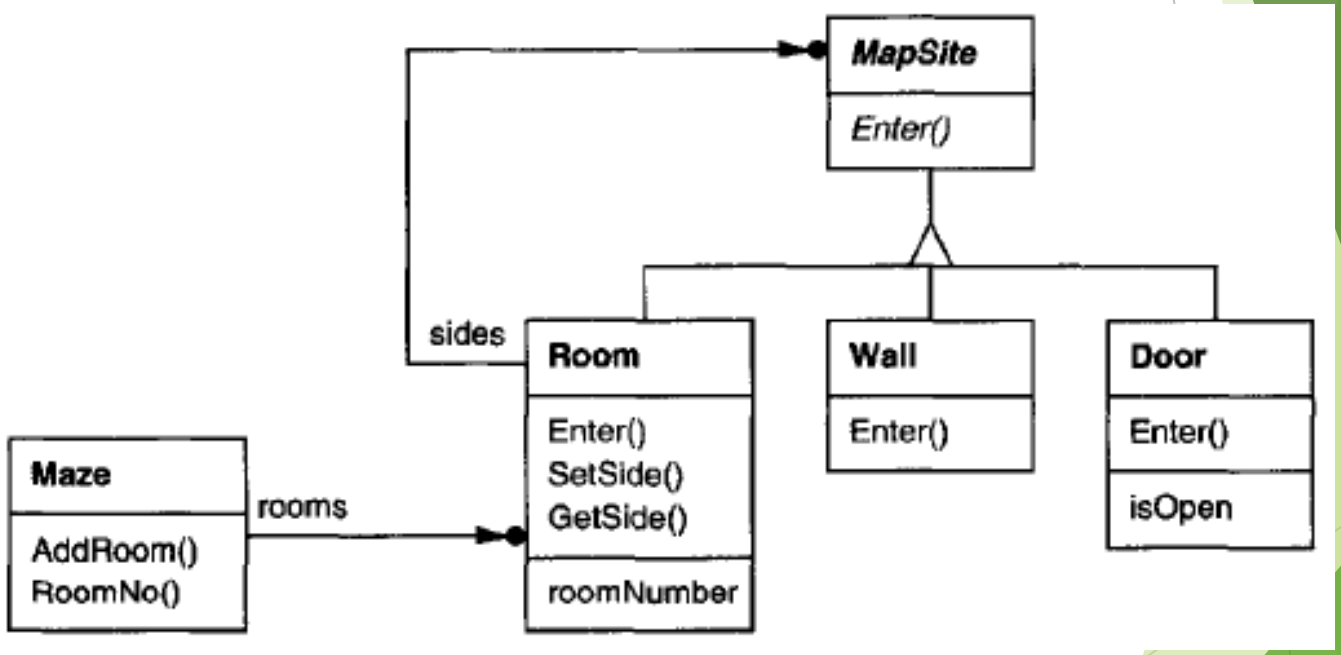
- Basic Idea:
  - “System should be independent of how its objects and products are created” (https://en.wikipedia.org/wiki/Creational_pattern)
- GOF Creational patterns
  - Abstract Factory: Create instances of several families of classes
  - Builder: Constructing a complex object step by step
  - Factory Method: creation through inheritance
  - Singleton: Ensuring one instance
  - Prototype: creation by making a copy

(https://sourcemaking.com/design_patterns/creational_patterns)



Maze

- Here we are concentrated on how maze get created.
- Our version of maze:
  - Maze is a set of rooms
  - A room knows about its neighbors. A neighbor could be:
    - A wall
    - A door to another room
  - Each room has four sides



```
class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    // ...
};
```

```
class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Abstract Factory

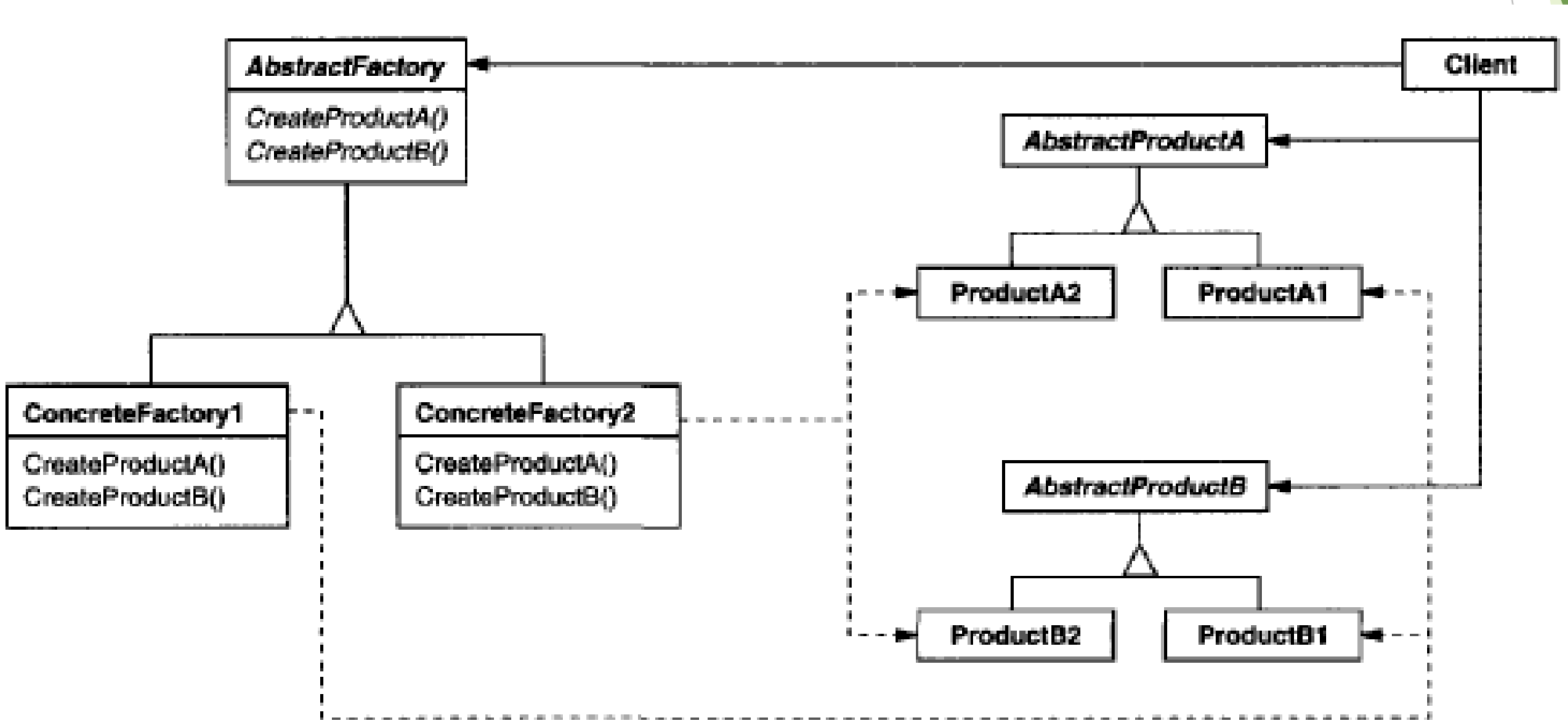
Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Applicability

- a system should be independent o f how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce th is constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory



```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

Builder

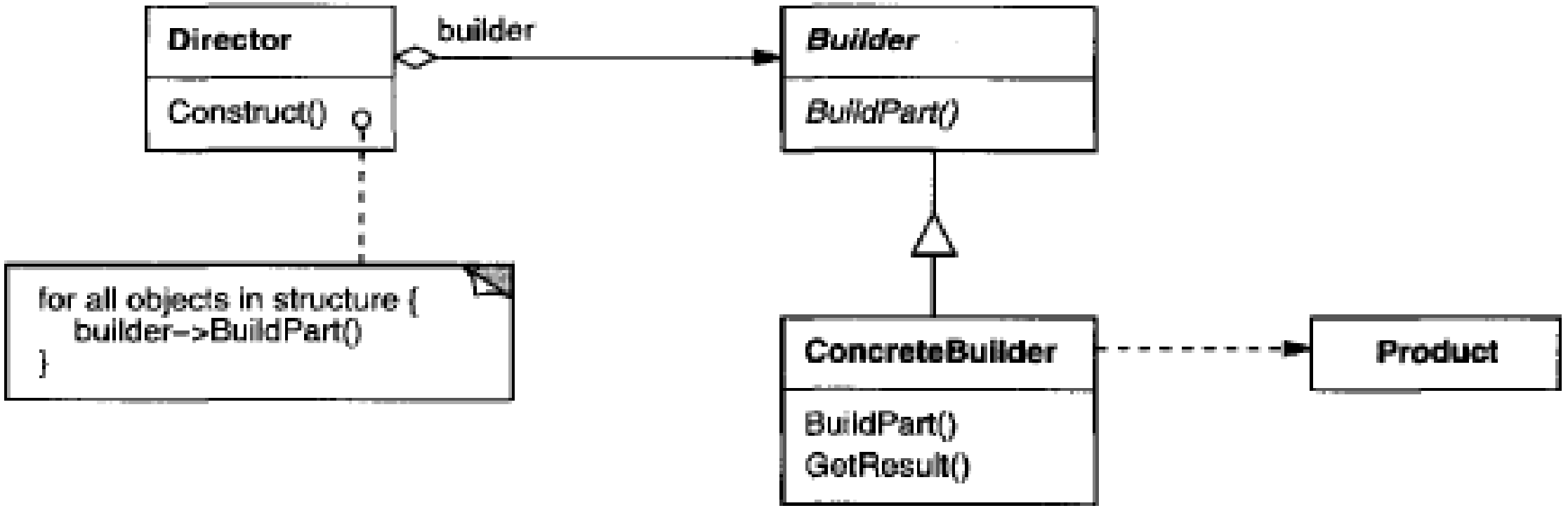
Intent

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

Applicability

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Builder


Director
builder
Construct()
Builder
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
BuildPart()
GetResult()
Product

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Factory Method

```
class MazeGame {
public:
    Maze* CreateMaze();

// factory methods:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Prototype

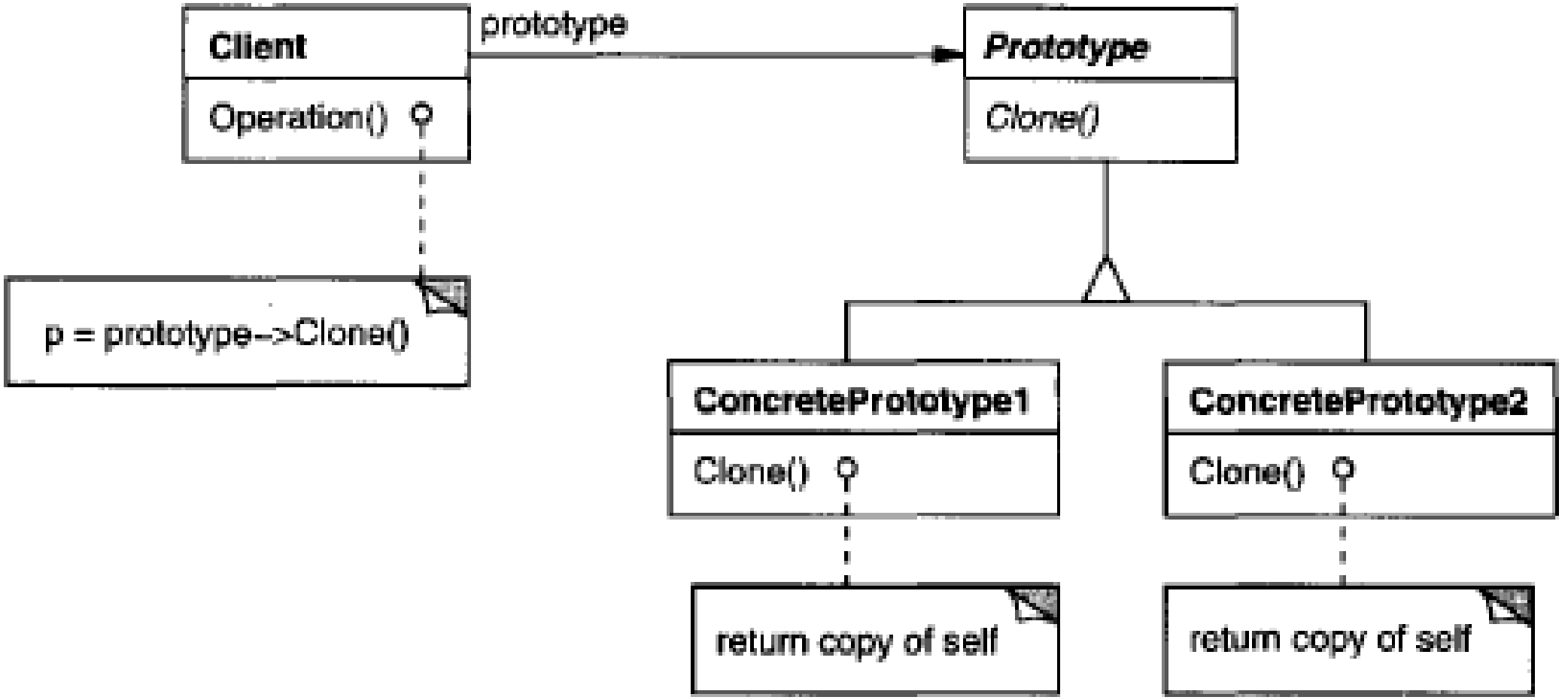
Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Applicability

- when the classes to instantiate are specified at run-time, for example, by dynamic loading ; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state..

Prototype


Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Project

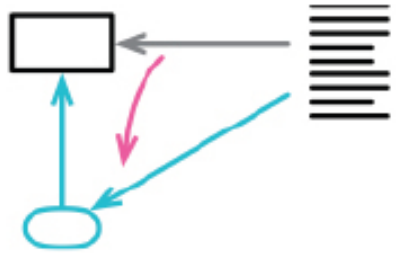
- Decide on a topic NOW
- Make a git-hub repository for the project NOW and make me a member/collaborator (rhasnani@yahoo.com)

Design Defects & Restructuring

Week 11: 18/19 Nov 2022

Rahim Hasnani

Basic Refactoring: Encapsulate Variable



```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner()       {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

- Create encapsulating functions to access and update the variable.
- For each reference to the variable, replace with a call to the appropriate encapsulating function.
- Restrict the visibility of the variable.
- How to prevent updates to data?

Basic Refactoring: Rename Variable


```
let a = height * width;
```

```
let area = height * width;
```

- If the variable is used widely, consider *Encapsulate Variable*
- Find all references to the Variable, and change every one

Basic Refactoring: Introduce Parameter Object

- If there isn't a suitable structure already, create one.
- Use *Change Function Declaration (124)* to add a parameter for the new structure.
- Test.
- Adjust each caller to pass in the correct instance of the new structure. Test after each one.
- For each element of the new structure, replace the use of the original parameter with the element of the structure. Remove the parameter. Test.


```
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}
```

```
function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```

Basic Refactoring: Combine Functions Into Class

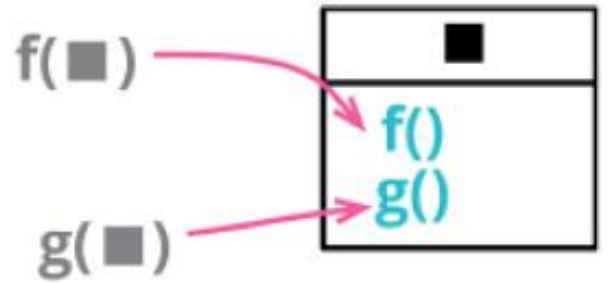
- Apply *Encapsulate Record (162)* to the common data record that the functions share.

  If the data that is common between the functions isn't already grouped into a record structure, use *Introduce Parameter Object* to create a record to group it together.

- Take each function that uses the common record and use *Move Function* to move it into the new class.

  Any arguments to the function call that are members can be removed from the argument list.

- Each bit of logic that manipulates the data can be extracted with *Extract Function* and then moved into the new class.

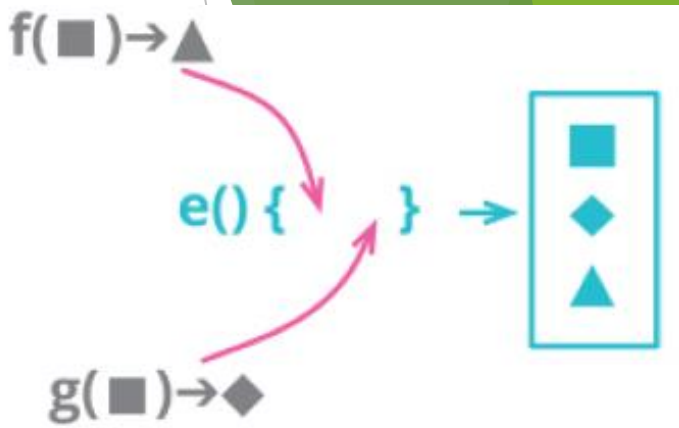



```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```

```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}
```

Basic Refactoring: Combine Functions Into Transform

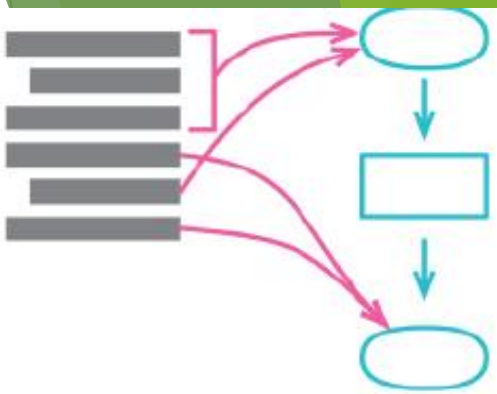
- Alternative to Combine Functions Into Class
- Rather than creating a new class, derived fields are added to the structure
- Problem occurs when data is changed after enriching…




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

Basic Refactoring: Split Phase



- Extract the second phase code into its own function.
- Test.
- Introduce an intermediate data structure as an additional argument to the extracted function.
- Test.
- Examine each parameter of the extracted second phase. If it is used by first phase, move it to the intermediate data structure. Test after each move.
- Apply *Extract Function* on the first-phase code, returning the intermediate data structure.

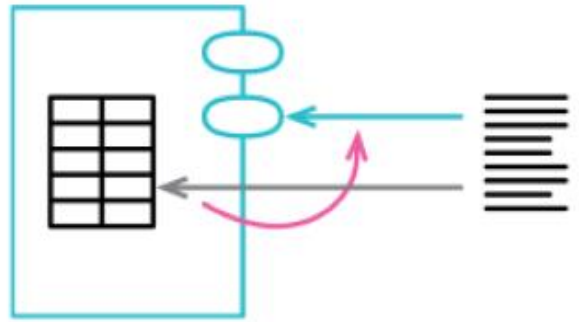
```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values =  aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

Encapsulate: Encapsulate Record

- Use *Encapsulate Variable* on the variable holding the record.

  Give the functions that encapsulate the record names that are easily searchable.
- Replace the content of the variable with a simple class that wraps the record. Define an accessor inside this class that returns the raw record. Modify the functions that encapsulate the variable to use this accessor.
- Test.
- Provide new functions that return the object rather than the raw record.
- For each user of the record, replace its use of a function that returns the record with a function that returns the object. Use an accessor on the object to get at the field data, creating that accessor if needed. Test after each change.
- Remove the class's raw data accessor and the easily searchable functions that returned the raw record.
- Test.
- If the fields of the record are themselves structures, consider using Encapsulate Record and *Encapsulate Collection* recursively.



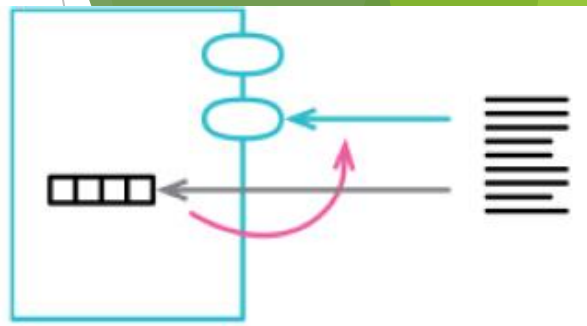
```
organization = {name: "Acme Gooseberries", country: "GB"};
```

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name()    {return this._name;}
  set name(arg) {this._name = arg;}
  get country()    {return this._country;}
  set country(arg) {this._country = arg;}
}
```

Encapsulate: Encapsulate Collection

- Apply *Encapsulate Variable* if the reference to the collection isn't already encapsulated.
- Add functions to add and remove elements from the collection.

  If there is a setter for the collection, use *Remove Setting Method* if possible. If not, make it take a copy of the provided collection.
- Run static checks.
- Find all references to the collection. If anyone calls modifiers on the collection, change them to use the new add/remove functions. Test after each change.
- Modify the getter for the collection to return a protected view on it, using a read-only proxy or a copy.
- Test.

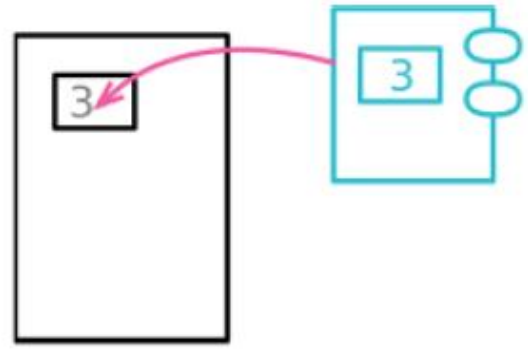


```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse)    { ... }
  removeCourse(aCourse) { ... }
```

Encapsulate: Replace Primitive with Object

- Apply *Encapsulate Variable* if it isn't already.
- Create a simple value class for the data value. It should take the existing value in its constructor and provide a getter for that value.
- Run static checks.
- Change the setter to create a new instance of the value class and store that in the field, changing the type of the field if present.
- Change the getter to return the result of invoking the getter of the new class.
- Test.
- Consider using *Rename Function* on the original accessors to better reflect what they do.
- Consider clarifying the role of the new object as a value or reference object by applying *Change Reference to Value* or *Change Value to Reference*.




```
orders.filter(o => "high" === o.priority
                || "rush" === o.priority);
```

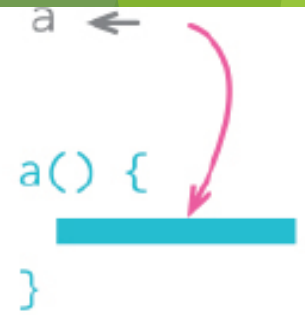
```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

Encapsulate: Replace Temp with Query

- Check that the variable is determined entirely before it's used, and the code that calculates it does not yield a different value whenever it is used.
- If the variable isn't read-only, and can be made read-only, do so.
- Test.
- Extract the assignment of the variable into a function.

  If the variable and the function cannot share a name, use a temporary name for the function.

  Ensure the extracted function is free of side effects. If not, use *Separate Query from Modifier*.
- Test.
- Use *Inline Variable* to remove the temp.



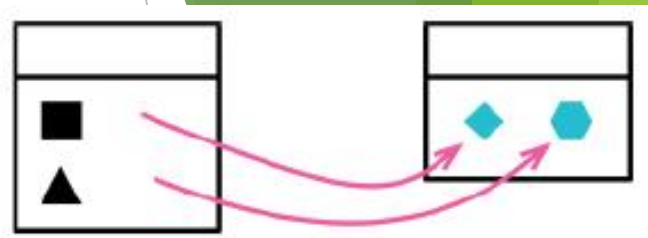

```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
get basePrice() {this._quantity * this._itemPrice;}

...

if (this.basePrice > 1000)
  return this.basePrice * 0.95;
else
  return this.basePrice * 0.98;
```

Encapsulate: Extract Class

- Decide how to split the responsibilities of the class.
- Create a new child class to express the split-off responsibilities.

  If the responsibilities of the original parent class no longer match its name, rename the parent.
- Create an instance of the child class when constructing the parent and add a link from parent to child.
- Use *Move Field* on each field you wish to move. Test after each move.
- Use *Move Function* to move methods to the new child. Start with lower-level methods (those being called rather than calling). Test after each move.
- Review the interfaces of both classes, remove unneeded methods, change names to better fit the new circumstances.
- Decide whether to expose the new child. If so, consider applying *Change Reference to Value* to the child class.

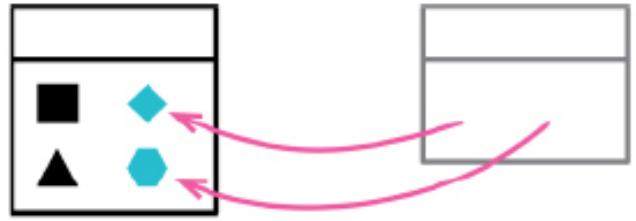


```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```

```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

Encapsulate: Inline Class

- In the target class, create functions for all the public functions of the source class.
- These functions should just delegate to the source class.
- Change all references to source class methods so they use the target class's delegators instead. Test after each change.
- Move all the functions and data from the source class into the target, testing after each move, until the source class is empty.
- Delete the source class and hold a short, simple funeral service.

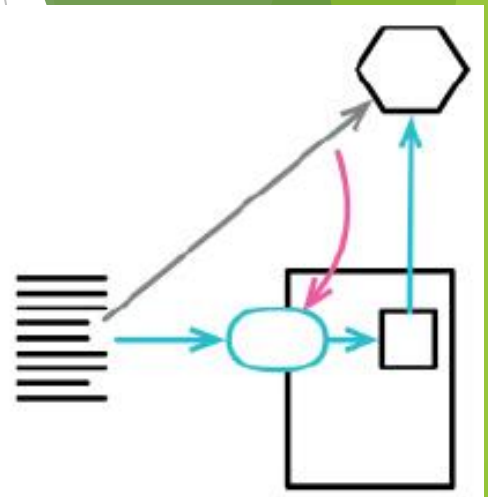


```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```

Encapsulate: Hide Delegate

- For each method on the delegate, create a simple delegating method on the server.
- Adjust the client to call the server. Test after each change.
- If no client needs to access the delegate anymore, remove the server's accessor for the delegate.
- Test.



```
manager = aPerson.department.manager;
```

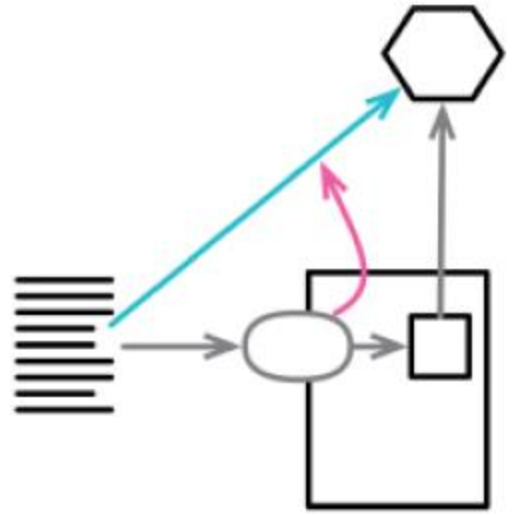
```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

Encapsulate: Remove Middle Man

- Create a getter for the delegate.
- For each client use of a delegating method, replace the call to the delegating method by chaining through the accessor. Test after each replacement.

  If all calls to a delegating method are replaced, you can delete the delegating method.

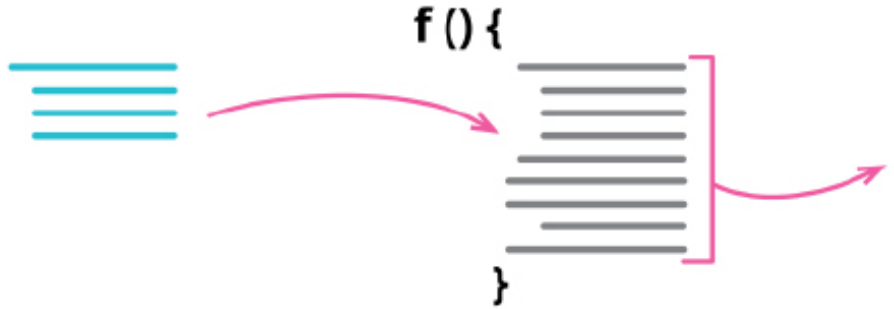
  With automated refactorings, you can use *Encapsulate Variable* on the delegate field and then *Inline Function* on all the methods that use it.



```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

```
manager = aPerson.department.manager;
```

Encapsulate: Substitute Algorithm



- Arrange the code to be replaced so that it fills a complete function.
- Prepare tests using this function only, to capture its behavior.
- Prepare your alternative algorithm.
- Run static checks.
- Run tests to compare the output of the old algorithm to the new one. If they are the same, you're done. Otherwise, use the old algorithm for comparison in testing and debugging.

```
function foundPerson(people) {
  for(let i = 0; i < people.length; i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
    }
  }
  return "";
}
```

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
  return people.find(p => candidates.includes(p)) || '';
}
```

Exercise

- Carry out refactoring

Design Defects & Restructuring

Week 12: 19/21 Nov 2022

Rahim Hasnani

Moving Features: Move Function

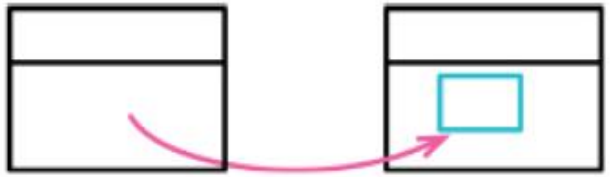
- Examine all the program elements used by the chosen function in its current context. Consider whether they should move too.
- Check if the chosen function is a polymorphic method.
- Copy the function to the target context. Adjust it to fit in its new home.
- Perform static analysis.
- Figure out how to reference the target function from the source context.
- Turn the source function into a delegating function.
- Test.
- Consider *Inline Function* on the source function.

```
class Account {
  get overdraftCharge() {...}
```

```
class AccountType {
    get overdraftCharge() {...}
```

Moving Features: Move Field

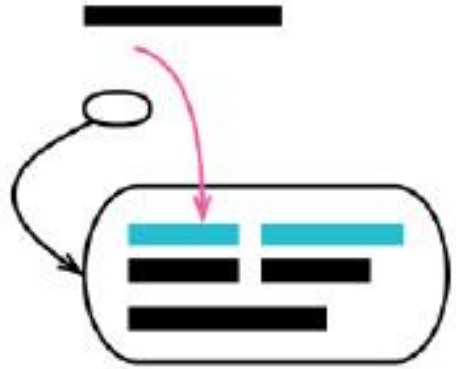
- Ensure the source field is encapsulated.
- Test.
- Create a field (and accessors) in the target.
- Run static checks.
- Ensure there is a reference from the source object to the target object.
- Adjust accessors to use the target field.
- Test.
- Remove the source field.
- Test.



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this._discountRate;}
```

```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this.plan.discountRate;}
```

Moving Features: Move Statements Into Functions



- If the repetitive code isn't adjacent to the call of the target function, use *Slide Statements* to get it adjacent.
- If the target function is only called by the source function, just cut the code from the source, paste it into the target, test, and ignore the rest of these mechanics.
- If you have more callers, use *Extract Function* on one of the call sites to extract both the call to the target function and the statements you wish to move into it. Give it a name that's transient, but easy to grep.
- Convert every other call to use the new function. Test after each conversion.
- When all the original calls use the new function, use *Inline Function (* to inline the original function completely into the new function, removing the original function.
- *Rename Function* to change the name of the new function to the same name as the original function. Or to a better name, if there is one.

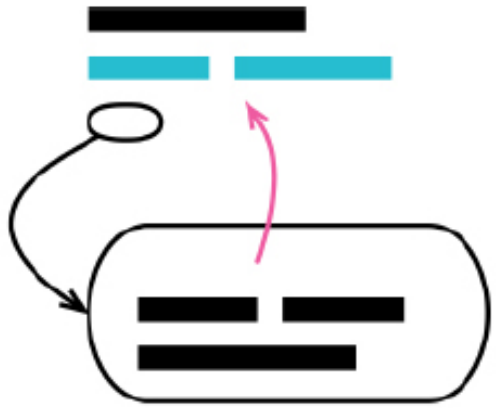
```
result.push(`<p>title: ${person.photo.title}</p>`);
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

Moving Features: Move Statements to Callers

- In simple circumstances, where you have only one or two callers and a simple function to call from, just cut the first line from the called function and paste (and perhaps fit) it into the callers. Test and you're done.
- Otherwise, apply *Extract Function* to all the statements that you *don't* wish to move; give it a temporary but easily searchable name.

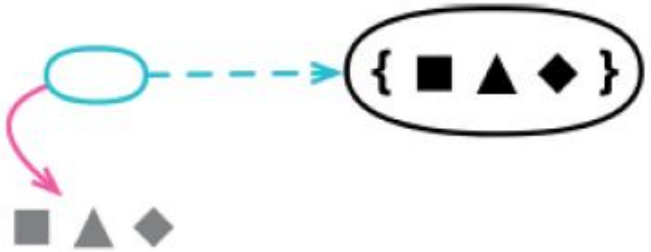
  If the function is a method that is overridden by subclasses, do the extraction on all of them so that the remaining method is identical in all classes. Then remove the subclass methods.
- Use *Inline Function* on the original function.
- Apply *Change Function Declaration* on the extracted function to rename it to the original name. Or to a better name, if you can think of one.



```
emitPhotoData(outStream, person.photo);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

Moving Features: Replace Inline Code with Function Call



```
let appliesToMass = false;
for(const s of states) {
  if (s === "MA") appliesToMass = true;
}
```

```
appliesToMass = states.includes("MA");
```

- Replace the inline code with a call to the existing function.
- Test.

Moving Features: Slide Statements

- Identify the target position to move the fragment to. Examine statements between source and target to see if there is interference for the candidate fragment. Abandon action if there is any interference.

  A fragment cannot slide backwards earlier than any element it references is declared.

  A fragment cannot slide forwards beyond any element that references it.

  A fragment cannot slide over any statement that modifies an element it references.

  A fragment that modifies an element cannot slide over any other element that references the modified element.

- Cut the fragment from the source and paste into the target position.
- Test.

```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```

```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

Moving Features: Split Loop

- Copy the loop.
- Identify and eliminate duplicate side effects.
- Test.

When done, consider *Extract Function* on each loop.

```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

Moving Features: Replace Loop with Pipeline

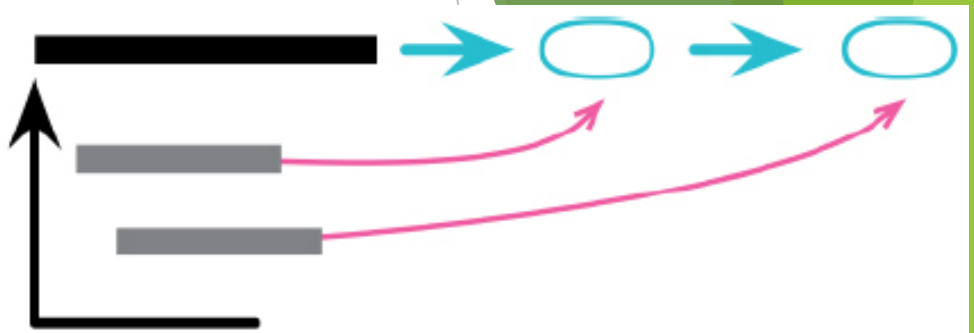
- Create a new variable for the loop's collection.

  This may be a simple copy of an existing variable.

- Starting at the top, take each bit of behavior in the loop and replace it with a collection pipeline operation in the derivation of the loop collection variable. Test after each change.

- Once all behavior is removed from the loop, remove it.

  If it assigns to an accumulator, assign the pipeline result to the accumulator.

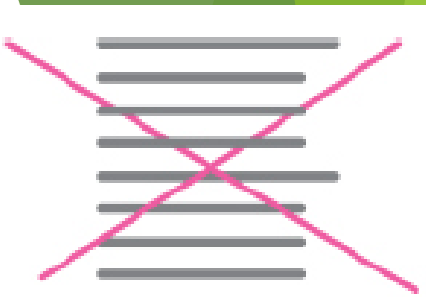


```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```

```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
;
```

Moving Features: Remove Dead Code

- If the dead code can be referenced from outside, e.g., when it's a full function, do a search to check for callers.
- Remove the dead code.
- Test.



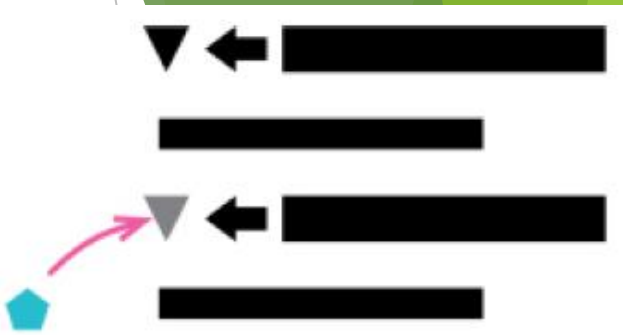
```
if(false) {
  doSomethingThatUsedToMatter();
}
```

Organizing Data: Split Variable

- Change the name of the variable at its declaration and first assignment.

  If the later assignments are of the form `i = i + something`, that is a collecting variable, so don't split it. A collecting variable is often used for calculating sums, string concatenation, writing to a stream, or adding to a collection.

- If possible, declare the new variable as immutable.
- Change all references of the variable up to its second assignment.
- Test.
- Repeat in stages, at each stage renaming the variable at the declaration and changing references until the next assignment, until you reach the final assignment.

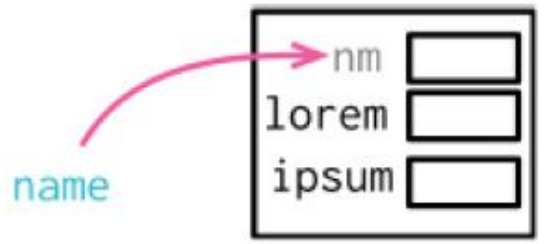


```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Organizing Data: Rename Field

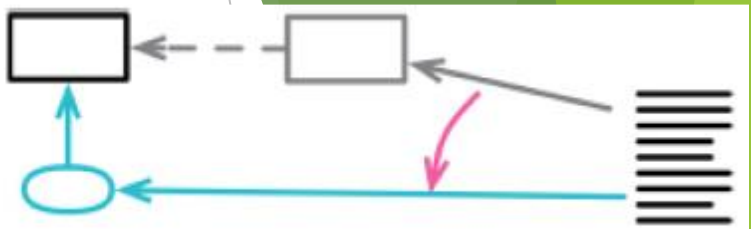
- If the record has limited scope, rename all accesses to the field and test; no need to do the rest of the mechanics.
- If the record isn't already encapsulated, apply *Encapsulate Record*.
- Rename the private field inside the object, adjust internal methods to fit.
- Test.
- If the constructor uses the name, apply *Change Function Declaration (124)* to rename it.
- Apply *Rename Function (124)* to the accessors.




```
class Organization {
  get name() {...}
}
```

```
class Organization {
  get title() {...}
}
```

Organizing Data: Replace Derived Variable with Query



- Identify all points of update for the variable. If necessary, use *Split* to separate each point of update.
- Create a function that calculates the value of the variable.
- Use *Introduce Assertion* to assert that the variable and the calculation give the same result whenever the variable is used.

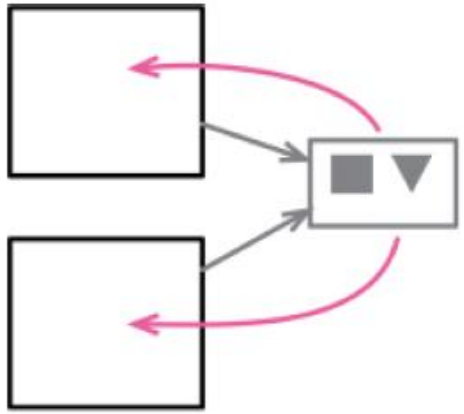
  If necessary, use *Encapsulate Variable* to provide a home for the assertion.
- Test.
- Replace any reader of the variable with a call to the new function.
- Test.
- Apply *Remove Dead Code* to the declaration and updates to the variable.

```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
  const old = this._discount;
  this._discount = aNumber;
  this._discountedTotal += old - aNumber;
}
```

```
get discountedTotal() {return this._baseTotal - this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

Organizing Data: Change Reference to Value

- Check that the candidate class is immutable or can become immutable.
- For each setter, apply *Remove Setting Method* .
- Provide a value-based equality method that uses the fields of the value object.
- Most language environments provide an overridable equality function for this purpose. Usually you must override a hash-code generator method as well.

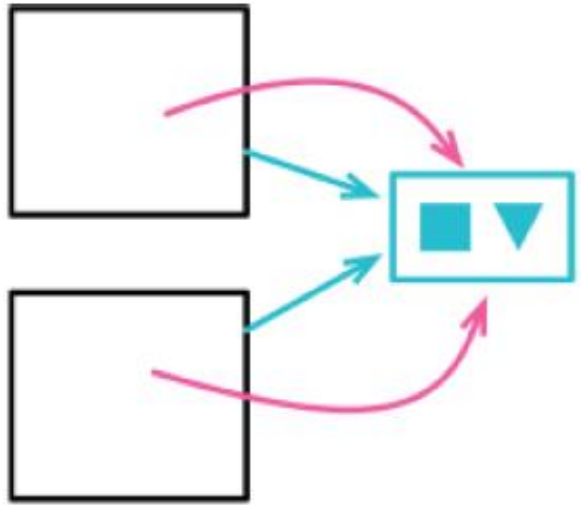


```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._price.currency);
  }
```

Organizing Data: Change Value to Reference

- Create a repository for instances of the related object (if one isn't already present).
- Ensure the constructor has a way of looking up the correct instance of the related object.
- Change the constructors for the host object to use the repository to obtain the related object. Test after each change



```
let customer = new Customer(customerData);
```

```
let customer = customerRepository.get(customerData.id);
```

More Methods…

- Covered directly from the book

Design Defects & Restructuring

Week 13: 26 Nov 2022

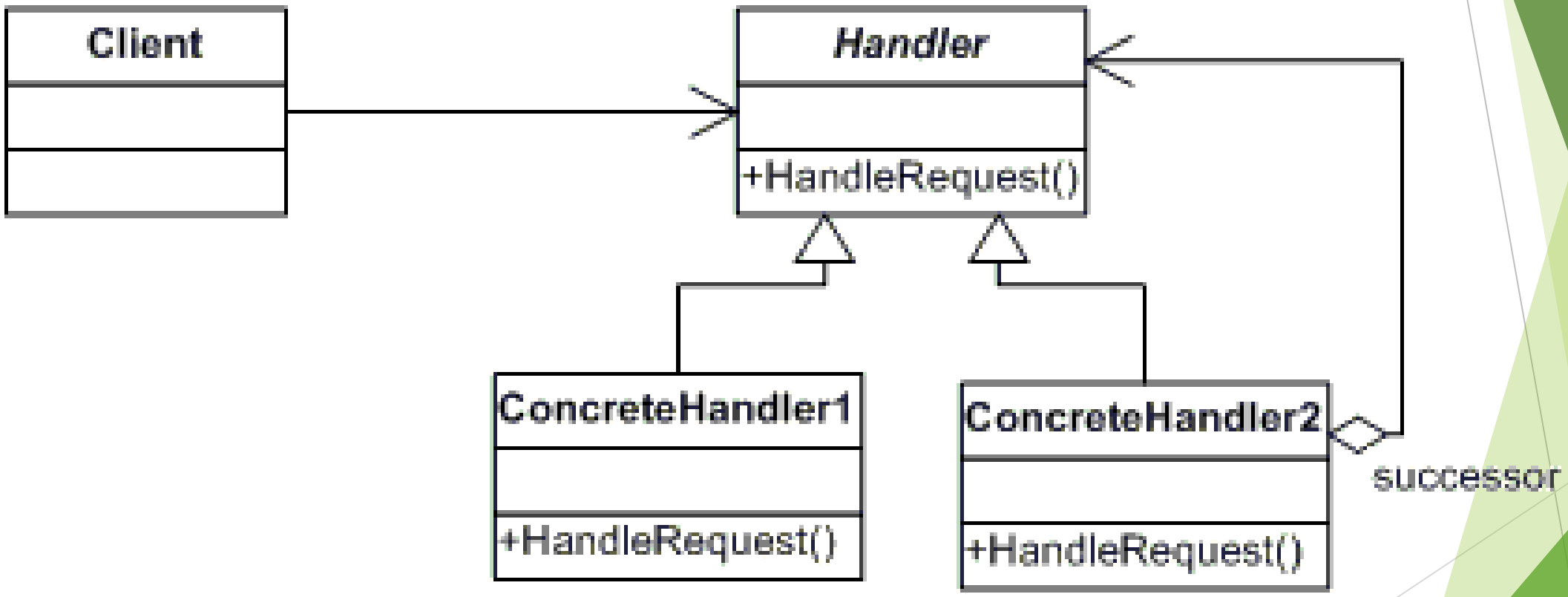
Rahim Hasnani

Chain of Responsibility

- Intent
  - 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
- Applicability
  - More than one object may handle a request, and the handler isn't known a priori
    - The handler should be ascertained automatically
  - You want to issue a request to one of several objects without specifying the receiver explicitly
  - The set of objects that can handle a request should be specified dynamically

Borrowed from Syed Yousuf's lectures

Chain of Responsibility



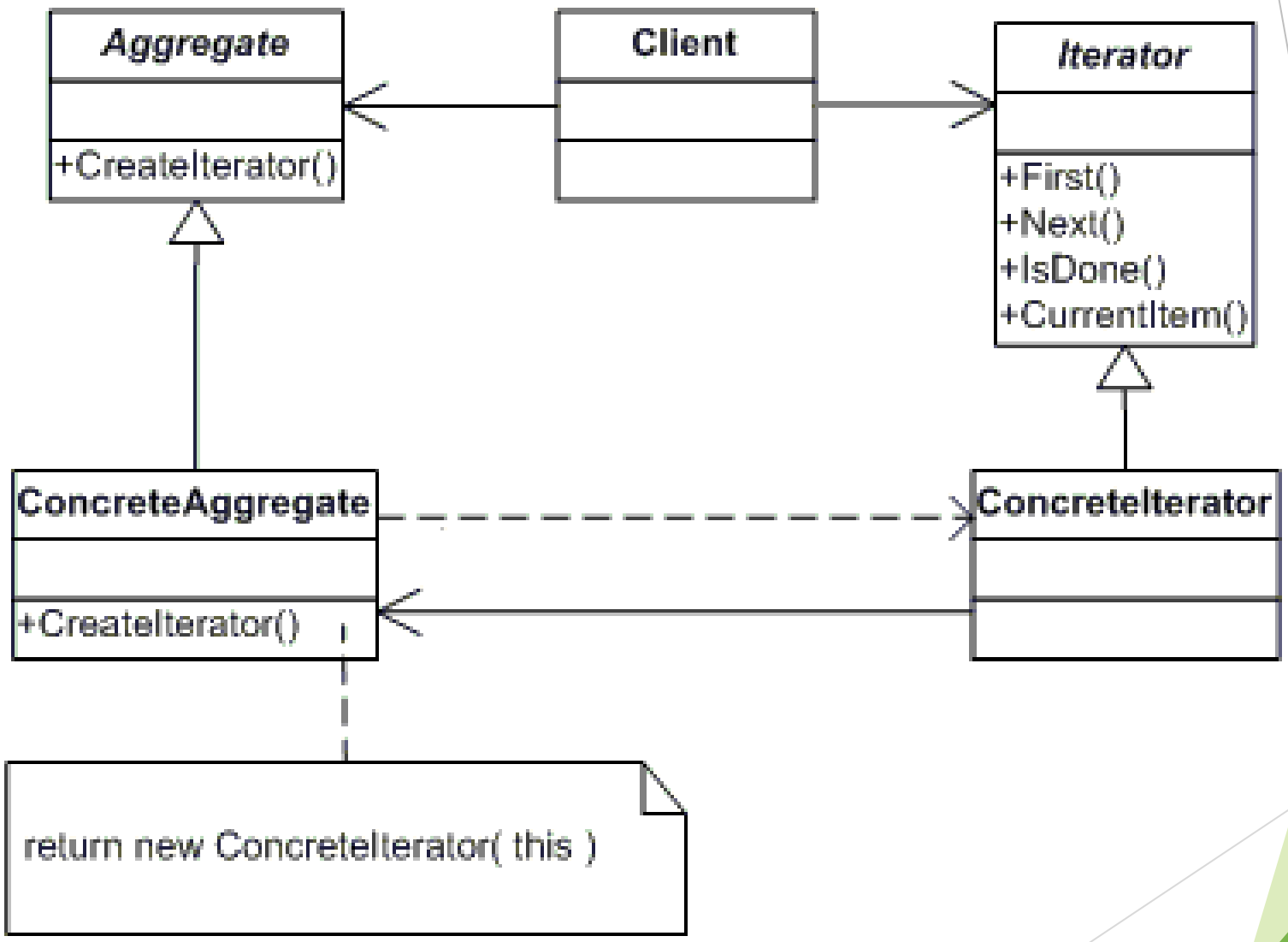
Borrowed from Syed Yousuf's lectures

Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Applicability
  - To access an aggregate object's contents without exposing its internal representation
  - To support multiple traversals of aggregate objects
  - To provide a uniform interface for traversing different aggregate structures (to support polymorphic iteration)

Borrowed from Syed Yousuf's lectures

Iterator



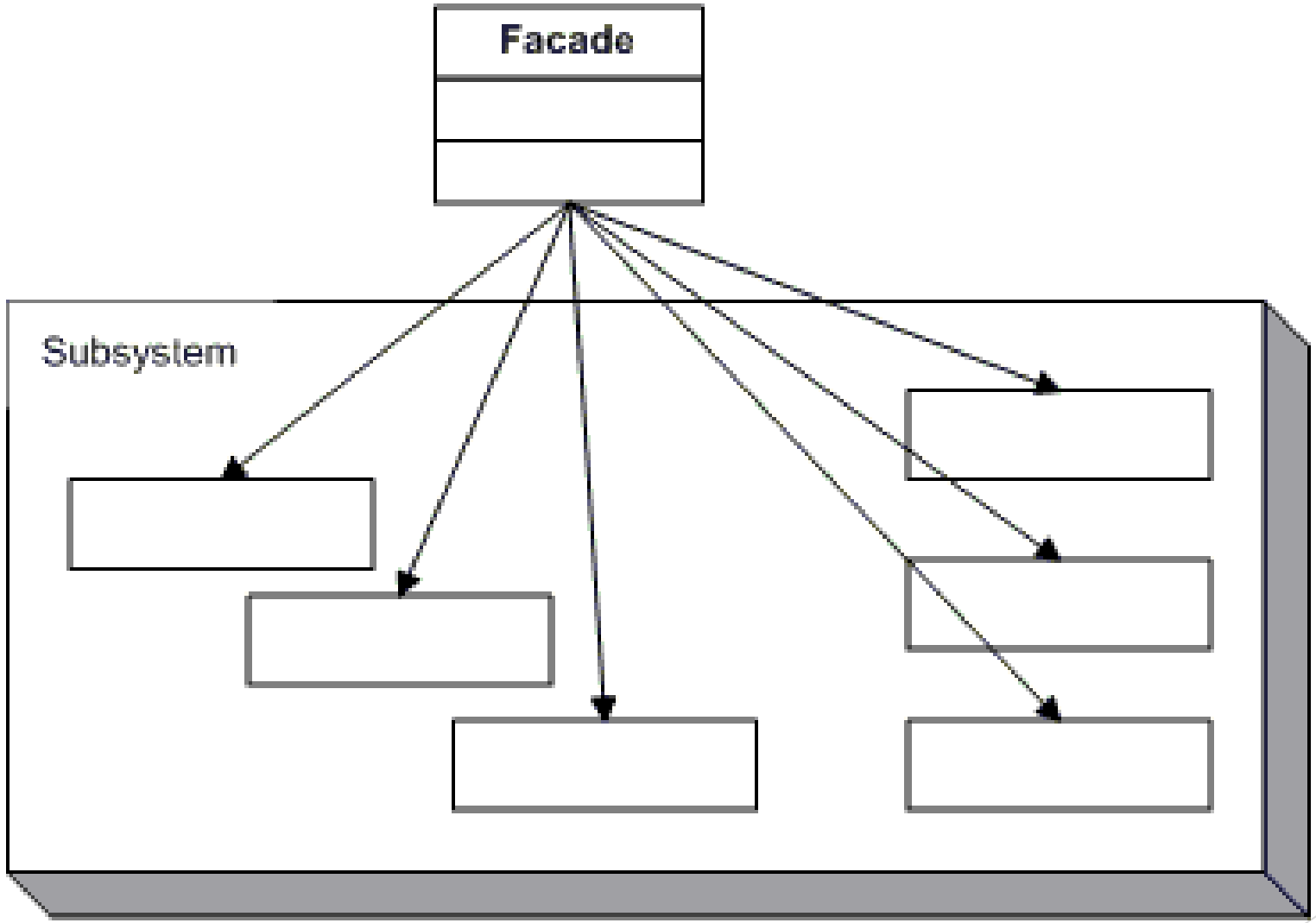
Borrowed from Syed Yousuf's lectures

Façade

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem
  - Façade defines a higher-level interface that makes the subsystem easier to use
- Applicability
  - You want to provide a simple interface to a complex subsystem
  - There are many dependencies between clients and the implementation classes of an abstraction
    - Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability
  - You want to layer your subsystems

Borrowed from Syed Yousuf's lectures

Façade



Borrowed from Syed Yousuf's lectures

Test Cases in Refactoring

Refactoring Tools

- https://blog.ndepend.com/top-10-visual-studio-refactoring-tips/
- https://www.youtube.com/watch?v=qod8aFrGSRE
- https://www.youtube.com/watch?v=G1S6NZfFvOg
- https://www.youtube.com/watch?v=WX8Rgqjny5A
- https://www.youtube.com/watch?v=WCMb3V7nxms

Today’s Exercise