

# SOFTWARE DESIGN

Prof. Mihaela Dînsoreanu

Contact: room D01, Baritiu 26-28

E-mail: mihaela.dinsoreanu@cs.utcluj.ro

# MANAGEMENT ISSUES

Time & Location:

- See Schedule on [www.ac.utcluj.ro](http://www.ac.utcluj.ro)
- Course files: moodle.cs.utcluj.ro

course enrollment key: **course\_PS/SD2020**

group key: **grouP\_30231** (adapt to your group #)

Grading:

- Project 20%
- Lab 20%
- Final Exam 60%

# YOUR TEACHING ASSISTANTS (TA)

30231 – Cristian Chira

30232 – Grigore Vlad

30233 – Anca Iordan

30234 – Anca Iordan

30235 – Daniel Ciugurean

30236 – Samuel Dolean

30237 – Anca Iordan

30238 – Anca Iordan

30239 – Paul Stanescu

302310 – Timotei Dolean

30431 – Lucian Braescu

30432 – Mihai Visan

30433 – Radu Tufisi

30434 – Richard Ardelean

CSC – Maria Potolea

# LAB SESSIONS

- Are COMPULSORY
- Maximum 3 absences allowed (BUT should be caught up)
- Only one assignment/lab session can be presented
- You need to get a grade  $\geq 5$  for the lab and project to attend the final exam
- **Attend the lab sessions only when your group is scheduled**

# RESEARCH

## Research for Diploma projects

(Deep) Machine learning applied in

- Neuroscience ((Explainable)Network analysis, Information coding, Spike sorting and burst detection)
- Language representation and understanding (ex. chatbots)
- IoT Data Analysis (ex. failure prediction, user profiling)
- Learning robots (imitation learning, reinforcement learning)

# PROJECT

- Decide
- If Research project
  - ⇒ Write an e-mail to any of {rodica.potolea@cs.utcluj.ro},  
mihaila.dinsoreanu@cs.utcluj.ro, camelia.lemnaru@cs.utcluj.ro }  
**by the end of the week (Sunday, 1<sup>st</sup> of March)** containing  
your name, group, relevant grades so far (i.e. Programming  
Techniques, Algorithms, etc.)
  - ⇒ We will get back to you with the next steps

# WHAT DO YOU EXPECT FROM THIS COURSE?

Please feel encouraged to tell/e-mail me any ideas/suggestions/complaints...

# REFERENCES [1]

## Software Architectures

- Juval Lowy, Righting software, O'Reilly, 2020
- Ian Gorton, Essential Software Architecture, Springer, second ed. 2011.
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley.
- Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 3<sup>rd</sup> edition, 2013.
- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, 2012
- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 2001. *Pattern-oriented system architecture, volume 1: A system of patterns*. Hoboken, NJ: John Wiley & Sons. [POSA book]
- Fowler Martin, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002

# REFERENCES [2]

## Design Patterns

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. AddisonWesley, 1995. [GoF]
- Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition), Prentice Hall, 2004, ISBN: 0131489062

## Courses

- B. Meyer (ETH Zurich)
- G. Kaiser (Columbia Univ. NY)
- I. Crnkovic (Sweden)
- (Univ. of Copenhagen)
- R. Marinescu (Univ. Timisoara)
- SaaS (Stanford)

# COURSE CONTENT [TENTATIVE]

1. Introduction. OOP Concepts. SOLID
2. Class design principles (GRASP). Package Design principles
- 3 – 5. Architectural Patterns
6. Live coding session
7. Midterm?
- 8 - 9. Enterprise applications patterns
- 10-12. Design Patterns
13. Quality Attributes
14. Exam review

# OBJECTIVES

After completing this course, you should be able to:

- **Identify the most relevant functional and non-functional requirements of a software system and document them**
- **Generate architectural alternatives for a problem by applying major software architectural styles and design patterns**
- **Analyze and select among them, based on well-known design principles and best practices**

# VALUE OF SOFTWARE?

## Behavior

“a program that works perfect now but is impossible to change”

- Urgent
- Not (always) important

## Architecture

“a program that doesn’t work perfect now but can be easily changed”

- Not (particularly) urgent
- Important

# WHAT IS (GOOD) ARCHITECTURE?

Supports:

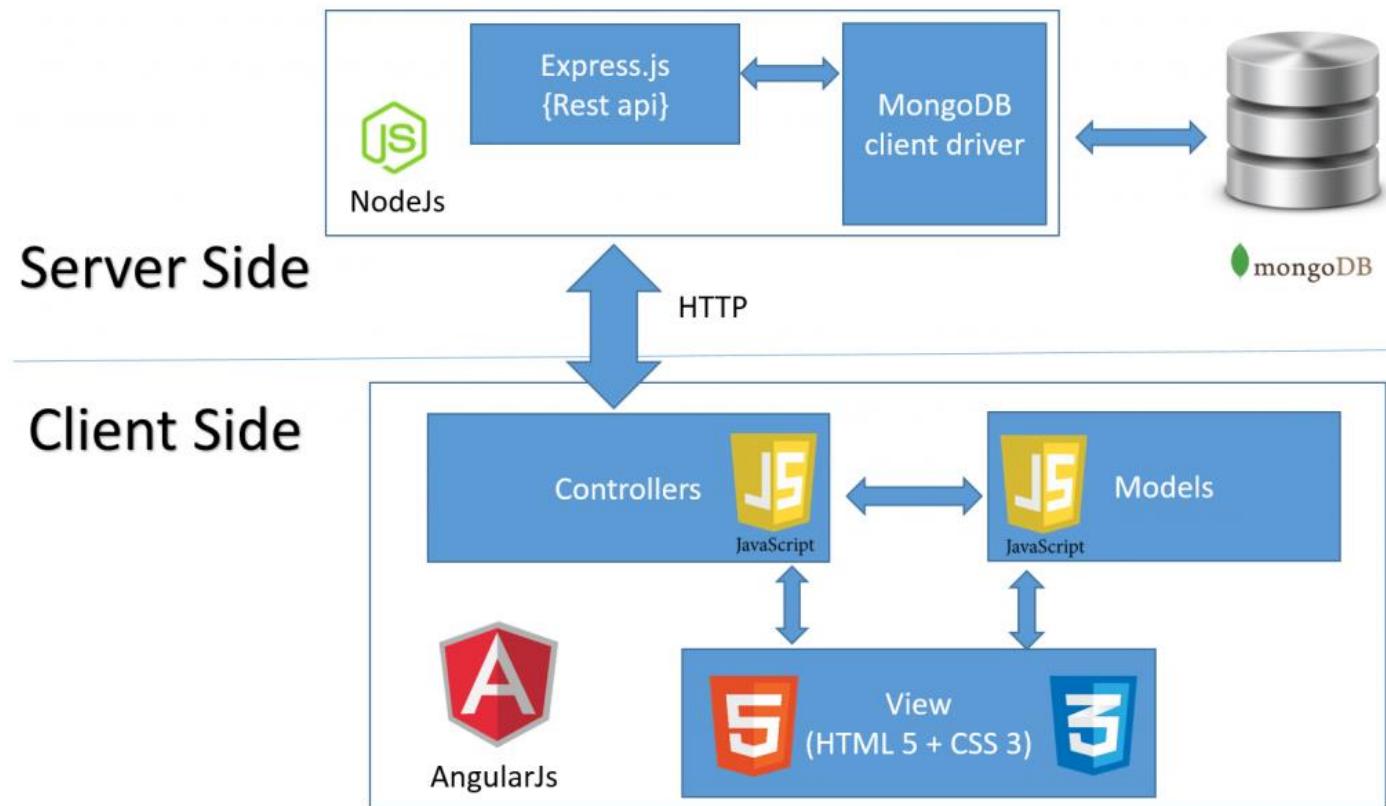
- The use cases and operation of the system (“screaming” architecture).
- The maintenance of the system.
- The development of the system.
- The deployment of the system.

By:

- Setting boundaries (decoupling)
- Leaving options open (separate policies from details)

# WHAT IS ARCHITECTURE NOT?

## Technology stack



# DECOUPLING LEVELS

## Source

- components all execute in the same address space,
- communicate with each other using simple function calls.
- a single executable loaded into computer memory

## Deployment

- independent deployable units (ex. jar files, DLLs, shared libraries)

## Service

- dependencies at the level of data structures,
- communication solely through network packets
- every execution unit is entirely independent of source and binary changes to others

# SOFTWARE DESIGN TECHNIQUES

What are Software Design Techniques?

- A **set of practices** for analysing, decomposing, and modularising software system architectures
- Characterized by structuring the system architecture on the basis of its **components** rather than the **actions** it performs.

# LEARNING SD TECHNIQUES

## **Junior Developer** (knows rules)

- knows algorithms, data structures and programming languages
- writes programs, although not always good ones

## **Senior Developer** (understands principles)

- understands software design & programming paradigms with pros and cons
- importance of cohesion, coupling, information hiding, dependency management etc.

## **Technical Architect** (applies patterns (i.e. proven solutions))

- develops design models
- understands how design solutions interact and can be integrated

# WHAT DO YOU NEED?

## Knowledge

- attending lectures AND reading books – terminology, concepts, principles, methods, and theories

## Understanding

- using your knowledge by applying it in hands-on activities, e.g., practical exercises, assignments, projects, discussions

## Skills

- actively and continuously work hard, gaining experience (practice!)

# TODAY'S OUTLINE

Basic OOP Review

SOLID Class Design Principles

# REFERENCES

[1] Martin, Robert C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017.

[2]

<http://butunclebob.com/Articles.UncleBob.PrinciplesOfOod>

[3] SOLID ebook

[4] <https://martinfowler.com/articles/injection.html>

# WHAT IS A CLASS?

A type that encapsulates

- State (Attributes)
- Constructors
- Behavior (Methods)

Class candidates:

- Person 
- Mihaela 
- AddAccount 

# HOW TO DECLARE A CLASS (JAVA)

```
class ClassName [extends ParentName implements InterfaceName(s)]  
{  
    [modifier(s)] type attribute1;  
    ...  
    [modifier(s)] return_type method1(param_list)  
        { //method body here}  
}
```

Access modifiers: public, protected, private

“Mutability” modifier: final

“Scope” modifier: static

# THE PERSON CLASS

```
class Person {  
    private int birthYear;  
    private String firstName, lastName;  
    private boolean employed;  
    private int nrOfLegs;  
  
    //constructor(s)  
    //setters & getters  
}
```

What should we make static?

What should we make final?

# THE PERSON CLASS

```
//code here  
  
class Person {  
    private final int birthYear;  
    private String firstName, lastName;  
    private boolean employed;  
    private static int nrOfLegs;  
  
    final  
  
    //constructor(s)  
    //setters & getters  
  
    ...  
}
```



# OVERLOADING METHODS

Define in a class, methods with the same name and different:

- Number of parameters
- Type of parameters
- Return type

```
class Person {  
    public int calculateAge() {  
        return Date.currentYear() -  
birthYear; }  
  
    public int calculateAge (int year )  
{  
    return year - birthYear; }  
  
    public float calculateAge () {  
        return Date.currentYear() -  
birthYear; } }
```

# WHAT IS AN OBJECT?

A specific entity of the type defined by the class.

⇒ Has specific values for the attributes

me is an object of type Person.

```
Person me = new Person();
```

```
me.firstName = "Mihaela"
```

```
me.lastName = "Dinsoreanu"
```

```
me.employed = true
```

```
me.numberOfLegs = ??
```

# HOW TO USE OBJECTS?

Call public methods to **query the object** (getters)

```
String name = me.getfirstName() + " " +  
me.getlastName();
```

```
int birthY = me.getbirthYear();
```

```
int age = me.calculateAge();
```

...

# HOW TO USE OBJECTS? (2)

Call public methods to **set attribute values** (setters)

```
me.setfirstName(fN);
```

```
me.setlastName(lN);
```

```
me.setBirthYear(bY);
```

```
me.setAge(int age);
```

...

How are parameters passed?

- By value !
  
- Primitive type?
- Reference?

```
public class Person {
    final int birthYear;
    String firstName, lastName;
    boolean employed;
    static int nrOfLegs;
    public Person (String fN, String lN, int bY, boolean e)
    {
        firstName = fN;
        lastName = lN;
        employed = e;
        birthYear = bY;
    }

    public void setFirstName(String n)
    {
        firstName = n;
    }
    public void setLastName(String n)
    {
        lastName = n;
    }

    public int calculateAge (int year ) {
        return year - birthYear;
    }
    public String toString(){
        return "The person "+firstName+ ' '+lastName + " is " +calculateAge(2016)+" years old!";
    }
    public static void display(Person p){
        System.out.println(p);
        p.setFirstName("Vasile");
        System.out.println("inside display "+p);
    }

    public static void main(String args[])
    {
        Person me = new Person ("Mihaela", "Dinsoreanu", 1970, true);
        display(me);
        System.out.println("outside display "+me);
    }
}
```

### Output - CMSC (run) ×



run:  
The person Mihaela Dinsoreanu is 46 years old!  
inside display The person Vasile Dinsoreanu is 46 years old!  
outside display The person Vasile Dinsoreanu is 46 years old!  
BUILD SUCCESSFUL (total time: 0 seconds)

# WHAT IS INHERITANCE?

The way to reuse CLASSES to create more specific classes

Represents the IS-A relationship

The attributes and methods of the superclass are inherited in the subclass

FINAL classes cannot be subclassed!

Examples:

- Student IS-A Person
- Dog IS-A Animal
- Truck IS-A Vehicle
- Square IS-A Rectangle



# OVERRIDING METHODS

Change the inherited code of the method

The method signature DOESN'T change!

Can all methods be overridden?

- FINAL methods cannot!

```
class Student extends Person {  
...  
    public String toString() {  
        return "This is student "+ firstName + " " +  
lastName;  
    }  
}
```

# WHAT IS COMPOSITION?

The way to reuse OBJECTS in order to create more complex objects.

Represents HAS-A relationship

Examples:

- House HAS-A Door
- Vehicle HAS-A Engine
- Person HAS-A Heart

```
class Person {  
    private Heart heart;  
    private String firstName, lastName;  
    ...  
    Person (Heart h, String fN, ...)  
}
```

```
class Heart {  
    private double pulse;  
    private double weight;  
    ..  
}
```

# INHERITING A SUPERCLASS

## What is inherited?

- Attributes
- Methods
- Constructors



## What can you do in a subclass?

- use the inherited fields and methods directly
- declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- declare new fields in the subclass that are not in the superclass.
- write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it. (NOT FINAL!!!)

# INHERITING A SUPERCLASS [2]

- write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- declare new methods in the subclass that are not in the superclass.
- write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword *super*.

## Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

```
class Base {  
    private int i;  
    public int getI() {return i;}  
    public void setI(int j) {i = j;}  
}  
  
public class Test extends Base {  
    public static void main(String args[]) {  
        Test t = new Test();  
        t.setI(5);  
        System.out.println(i);  
        System.out.println(t.getI());  
    } }
```

# POLYMORPHISM

The possibility to consider an instance as having different types. NOT ANY TYPE!!!!

```
String display(Person p) {  
    System.out.println(p);  
}
```

```
Person me, you;  
me = new Person();  
you = new Student();
```

`display(me); => “me@32342323”`

`display(you); => “This is student .....`

# CLASS DESIGN PRINCIPLES

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion



# SINGLE RESPONSIBILITY

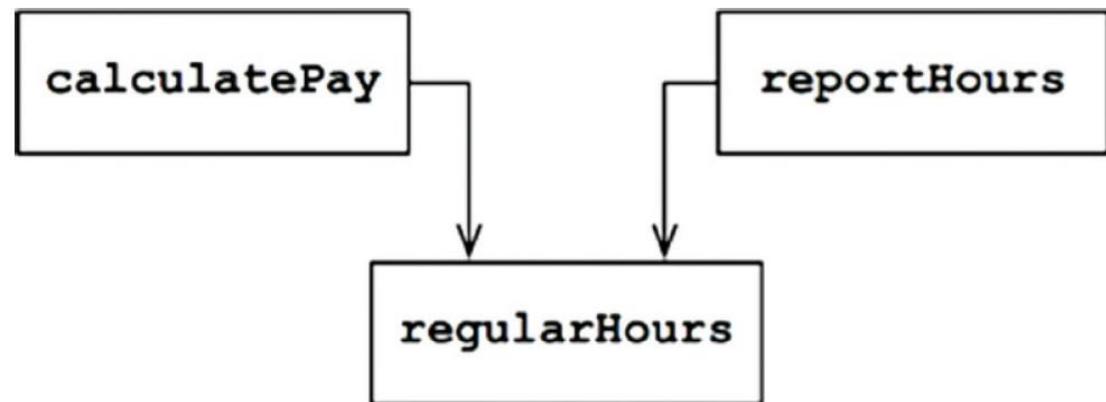
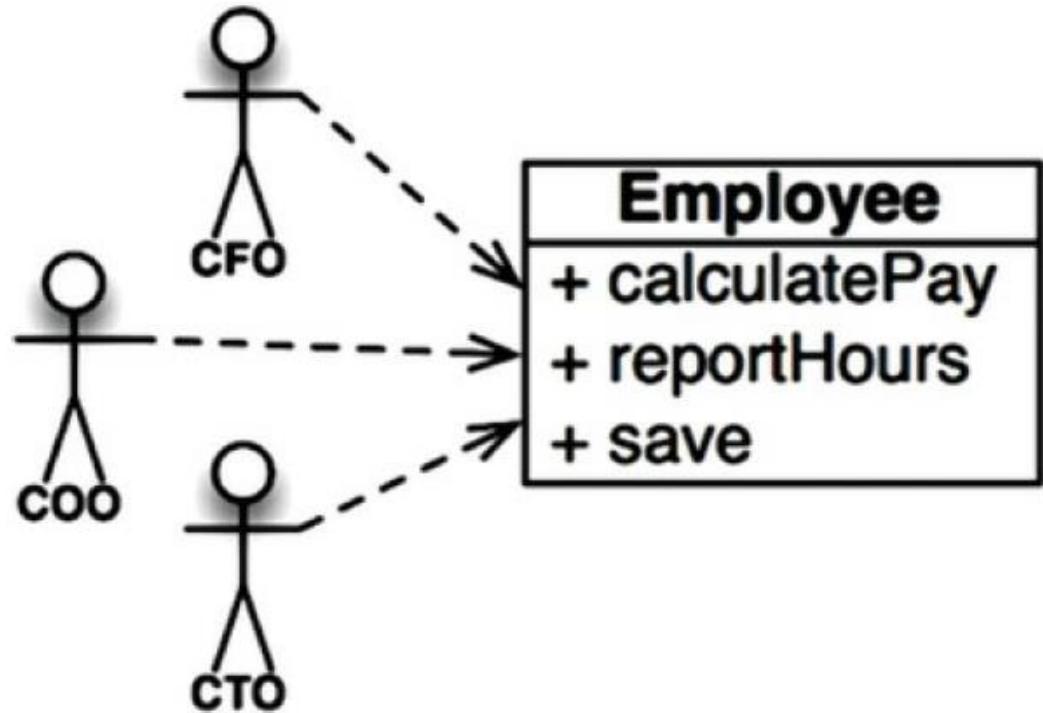
A module should have one, and only one, **reason to change**.

A module should be **responsible to one**, and only one, **user or stakeholder**.

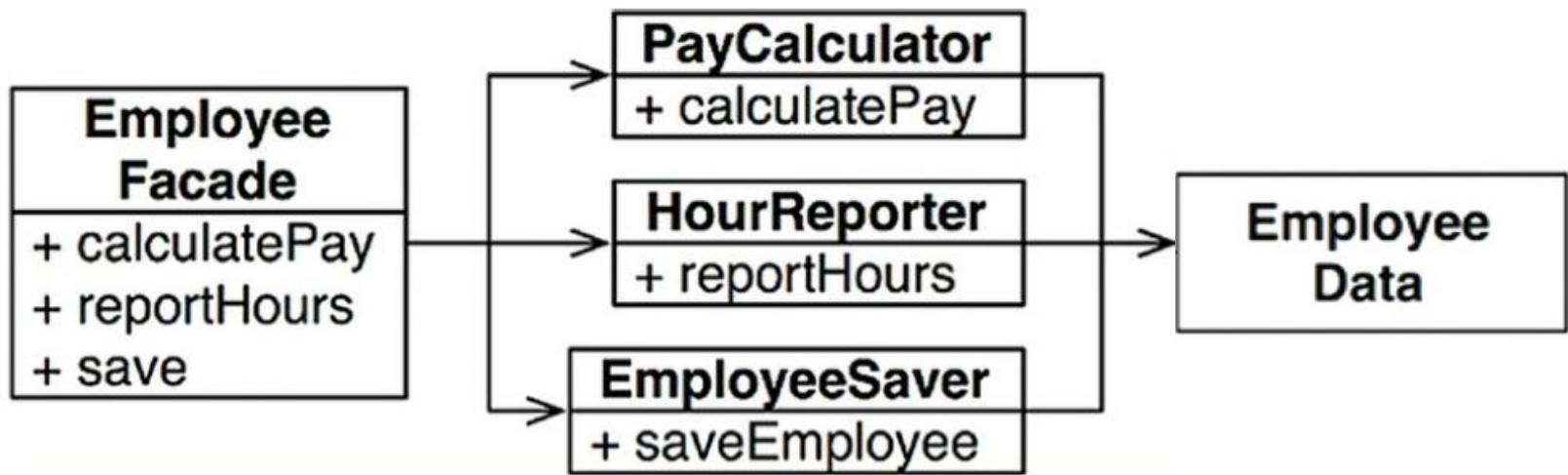
A module should be **responsible to one**, and only one, **actor**.



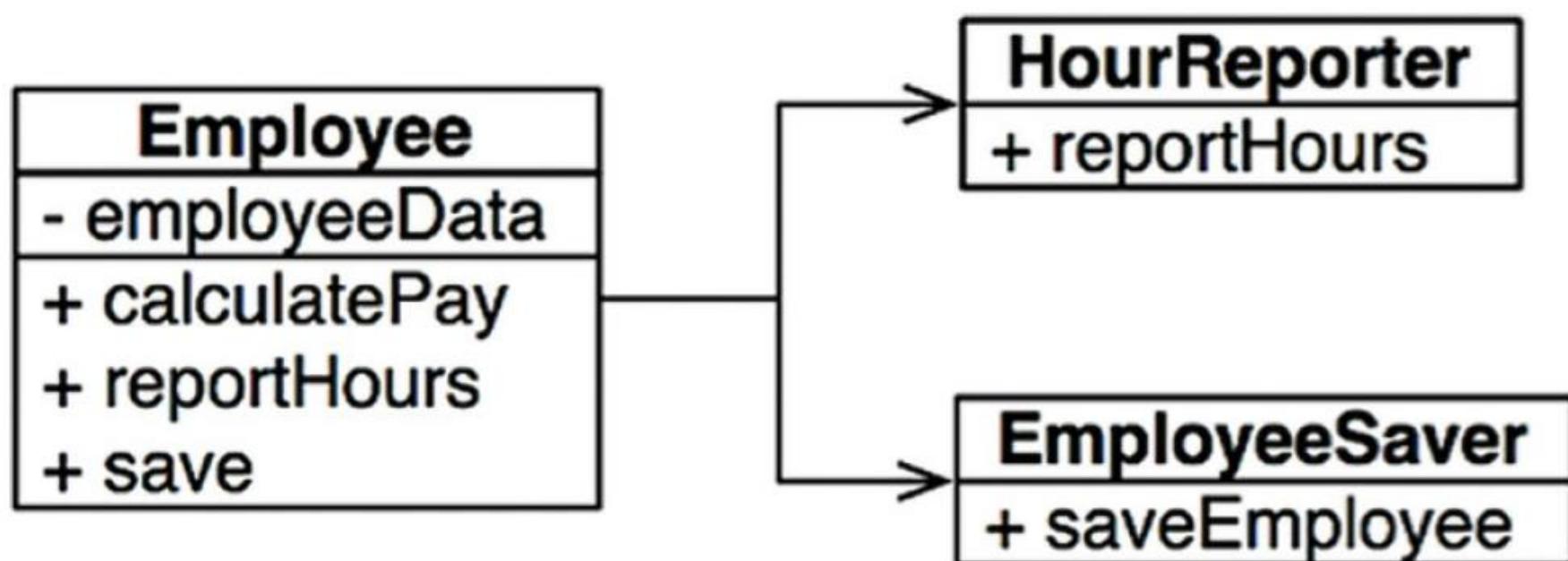
# EXAMPLE



# SOLUTION(S)



OR...



# OPEN-CLOSED PRINCIPLE (OCP)

A software artifact should be open for extension but closed for modification.

EXTENSION??

- by inheritance?
- by composition?



# EXAMPLE

What if a new type of report is needed?

```
class Report {  
    enum Type {  
        ORDERS_PER_DAY, CONVERSION_RATES  
    }  
  
    Type type;  
  
    String generate() {  
        ...  
        switch (type) {  
            case ORDERS_PER_DAY:  
                // do stuff  
                break;  
            case CONVERSION_RATES:  
                // do stuff  
                break;  
        }  
        ...  
    }  
}
```

# SOLUTION

Generate abstraction

```
interface Report {  
    String generate();  
}
```

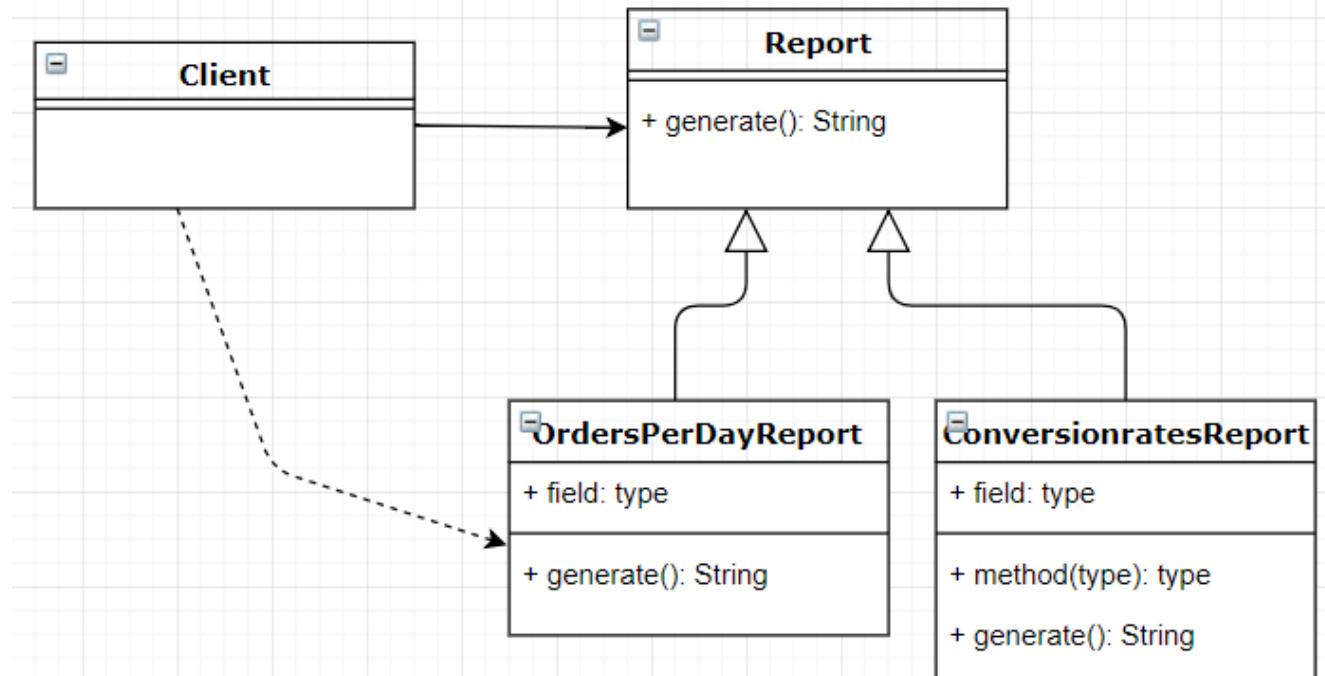
Implement the abstraction

```
class OrdersPerDayReport implements Report {  
    public String generate() {  
        // do stuff  
    }  
}  
  
class ConversionRatesReport implements Report {  
    public String generate() {  
        // do stuff  
    }  
}
```

# TECHNIQUE

Dynamic polymorphism

Dependency management



Static polymorphism

- Templates, generics

# WHAT IF...

- ... another column has to be added into the report?
- ... the report format should be different if displayed in a web interface or printed?
- ... the same report should be more/less detailed depending on the user?

- ⇒ The challenge is to decide what to close!
- ⇒ Strategic closure

# STRATEGIC CLOSURE

Use abstraction to gain explicit closure

- provide class methods which can be dynamically invoked to determine *general* policy decisions
- design using abstract ancestor classes

Use "Data-Driven" approach to achieve closure

- place volatile policy decisions in a separate location (e.g. a configuration file or a separate object)
- minimizes future change locations

# LISKOV SUBSTITUTION PRINCIPLE (LSP)

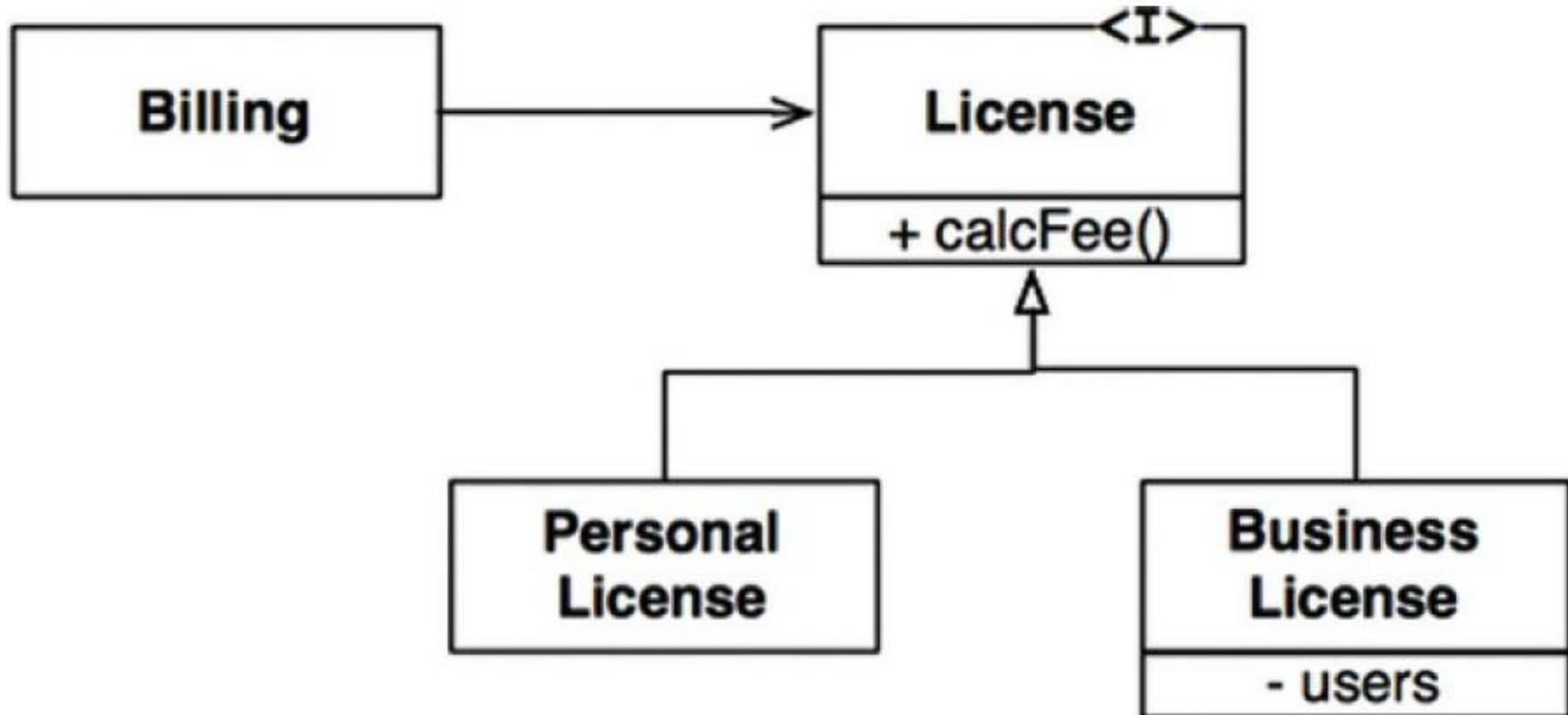
*“What is wanted here is something like the following substitution property:*

*If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.”*

[Barbara Liskov, 1988]

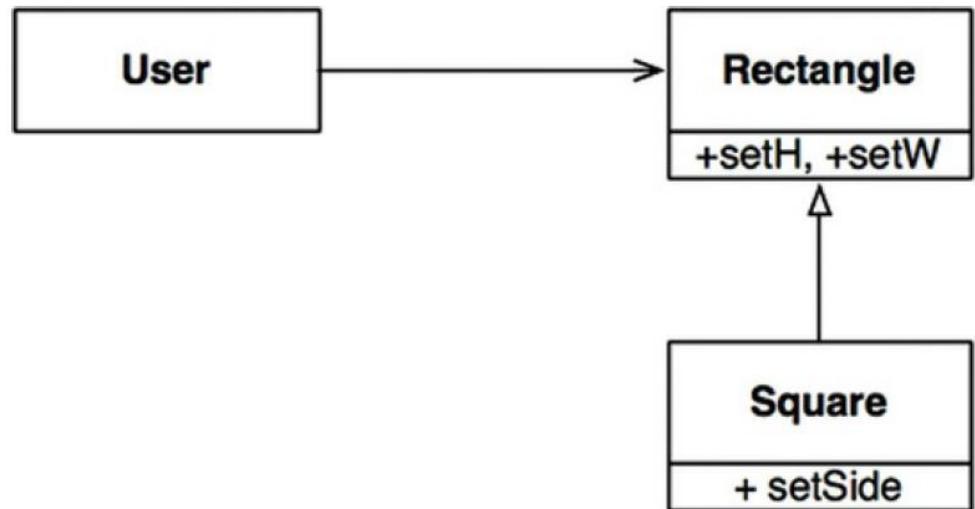


# EXAMPLE



# LSP VIOLATION

```
class Rectangle
{
    private:
        double width;
        double height;
    public:
        void setW(double w) ...
        void setH (double h) ...
}
```



class Square inherits Rectangle ?

# IS-A RELATIONSHIP REFERS TO BEHAVIOR

Override setW () and setH ()

=> Duplicated code

Problem! Static binding (C++)

```
void g(Rectangle& r)
{
    r.setW(4);
    r.setH(5);
}
```

# PROBLEM CONTINUED

## Dynamic binding (Java)

```
class Rectangle
{
    private double width;
    private double height;

    public void setW (double w) ...
    public void setH (double h) ...
}

void g(Rectangle r)
{
    r.setW(4);
    r.setH(5);
    assert(r.getW()*r.getH()== 20);
}
```

# DESIGN BY CONTRACT [BERTRAND MEYER]

Basic notation: ( $P$ ,  $Q$ : assertions, i.e. properties of the state of the computation.  $A$ : instructions).

$$\{P\} A \{Q\}$$

Total correctness: Any execution of  $A$  started in a state satisfying  $P$  will terminate in a state satisfying  $Q$ .

Design by contract

- 1. Preconditions  $P$  of the derived class method are no stronger than the base class method.**
- 2. Postconditions  $Q$  of the derived class method are no weaker than the base class method.**

# LSP HEURISTICS

**It is illegal for a derived class, to override a base-class method with a NOP method**

NOP = a method that does nothing

**Solution 1: Inverse Inheritance Relation**

- if the initial base-class has only additional behavior

**Solution 2: Extract Common Base-Class**

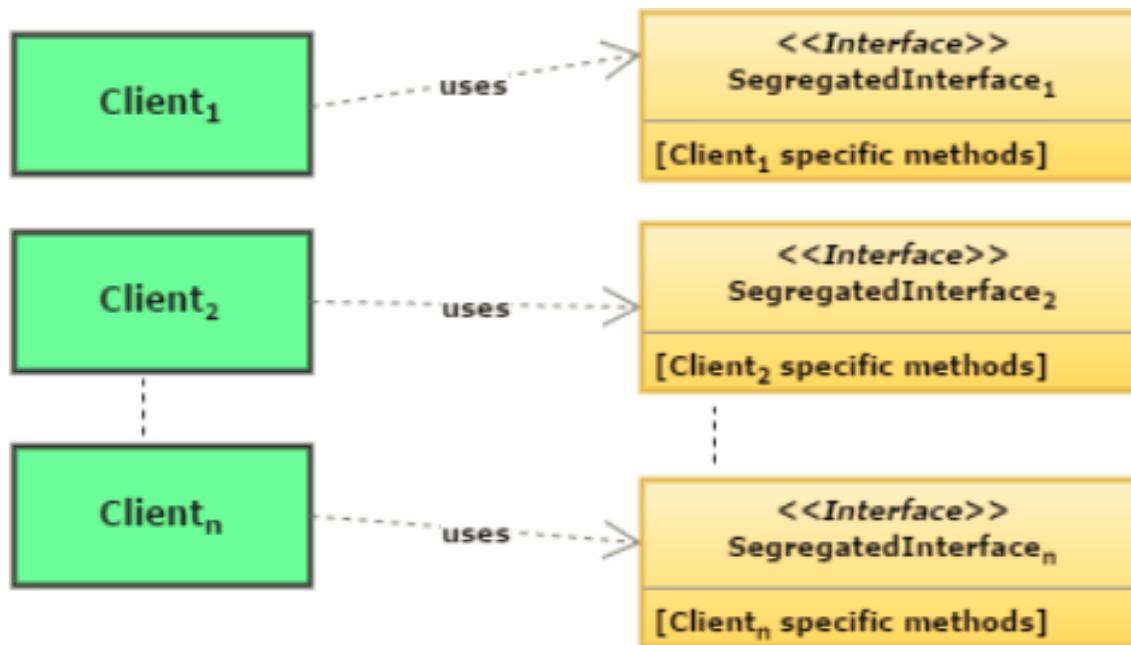
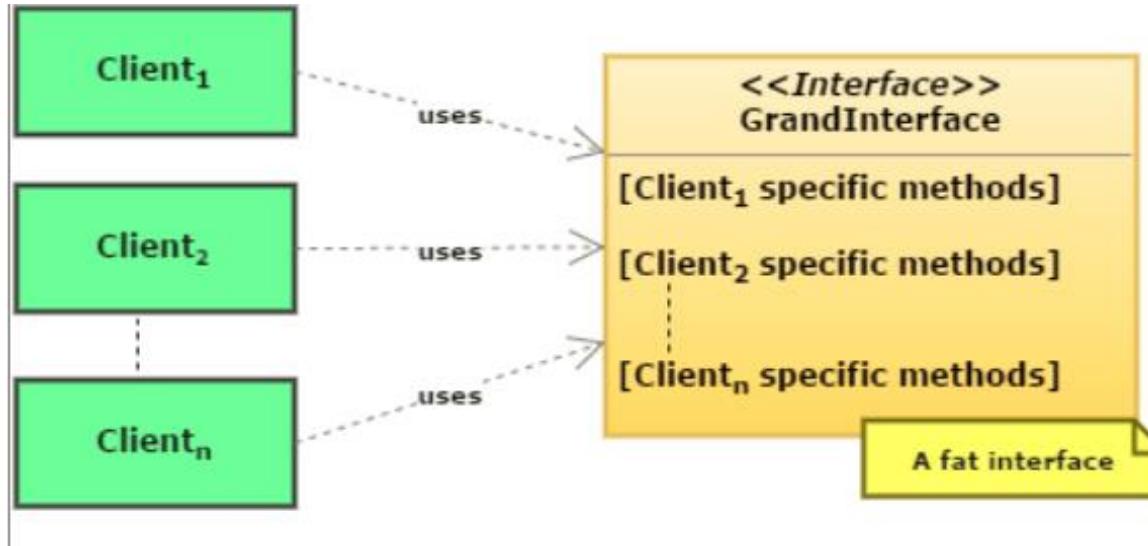
- if both initial and derived classes have different behaviors

# INTERFACE SEGREGATION PRINCIPLE (ISP)

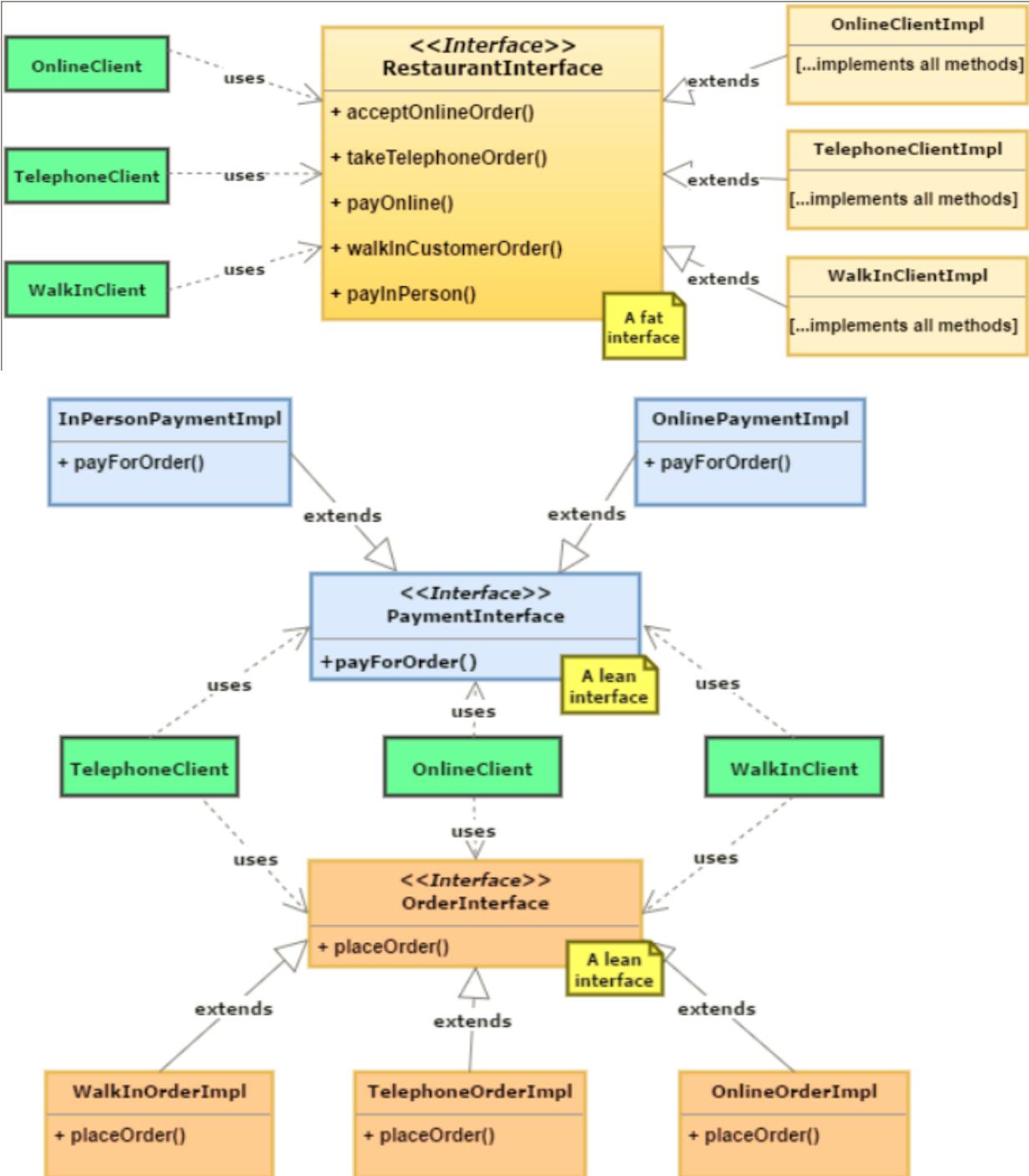
Clients should not be forced to depend upon interfaces that they don't use.



# ISP

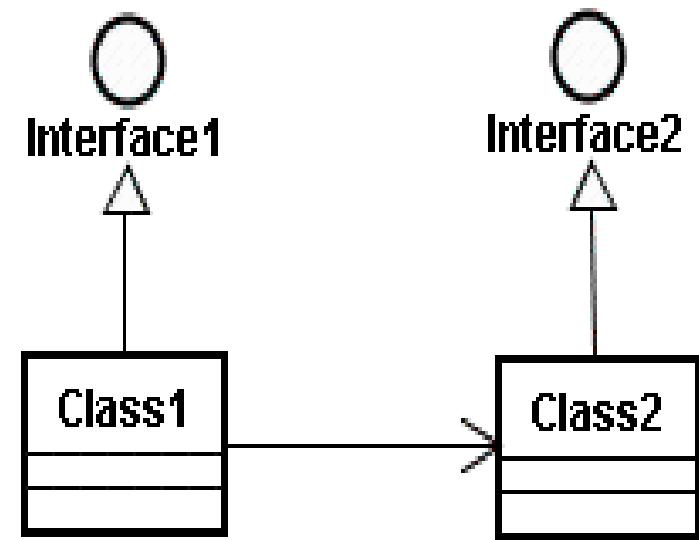
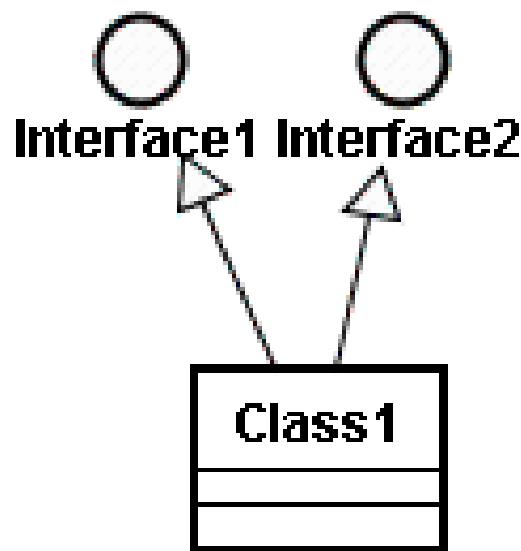


# ISP EXAMPLE



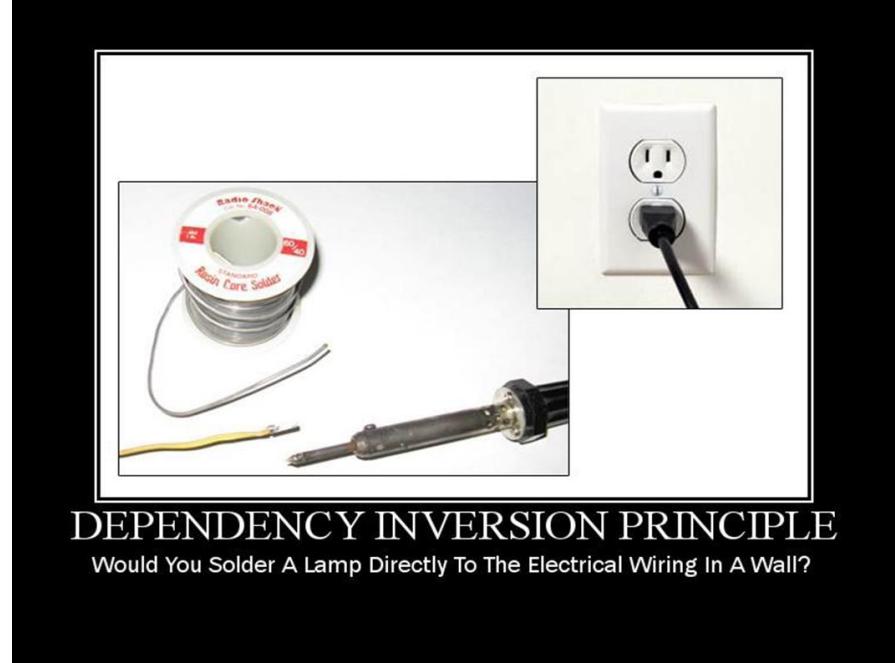
# ISP EXAMPLE

Separation thru Multiple Inheritance vs. separation thru delegation



# DEPENDENCY INVERSION PRINCIPLE (DIP)

- I.High-level modules should **not** depend on low-level modules.  
Both should depend on abstractions.
- II.Abstractions should **not** depend on details. Details should depend on abstractions.



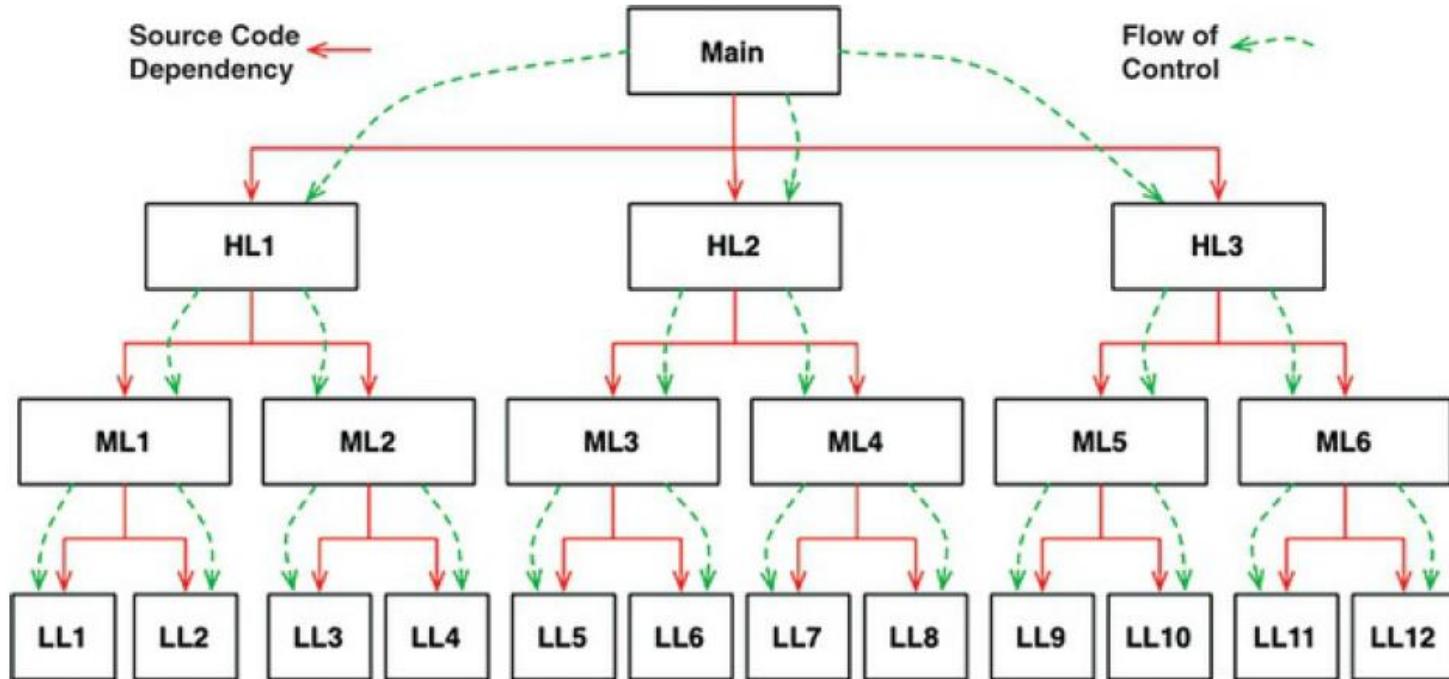
# DEPENDENCY INVERSION MOTIVATION

Traditional calling tree Main

#include (C++)

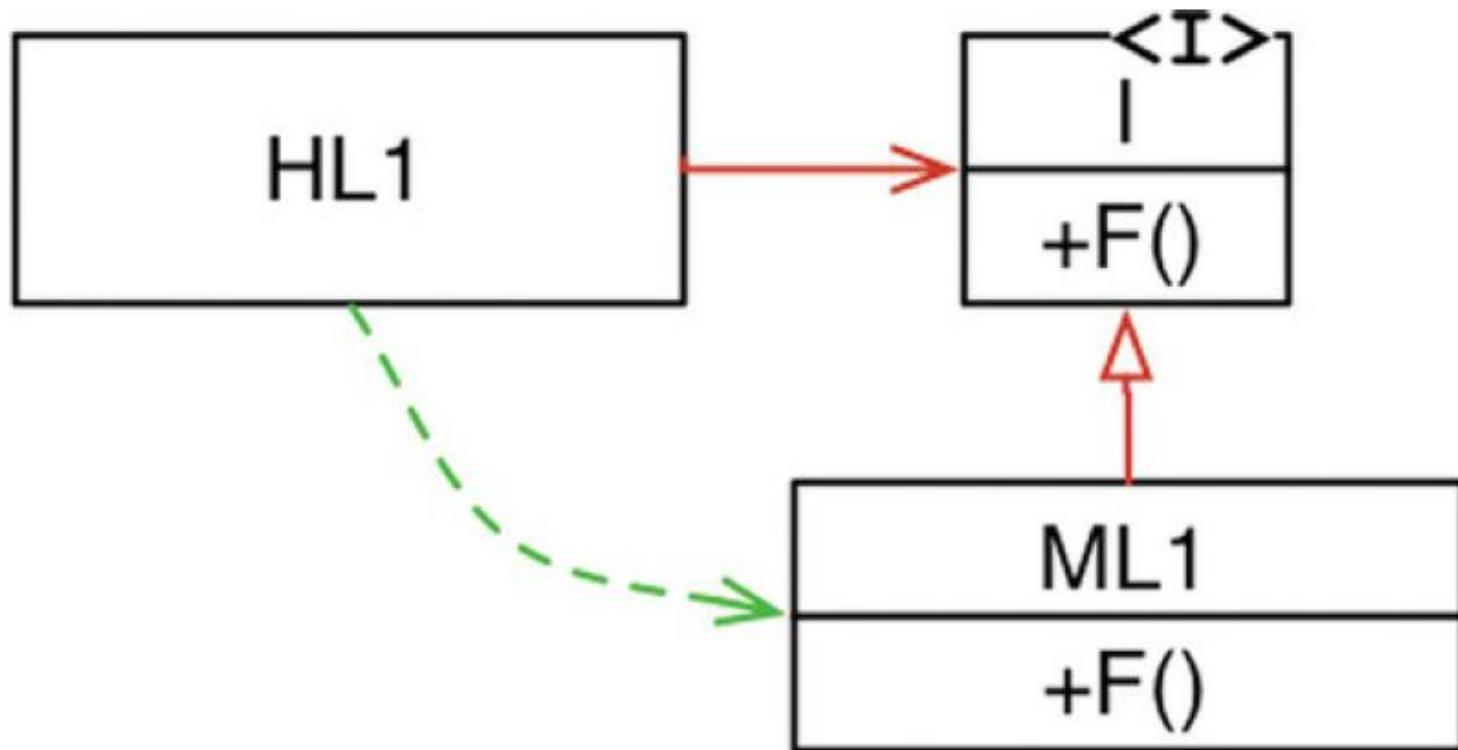
import (Java)

using (C#)

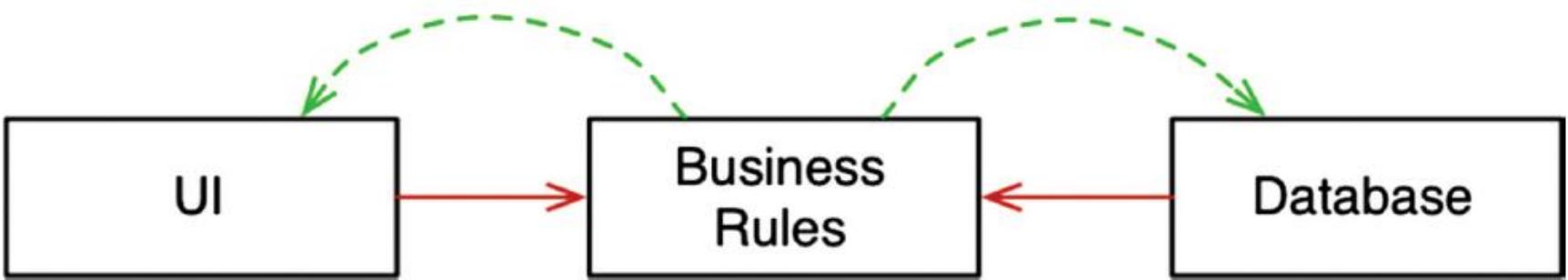


# DEPENDENCY INVERSION

The power of polymorphism!

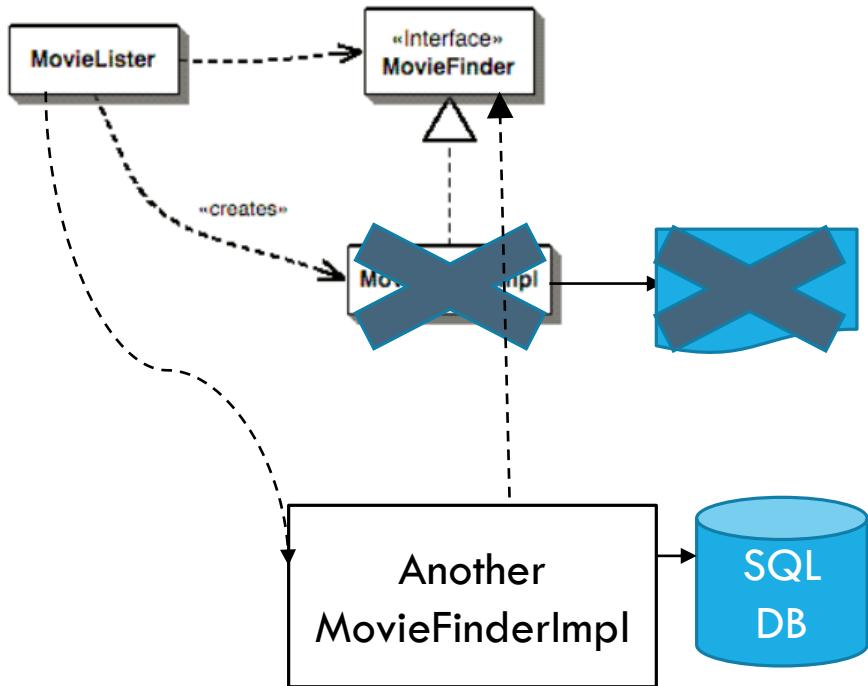


# DEPENDENCY INVERSION IN LAYERS



- The business rules, the UI, and the database can be compiled into three separate components or deployment units (e.g., jar files, DLLs, etc.)
- The component containing the business rules will not depend on the components containing the UI and database => The business rules can be *deployed independently* of the UI and the database.
- Changes to the UI or the database need not have any effect on the business rules.
- If the modules in your system can be deployed independently, then they can be developed *independently* by different teams.

# DEPENDENCY INJECTION



```
public class MovieLister {  
    private MovieFinder finder;  
  
    public MovieLister() {  
        this.finder = new  
MovieFinderImpl("movies.txt")  
    } ... }
```

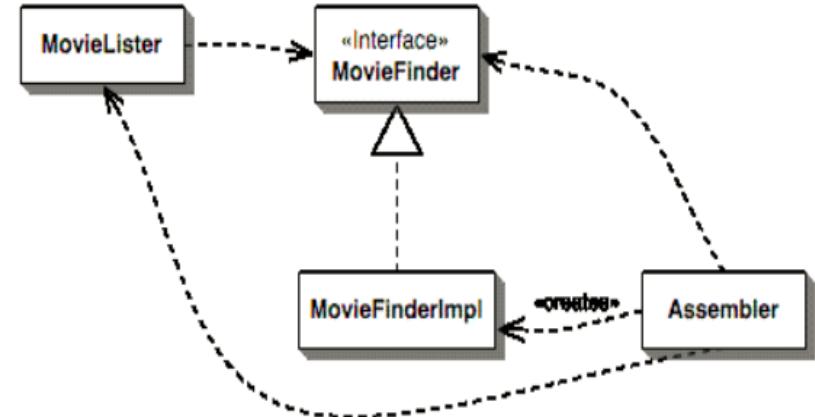
# TYPES OF DEPENDENCY INJECTION

## Constructor

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder; }
```

```
class TextMovieFinder implements MovieFinder  
    public TextMovieFinder(String filename) {  
        this.filename = filename; }
```

```
//configuration code in a different class  
private MutablePicoContainer  
configureContainer() {  
    MutablePicoContainer pico = new  
DefaultPicoContainer();  
    Parameter[] finderParams = {new  
ConstantParameter("movies.txt")};  
  
    pico.registerComponentImplementation(MovieFinder.class,  
TextMovieFinder.class,  
finderParams);  
  
    pico.registerComponentImplementation(MovieLister.class);  
    return pico;  
}
```



```
//test the code
```

```
MutablePicoContainer pico =  
configureContainer();  
  
MovieLister lister =  
(MovieLister)  
pico.getComponentInstance(MovieLister.class);
```

# SETTER DI WITH SPRING

```
public class MovieLister {  
    private MovieFinder finder;  
    public void setFinder(MovieFinder finder) {  
        this.finder = finder; } }  
  
class TextMovieFinder...  
...  
    public void setFilename(String filename) {  
        this.filename = filename;  
    }  
  
//test  
public void testWithSpring() throws Exception  
{  
    ApplicationContext ctx = new  
FileSystemXmlApplicationContext("spring.xml");  
    MovieLister lister = (MovieLister)  
ctx.getBean("MovieLister");  
}
```

```
//configuration  
<beans>  
    <bean id="MovieLister"  
class="spring.MovieLister">  
        <property  
name="finder">  
            <ref  
local="MovieFinder"/>  
        </property>  
    </bean>  
    <bean id="MovieFinder"  
class="spring.TextMovieFinder">  
        <property  
name="filename">  
            <value>movies1.txt</value>  
        </property>  
    </bean>  
</beans>
```

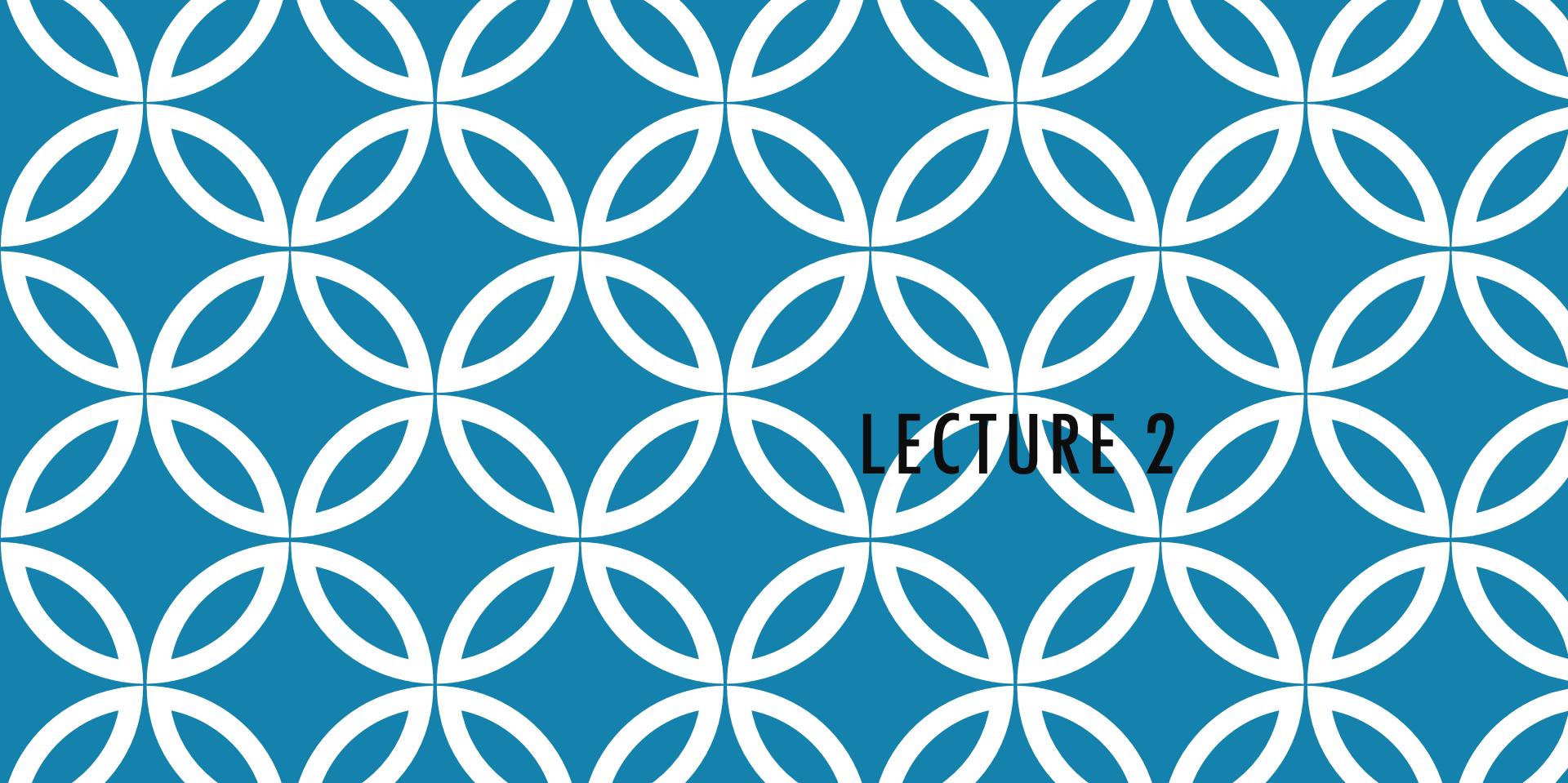
# WRAP-UP

Our objective is to develop GOOD software architectures

EACH PROBLEM HAS SEVERAL SOLUTIONS!

- Design
- Technology
- Code
- Deployment

Basic Design Principles have to be considered!



# LECTURE 2

---

Design principles

# CONTENT

General Responsibility Assignment Principles  
(GRASP)

Package Design

- Cohesion Principles
- Coupling Principles

# REFERENCES

- Robert Martin  
<http://butunclebob.com/Articles.UncleBob.PrinciplesOfOod>
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley
- Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Ed, Addison Wesley, 2004 – Chapters 17, 18.

## Courses

- B. Meyer (ETH Zurich)
- R. Marinescu (Univ. Timisoara)

# LAST TIME

OOP concepts +

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

# CHALLENGE

Overloading and polymorphism

Compiler error?

Output?

```
1    class Base{
2        @Override
3        public String toString() { return "This is base";}
4
5
6    }
7
8    class Sub extends Base{
9        @Override
10       public String toString() { return "This is subclass";}
11   }
12
13
14    class Host{
15        public String print(Base b)
16        {
17            return b.toString() + " din metoda cu base";
18        }
19        public String print(Sub b)
20        {
21            return b.toString() + " din metoda cu sub";
22        }
23
24
25    public class Polytest {
26        public static void main(String[] args){
27            Base abase = new Base();
28            Sub asub = new Sub();
29            Host h = new Host();
30            System.out.println(h.print(abase));
31            System.out.println(h.print(asub));
32        }
33    }
34
35
36
37
38
39
40
41
42 }
```

Output - CMSC (run) x



```
run:
This is base din metoda cu base
This is subclass din metoda cu sub
```

# CONTINUED

Delete one method

Error?

Output?

```
0   class Base{
13    @Override
14    public String toString() { return "This is base";}
15
16  }
17
18  class Sub extends Base{
19    @Override
20    public String toString() { return "This is subclass";}
21  }
22
23  class Host{
24    public String print(Base b)
25    {
26      return b.toString() + " din metoda cu base";
27    }
28    /*public String print(Sub b)
29    {
30      return b.toString() + " din metoda cu sub";
31    }*/
32  }
33
34  public class Polytest {
35    public static void main(String[] args){
36      Base abase = new Base();
37      Sub asub = new Sub();
38      Host h = new Host();
39      System.out.println(h.print(abase));
40      System.out.println(h.print(asub));
41    }
42  }
```

Output - CMSC (run) x

```
run:
This is base din metoda cu base
This is subclass din metoda cu base
```

# HOW TO FAIL WITH SOLID

**Single Responsibility Principle:** Each class has one (*very small*) specific responsibility. That can lead to thousands of classes.

**Open-Closed Principle:** Use *inheritance* to add every new feature.

**Interface Segregation Principle:** Everything is single-inheritance based

**Dependency Inversion Principle:** Everything is abstracted (data-driven to the extreme).

=> Every single class is ultimately inherited out of the same family tree, sometimes *hundreds* of layers deep.

# GRASP

General Responsibility Assignment Software Patterns

OO system = objects sending messages to other objects to complete operations.

Issues:

- **Responsibilities** assigned to objects
- **Interaction** ways between objects

# RESPONSIBILITIES

## **Knowing** responsibilities:

- knowing about private encapsulated data;
- knowing about related objects;
- knowing about things it can derive or calculate;

## **Doing** responsibilities:

- doing something itself;
- initiating action in other objects;
- controlling and coordinating activities in other objects;

A responsibility is not the same as a method, but methods are implemented to fulfil responsibilities.

# GRASP: GENERAL PRINCIPLES IN ASSIGNING RESPONSIBILITIES

From Craig Larman's 9 principles:

- **Expert**
- **Creator**
- **Controller**
- **Low Coupling**
- **High Cohesion**
- **Polymorphism**
- **Pure Fabrication**
- **Indirection**
- **Don't Talk to Strangers (Law of Demeter)**

# CASE STUDY – POS SYSTEM

A Point-Of-Sale (POS) system is an application used (in part) to record sales and handle payments; it is typically used in a retail store.

It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control.

# MODELS

Challenge: How do we get from the narrative requirements to an implementable model?

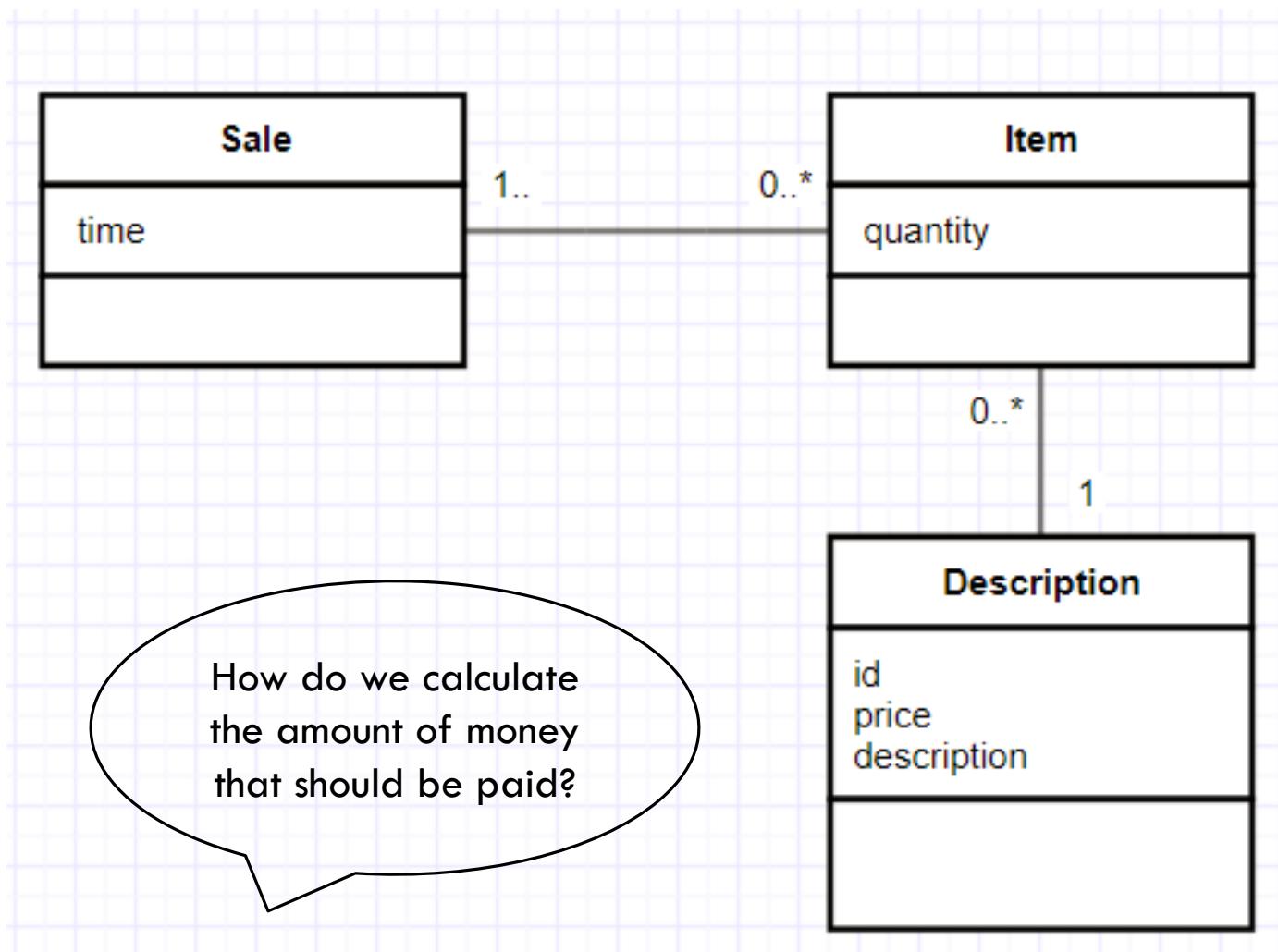
Domain Model

- Sale
- Item
- Payment
- Product
- Register

Design Model

- ?

# DESIGN MODEL



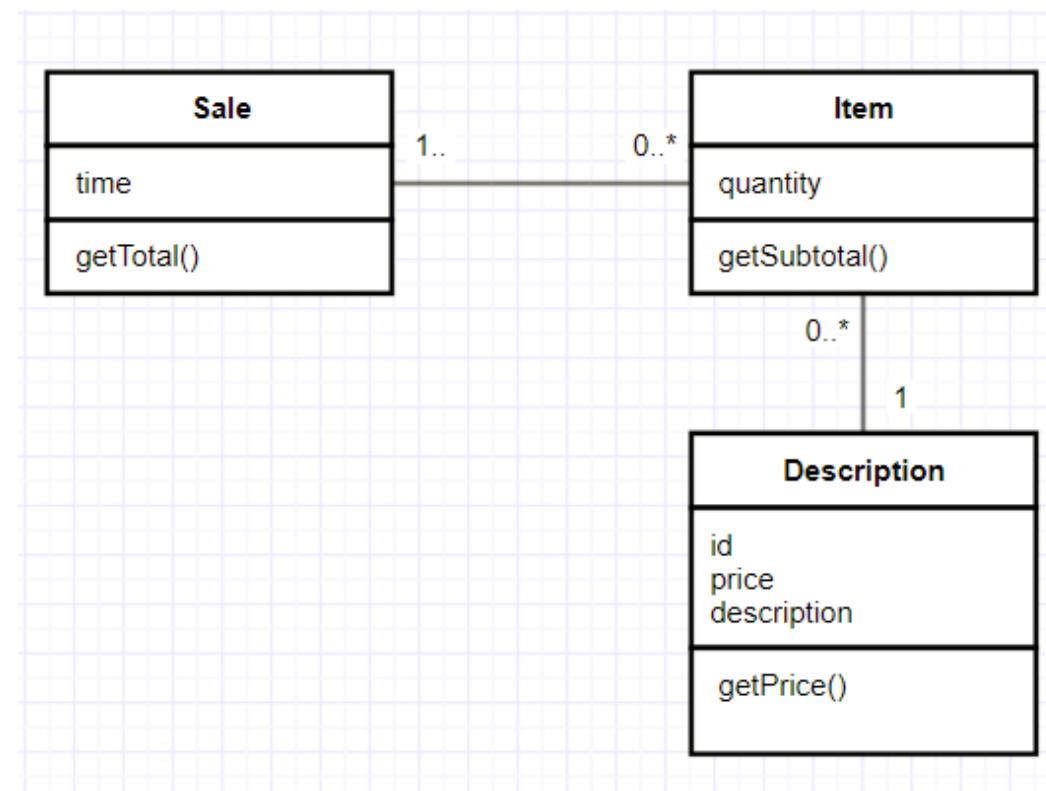
# INFORMATION EXPERT

**Problem:** What is the most basic principle by which responsibilities are assigned in object-oriented design?

**Solution:** Assign a responsibility to the information expert - the class that **has the necessary information** to fulfil the responsibility.

# INFORMATION EXPERT EXAMPLE

In the POS application, what class has the information needed to calculate the total amount of money?



SALE

# CONCLUSION

## Discussion

- Expert - most used principle in the assignment of responsibilities
- Information is spread across different objects => they need to interact

## Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks => supports low coupling
- Behavior is distributed across the classes that have the required information => more cohesive "lightweight" class definitions

# CREATOR

**Problem:** Who should be responsible for creating a new instance of some class?

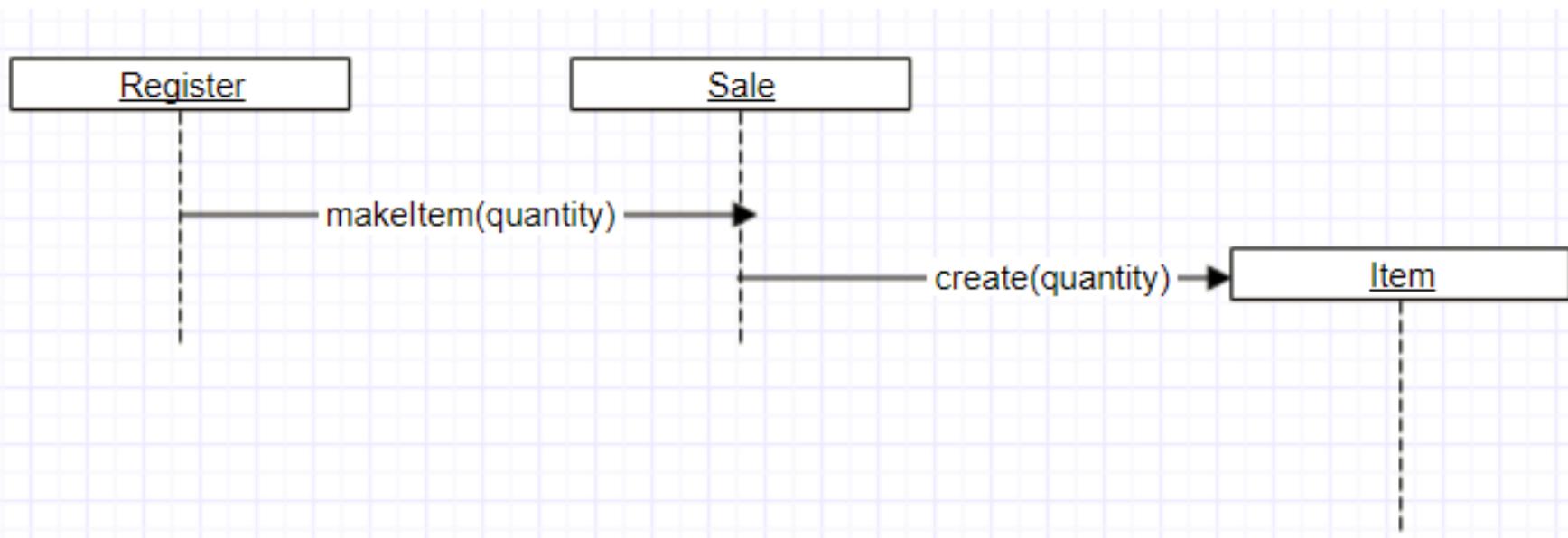
**Solution:** Assign class B the responsibility to create an instance of class A if one of the following is true:

- B contains A objects
- B closely uses A objects
- B has the initialising data that will be passed to A when it is created.

# CREATOR

**Example:** In the POS application who should be responsible for creating an Item instance?

SALE



# CONCLUSION

## Discussion:

- Creator guides assigning responsibilities related to the creation of objects.
- Sometimes a creator is the class that has the initialising data that will be used during creation.

For example, who should be responsible to create a Payment instance ?

## Benefits:

- Low Coupling is supported

## Issues:

- Complex creation procedures
- Solution ?

# CONTROLLER

**Problem:** Who should be responsible for handling a system event?

- A system event is a high level event generated by an external actor.
- A Controller is a non-user interface object responsible for handling a system event.

# CONTROLLER

**Solution:** Assign the responsibility for handling a system event to a class representing one of the following choices:

- Facade controller
  - handles all the events
  - represents the overall "system"
- Role controller
  - handles the events associated to the role
  - represents a person in the real-world
- Use-case controller
  - handles the events associated to a use-case
  - represents an artificial handler

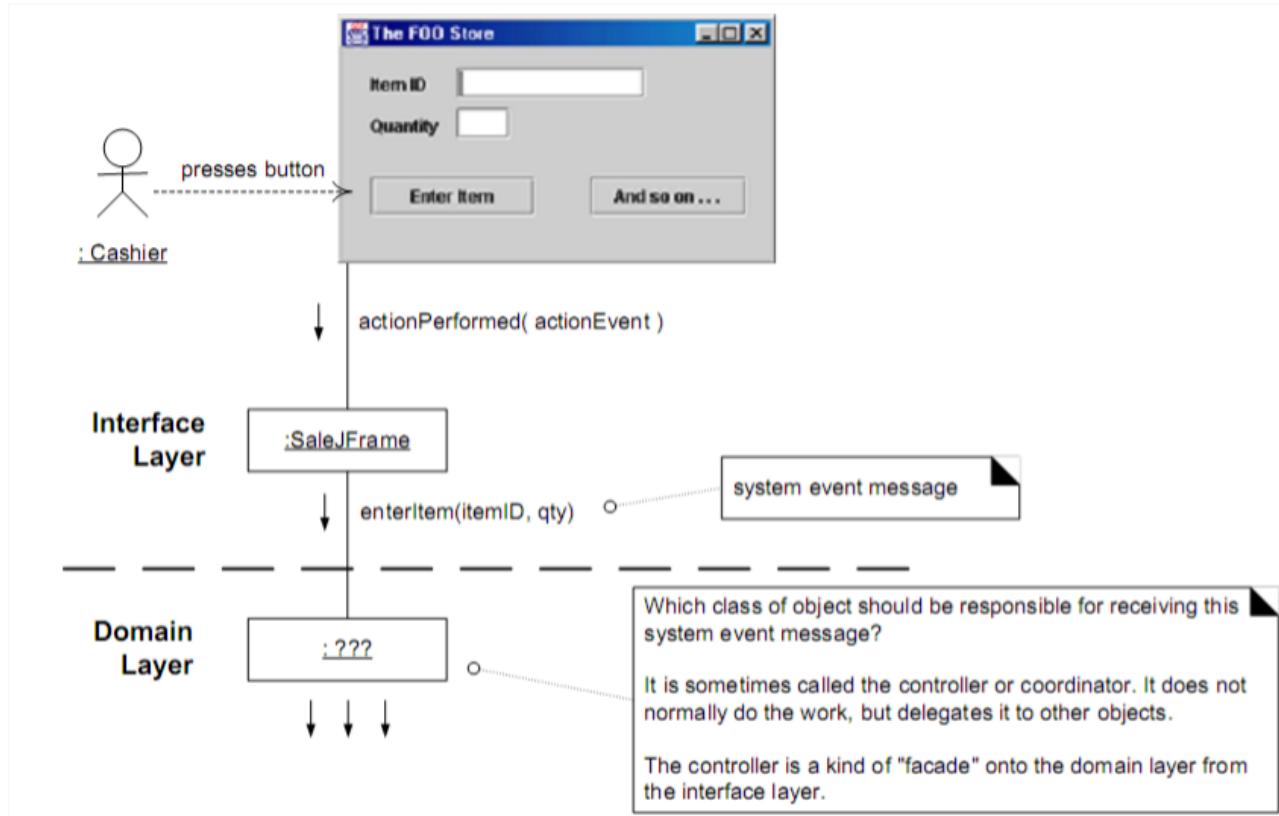
# CONTROLLER

**Example:** In the point of sale application the current system operations have been identified as:

`endSale()`

`enterItem()`

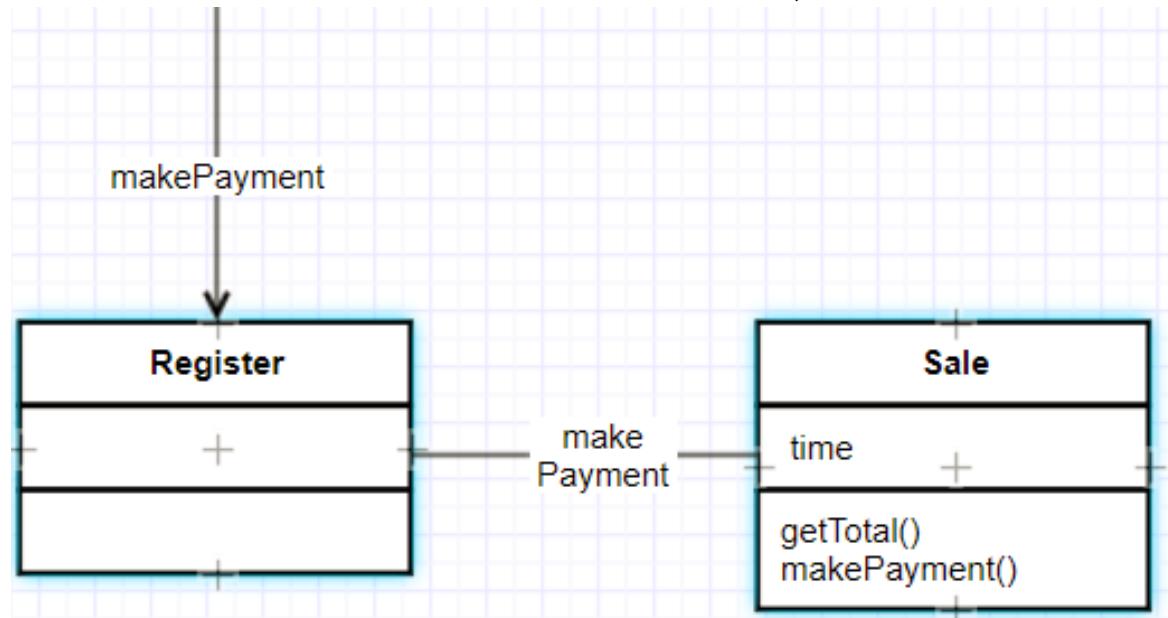
`makePayment()`



# SOLUTION

By the Controller pattern, here are some choices:

- Register, POSSystem: represents the overall "system" (Facade controller)
- ProcessSaleSession, ProcessSaleHandler: represents a handler of all system events of a use case scenario (Use-case controller)



# DISCUSSION

- Usually a controller delegates to other objects the work that needs to be done; it coordinates or controls the activity.
- Facade controllers are suitable when there are not "too many" system events
- A use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling
- Controller class is called bloated if
  - The class is overloaded with too many responsibilities.
    - Solution – Add more controllers
  - Controller class performs many tasks itself.
    - Solution – controller class has to delegate things to others.

# CONCLUSIONS

## Benefits:

Increased potential for reusable components:

- it ensures that business or domain processes are handled by the layer of domain objects rather than by the interface layer.
- the application is not bound to a particular interface.

Reason about the state of the use case:

- As all the system events belonging to a particular use case are assigned to a single class, it is easier to control the sequence of events that may be imposed by a use case (e.g. MakePayment cannot occur until EndSale has occurred).

# LOW COUPLING

How strongly are the objects connected to each other?

Coupling = object depending on other object.

When the independent element changes, it affects the dependent one.

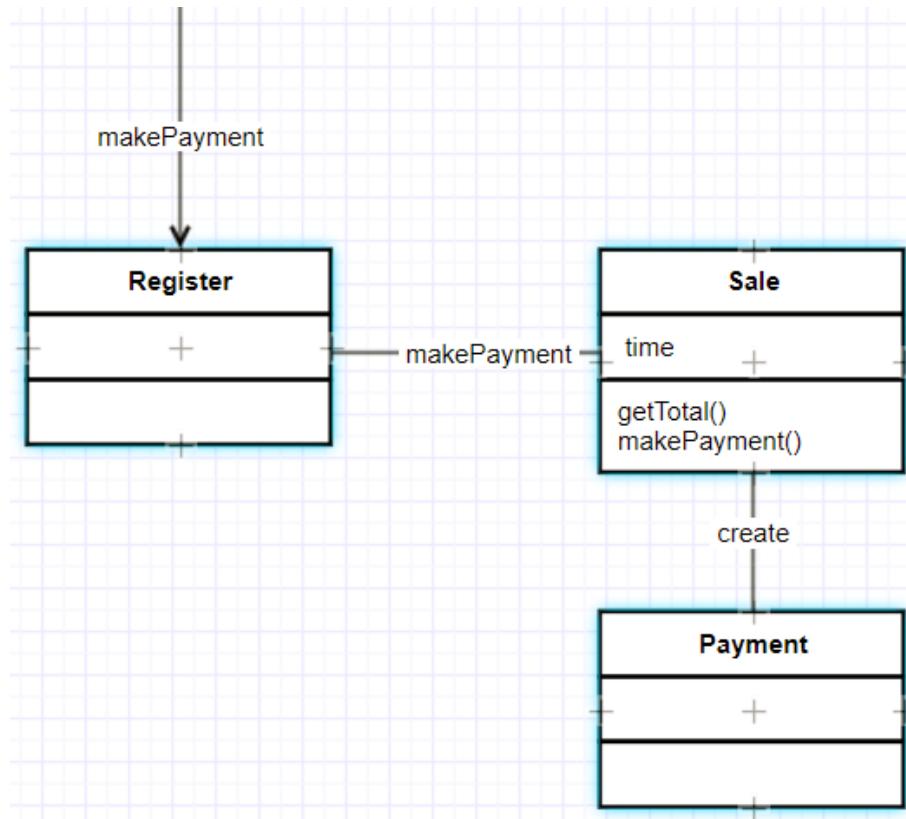
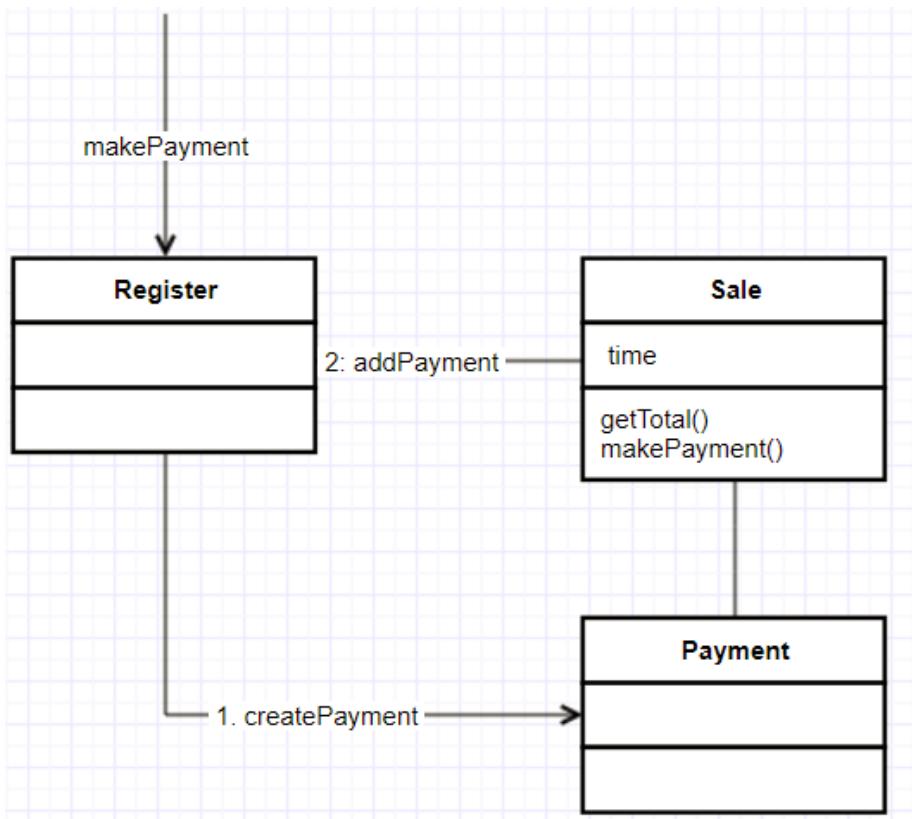
Prefer low coupling – assign responsibilities so that coupling remains low.

# COUPLING

TypeX depends on TypeY if

- TypeX has an **attribute that refers** to a TypeY instance, or TypeY itself.
- TypeX has a **method which references** an instance of TypeY, or TypeY itself, by any means. (Typically include a parameter or local variable of type TypeY, or the returned object is an instance of TypeY.)
- TypeX is a direct or indirect **subclass** of TypeY.
- TypeY is an interface, and TypeX **implements** that interface.

# WHICH IS BETTER COUPLED?



# MEASURING COUPLING

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }
}
```

Code Metrics Results

Hierarchy	Class Coupling
ConsoleApplication6	0
ConsoleApplication	0
Person	0

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }
}

class PersonStuff
{
    void DoSomething()
    {
        Person myPerson = new Person();
        myPerson.Age = 23;
        myPerson.Name = "Bubba";
        myPerson.Salary = 20.34;
    }
}
```

Code Metrics Results

Hierarchy	Class Coupling
ConsoleApplication6 (D)	1
ConsoleApplication6	1
Person	0
PersonStuff	1
DoSomething()	1
PersonStuff()	1

```
class PersonStuff
{
    Person myPerson = new Person();

    void DoSomething()
    {
        myPerson.Age = 23;
        myPerson.Name = "Bubba";
        myPerson.Salary = 20.34;
    }
}
```

Code Metrics Results

Hierarchy	Class Coupling
ConsoleApplication6 (D)	1
ConsoleApplication6	1
Person	0
PersonStuff	1
DoSomething()	1
PersonStuff()	1

# HIGH COHESION

How are the operations of any element functionally related?

Related responsibilities are placed into one manageable unit.

Correlated to SRP!

## Benefits

- Easily understandable and maintainable.
- Code reuse
- Low coupling

# COHESIVE?



makePayment

Register

2: addPayment

Sale

time

getTotal()  
makePayment()

1. createPayment

Payment

makePayment

Register

+

makePayment

Sale

time  
getTotal()  
makePayment()

create

Payment

+

+

# LACK OF COHESION (LCOM)

LCOM measures the dissimilarity of methods in a class by instance variable or attributes.

- *Functional cohesion* - the design unit (module) performs a single well-defined function or achieves a single goal.
- *Sequential cohesion* - the design unit performs more than one function, but these functions occur in an order prescribed by the specification, i.e. they are strongly related.
- *Communication cohesion* - a design unit performs multiple functions, but all are targeted on the same data.

# LCOM CONT'D

- *Procedural cohesion* - a design unit performs multiple functions that are procedurally related. The code in each module represents a single piece of functionality defining a control sequence of activities.
- *Temporal cohesion* - a design unit performs more than one function, and they are related only by the fact that they must occur within the same time span (ex. a design that combines all data initialization into one unit and performs all initialization at the same time even though it may be defined and utilized in other design units).
- *Logical cohesion* - a design unit that performs a series of similar functions (ex. the Java class `java.lang.Math`)

# LCOM4

LCOM4 measures the number of "*connected components*" in a class.

A connected component is a set of related methods (and class-level variables).

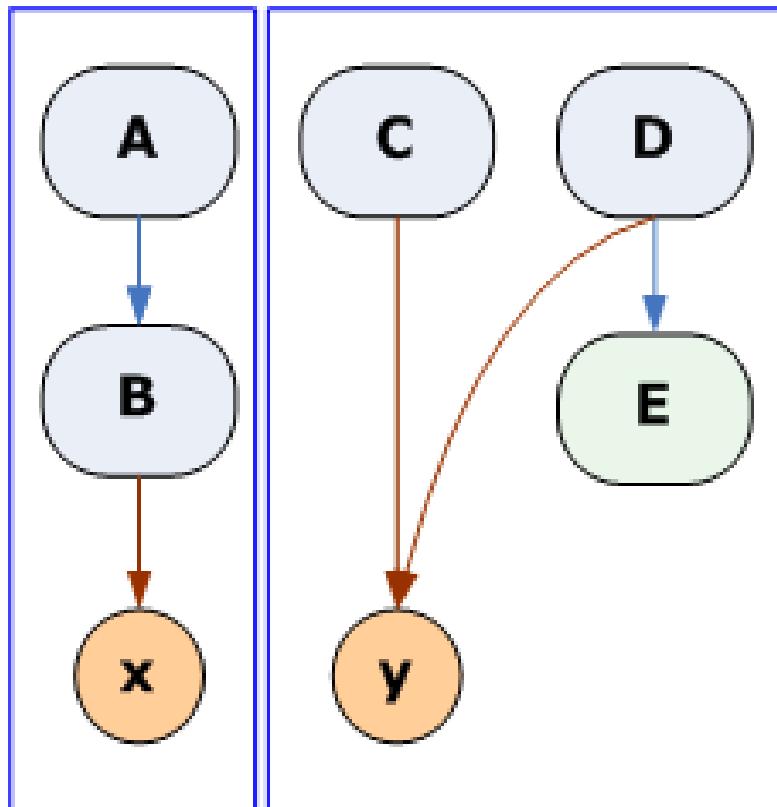
Methods **a** and **b** are related if:

- they both access the same class-level variable, or
- **a** calls **b**, or **b** calls **a**.

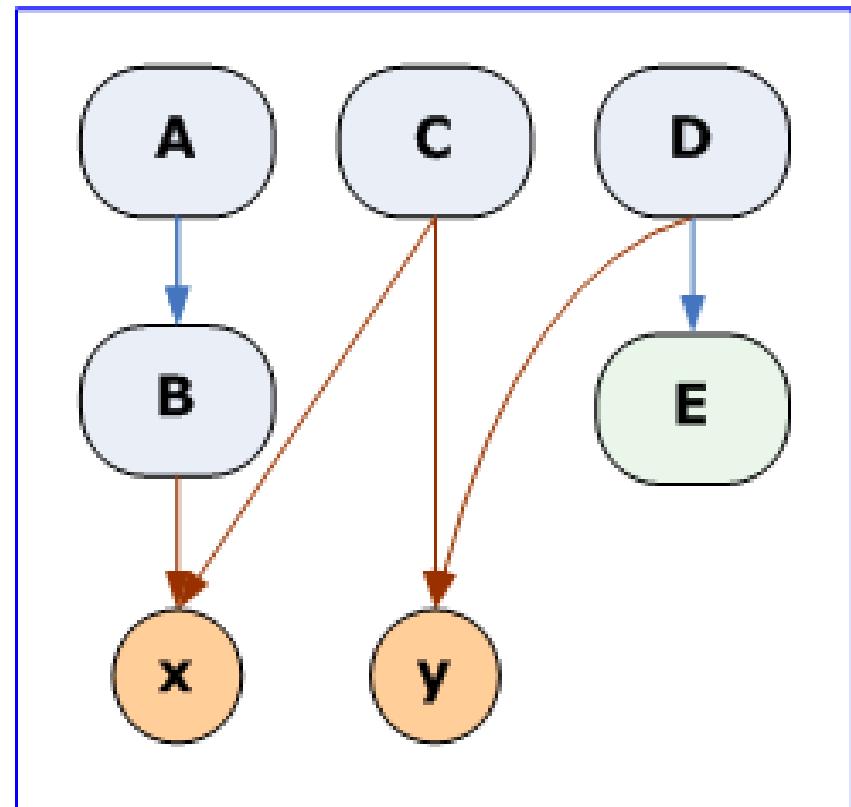
**There should be only one connected component in each class.**

If there are 2 or more components, the class should be split into so many smaller classes.

# LCOM4



$\text{LCOM4} = 2$



$\text{LCOM4} = 1$

# LAW OF DEMETER

## Weak Form

Inside of a method M of a class C, data can be accessed and messages can be sent to only the following objects:

- **this and super**
- **data members (attributes) of class C**
- **parameters** of the method M
- **objects created** within M
  - by calling directly a constructor
  - by calling a method that creates the object
- **global variables**

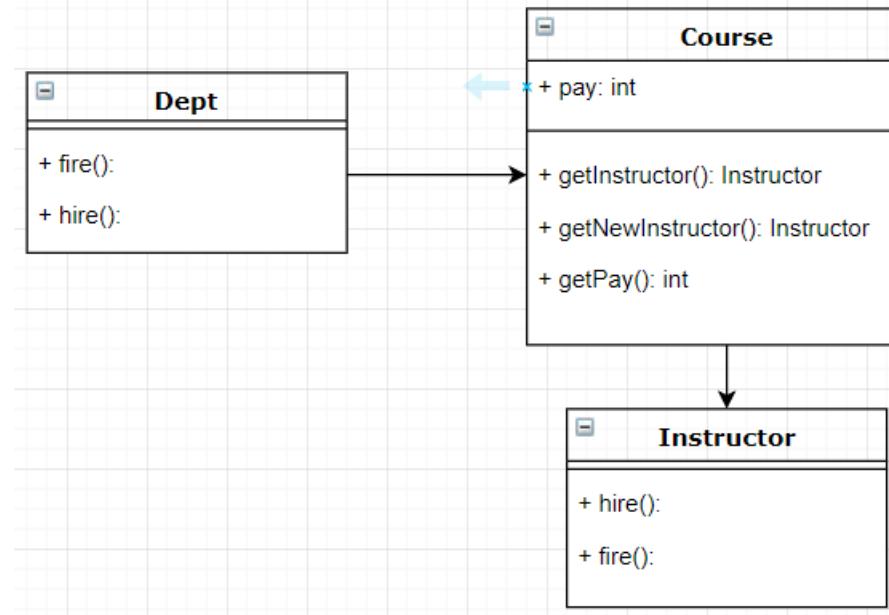
## Strong Form

In addition to the Weak Form, you are not allowed to access directly inherited members

# LOD EXAMPLE

```
class Demeter {  
    private A a;  
  
    public void example(B b)  
    {  
        C c;  
        c = func();  
        b.invert();passed parameters  
        a = new A();  
        a.setActive();attribute  
        c.print();local variable  
    }  
}
```

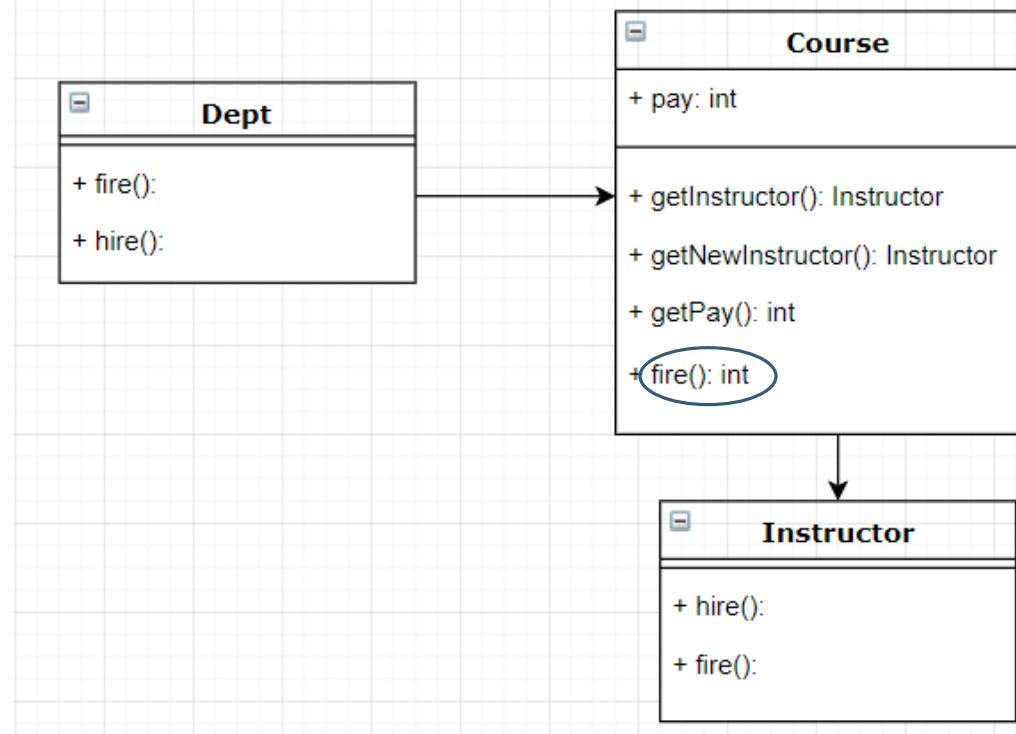
# LOD COUNTER EXAMPLE



```
class Course
{
    Instructor boring = new Instructor();
    int pay = 5;
    public Instructor getInstructor() { return boring; }
    public Instructor getNewInstructor() { return new Instructor(); }
    public int getPay() { return pay; }
}

class Dept {
    Course test = new Course();
    public void fire() { test.getInstructor().fire(); }
    public void hire() { test.getNewInstructor().hire(); }
    public int raisePay() { return test.getpay() + 10; }
}
```

# LOD GOOD EXAMPLE



```
class Course {
    Instructor boring = new Instructor();
    int pay = 5;
    public Instructor fire(){ boring.fire(); }
    public Instructor getNewInstructor() { return new Instructor(); }
    public int getPay() { return pay ; }
}

class Dept {
    Course test = new Course();
    public void fire() {test.fire();}
        public void hire() {test.getNewInstructor().hire();}
        public int raisePay() { return test.getpay() + 10;}
}
```

# LOD FOR CHILDREN

*You can play with yourself.*

*You can play with your own toys*

*You can play with toys that were given to you.*

*You can play with toys you've made yourself.*

# LOD BENEFITS

## Coupling Control

- reduces data coupling

## Information hiding

- prevents from retrieving subparts of an object

## Information restriction

- restricts the use of methods that provide information

## Few Interfaces

- restricts the classes that can be used in a method

## Explicit Interfaces

- states explicitly which classes can be used in a method

# ACCEPTABLE LOD VIOLATIONS

If optimization requires violation

- Speed or memory restrictions

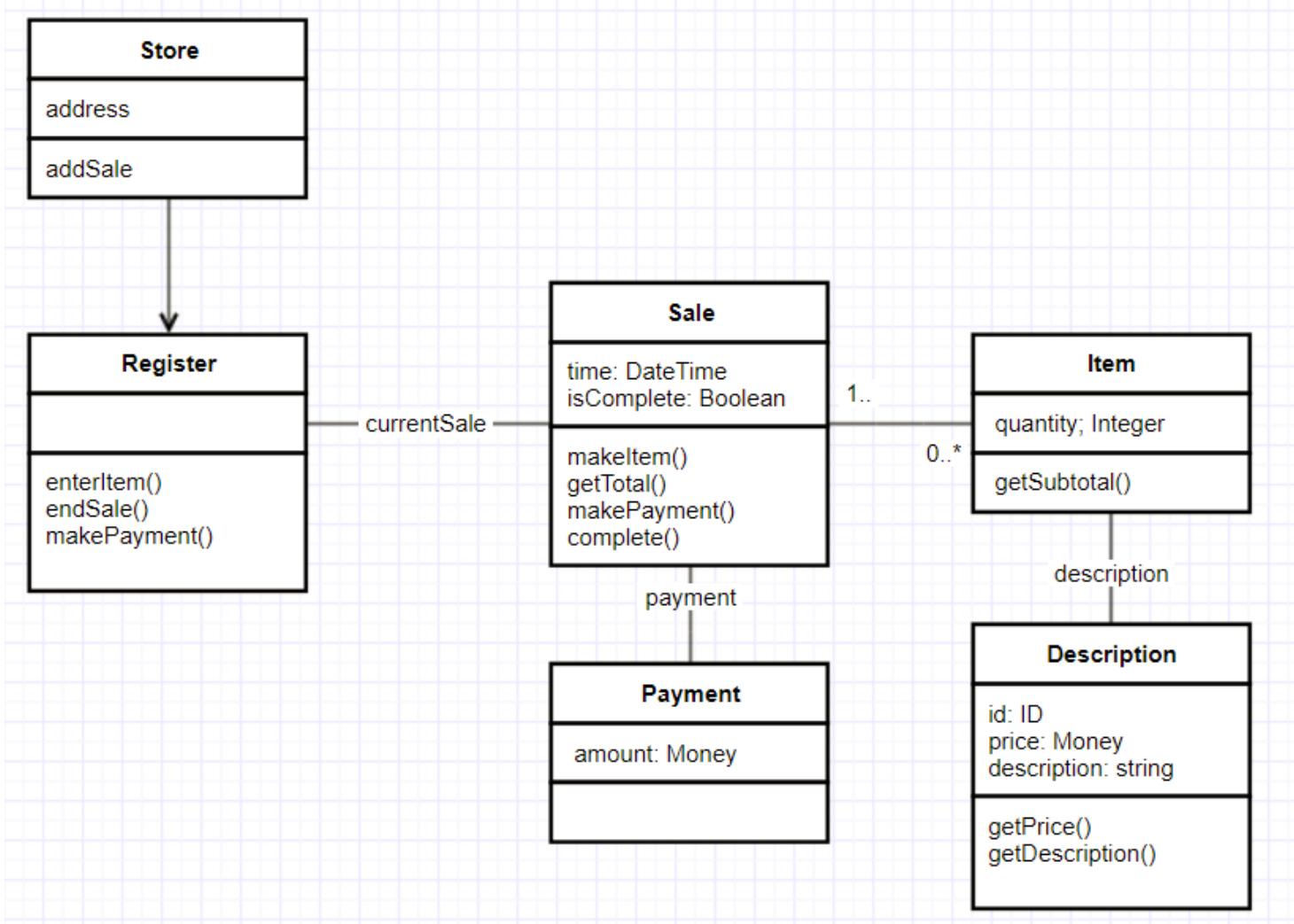
If module accessed is a fully stabilized “Black Box”

- No changes to interface can reasonably be expected due to extensive testing, usage, etc.

Otherwise, do not violate this law!!

- Long-term costs will be very prohibitive

# POS FINAL DESIGN



# HIGHER-LEVEL DESIGN

Dealing with *large-scale systems*

- team of developers, rather than an individual

Classes are a valuable but not sufficient mechanism

- too *fine-grained* for organizing a large scale design
- need mechanism that impose a higher level of order

## Packages

- a logical grouping of declarations that can be imported in other programs
- containers for a group of classes (UML)
- reason at a higher-level of abstraction

# ISSUES OF HIGHER-LEVEL DESIGN

## Goal

- *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages

## Issues

- What are the best partitioning criteria?
- What principles govern the design of packages?
  - *creation* and *dependencies* between packages

## Approach

- Define principles that govern package design
  - the creation and interrelationship and use of packages

# PRINCIPLES OF OO HIGHER-LEVEL DESIGN

## Cohesion Principles

- Reuse/Release Equivalency Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)

## Coupling Principles

- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

# WHAT IS REALLY REUSABILITY ?

Does copy-paste mean reusability?

- Disadvantage: **You own that copy!**
  - you must change it, fix bugs.
  - eventually the code diverges

Martin's Definition:

- *I reuse code if, and only if, I never need to look at the source-code*
- treat reused code like a *product* ⇒ don't have to maintain it

Clients (re-users) may decide to use a newer version of a component release

# REUSE/RELEASE EQUIVALENCY PRINCIPLE (REP)

***The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused. [R. Martin]***

***Either all the classes in a package are reusable or none of it is! [R. Martin]***

# WHAT DOES THIS MEAN?

Reused code = product

- Released, named and maintained by the producer.

Programmer = client

- Doesn't have to maintain reused code
- Doesn't have to name reused code
- May choose to use an older/newer release

# THE COMMON REUSE PRINCIPLE

*All classes in a package [library] should be reused together. If you reuse one of the classes in the package, you reuse them all. [R.Martin]*

*If I depend on a package, I want to depend on every class in that package! [R.Martin]*

# WHAT DOES THIS MEAN?

Criteria for grouping classes in a package:

- Classes that tend to be **reused** together.

Packages have physical representations (shared libraries, DLLs, assembly)

- Changing just one class in the package => re-release the package => revalidate the application that uses the package.

# COMMON CLOSURE PRINCIPLE (CCP)

*The classes in a package should be closed against the same kinds of changes.*

*A change that affects a package affects all the classes in that package*

[R. Martin]

# WHAT DOES THIS MEAN?

Another criteria of grouping classes: **Maintainability!**

- Classes that tend to change together for the same reasons
- Classes highly dependent

SRP at packages level

# REUSE VS. MAINTENANCE

REP and CRP makes life easier for **reuser**

- packages very small

CCP makes life easier for **maintainer**

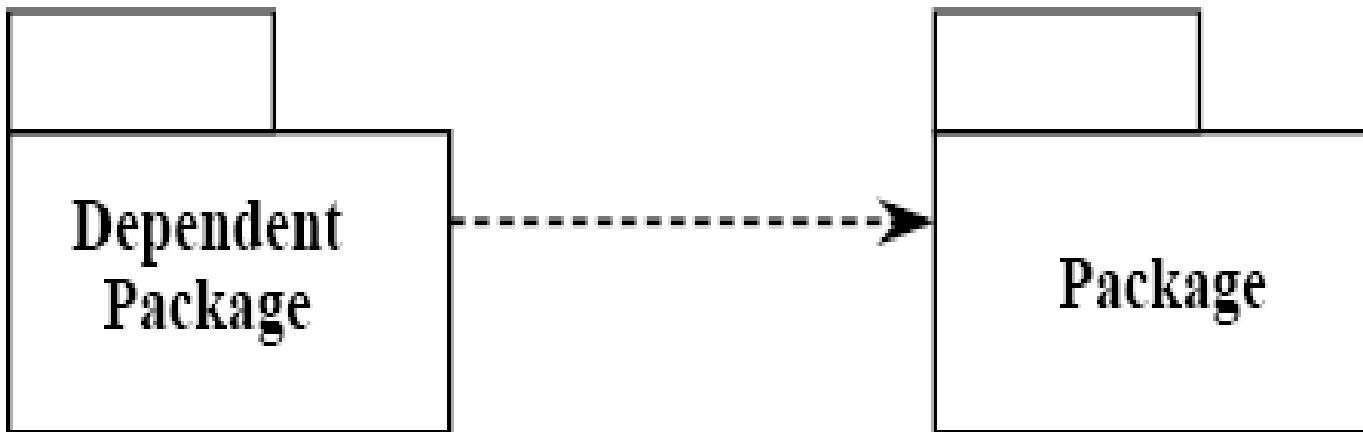
- larger packages

**Packages are not fixed in stone**

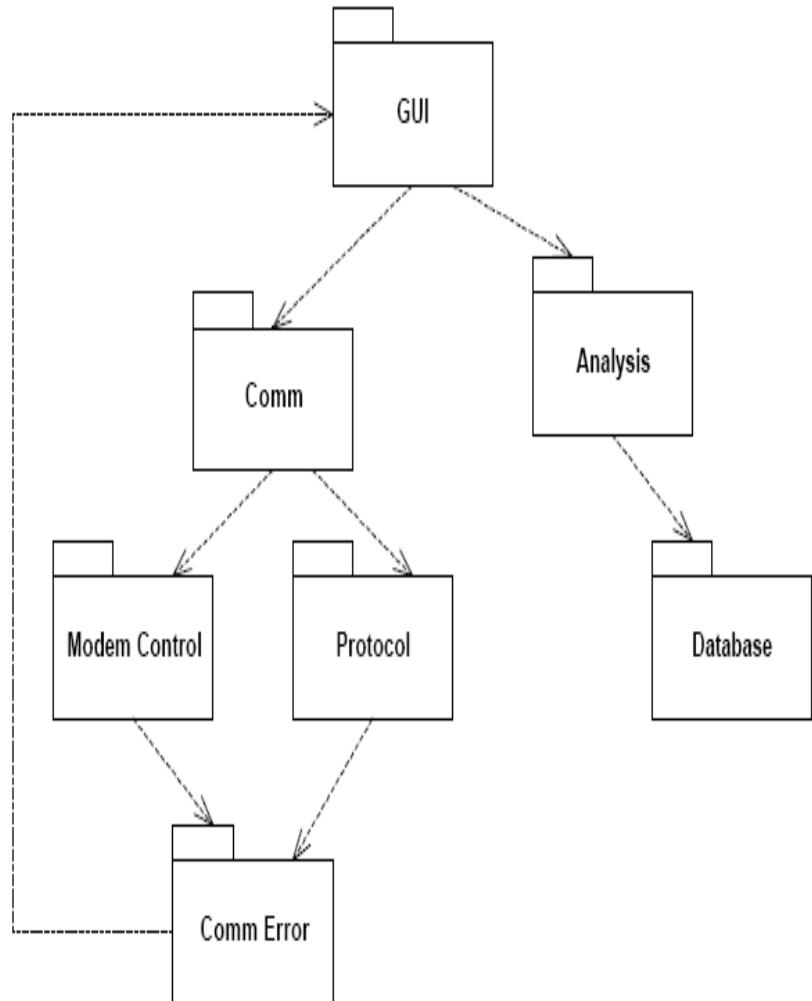
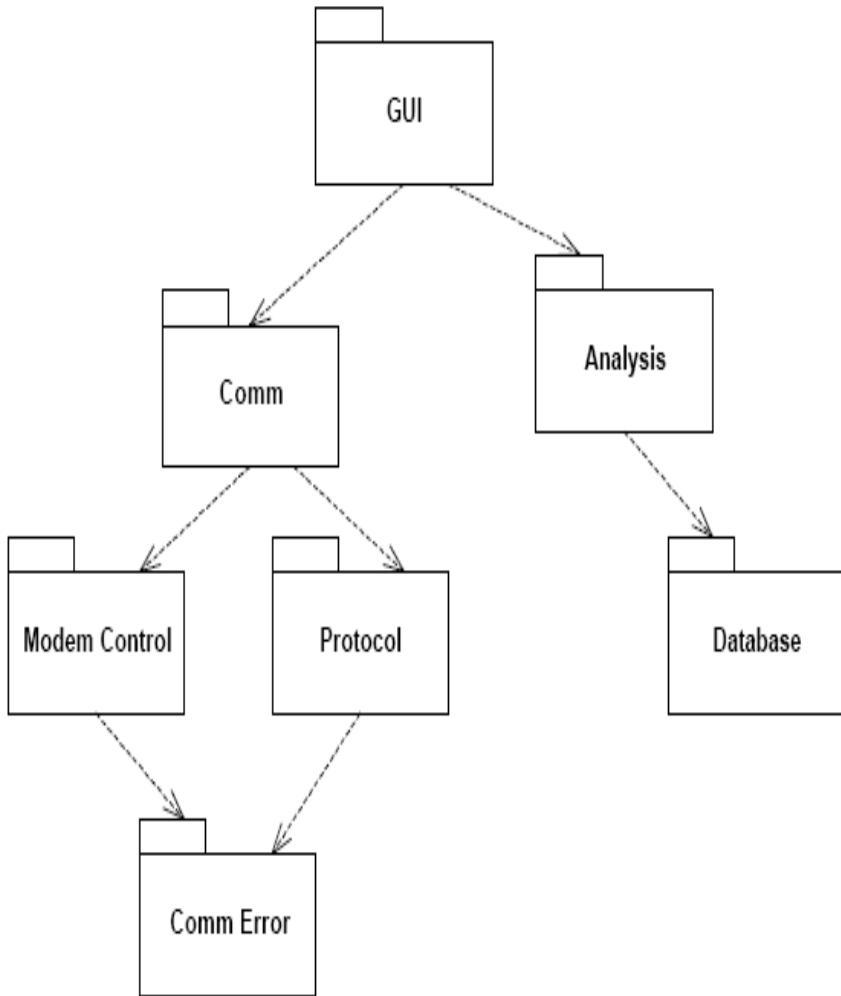
- early in project focus on CCP
- later when architecture stabilizes: focus on REP and CRP

# ACYCLIC DEPENDENCIES PRINCIPLES (ADP)

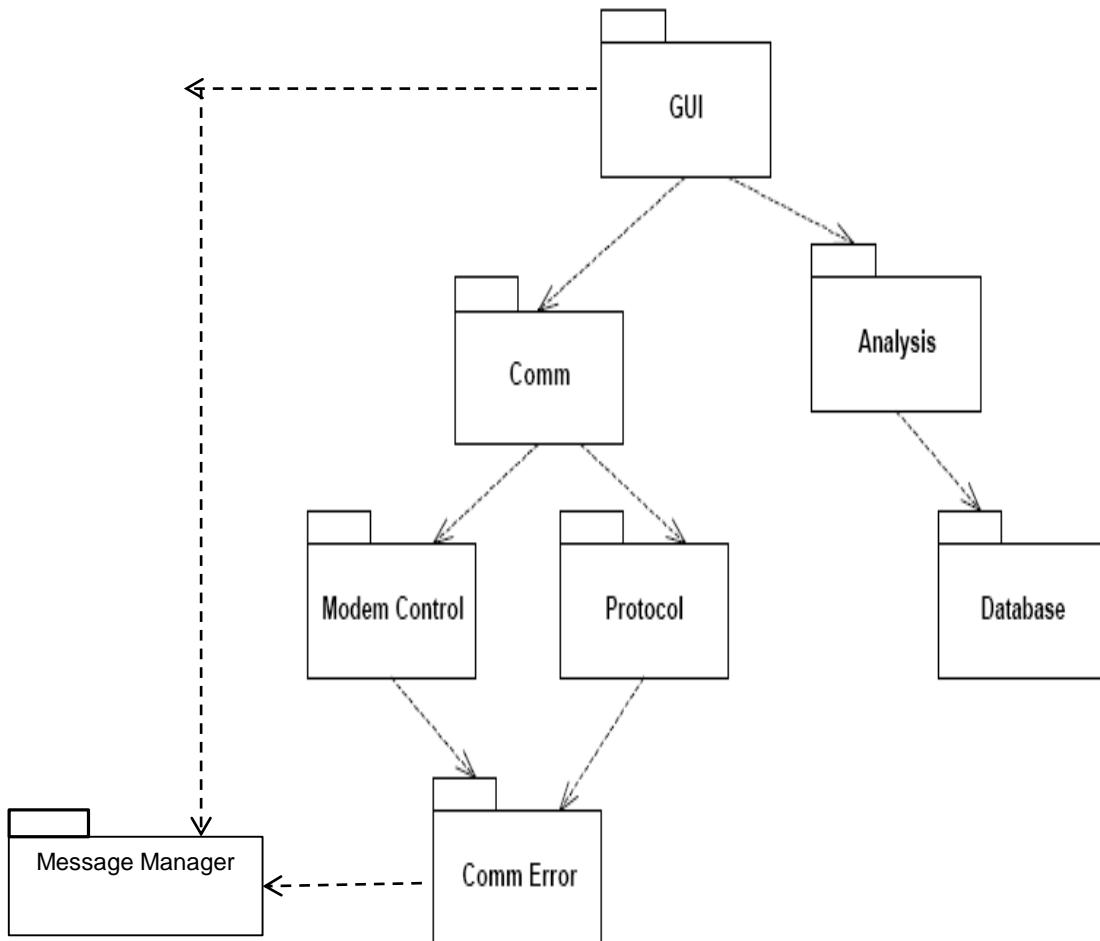
*The dependency structure for released component must be a Directed Acyclic Graph (DAG). There can be no cycles. [R. Martin]*



# DEPENDENCY GRAPHS



# BREAKING THE CYCLE

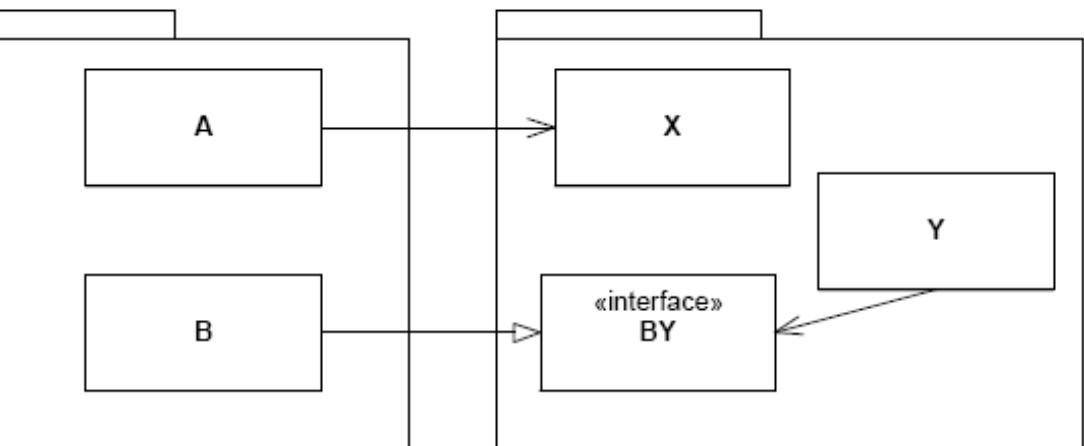
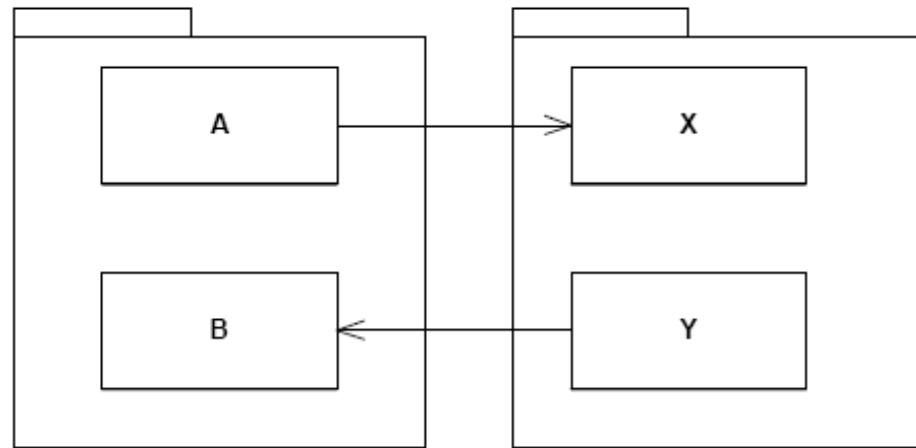


Take out of the GUI  
package the classes that  
Comm Error depend on

=> Add a new package  
(i.e. Message Manager)

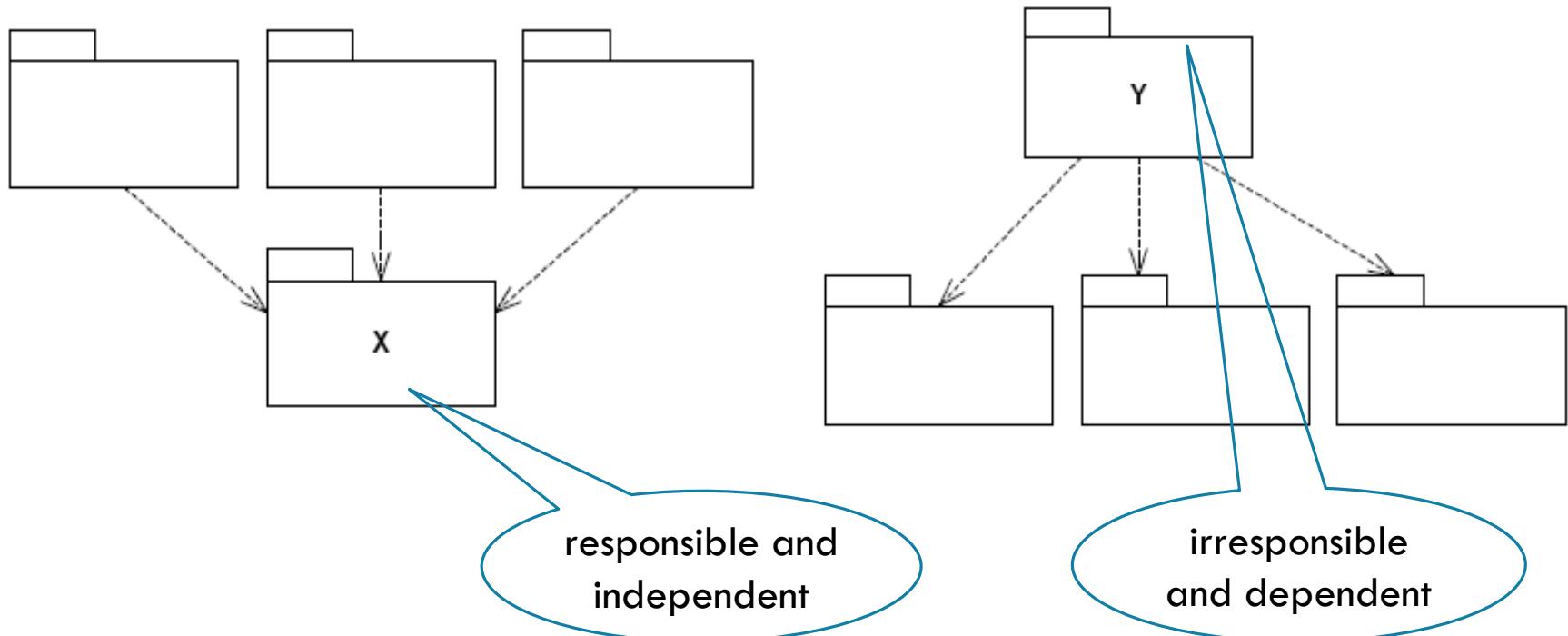
# BREAKING THE CYCLE

DIP + ISP



# STABILITY

Stability is related to the amount of work in order to make a change.



Driven by Responsibility and Independence

# STABILITY METRICS

$C_a$  – Afferent coupling (incoming dependencies)

- How responsible am I?

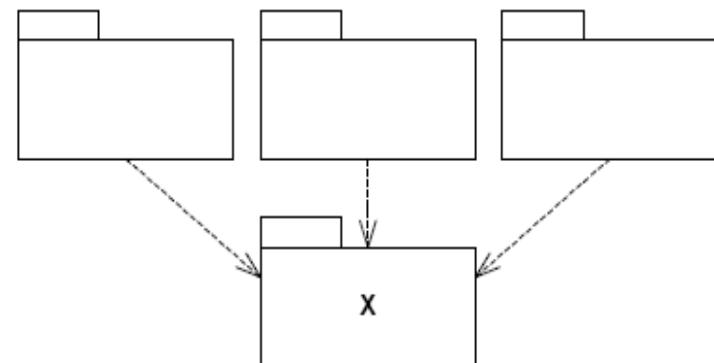
$C_e$  – Efferent coupling (outgoing dependencies)

- How dependant am I?

$$\text{Instability } I = C_e / (C_a + C_e)$$

Example for X:

$$C_a = 3, C_e = 0 \Rightarrow I = 0 \text{ (very stable)}$$



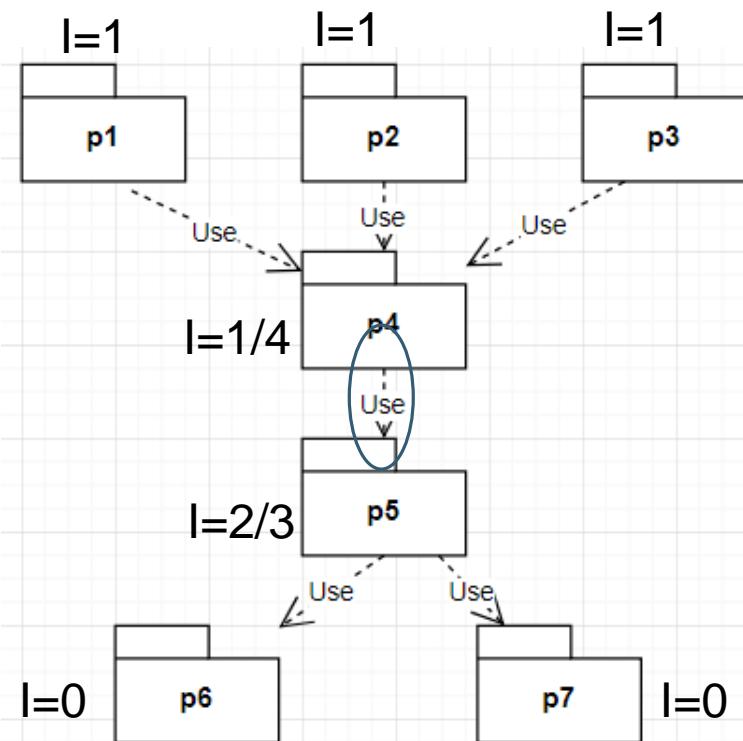
# STABLE DEPENDENCY PRINCIPLE (SDP)

Depend in the direction of stability.

What does this mean?

- Depend upon packages whose I is lower than yours.

## Counter-example



# WHERE TO PUT HIGH-LEVEL DESIGN?

High-level architecture and design decisions don't change often

- shouldn't be volatile  $\Rightarrow$  place them in stable packages
- design becomes hard to change  $\Rightarrow$  *inflexible* design

**How can a totally stable package ( $I = 0$ ) be flexible enough to withstand change?**

Answer: ***The Open-Closed Principle***

- Classes that can be extended without modifying them  
 $\Rightarrow$  **Abstract Classes**

# STABLE ABSTRACTIONS PRINCIPLE (SAP)

Stable packages should be abstract packages.

What does this mean?

- Stable packages should be depended upon
- Flexible packages should be dependent
- OCP => Stable packages should be highly abstract

# ABSTRACTNESS METRICS

$N_c$  = number of classes in the package

$N_a$  = number of abstract classes in the package

*Abstractness A* =  $N_a/N_c$

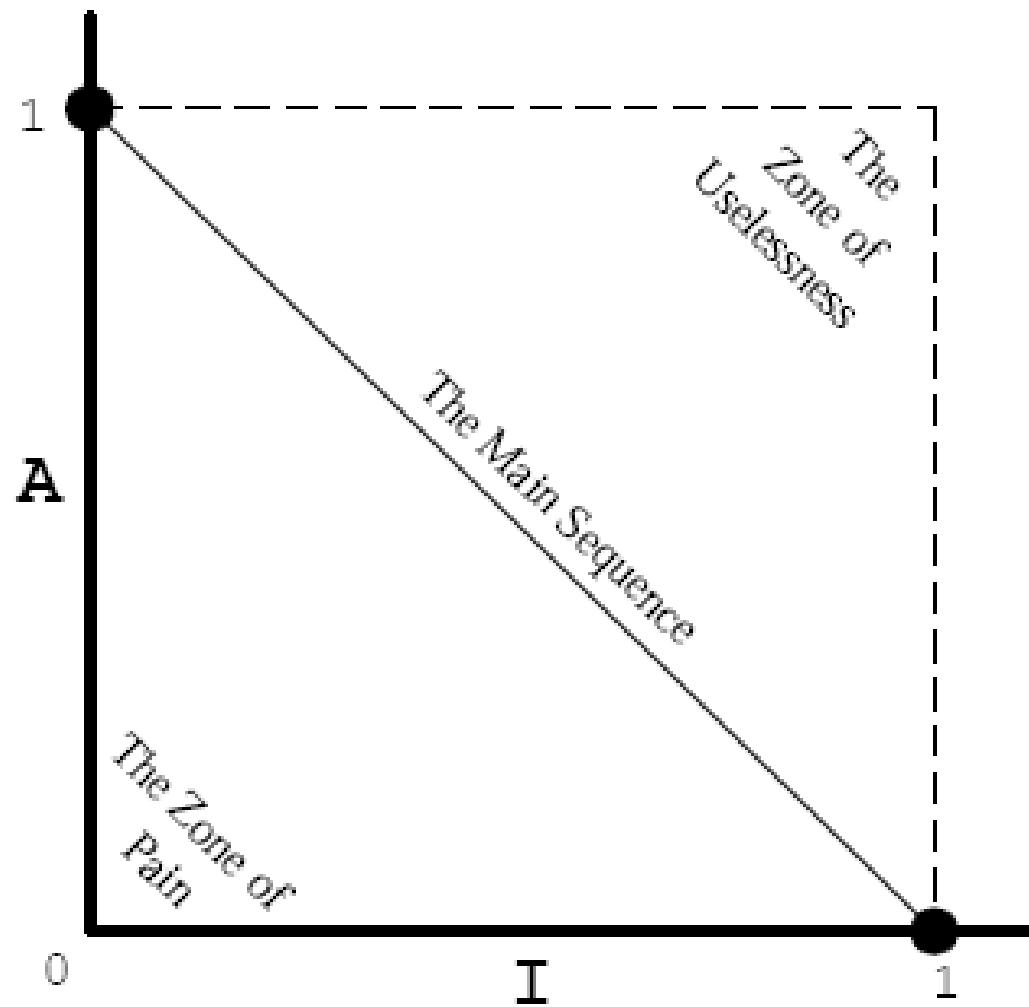
Example:

- All classes are concrete  $N_a = 0 \Rightarrow A = 0$

What about hybrid classes?

# THE MAIN SEQUENCE

$I$  should increase as  $A$  decreases



# THE MAIN SEQUENCE

## Zone of Pain

- highly stable and concrete  $\Rightarrow$  rigid
- famous examples:
  - database-schemas (volatile and highly depended-upon)
  - concrete utility libraries (instable but non-volatile)

## Zone of Uselessness

- instable and abstract  $\Rightarrow$  useless
  - no one depends on those classes

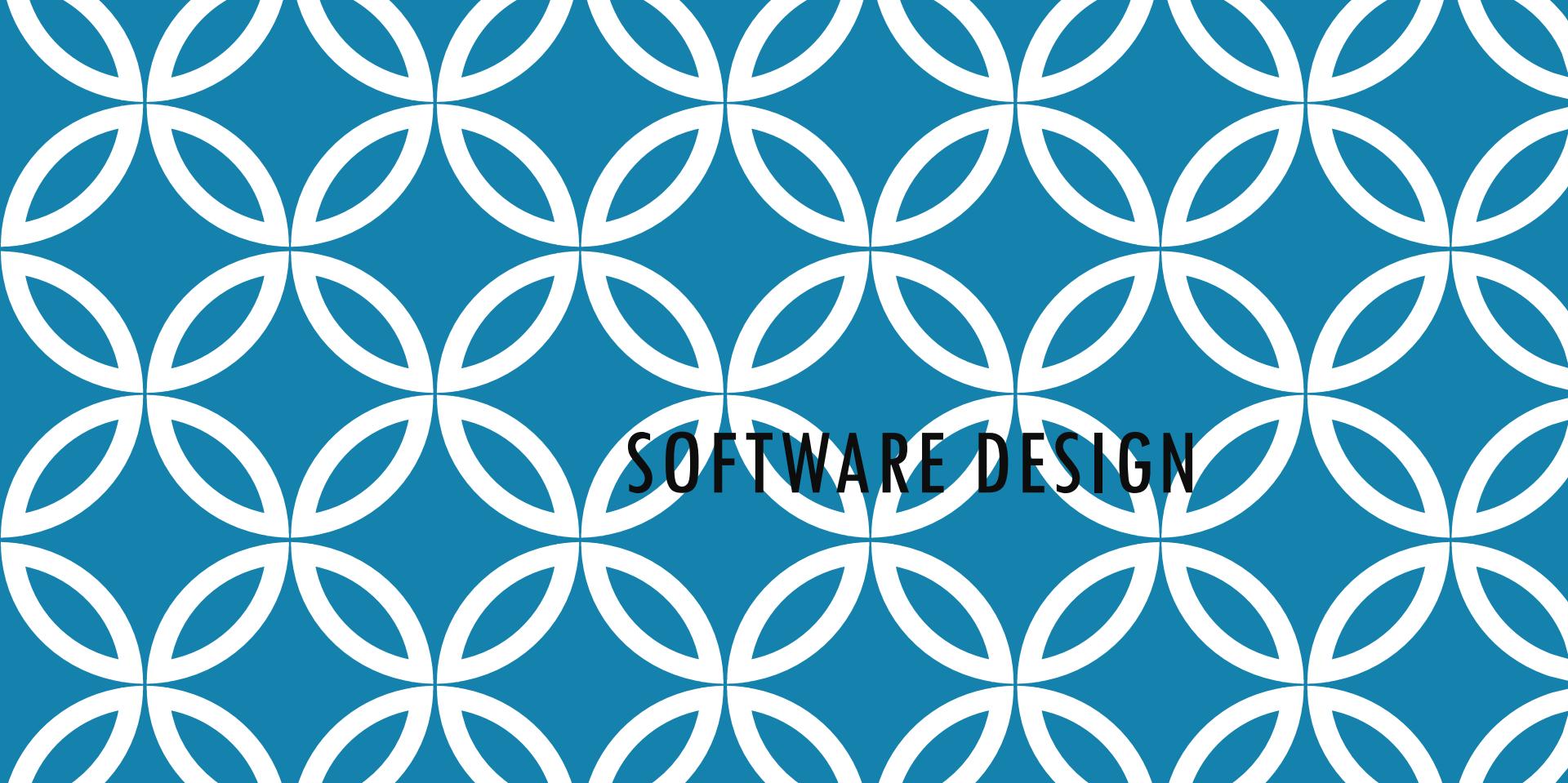
## Main Sequence

- maximizes the distance between the zones we want to avoid
- depicts the balance between abstractness and stability.

# WRAP-UP

Principles for good class design related to

- Assigning Responsibilities (GRASP)
- Package design
- Coupling
- Cohesion



# SOFTWARE DESIGN

Architectural Patterns 1

# CONTENT

## Architectural Patterns

- Layers
  - Client-Server
- Event-driven
  - Broker
  - Mediator
- MVC (and variants)

# REFERENCES

## Books

- Martin, Robert C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017.
- Mark Richards, *Software Architecture Patterns*, O'Reilly, 2015 [SAP]
- Taylor, R., Medvidovic, N., Dashofy, E., *Software Architecture: Foundations, Theory, and Practice*, 2010, Wiley [Taylor]
- F. Buschmann et. al, *PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns*, Wiley&Sons, 2001.[POSA]
- Artem Syromiatnikov, *A Journey Through the Land of Model-View-\* Design Patterns*, MSc Thesis, 2014.
- Reid Holmes, *MVC/MCP*, Univ. of Waterloo course materials

## Online resources

- O. Shelest, *MVC, MVP, MVVM design patterns*,
- <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/>
- [http://www.codeproject.com/KB/architecture/MVC\\_MVP\\_MVVM\\_design.aspx](http://www.codeproject.com/KB/architecture/MVC_MVP_MVVM_design.aspx)
- <http://www.tinmegali.com/en/model-view-presenter-mvp-in-android-part-2/>
- <https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/>
- [https://www.infoq.com/articles/no-more-mvc-frameworks?utm\\_source=infoq&utm\\_campaign=user\\_page&utm\\_medium=link](https://www.infoq.com/articles/no-more-mvc-frameworks?utm_source=infoq&utm_campaign=user_page&utm_medium=link)

# LAST TIME

## GRASP

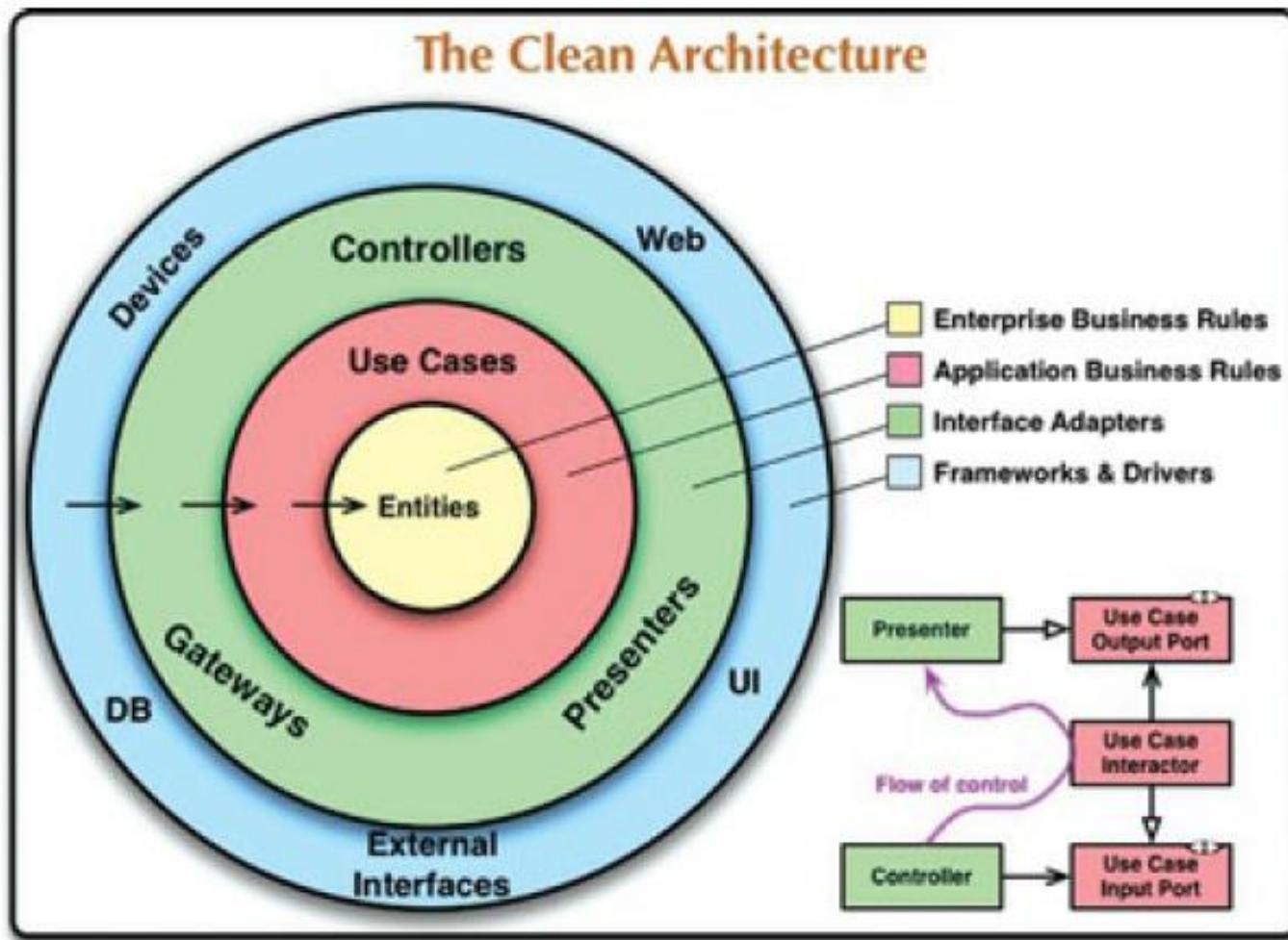
### Package design principles

- Cohesion
- Coupling

# CLEAN ARCHITECTURE

- *Independent of frameworks.* Frameworks are tools, they do not drive your architecture.
- *Testable.* The business rules can be tested without the UI, database, web server, or any other external element.
- *Independent of the UI.* The UI can change easily, without changing the rest of the system.
- *Independent of the database.* You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else.
- *Independent of any external agency.* In fact, your business rules don't know anything at all about the interfaces to the outside world.

# DEPENDENCIES IN A CLEAN ARCHITECTURE



# ARCHITECTURAL PATTERNS

## Definition

- An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

[POSA, vol.1]

Different books present different taxonomies of patterns

# STRUCTURAL AP

From Mud to Structure

Direct mapping Requirements -> Architecture?

Problems: non-functional qualities like

- availability
- reliability
- maintainability
- understandability...

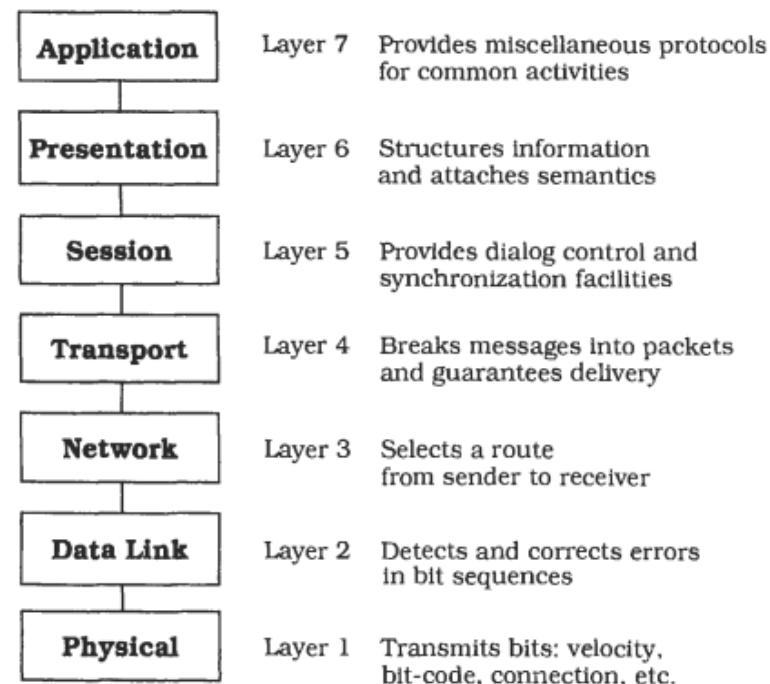
# LAYERS

## Definition

- The **Layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

## Example

- Networking: OSI 7-Layer Model



# DESIGNING LAYERS

Addressed problems

- High-level and low-level operations
- High-level operations rely on low-level ones
  - ⇒ Vertical structuring
- Several operations on the same level on abstraction but highly independent
  - ⇒ Horizontal structuring

# CONSTRAINTS

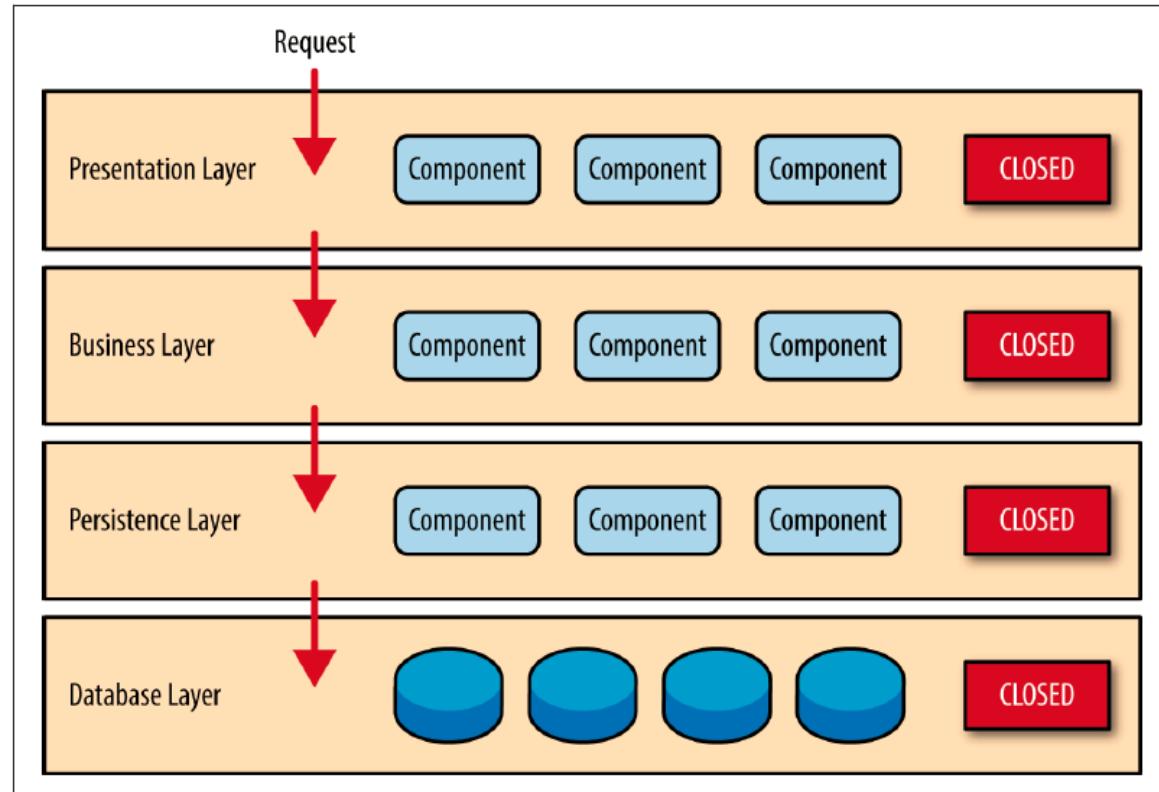
- Late source code/design **changes should not ripple through the system.**
- **Interfaces should be stable** and may even be prescribed by a standards body.
- **Parts of the system should be exchangeable.**
- It may be necessary to **build other systems** at a later date **with the same low-level issues** as the system you are currently designing.

# CONSTRAINTS [2]

- **Similar responsibilities should be grouped** to help understandability and maintainability.
- There is **no 'standard' component granularity**.
- Complex components need further **decomposition**.
- The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

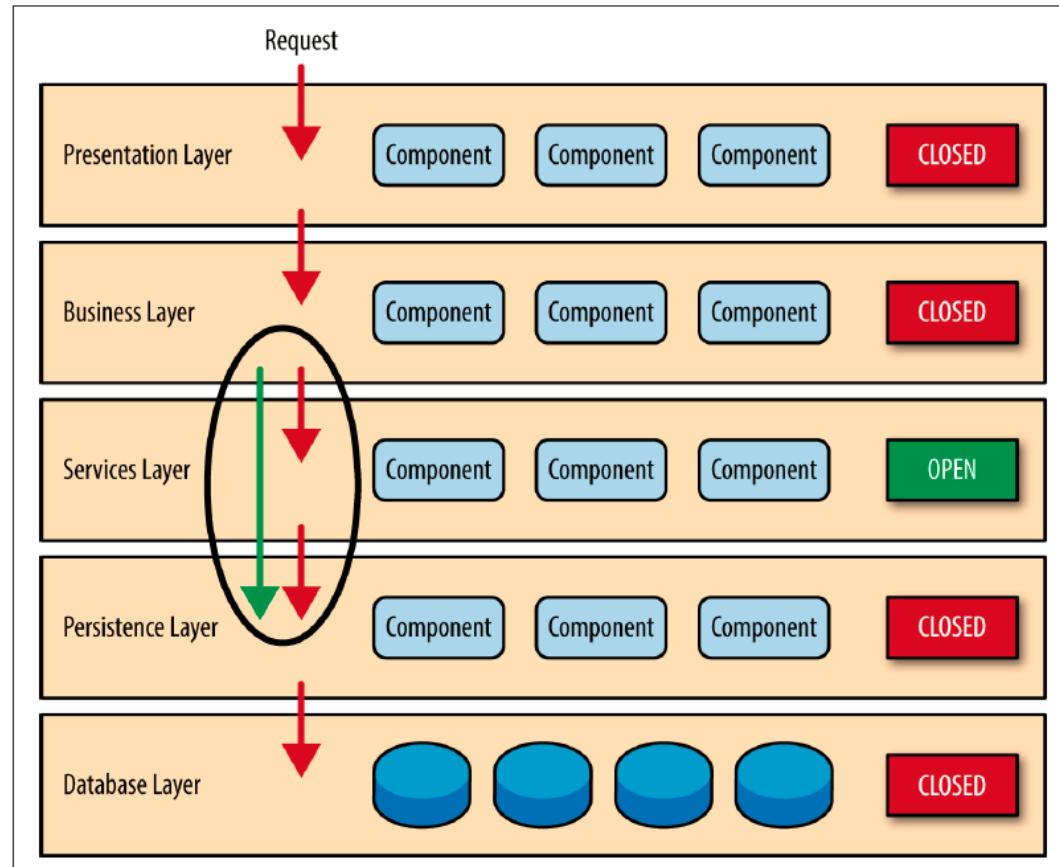
# KEY CONCEPTS

- Closed layers
- Layers of isolation: changes made in one layer of the architecture don't impact components in other layers



# VARIANTS

- **open layered system** (any component calls any other component)
- **semi-closed/semi-open architecture** allows calling more than one layer down



# EXAMPLE

## Customer Screen:

- Java Server Faces
- ASP (MS)

## Customer Delegate:

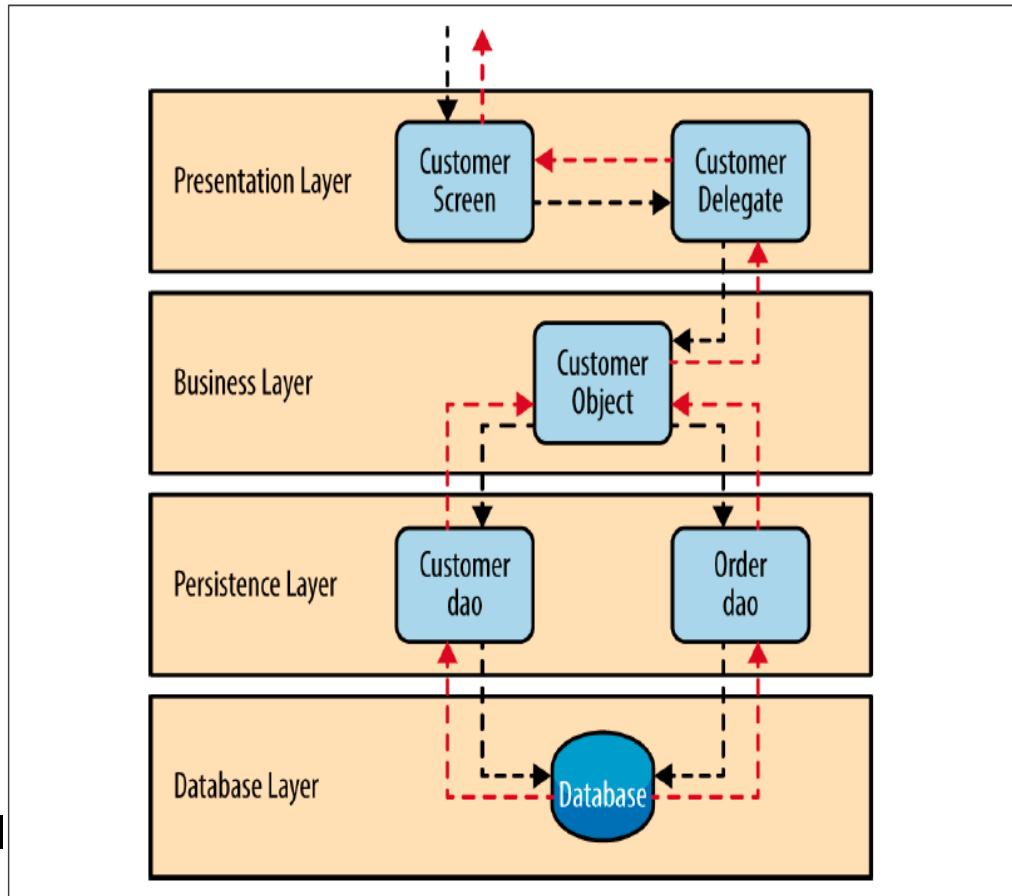
- managed bean component

## Customer Object:

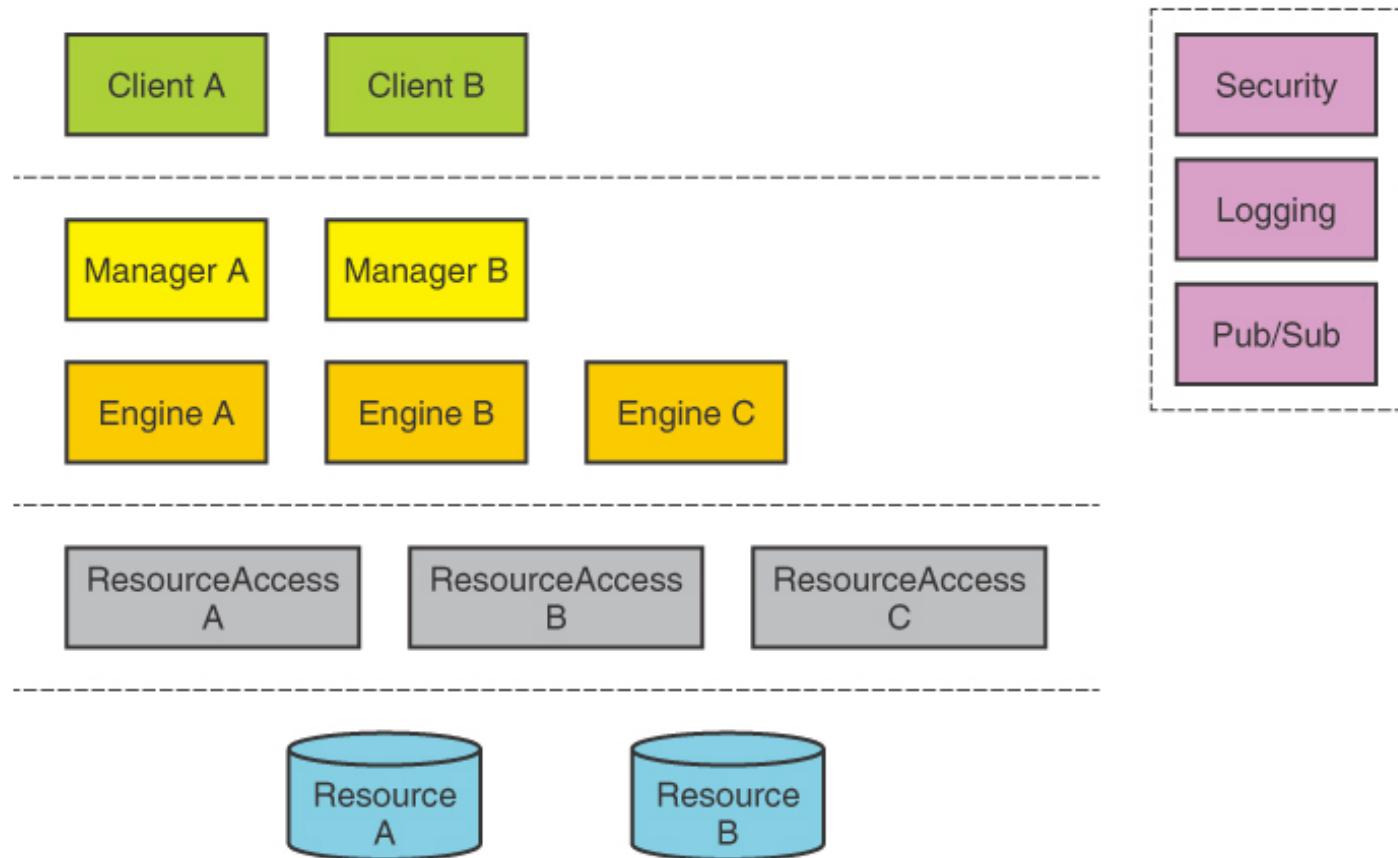
- Local Spring Bean
- Remote EJB3 component
- C# (MS)

## DAOs:

- POJOs
- MyBatis XML Mapper files
- Objects encapsulating raw JDBC calls or Hibernate queries
- ADO (MS)



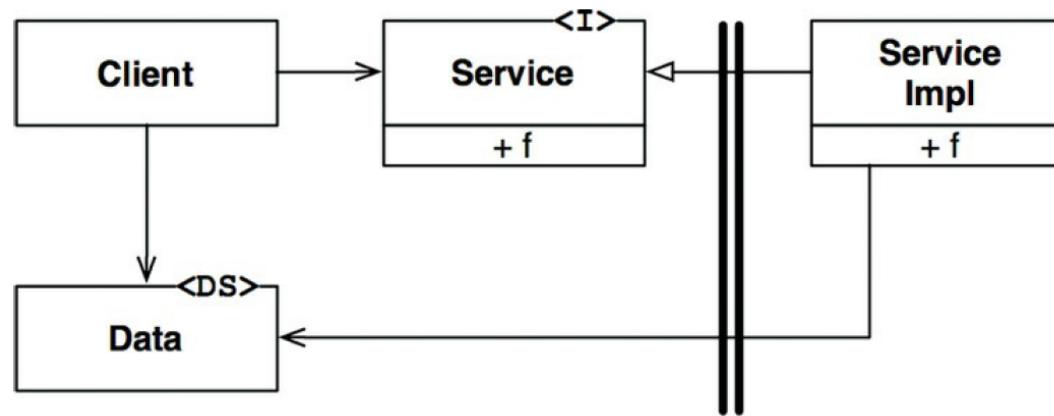
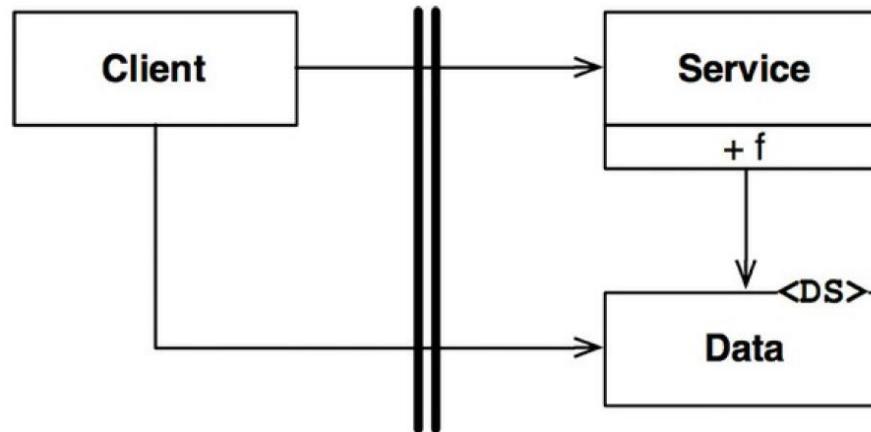
# RELAXING THE RULES



# DESIGN STEPS

- Define abstraction criterion for grouping tasks into layers
- Determine number of abstraction levels
- Name the layers and assign tasks to each of them
- Specify services (from a layer to another)
- Refine layering (iterate steps above)
- Specify interface for each layer
- Structure individual layers
- Specify communication between adjacent layers
- Decouple adjacent layers (callbacks for bottom-up)
- Design error-handling strategy

# DEPENDENCY MANAGEMENT

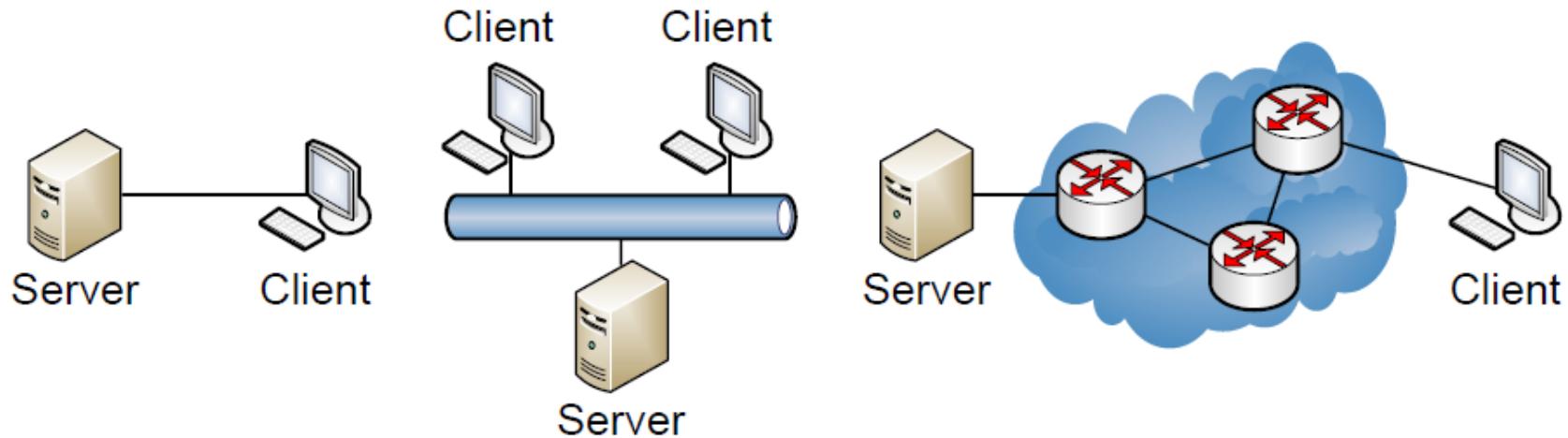


# CLIENT-SERVER

Partition tasks and workload between provider of a resource (server) and requester (client)

Client does not share resources

Server hosts one or more server programs



# STRUCTURE

## Server

- Provides function or service (E.g. web server, web page, file server)
- Can service multiple clients

## Client

- Consumes services
- Interacts with server to retrieve data
- Must understand response based on application protocol

**Both can process data**

# TYPES OF SERVERS AND CLIENTS

## Servers

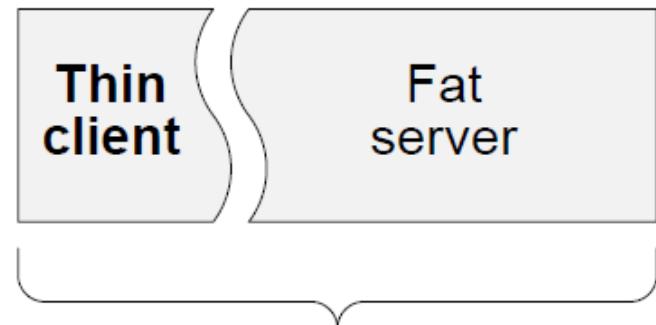
- Iterative (UDP-based servers, ex. Internet services like echo, daytime)
- Concurrent (TCP-based servers, ex. HTTP, FTP)
  - Thread-per-client
  - Thread pool

## Clients

- Thin
- Fat



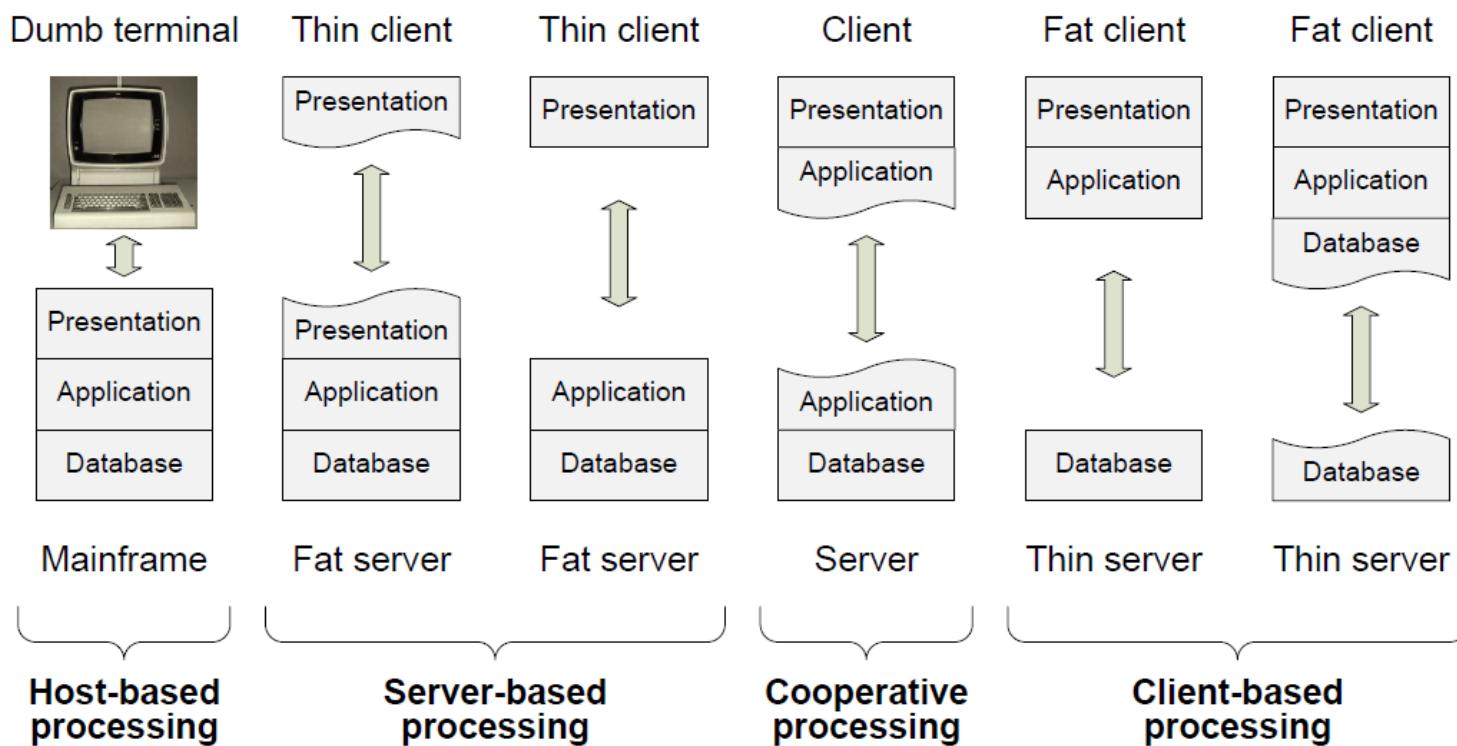
Functionality &  
processing load



Functionality &  
processing load

# LAYERS VS. TIERS

- Layers – logical (ex. presentation, business logic, data access)
- Tiers – physical

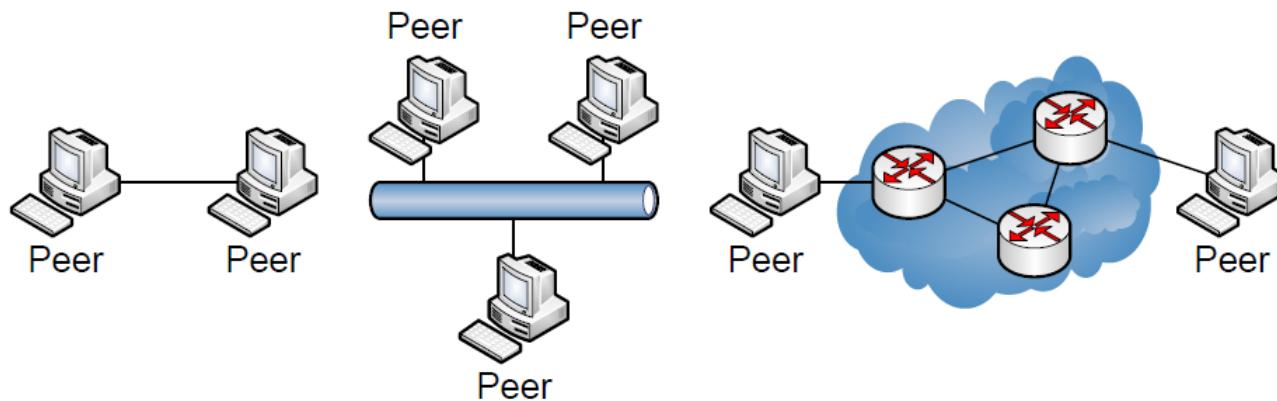


# CONSEQUENCES ON QUALITY ATTRIBUTES

Quality Attribute	Issues
Availability	Servers in each tier can be replicated, so that if one fails, others remain available. Overall the application will provide a lower quality of service until the failed server is restored.
Failure handling	If a client is communicating with a server that fails, most web and application servers implement transparent failover. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request.
Modifiability	Separation of concerns enhances modifiability, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other tiers.
Performance	This architecture has proven high performance. Key issues to consider are the amount of concurrent threads supported in each server, the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.
Scalability	As servers in each tier can be replicated, and multiple server instances run on the same or different servers, the architecture scales out and up well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.

# PEER-TO-PEER

- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages



# PEER-TO-PEER [2]

- **Topology:** Network (may have redundant connections between peers) can **vary arbitrarily and dynamically**
- Supports **decentralized** computing with flow of control and resources distributed among peers.
- Highly **robust** in the face of failure of any given node.
- **Scalable** in terms of access to resources and computing power.
- **Drawbacks:**
  - Poor security
  - Nodes with shared resources have poor performance

# EVENT-DRIVEN (DISTRIBUTED) ARCHITECTURES

- (Distributed) **asynchronous** architecture pattern
- Highly **scalable** applications
- Highly **adaptable** by integrating highly decoupled, single-purpose event processing components that asynchronously receive and process events
- 2 main topologies
  - Broker
  - Mediator

# BROKER

## Definition

- The Broker architectural pattern can be used to structure distributed software systems with **decoupled components** that interact by remote service invocations. A broker component is responsible for coordinating communication.

## Example

- SOA

# BROKER

## Context

- The environment is a **distributed** and possibly **heterogeneous** system with **independent, cooperating components**.

## Problems

- System = set of decoupled and inter-operating components
- Inter-process communication
- Services for adding, removing, exchanging, activating and locating components are also needed.

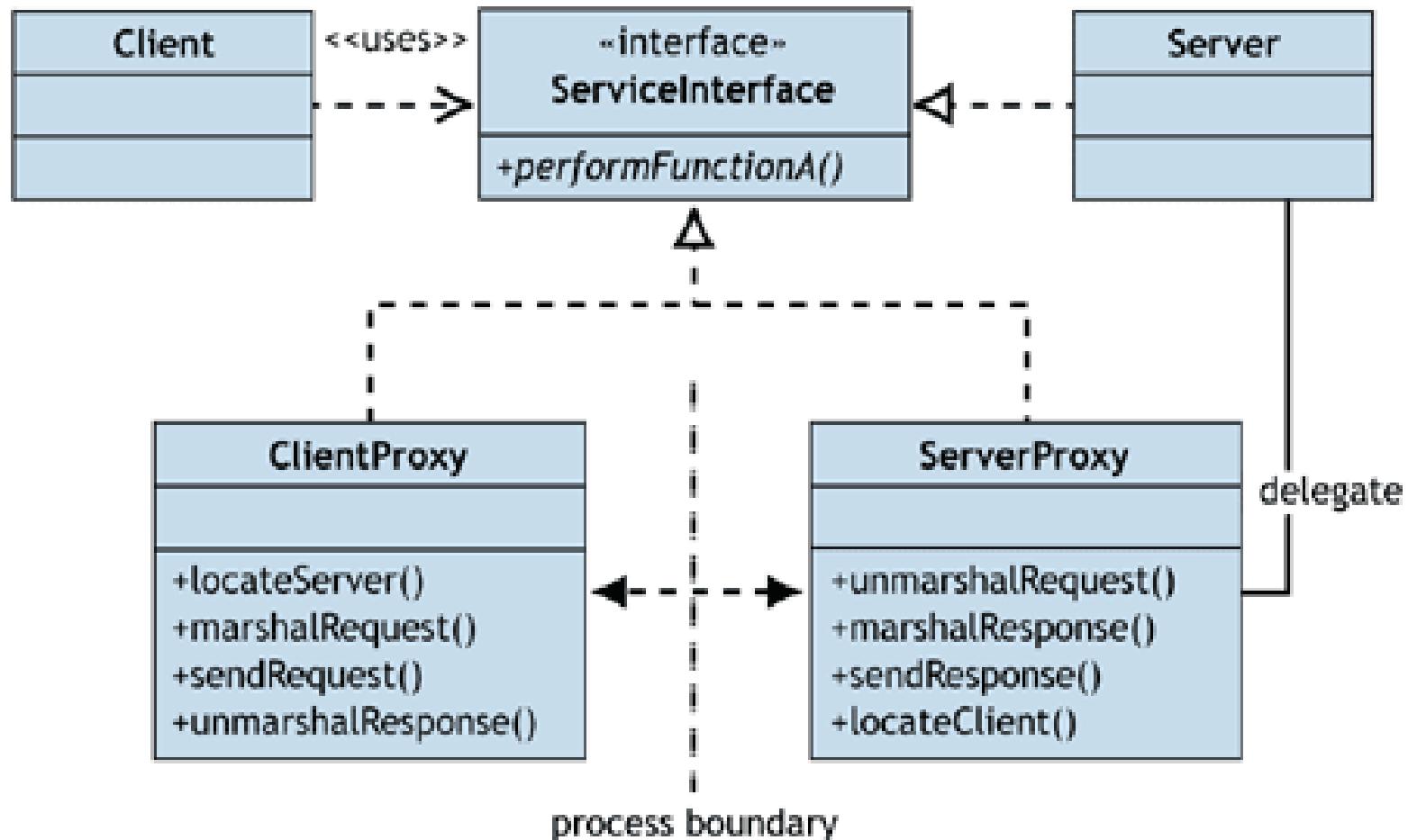
# CONSTRAINTS

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

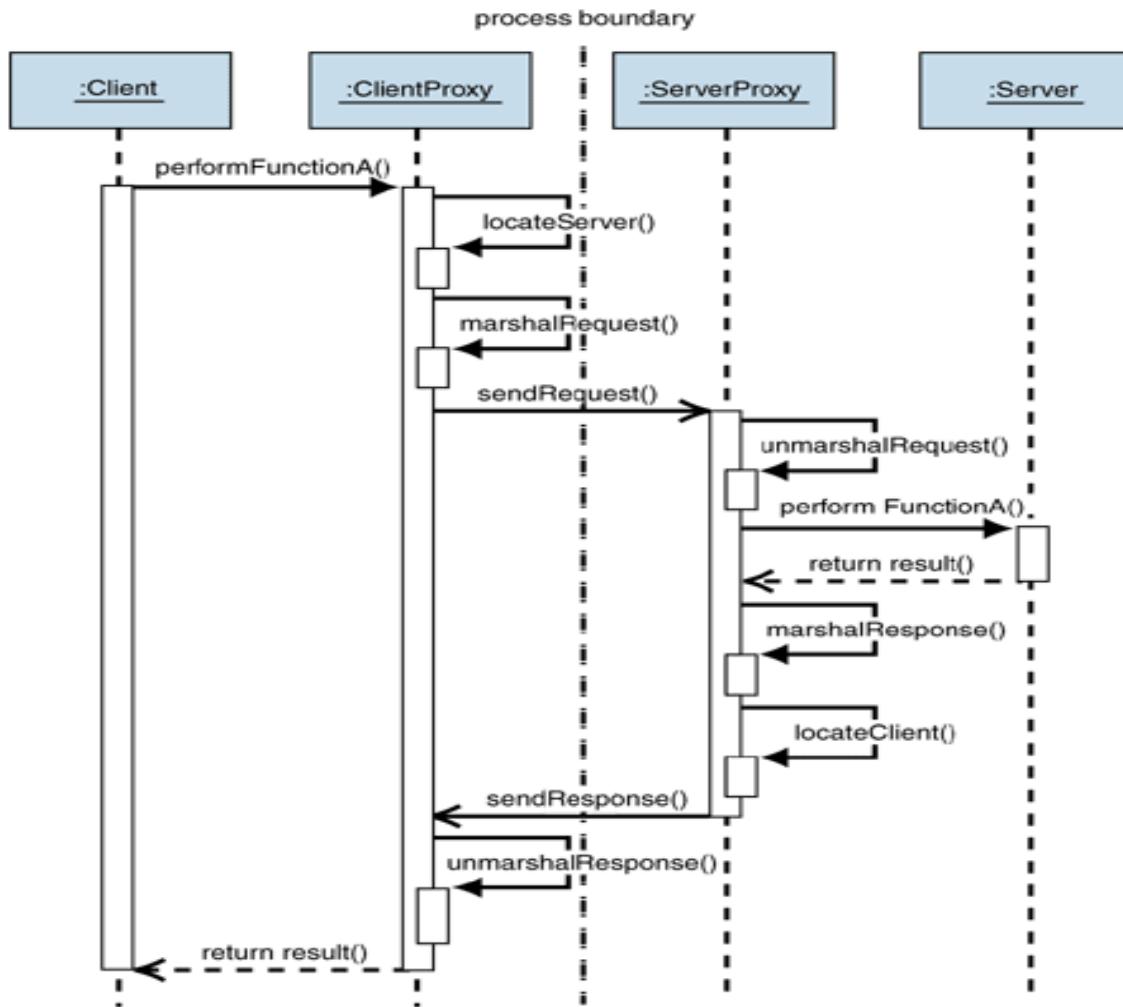
# NON-DISTRIBUTED SYSTEM [MSDN]



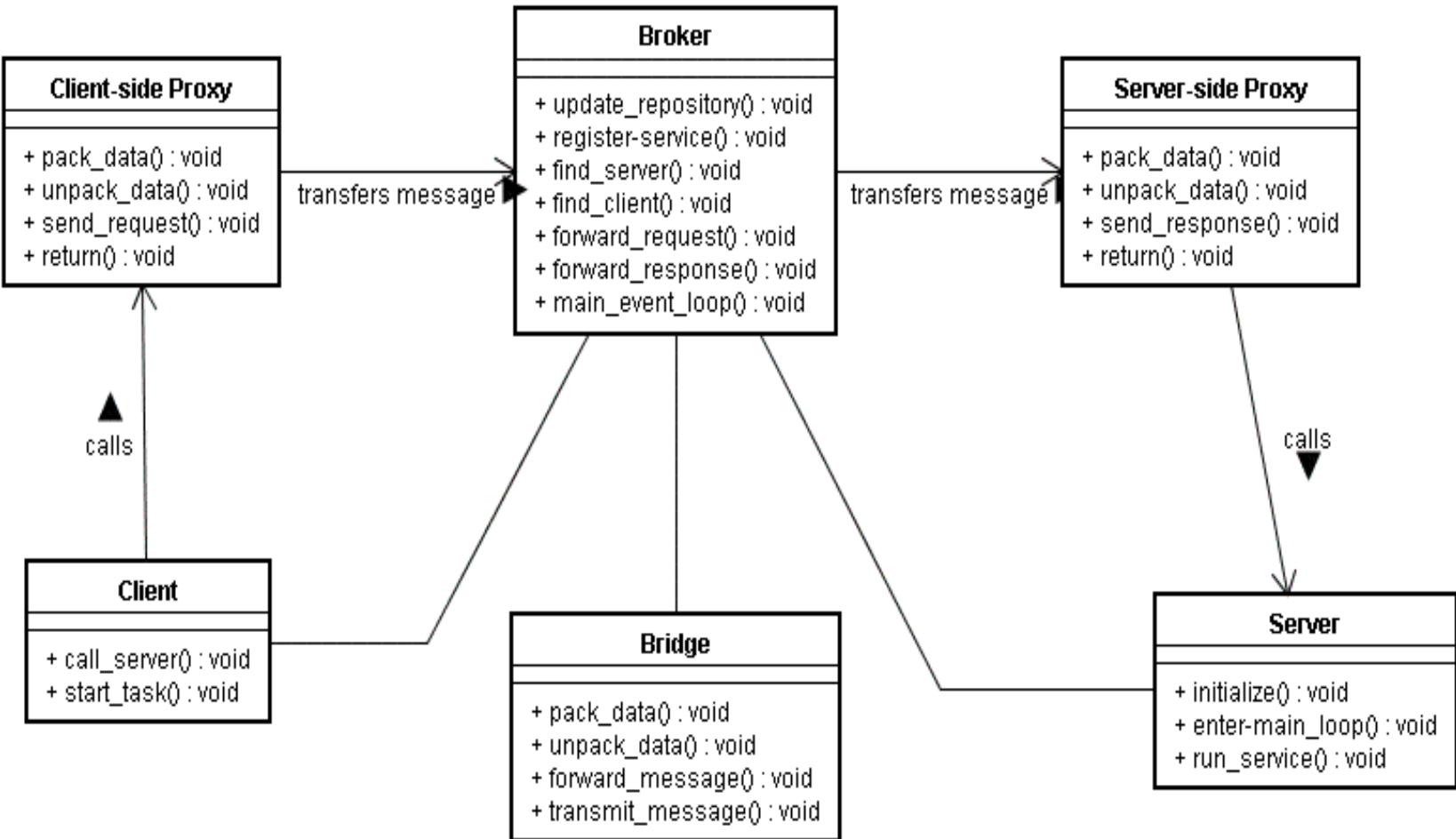
# DISTRIBUTED SYSTEM [MSDN]



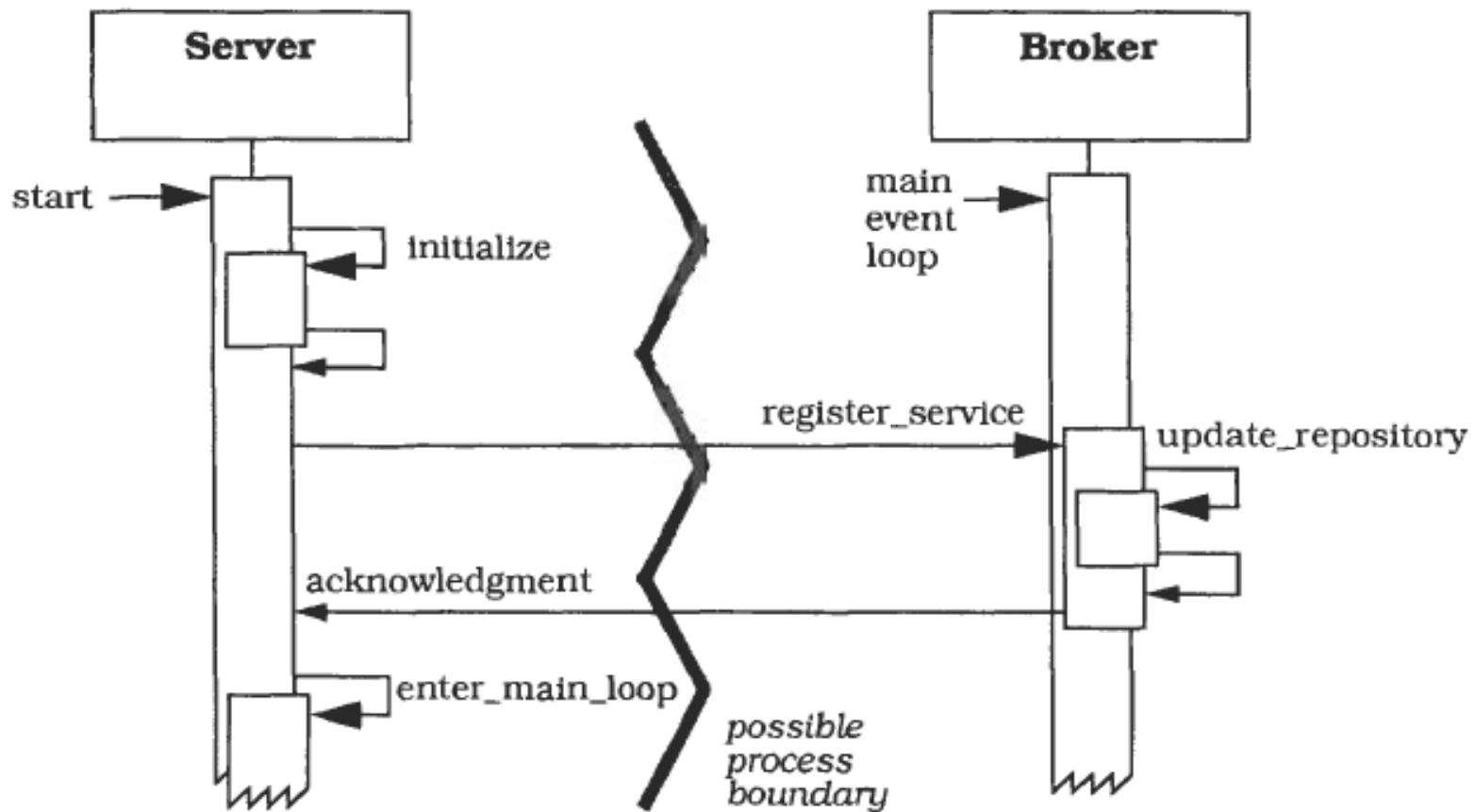
# DISTRIBUTED SYSTEM – PROBLEMS?



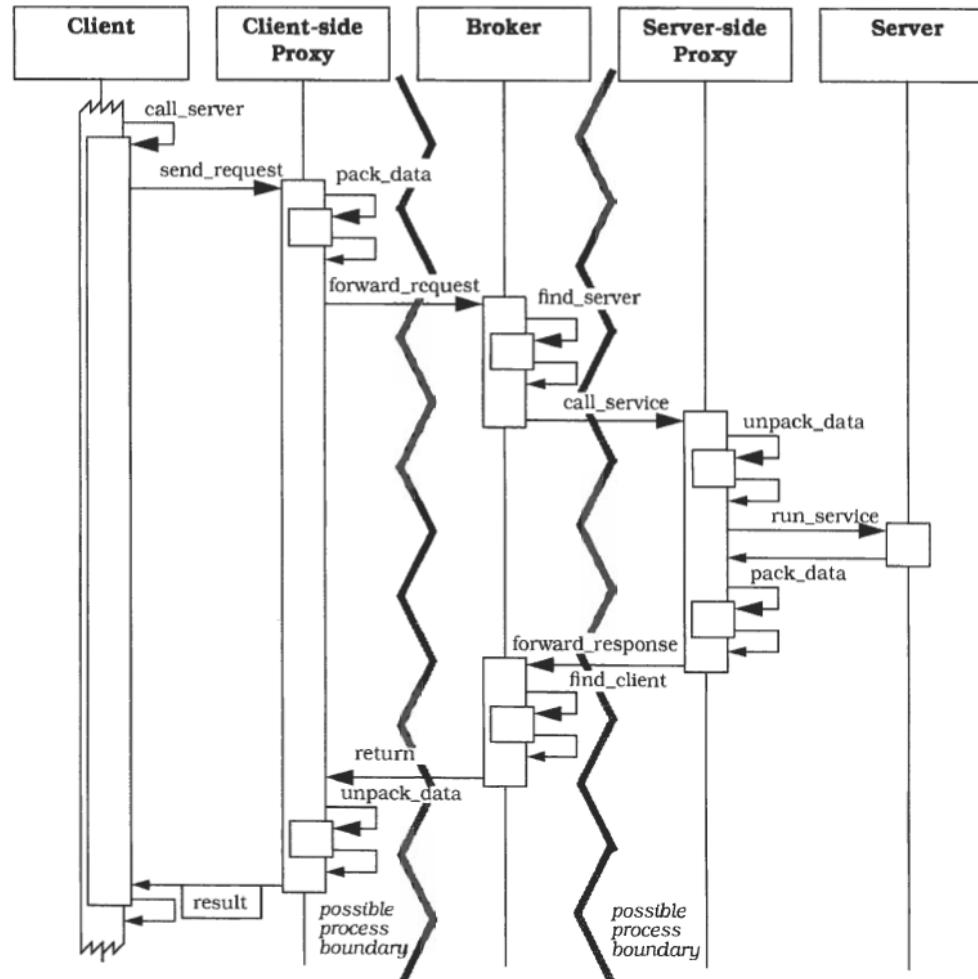
# SOLUTION



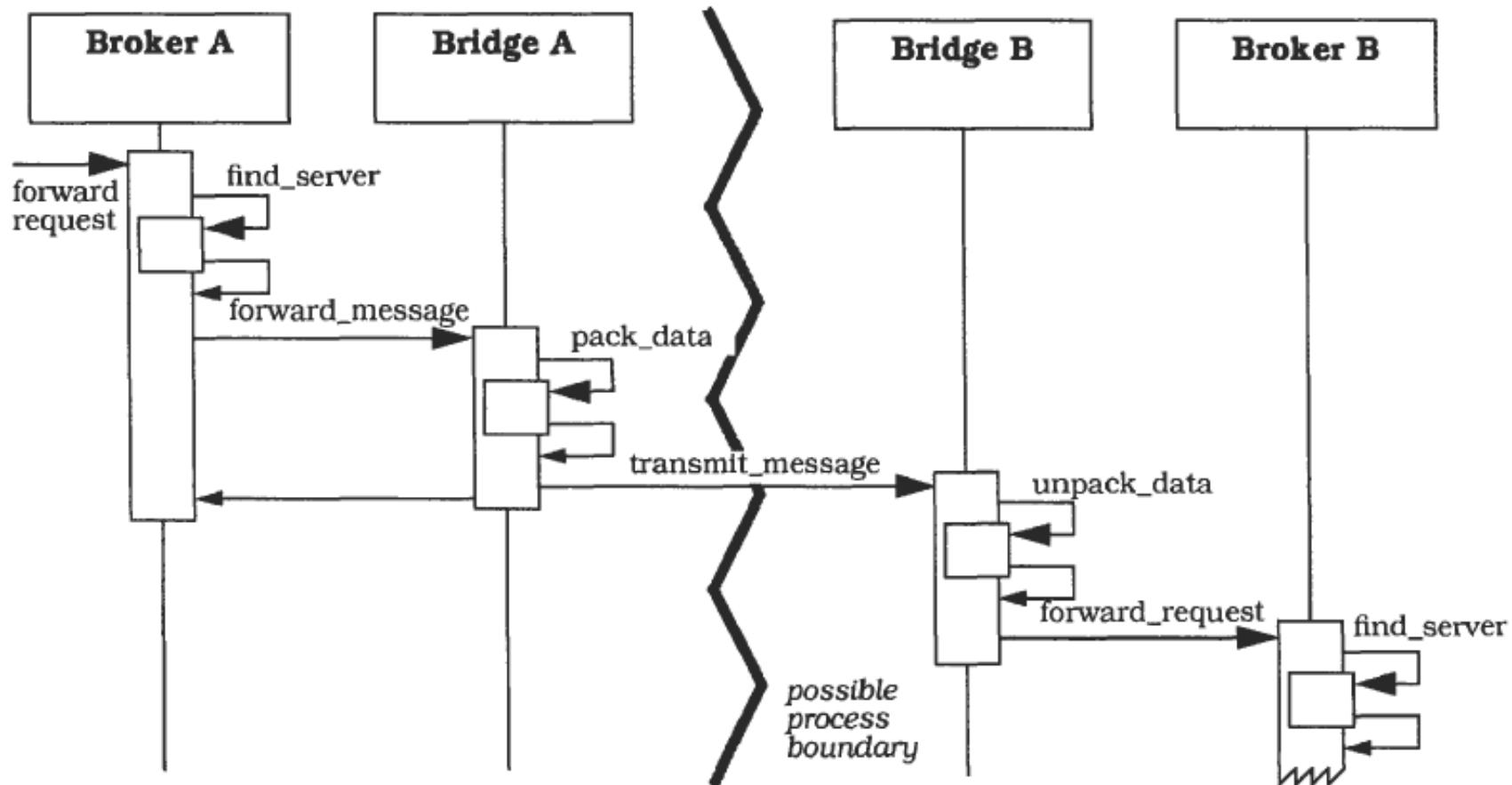
# SCENARIO I – SERVER REGISTRATION



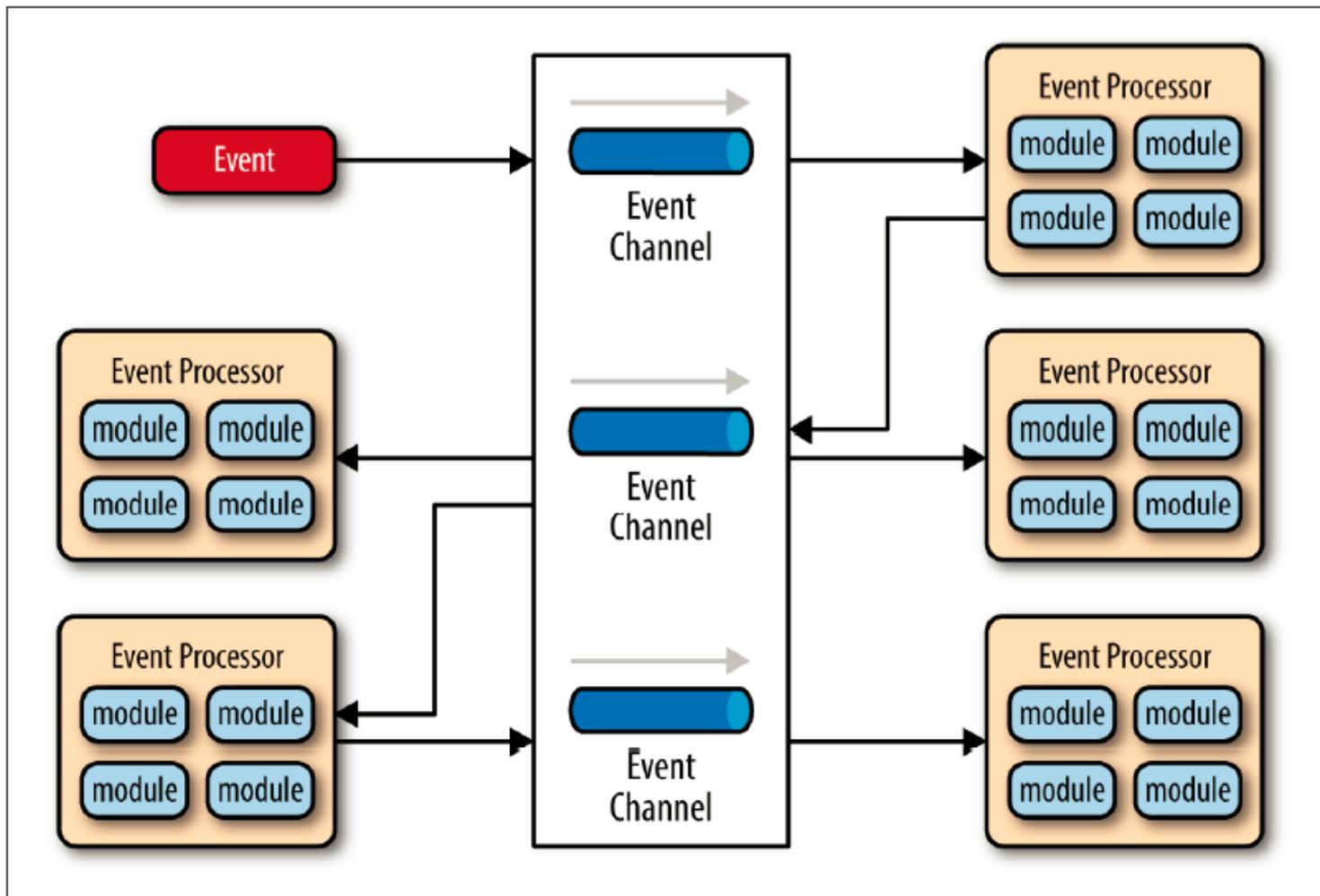
# SCENARIO II – BROKER CONNECTING CLIENT AND SERVER



# SCENARIO III – BRIDGE CONNECTING BROKERS



# EVENT-DRIVEN PERSPECTIVE



# CONSEQUENCES

## Benefits

- Location transparency
- Changeability and extensibility of components
- Portability of a Broker System
- Interoperability between Broker Systems
- Reusability

## Liabilities

- Reliability – remote process availability, lack of responsiveness, broker reconnection
- Lack of atomic transactions for a single business process

# CONSEQUENCES ON QUALITY ATTRIBUTES

Quality Attribute	Issues
Availability	To build high availability architectures, brokers must be replicated. This is typically supported using similar mechanisms to messaging and publish-subscribe server clustering.
Failure handling	As brokers have typed input ports, they validate and discard any messages that are sent in the wrong format. With replicated brokers, senders can fail over to a live broker should one of the replicas fail.
Modifiability	Brokers separate the transformation and message routing logic from the senders and receivers. This enhances modifiability, as changes to transformation and routing logic can be made without affecting senders or receivers.
Performance	Brokers can potentially become a bottleneck, especially if they must service high message volumes and execute complex transformation logic. Their throughput is typically lower than simple messaging with reliable delivery.
Scalability	Clustering broker instances makes it possible to construct systems scale to handle high request loads.

# MEDIATOR

- for events that have multiple steps and require some level of orchestration to process the event

Event queue (hosts initial events)

- Message queue
- Web service endpoint

Events

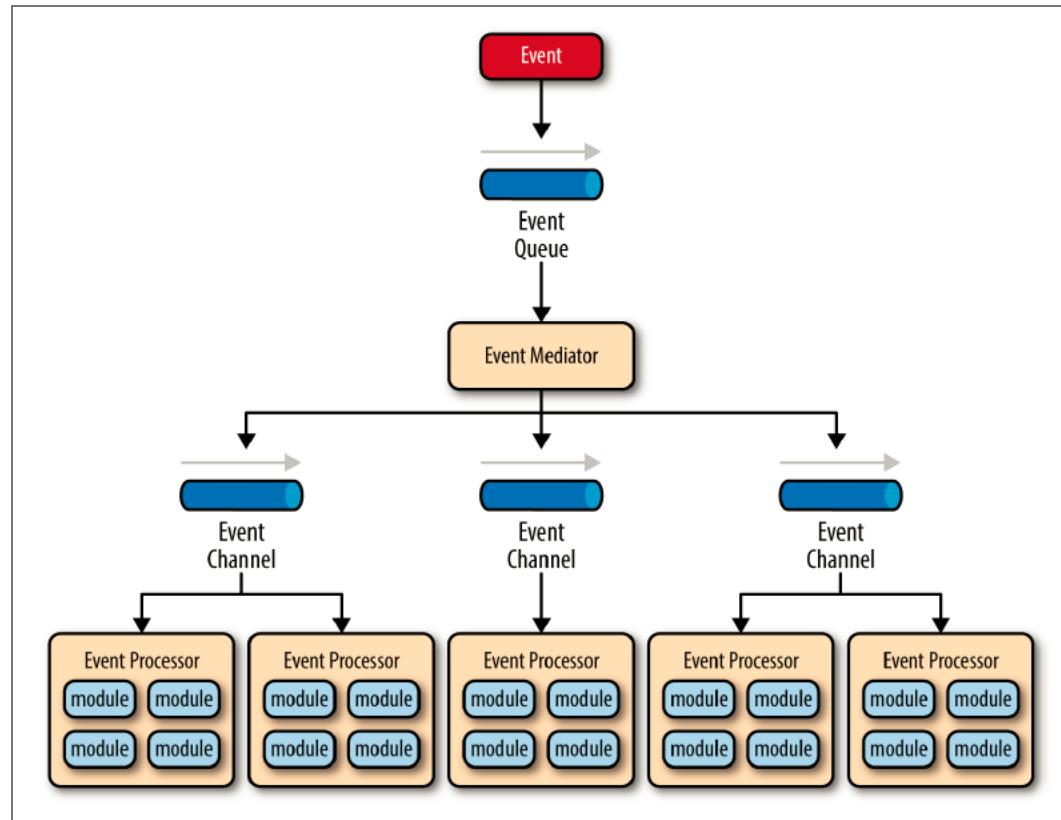
- Initial
- Processing

Event channel (passes processing events)

- Message queue
- Message topic

Event processor

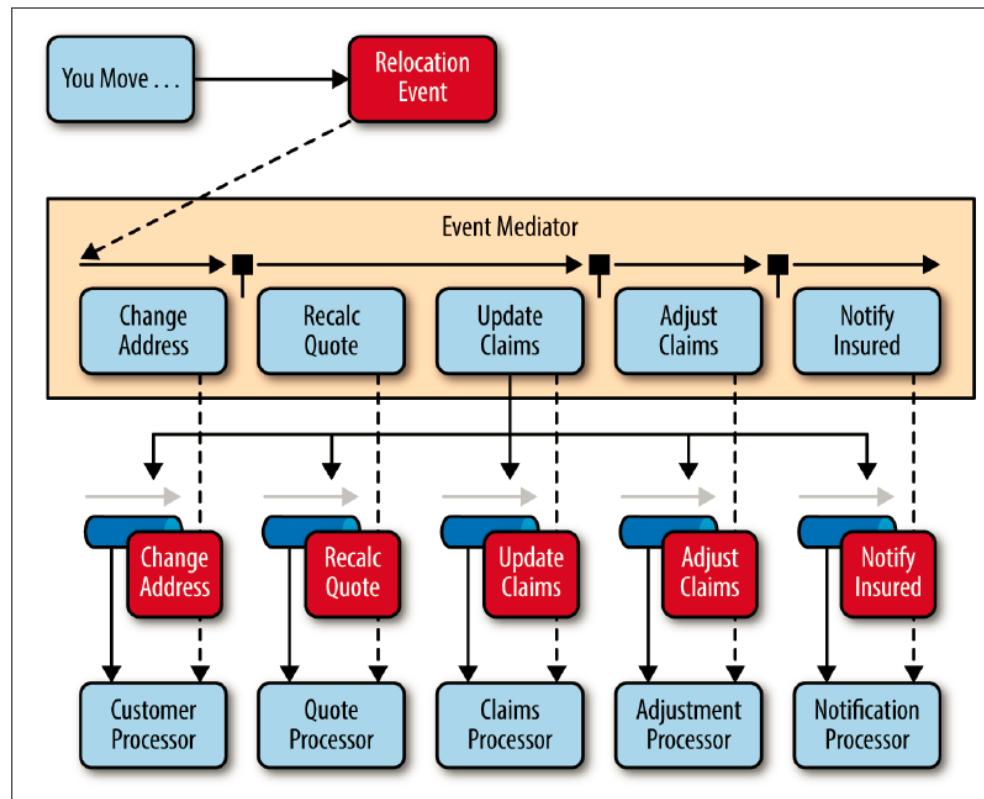
- self-contained, independent, highly decoupled business logic components



# EXAMPLE (INSURANCE APP)

## Event mediator

- open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB
- BPEL (business process execution language) coupled with a BPEL engine (ex. Apache ODE)
- business process manager (BPM) (ex. jBPM)



# INTERACTIVE SYSTEMS

## Context

- **Interactive applications with a flexible human-computer interface.**

## Problem

- User interfaces are especially prone to change requests.
- Different users place conflicting requirements on the user interface.

# MODEL VIEW CONTROLLER (MVC)

## Solution

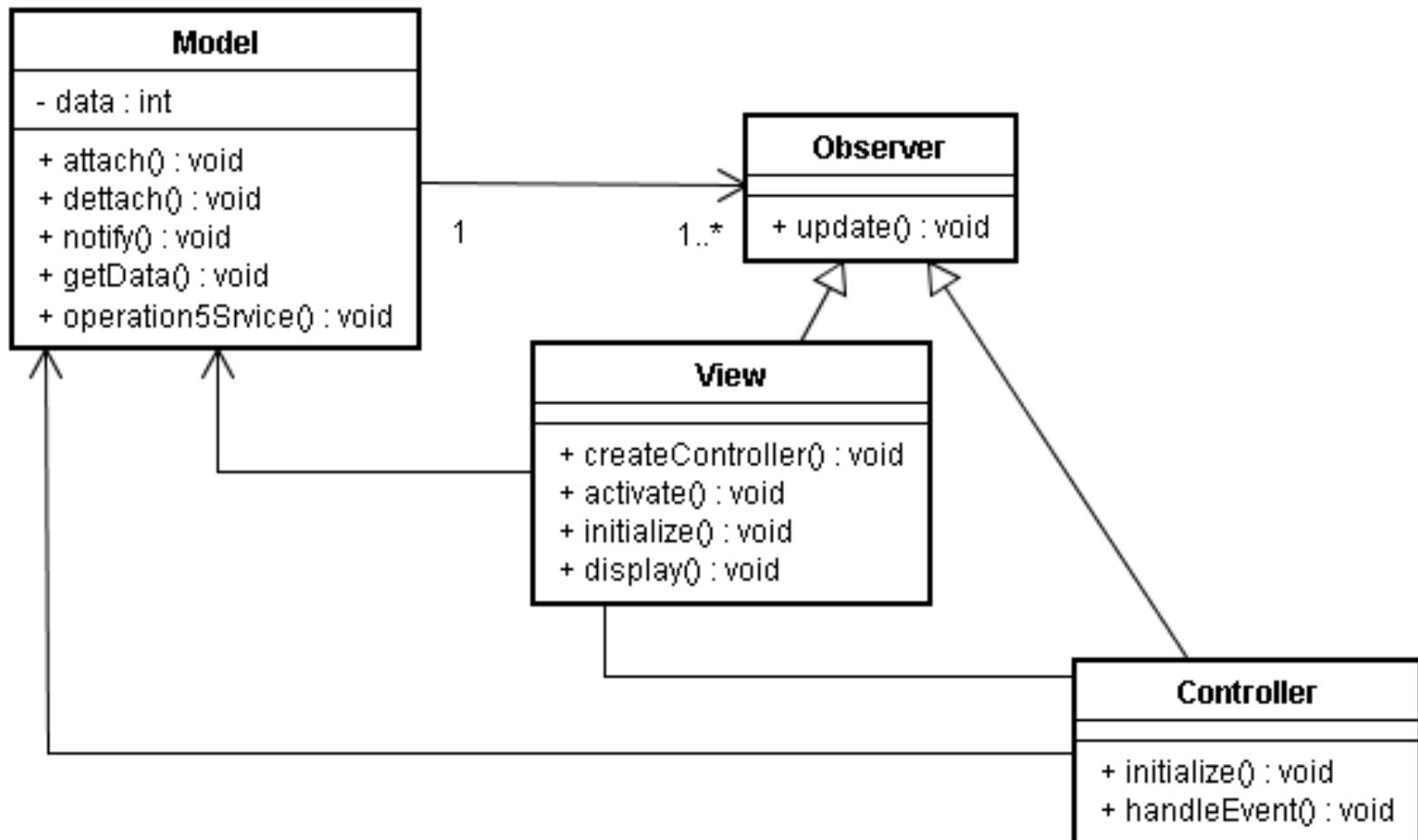
- 3 areas: handle input, processing, output
- The **Model** component encapsulates core data and functionality (processing).
- **View** components display information to the user. A view obtains the data from the model (output).
- Each view has an associated **Controller** component. Controllers handle input.

# MVC

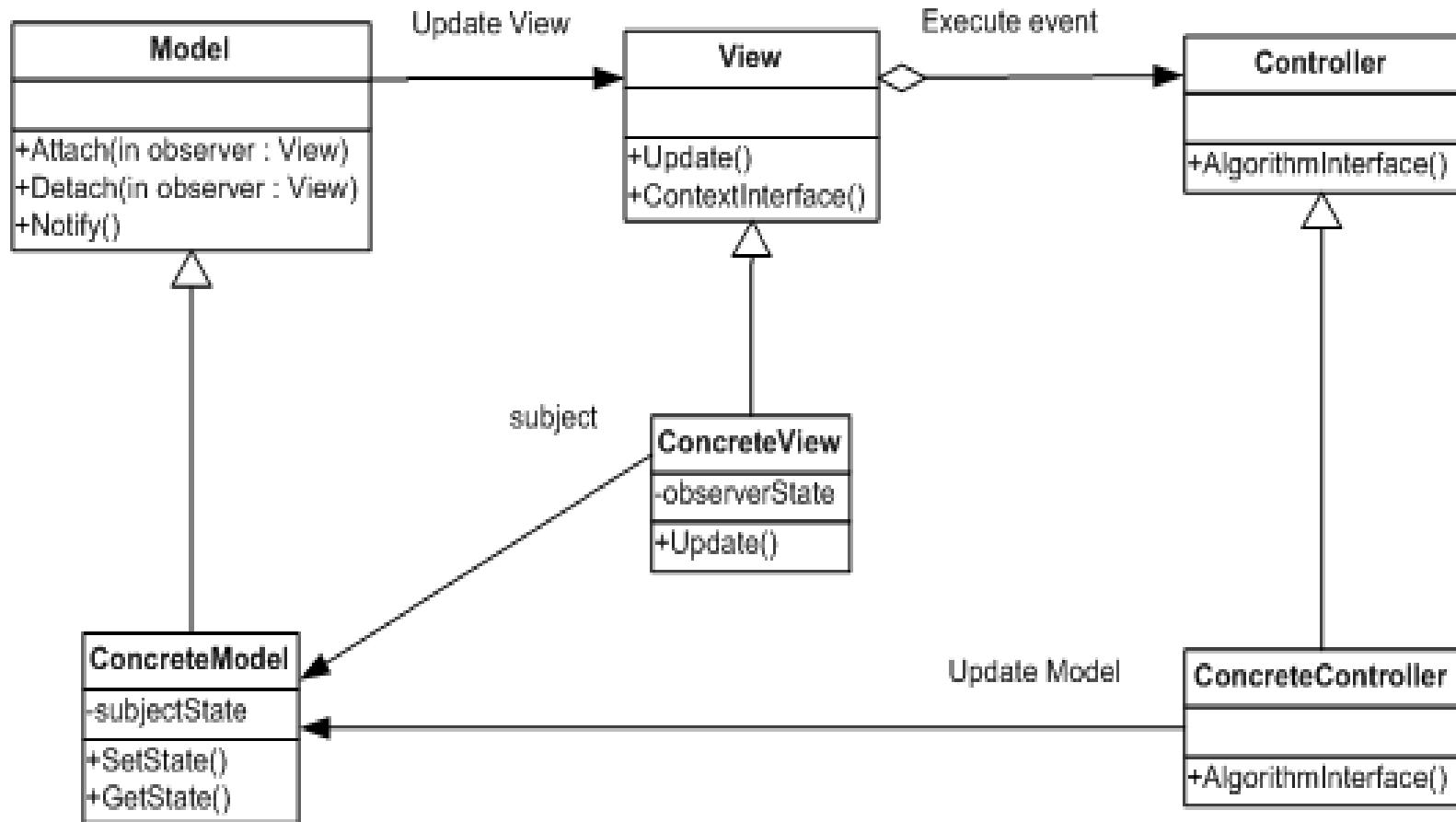
## Constraints

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behavior of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

# MVC STRUCTURE



# MVC MORE DETAILED



# THE MODEL

- encapsulates and manipulates the **domain data** to be rendered
- has **no idea how to display** the information is has **nor does it interact** with the user or receive any user input
- encapsulates the functionality necessary to manipulate, obtain, and deliver that data to others, *independent* of any user interface or any user input device

# THE VIEW

- Is a **specific visual rendering** of the information contained in the model (graphical, text-based).
- **Multiple views** may present multiple renditions of the data in the model
- Each view is **dependent** of a model
- When the model changes, all dependent views are updated

# THE CONTROLLER

- handles user input. They “listen” for user direction, and *handle* requests using the model and views
- often watches mouse events and keyboard events
- allows the decoupling of the model and its views, allowing views to simply render data and models to simply encapsulate data
- is “paired up” with collections of view types, so that a “pie graph” view would be associated with its own “pie graph” controller, etc.
- **The behavior of the controller is dependent upon the state of the model**

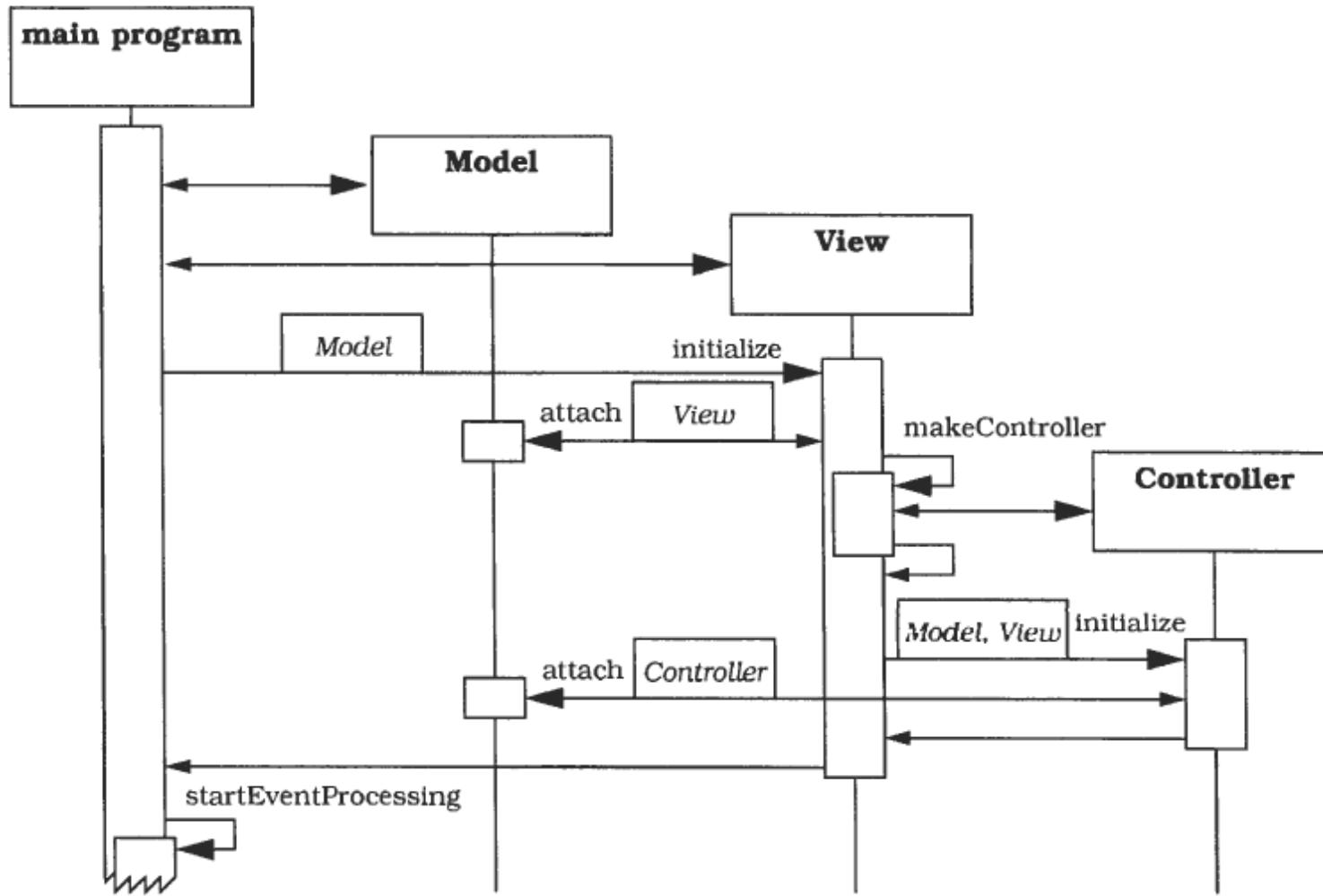
# HOW DOES THIS WORK?

- The **model** has a list of **views** it supports
- Each **view** has a reference to its **model**, as well as its supporting **controller**
- Each **controller** has a reference to the **view** it controls, as well as to the **model** the view is based on. However, models know nothing about controllers.
- On user input, the controller notifies the model which in turn notifies its **views** of a change

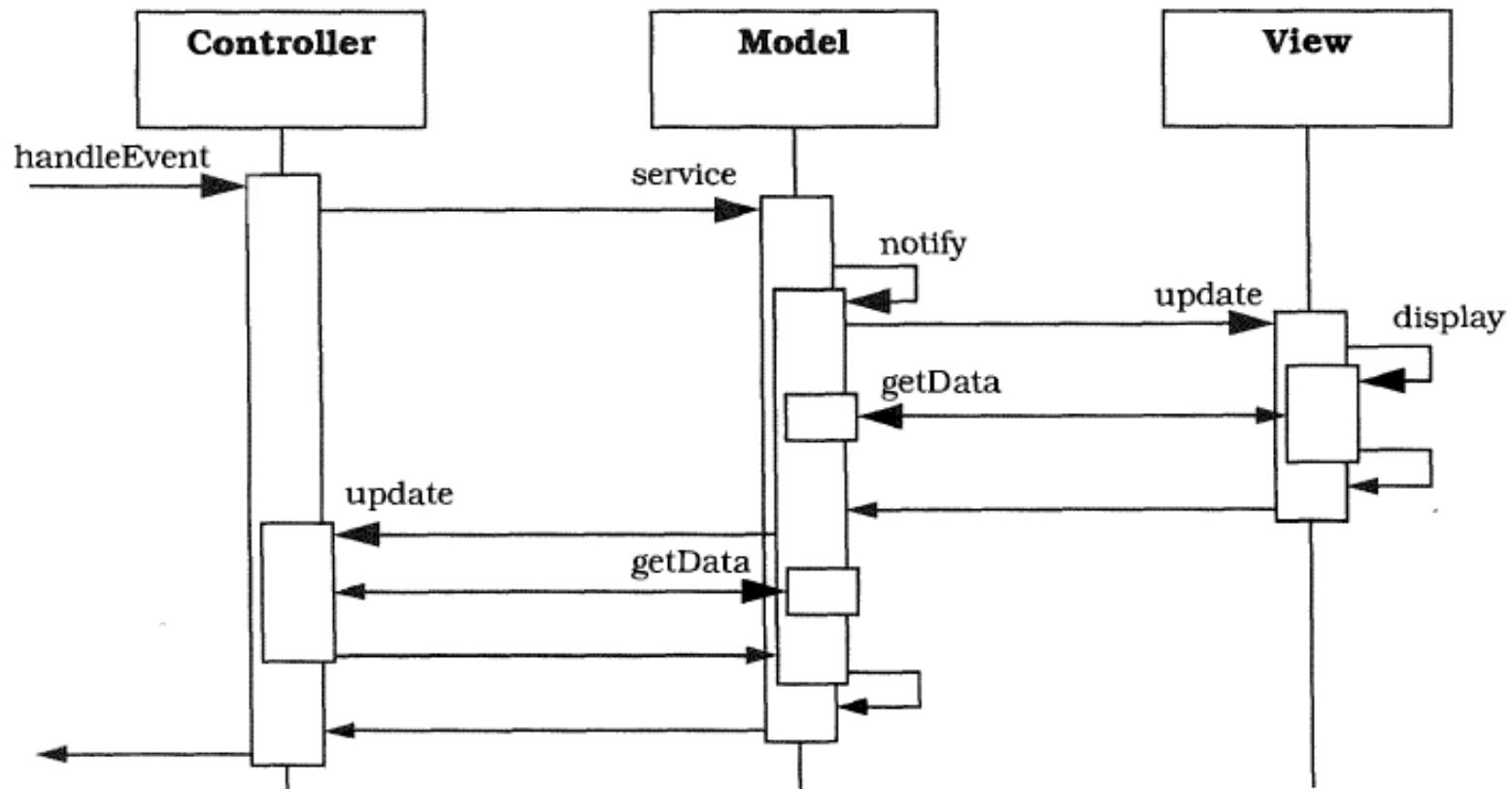
Changing a controller's view will give a different *look*.

Changing a view's controller will give a different *feel*.

# SCENARIO I – CREATING THE M,V,C OBJECTS



# SCENARIO II – EVENT HANDLING



# CONSEQUENCES

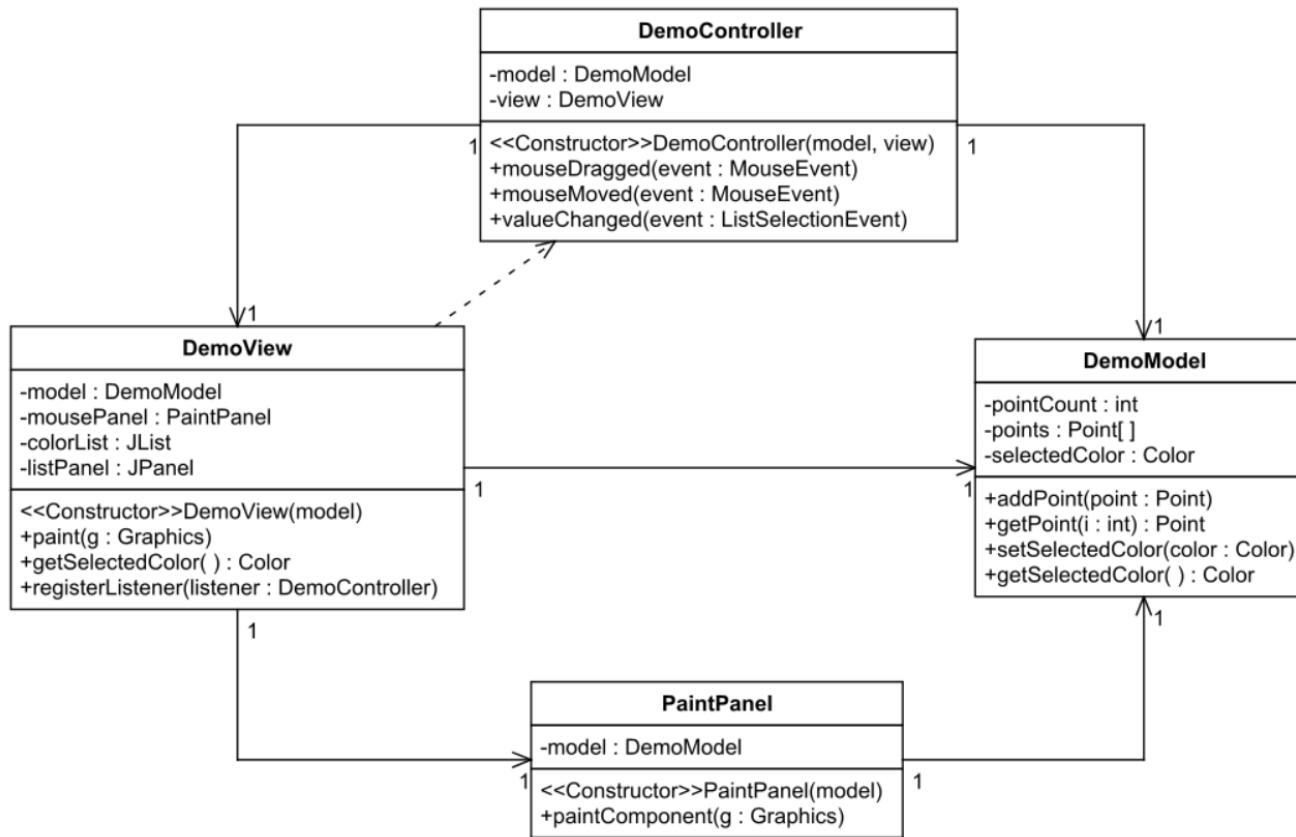
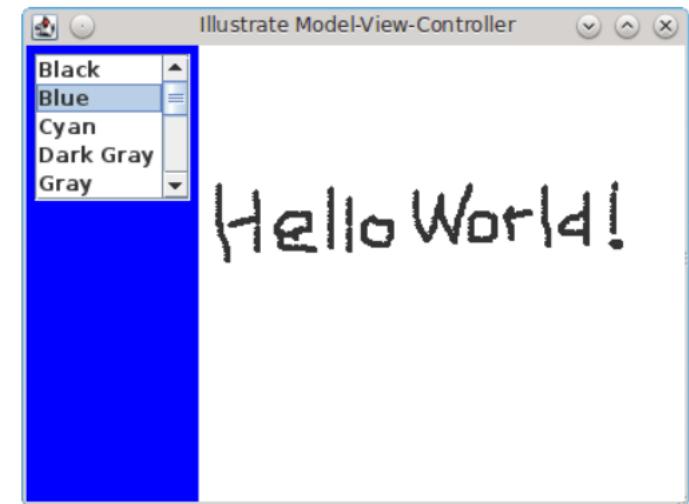
## Benefits

- Multiple views of the same model.
- Synchronized views.
- 'Pluggable' views and controllers
- Exchangeability of 'look and feel'.
- Framework potential.

## Liabilities

- Increased complexity.
- Potential for excessive number of updates.
- Intimate connection between view and controller.
- Close coupling of views and controllers to a model
- Inefficiency of data access in view.
- Inevitability of change to view and controller when porting.

# MVC EXAMPLE



# MODEL VIEW PRESENTER

## Intent

- Separation between
  - Data
  - Business Logic
  - UI
- Enforce single responsibility: M-V-P
- Reduce coupling
- Facilitate isolated testing

# MVP STRUCTURE

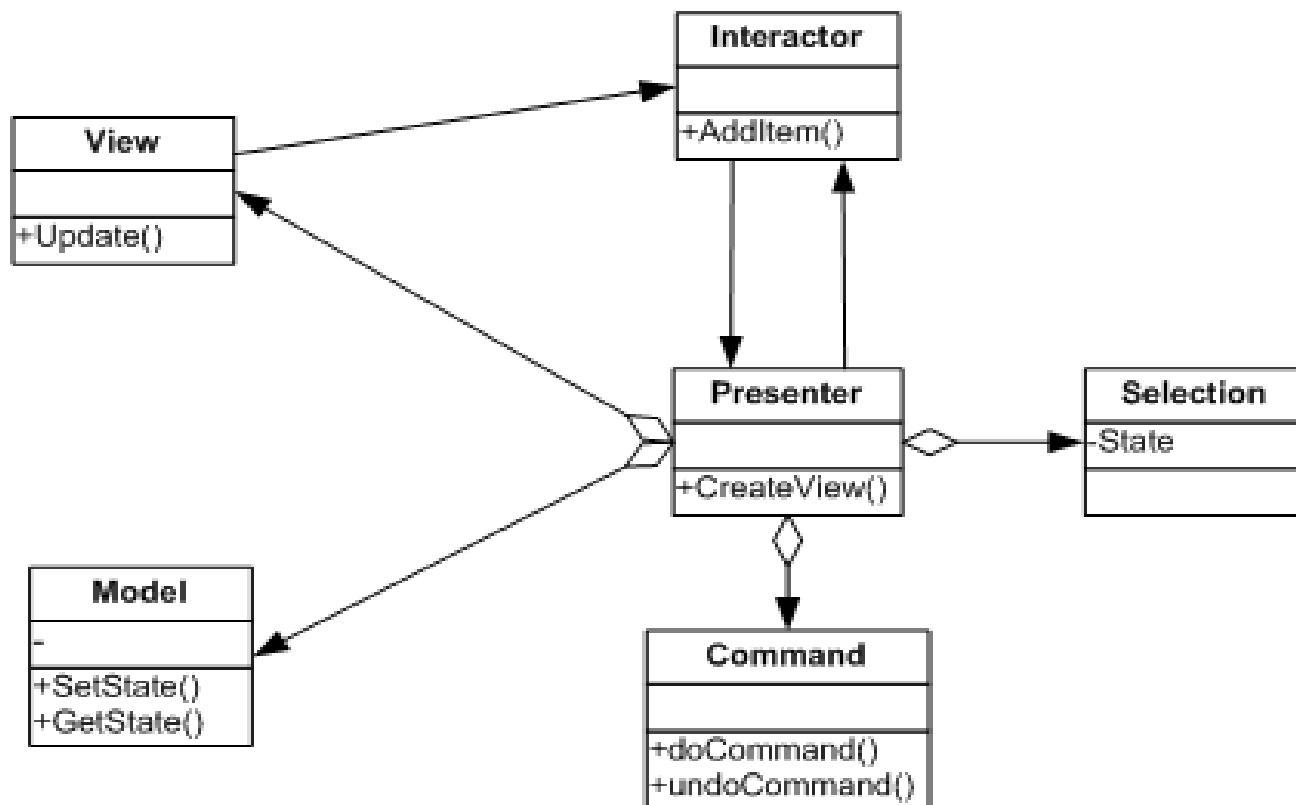
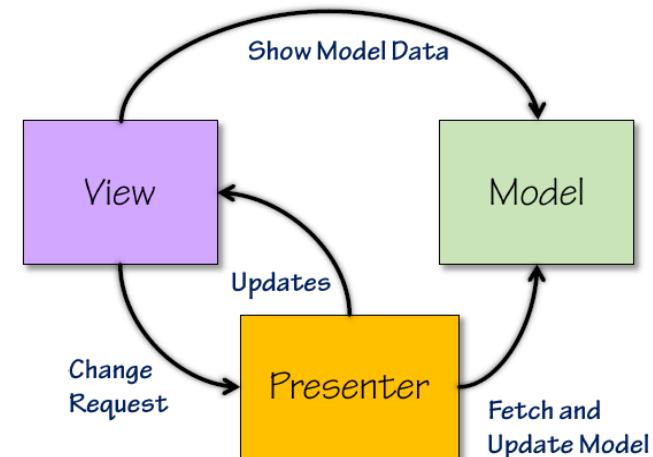
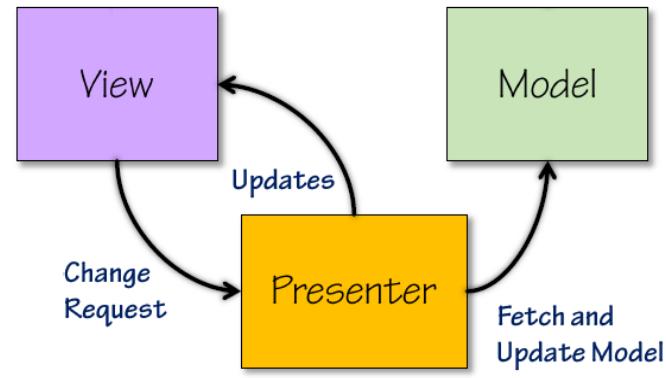


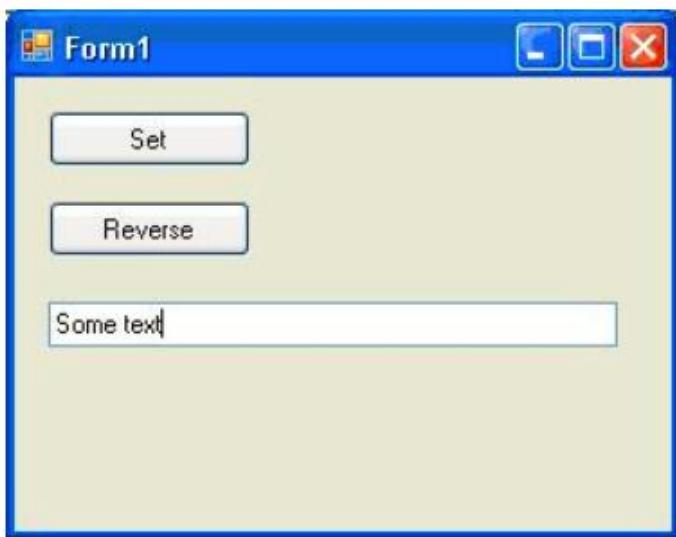
Figure 6: MVP

# VARIANTS

- View and Model are decoupled
  - Presenter coordinates events between View and Model
  - Passive View
- 
- View and Model no longer separated
  - Presenter coordinates events between View and Model
  - Model contains only what View needs
  - Presenter interacts with Domain Layer/Data Access



# MVP EXAMPLE



```
namespace ModelViewPresenter
{
    public partial class Form1 : Form, IView
    {
        private Presenter presenter = null;
        private readonly Model m_Model;

        public Form1(Model model)
        {
            m_Model = model;
            InitializeComponent();
            presenter = new Presenter(this, m_Model);
            SubscribeToModelEvents();
        }

        public string TextValue
        {
            get
            {
                return textBox1.Text;
            }
            set
            {
                textBox1.Text = value;
            }
        }

        private void Set_Click(object sender, EventArgs e)
        {
            presenter.SetTextValue();
        }

        private void Reverse_Click(object sender, EventArgs e)
        {
            presenter.ReverseTextValue();
        }

        private void SubscribeToModelEvents()
        {
            m_Model.TextSet += m_Model_TextSet;
        }

        void m_Model_TextSet(object sender, CustomArgs e)
        {
            this.textBox1.Text = e.m_after;
            this.label1.Text = "Text changed from " + e.m_before + " to " + e.m_after;
        }
    }
}
```

Model injection in constructor

Creates Presenter

```

namespace ModelViewPresenter
{
    public class Presenter
    {
        private readonly IView m_View;
        private IModel m_Model;

        public Presenter(IView view, IModel model)
        {
            this.m_View = view;
            this.m_Model = model;
        }

        public void ReverseTextValue()
        {
            string reversed = ReverseString(m_View.TextValue);
            m_Model.Reverse(reversed);
        }

        public void SetTextValue()
        {
            m_Model.Set(m_View.TextValue);
        }

        private static string ReverseString(string s)
        {
            char[] arr = s.ToCharArray();
            Array.Reverse(arr);
            return new string(arr);
        }
    }
}

```

```

namespace ModelViewPresenter
{
    public class Model : IModel
    {
        private string m_TextValue;

        public event EventHandler<CustomArgs> TextSet;
        public event EventHandler<CustomArgs> TextReverse;

        public Model()
        {
            m_TextValue = "";
        }

        public void Set(string value)
        {
            string before = m_TextValue;
            m_TextValue = value;
            RaiseTextSetEvent(before, m_TextValue);
        }

        public void Reverse(string value)
        {
            string before = m_TextValue;
            m_TextValue = value;
            RaiseTextSetEvent(before, m_TextValue);
        }

        public void RaiseTextSetEvent(string before, string after)
        {
            TextSet(this, new CustomArgs(before, after));
        }
    }

    public class CustomArgs : EventArgs
    {
        public string m_Before { get; set; }
        public string m_After { get; set; }

        public CustomArgs(string before, string after)
        {
            m_Before = before;
            m_After = after;
        }
    }
}

```

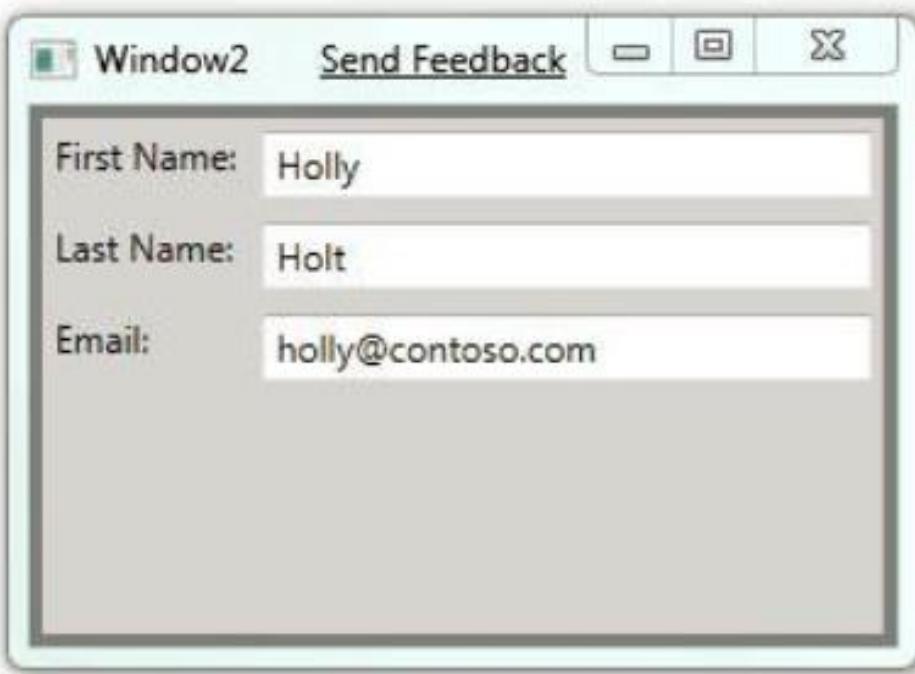
# MVVM

## Motivation

- Need to share a project with a designer, and flexibility for design work and development work to happen near-simultaneously
- Thorough unit testing for your solutions
- Important to have reusable components, both within and across projects
- Flexibility to change the user interface without having to refactor other logic in the code base

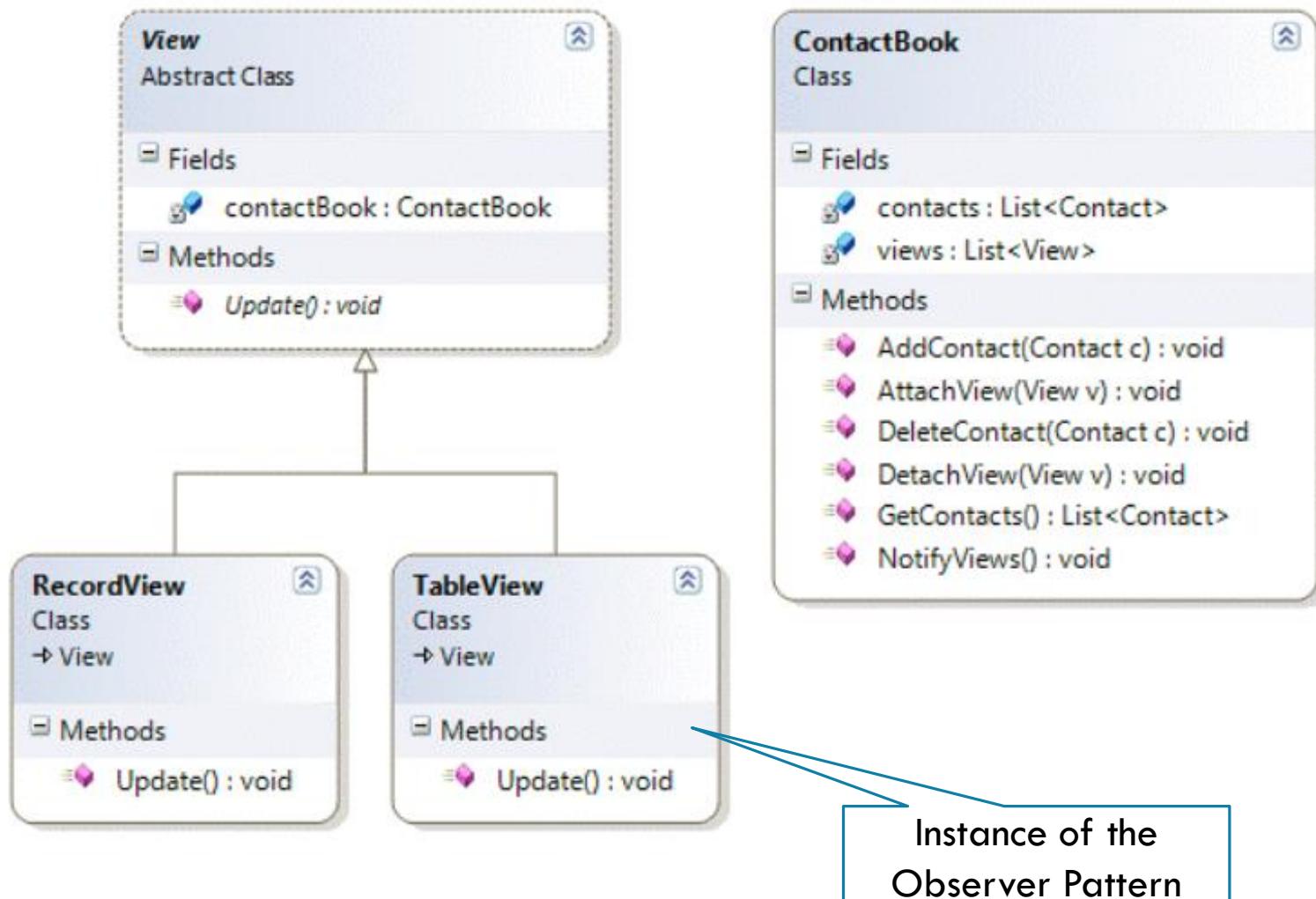
# MV\* EXAMPLE

## UI Contacts App



[[https://blogs.msdn.microsoft.com/ivo\\_manolov/2012/03/17/model-view-viewmodel-mvvm-applications-general-introduction/](https://blogs.msdn.microsoft.com/ivo_manolov/2012/03/17/model-view-viewmodel-mvvm-applications-general-introduction/)]

# M-V SEPARATION



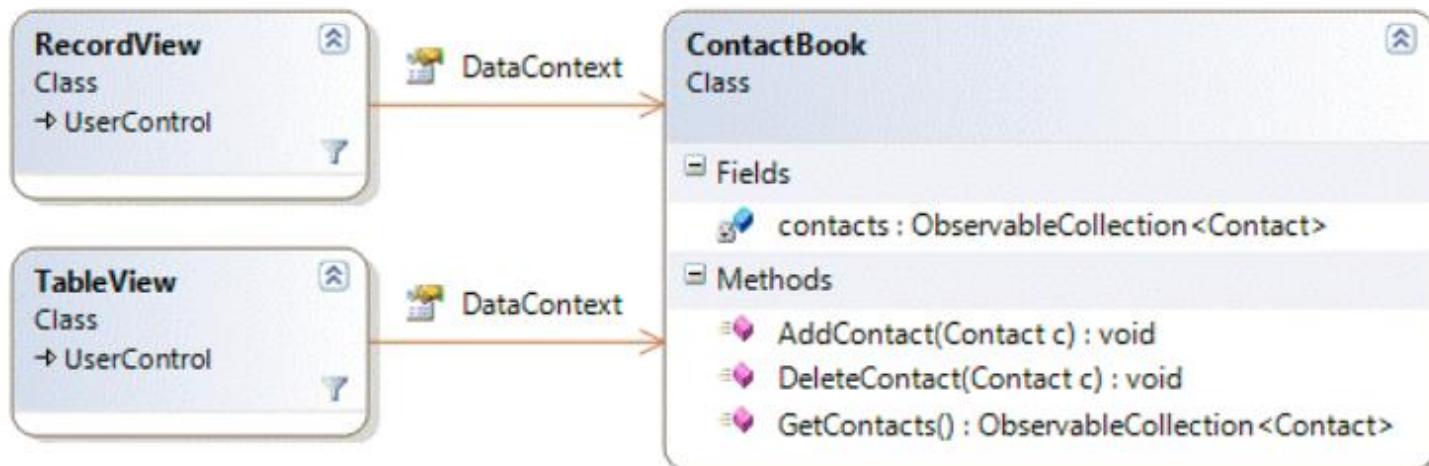
# SIMPLISTIC WPF /SILVERLIGHT DESIGN

## New features:

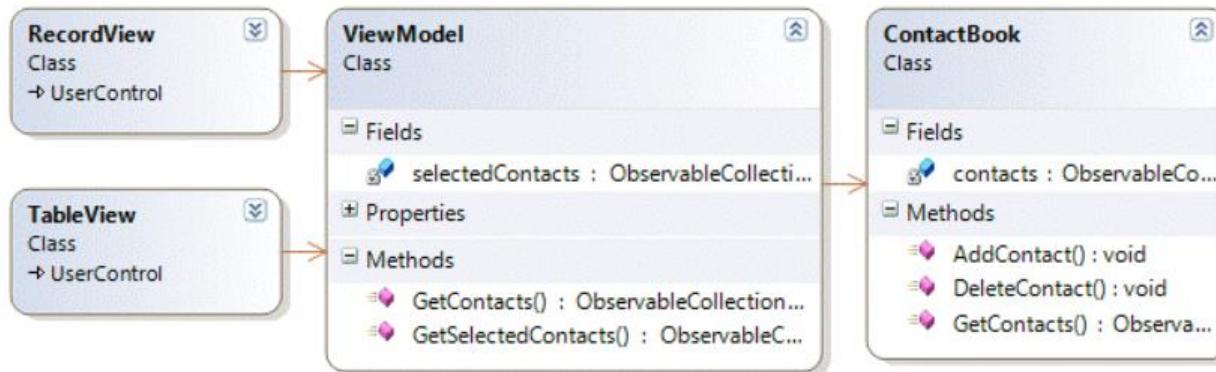
1. Track the selected item, so that we update RecordView whenever the selection in TableView changes, and vice versa.
2. Enable or disable parts of the UI of RecordView and TableView based on some rule (for example, highlight any entry that has an e-mail in the live.com domain).

*Databinding* is the ability to bind UI elements to any data.

*Commands* provide the ability to notify the underlying data of changes in the UI.

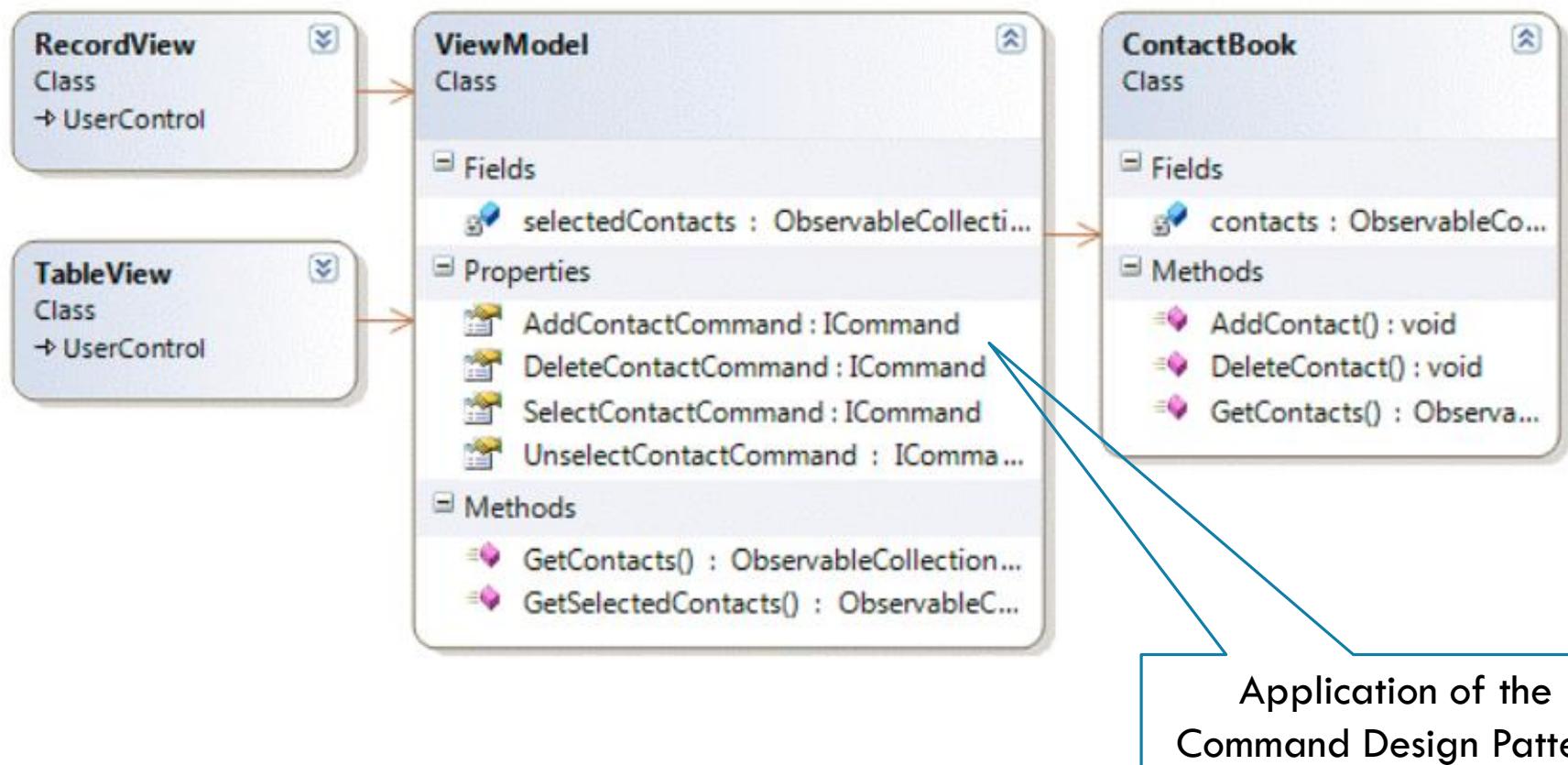


# MVVM DESIGN



- The views know of the ViewModel and bind to its data, to be able to reflect any changes in it.
- The ViewModel has no reference to the views—it holds only a reference to the model.
- For the views, the ViewModel acts both as
  - a façade to the model,
  - a way to share state between views (selectedContacts in the example).
- The ViewModel often exposes commands that the views can bind to and trigger.

# COMMANDS IN MVVM APPS



# MVVM

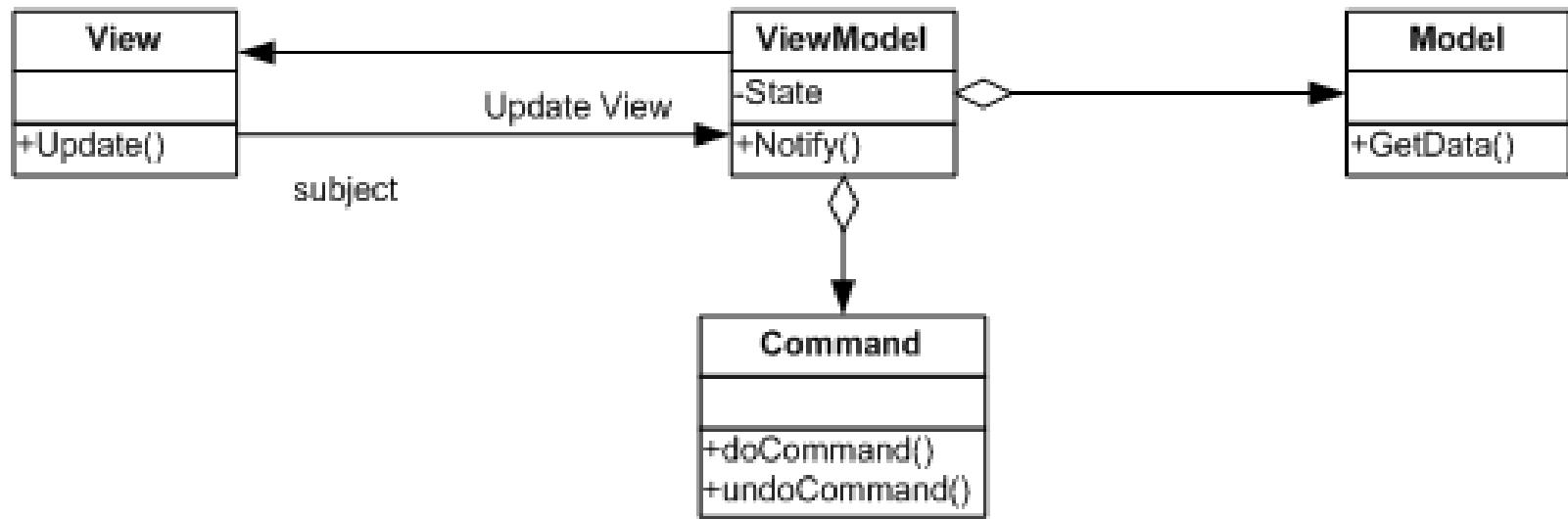
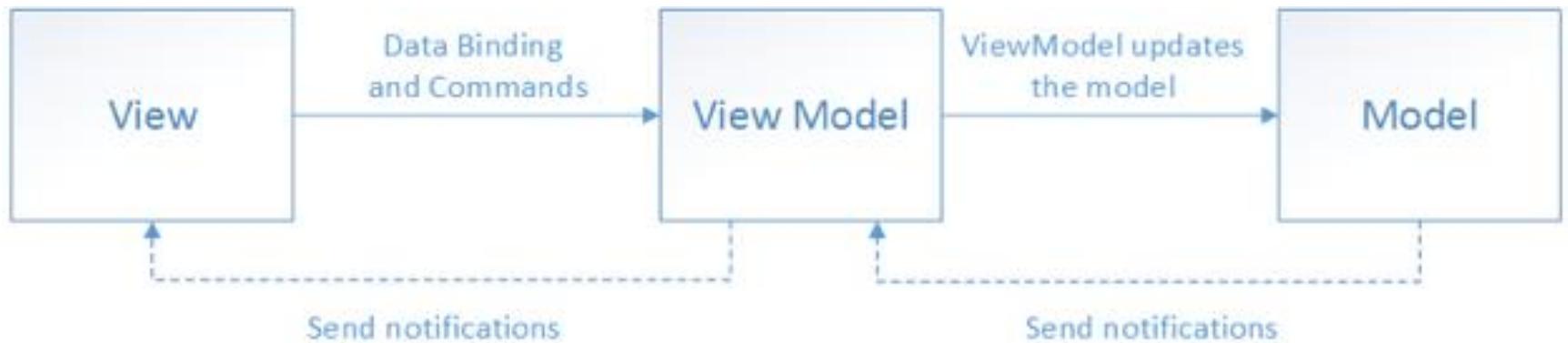


Figure 8: MVVM

# MVVM STRUCTURE



## Model

- Represents the data
  - 2 perspectives
    - OO approach: Domain Objects
    - Data-centric approach: XML, DAL, etc.
  - No knowledge of how and where it will be presented
  - Implements `INotifyPropertyChanged`

# MVVM STRUCTURE

## ViewModel

- Contains Properties, Commands, etc. to facilitate communication between View and Model
- Properties that expose instances of Model objects
- Commands and events that interact with Views for user interactions
- Implements INotifyPropertyChanged
- Pushes data up to the View

## View

- Visual display
- Buttons, windows, graphics, etc.
- Responsible for displaying data
- Not responsible for retrieving data, business logic, business rules or validation
- Contains binding extensions
  - Identify data points represented to the user
  - Point to the names of the data point properties

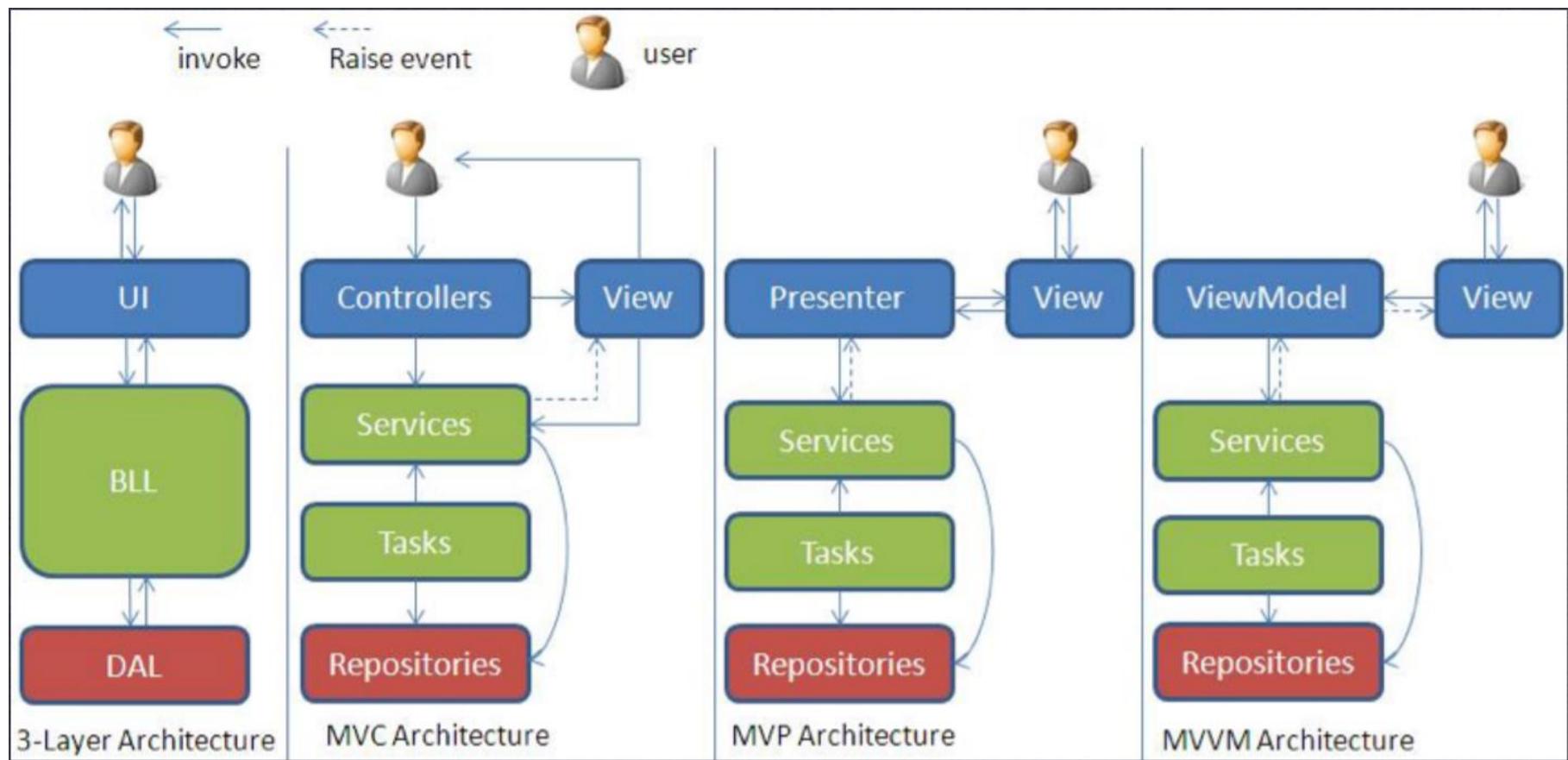
# MVVM DISCUSSION

- MVVM is a derivative of MVC that takes advantage of particular strengths of the Windows Presentation Foundation (WPF) architecture
- Separates the Model and the View by introducing an abstract layer between them: a “View of the Model,” or ViewModel.
- The typical relationship between a ViewModel and the corresponding Views is one-to-many, but not always.
- There are situations when one ViewModel is aware of another ViewModel within the same application. When this happens, one ViewModel can represent a collection of ViewModels.

# MVVM BENEFITS

- A ViewModel provides a **single store for presentation** policy and state, thus improving the reusability of the Model (by decoupling it from the Views) and the replaceability of the Views (by removing specific presentation policy from them).
- MVVM improves the overall **testability** of the application.
- MVVM also improves the “**mockability**” of the application.
- MVVM is a very **loosely coupled** design. The View holds a reference to the ViewModel, and the ViewModel holds a reference to the Model. The rest is done by the data-binding and commanding infrastructure of WPF.

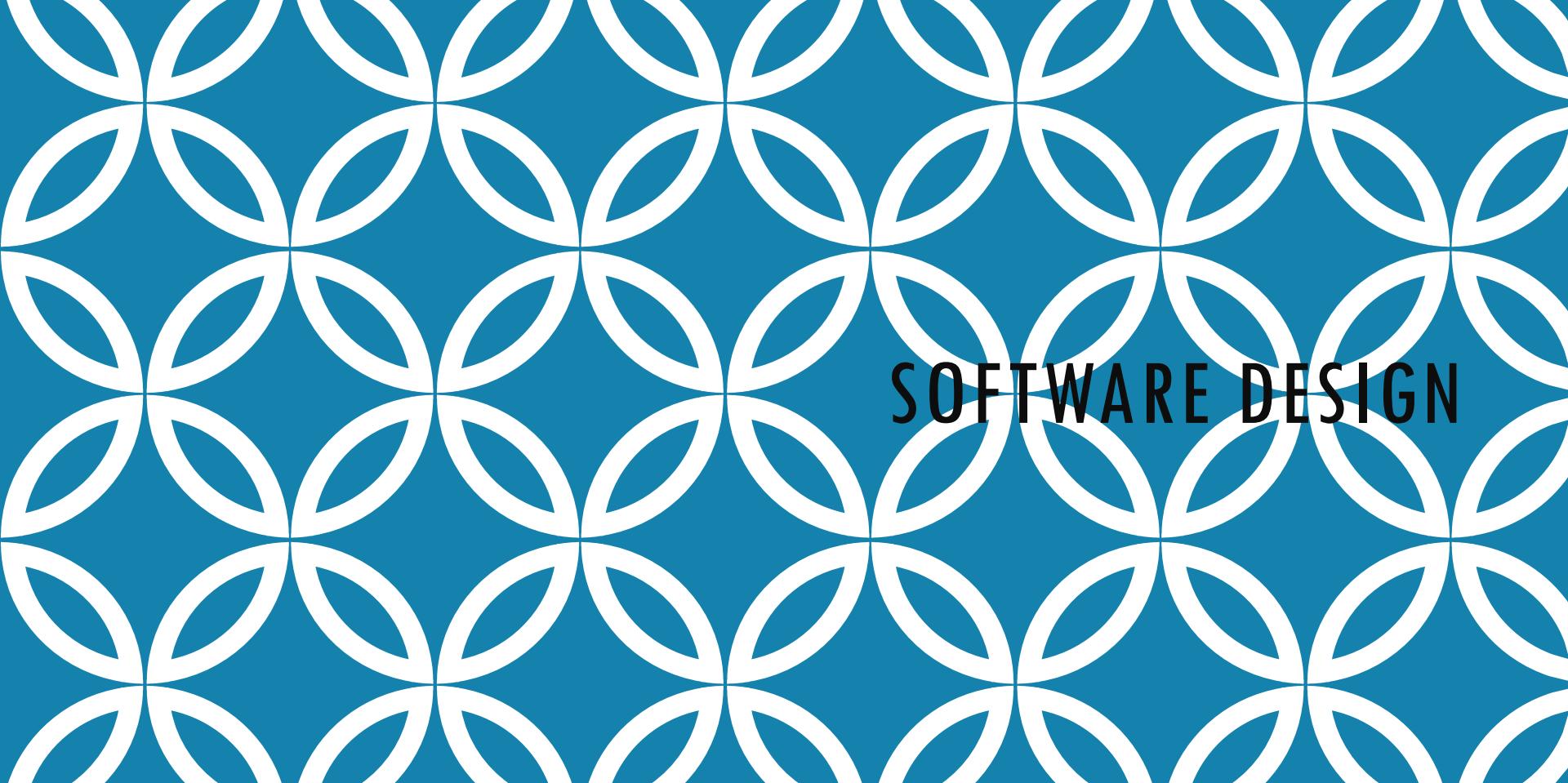
# MV\* IN A NUTSHELL



# WRAP-UP

## Architectural Patterns

- High-level design of an architecture
- Patterns may have several variants
- Need to be adapted/customized for the specific needs
- Use more basic design patterns



# SOFTWARE DESIGN

Business logic

# CONTENT

- Organizing the Business Logic
  - Domain driven design
  - Service driven design

# REFERENCES

- Martin Fowler et. al, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2003 [Fowler]
- Abel Avram, Floyd Marinescu, *Domain Driven Design Quickly*, InfoQ Enterprise Software Development, 2006 [DDDQ]
- Vaughn Vernon, *Domain Driven Design Distilled*, Addison Wesley, 2016. [DDDD]
- Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*
- Mark Richards, *Microservices vs. Service-Oriented Architecture* O'Reilly, 2016
- David Patterson, Armando Fox, *Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing*, Alpha Ed.[Patterson]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- Armando Fox, David Patterson, and Koushik Sen, *SaaS Course Stanford*, Spring 2012 [Fox]
- Jacques Roy, *SOA and Web Services*, IBM
- Mark Bailey, *Principles of Service Oriented Architecture*, 2008
- Erl, Thomas. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Pearson Education. 2016
- Erl, Thomas. *SOA Design Patterns*, Prentice Hall, 2009.

<http://soapatterns.org>

# PATTERNS FOR ENTERPRISE APPLICATIONS

## [FOWLER]

### Enterprise Applications

- Persistent data
- Volume of data
- Concurrent access
- Complex user interface
- Integration with other applications
  - Conceptual dissonance
- Business logic

# ENTERPRISE APPLICATIONS

Example: B2C online retailer

- High volume of users: scalability

Example: processing of leasing agreements

- Complicated business logic
- Rich-client interface
- Complicated transaction behavior

Example: expense tracking for small company

# PRINCIPAL LAYERS

See pattern Layers in [POSA]

Here: applied to enterprise applications

Presentation logic

- Interaction with user
- Command-line or rich client or Web interface

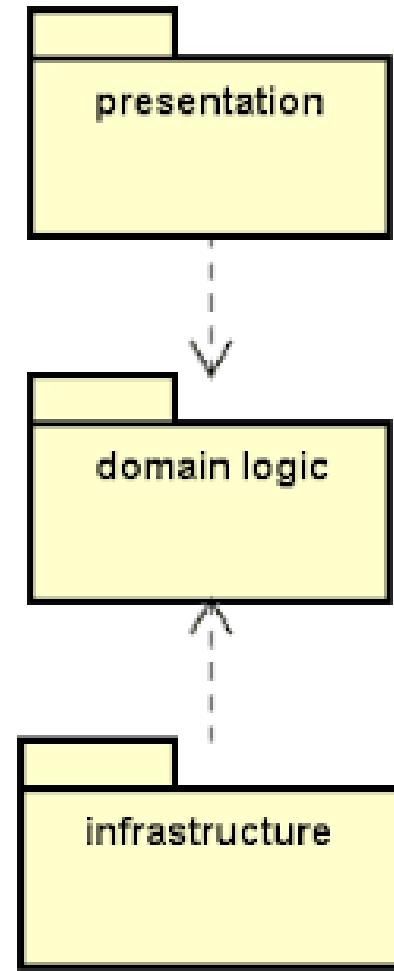
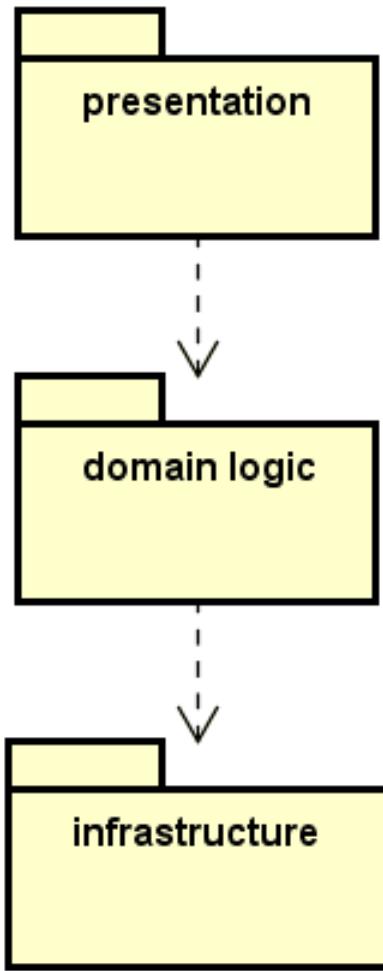
Business logic

- Validation of input and calculation of results

Data access logic

- Communication with databases and other applications

# DEPENDENCIES?



# MORE DETAILED LAYERING

Presentation

---

Controller/Mediator

---

Business Logic

---

Data Mapping

---

Data Source

# BUT FIRST...

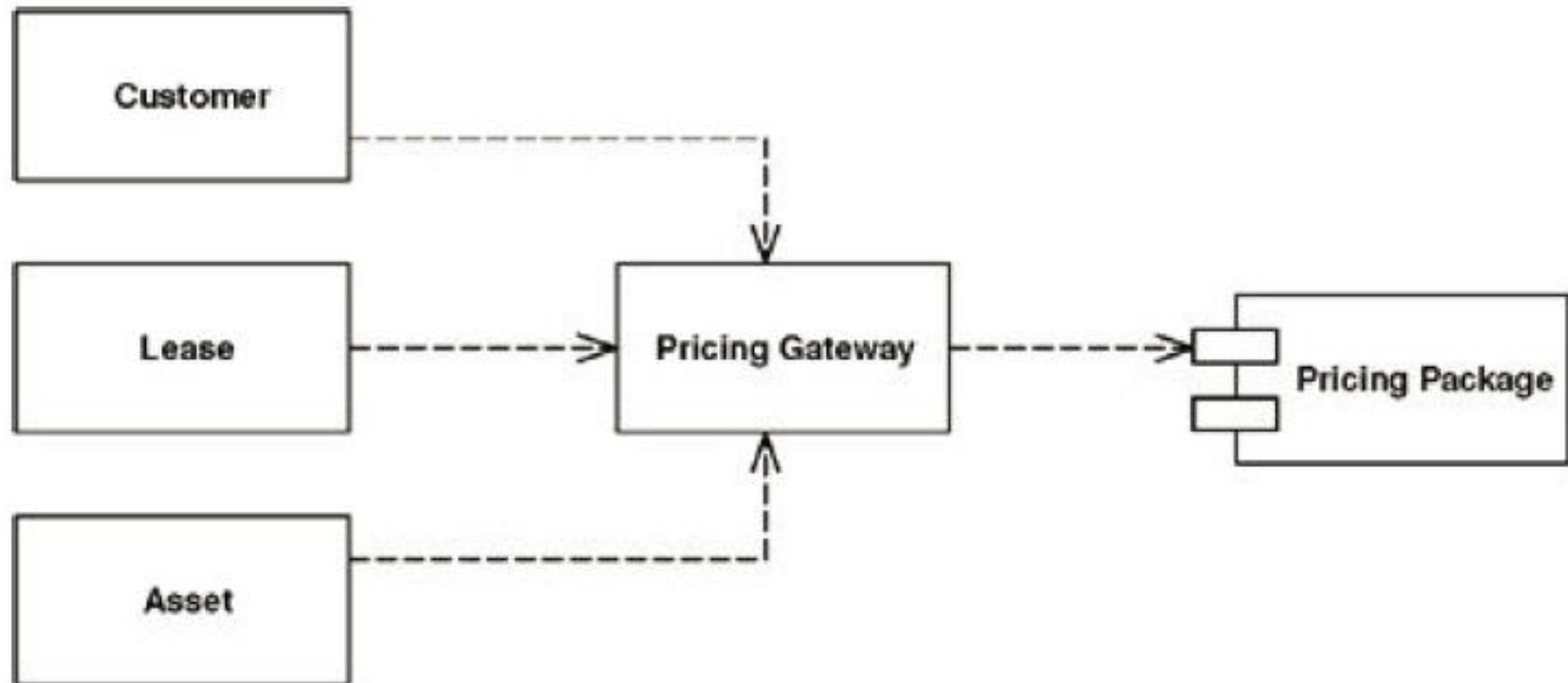
Some basic patterns

Gateway

Record Set

# GATEWAY

*An object that encapsulates access to an external system or resource*



# GATEWAY – HOW IT WORKS

- External resources - each with its own API
- Wraps all the API specific code into a Gateway whose interface is defined by the client
- Should be simple and minimal. Additional complex logic should be placed in the client
- Can be generated.
- Examples: Gateways to access Databases (DAOs)

# GATEWAY - BENEFITS

- Easier handling of awkward API's
- Easier to swap out one kind of resource for another
- Easier to test by giving you a clear point to deploy *Service Stubs* (A *stand-in implementation of an external service*)

# GATEWAY - EXAMPLE

Build a gateway to an interface that just sends a message using the message service

```
int send(String messageType, Object[] args);
```

Confirmation message

```
messageType = 'CONFIRM';  
args[0] = id;  
args[1] = amount;  
args[2] = symbol;
```

# BETTER...

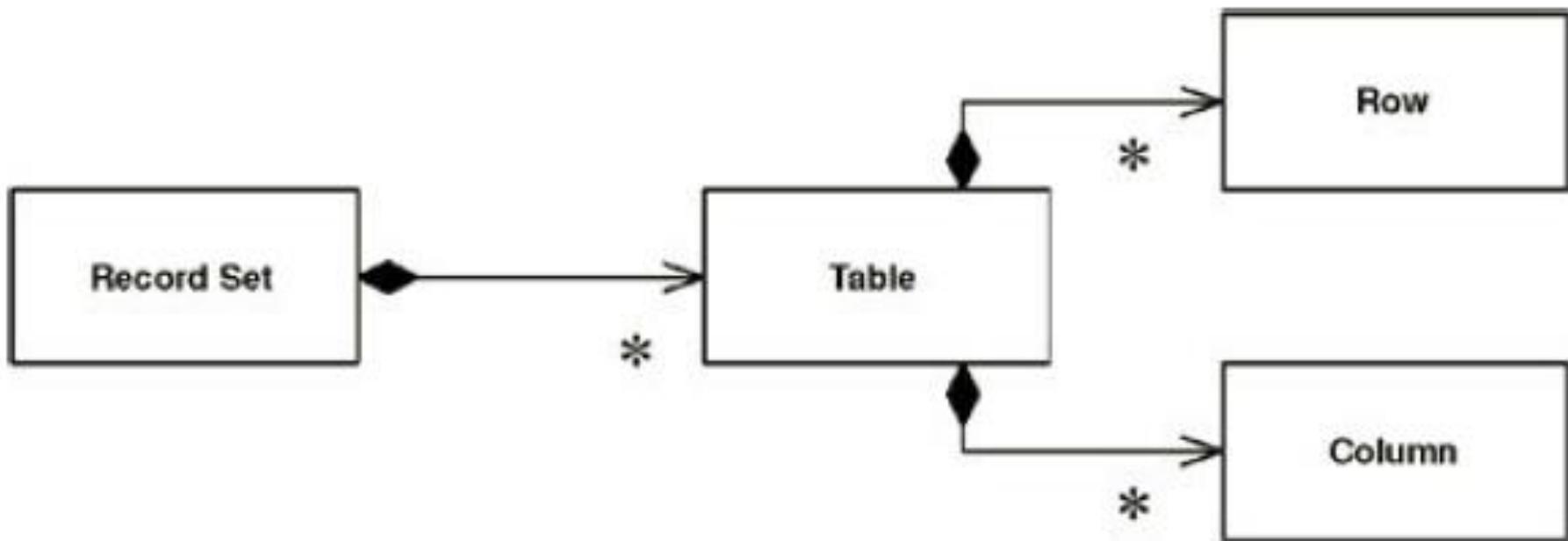
```
public void sendConfirmation(String orderID, int amount, String symbol);  
  
class Order...  
  
    public void confirm() {  
        if (isValid()) Environment.getMessageGateway().sendConfirmation(id, amount,  
symbol);  
    }  
  
class MessageGateway...  
  
    protected static final String CONFIRM = "CNFRM";  
    private MessageSender sender;  
    public void sendConfirmation(String orderID, int amount, String symbol) {  
        Object[] args = new Object[]{orderID, new Integer(amount), symbol};  
        send(CONFIRM, args);  
    }  
    private void send(String msg, Object[] args) {  
        int returnCode = doSend(msg, args);  
        if (returnCode == MessageSender.NULL_PARAMETER)
```

# GATEWAY VS. FAÇADE VS. ADAPTER

- The *façade* is usually done by the **writer of the service** for general use, while a *Gateway* is written by the **client** for their particular use.
- A *façade* always implies a **different interface** to what it's covering, while a *Gateway* may copy the wrapped interface entirely, being used for substitution or testing purposes
- Adapter alters an implementation's interface to match **another interface** which you need to work with.
- With *Gateway* there usually **isn't an existing interface**, although you might use an adapter to map an implementation to an existing *Gateway* interface. In this case the adapter is part of the implementation of the *Gateway*.

# RECORD SET

*An in-memory representation of tabular data*



# RECORD SET – HOW IT WORKS

- Provides an in memory structure that looks exactly like the result of a SQL query, but can be generated and manipulated by other parts of the system.
- Usually provided by the framework/platform

Examples:

- `ResultSet` in Java

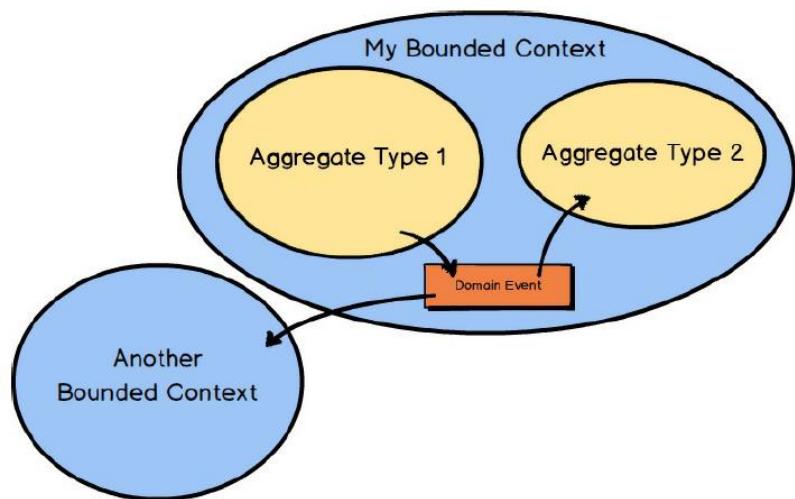
# BUSINESS LOGIC LAYER

“ ...It involves *calculations* based on inputs and stored data, *validation* of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch ...” [Fowler]

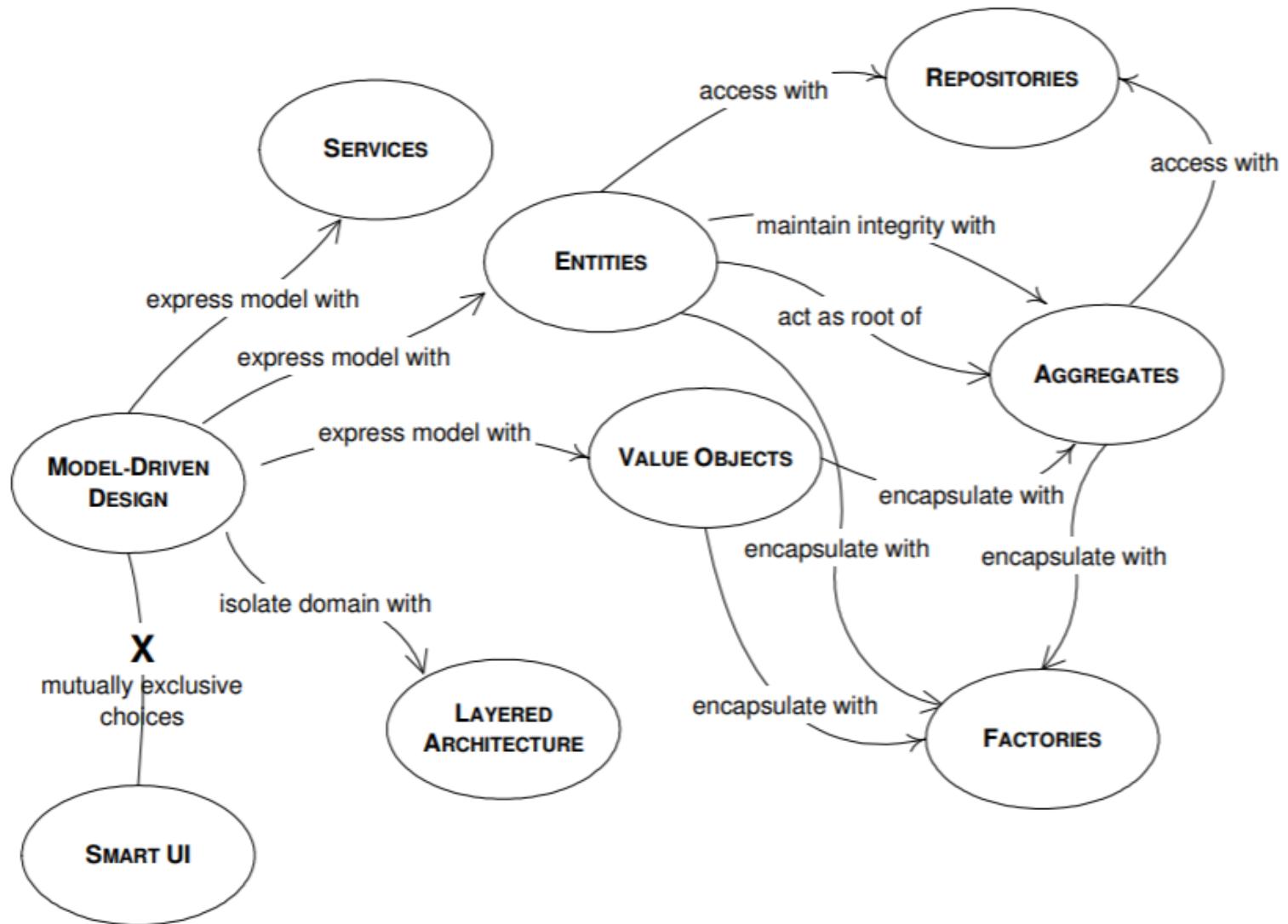
Also called Domain Layer

# DOMAIN DRIVEN DESIGN

- Strategic Design
  - Subdomains (problem perspective)
    - Core Domain
    - Support Domain
    - Generic Domain
  - Bounded Context (solution perspective)
  - Context Mapping
- Tactical Design
  - Aggregates
  - Domain Events



# MODEL DRIVEN DESIGN [DDDQ]

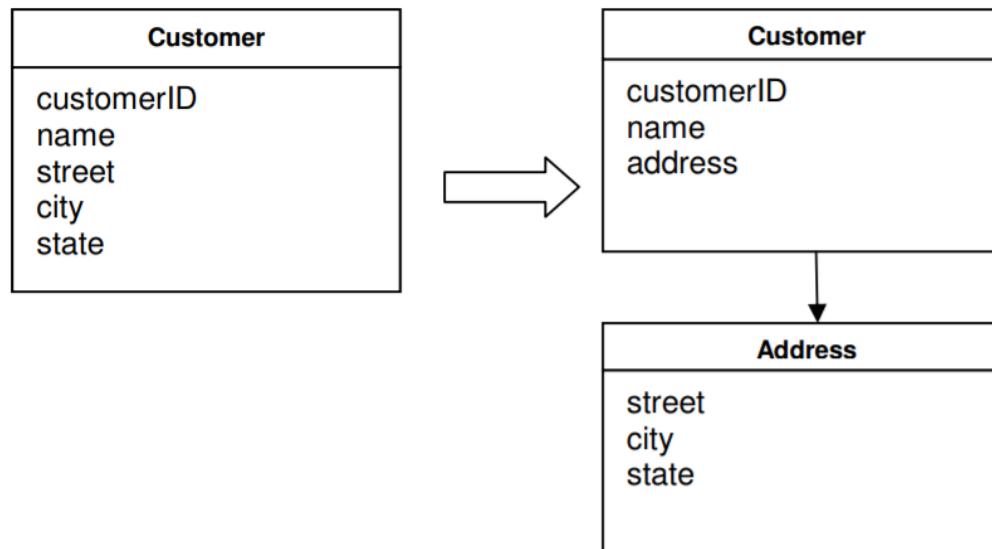


# ENTITIES

- Have an identity that remains unchanged through the states of the application
- Focus on identity and continuity, not on values of the attributes
- The identity can be
  - An attribute
  - A combination of attributes
  - An artificial attribute
- Examples:
  - Person
  - Account

# VALUE OBJECTS

- Do not have identities
- The values of the attributes are important
- Easily created and discarded (no tracking is needed)
- Highly recommended to make them immutable => can be shared!



# ENTITY OR VALUE OBJECT?

- Entity

- Need to decide how to uniquely identify it
- Need to track it
- Need one instance for each object => affects performance

Ex. Customer object

- Value Object

- If the Value Object is shareable, it should be immutable
- Can contain other Value Objects and even references to Entities

# SERVICES

- Contain operations that do not belong to any Entity/Value object
- Do not have internal states
- Operate on Entity/Value objects becoming a point of connection => loose coupling between objects

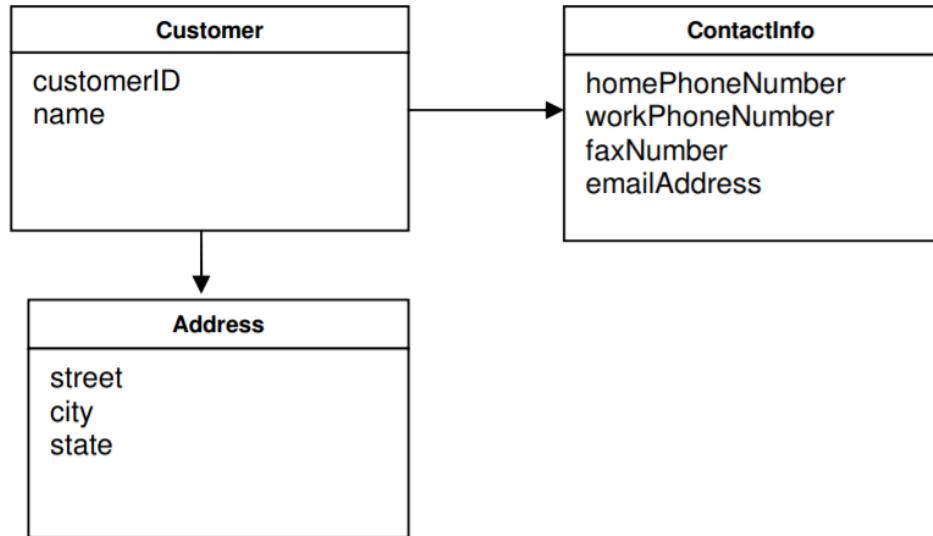
# SERVICES

## Characteristics:

- The operation performed by the Service refers to a domain concept which does not naturally belong to an Entity or Value Object.
- The operation performed refers to other objects in the domain.
- The operation is stateless.

Ex. transferring money from one account to another.

# AGGREGATES

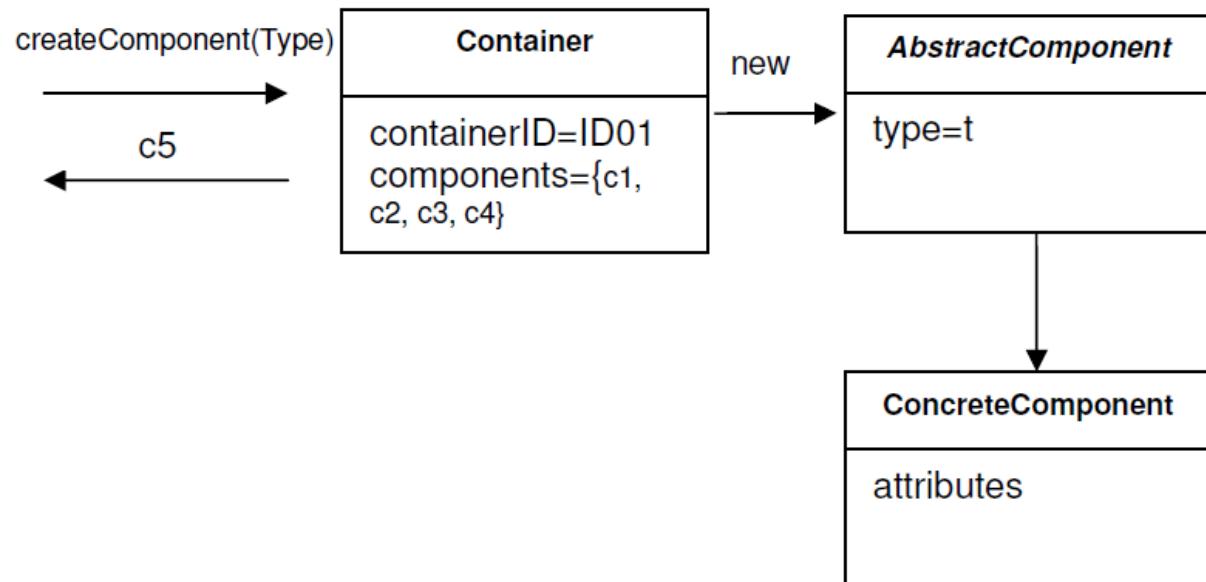


- A domain pattern used to define object ownership and boundaries
- Associations
  - One-to-one
  - One-to-many
  - Many-to-many
- Groups associated objects that represent one unit with regard to data change => defines a **transactional consistency boundary**
- Each aggregate has one root (an Entity)
- Only the root is accessible from outside

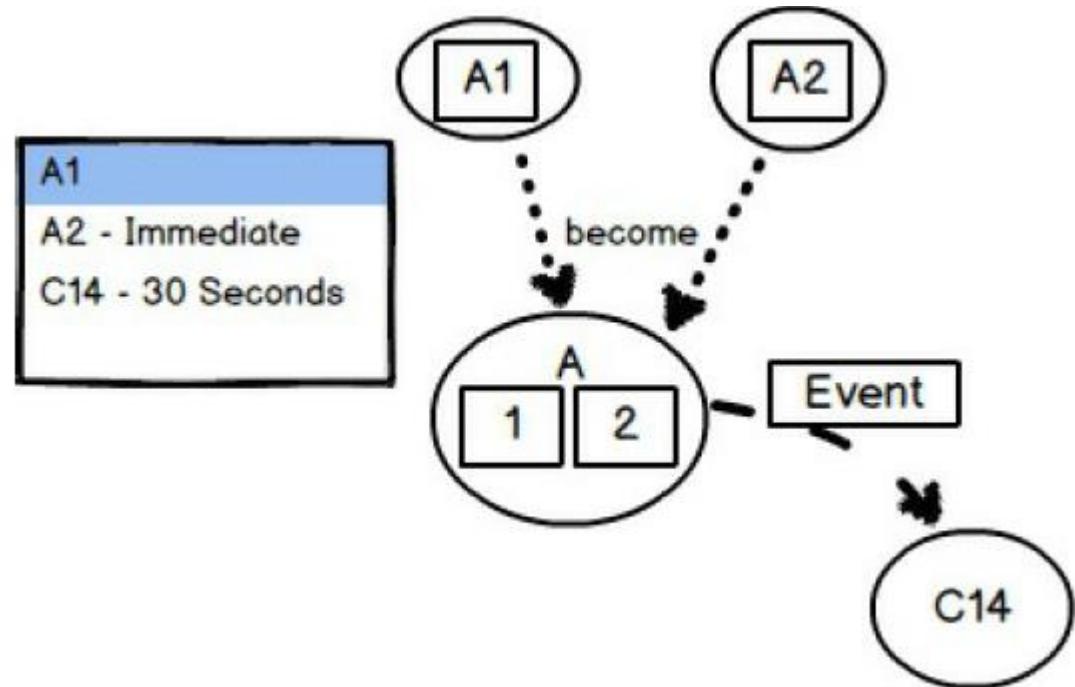
# DISCUSSION

- Root Entity has global identity, inner entities have local identity
- Root Entity enforces invariants
- If the root is deleted => all aggregated objects are deleted, too
- Creating aggregates => atomic process

⇒ Use Factories



# AGGREGATES



## Rules of thumb

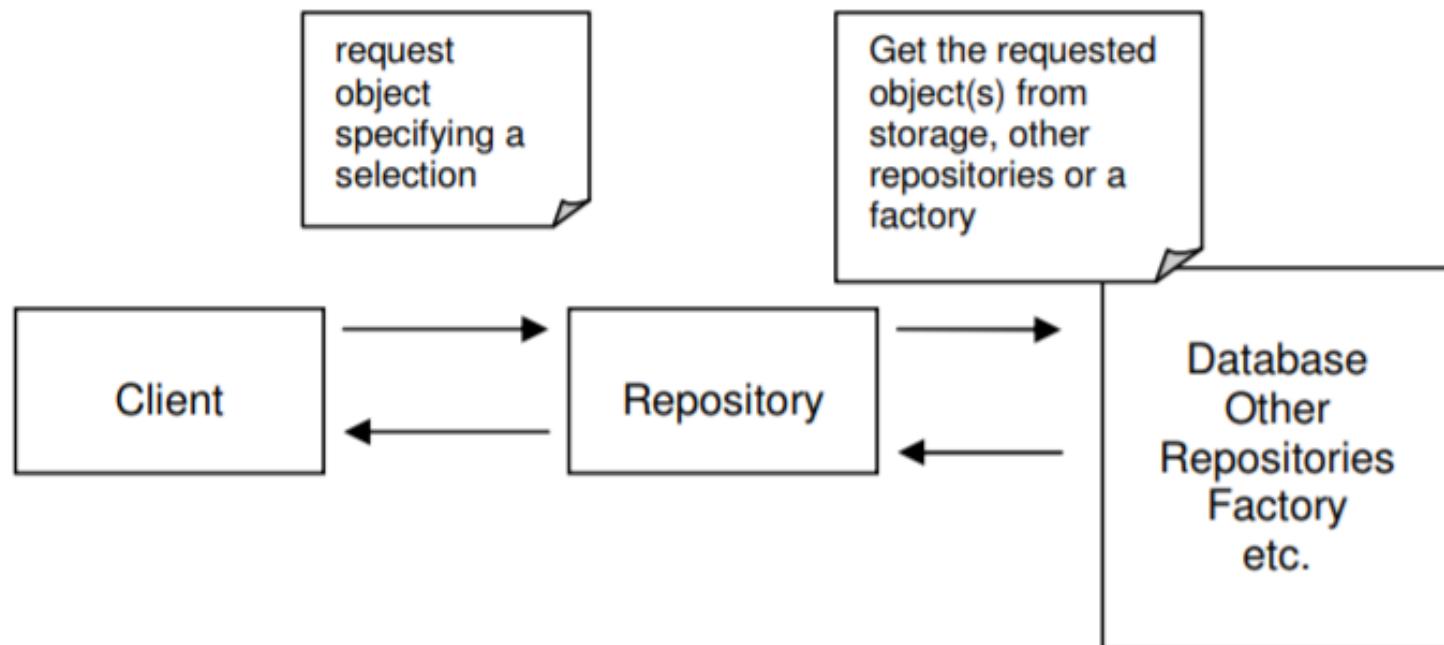
- Protect business invariants inside Aggregate boundaries.  
(transactional consistency within Aggregate)
- Design small Aggregates.
- Reference other Aggregates by identity only.
- Update other Aggregates using eventual consistency.  
(transactional consistency across Aggregates)

# REPOSITORY

- How do we get the object (i.e. Entity, Value)?
  - Create it (Factories)
  - Obtain it
    - If it is a Value Object => need the root of the Aggregate
    - If it is an Entity => can be obtained directly from the database.
- Problems:
  - Dependency on the database structure
  - Mixing database access into the domain logic

# REPOSITORY

Repository encapsulates all the logic needed to obtain object references (either by already storing them, or by getting them from the storage).

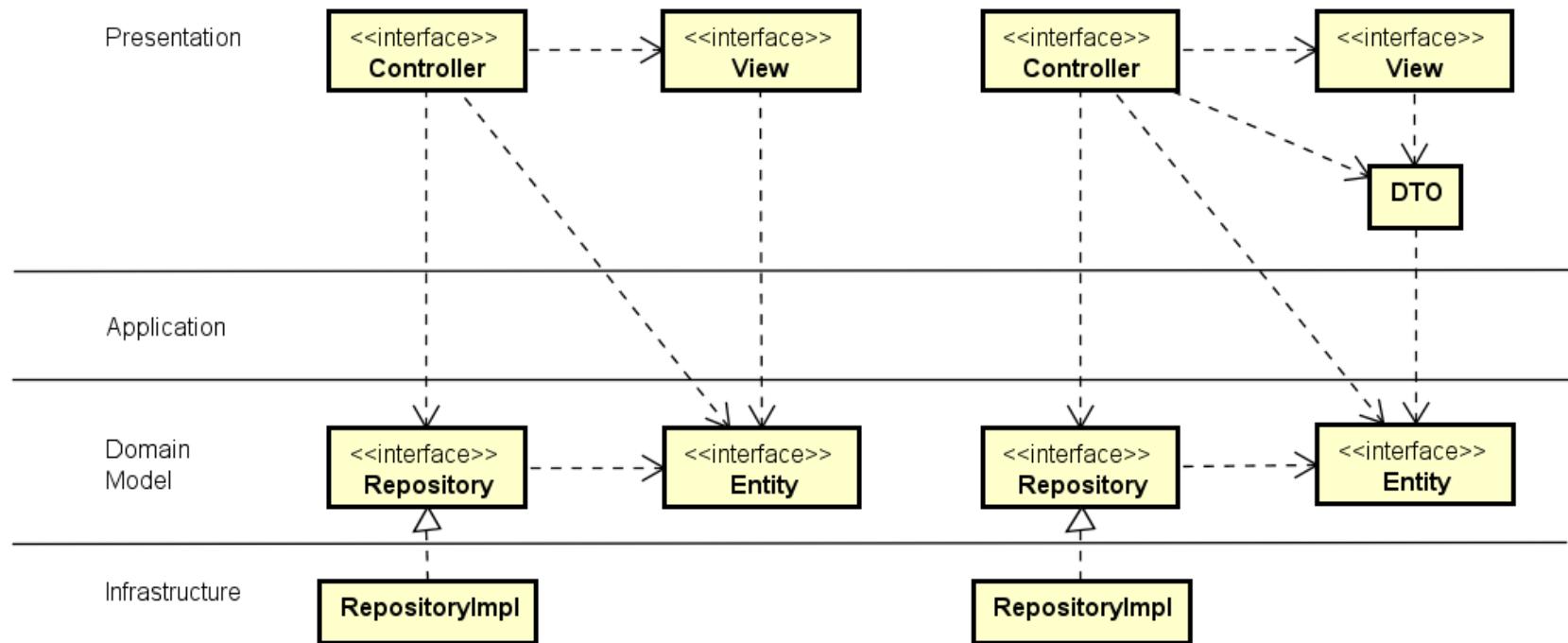


# DISCUSSION

How complex should the Domain Model be?

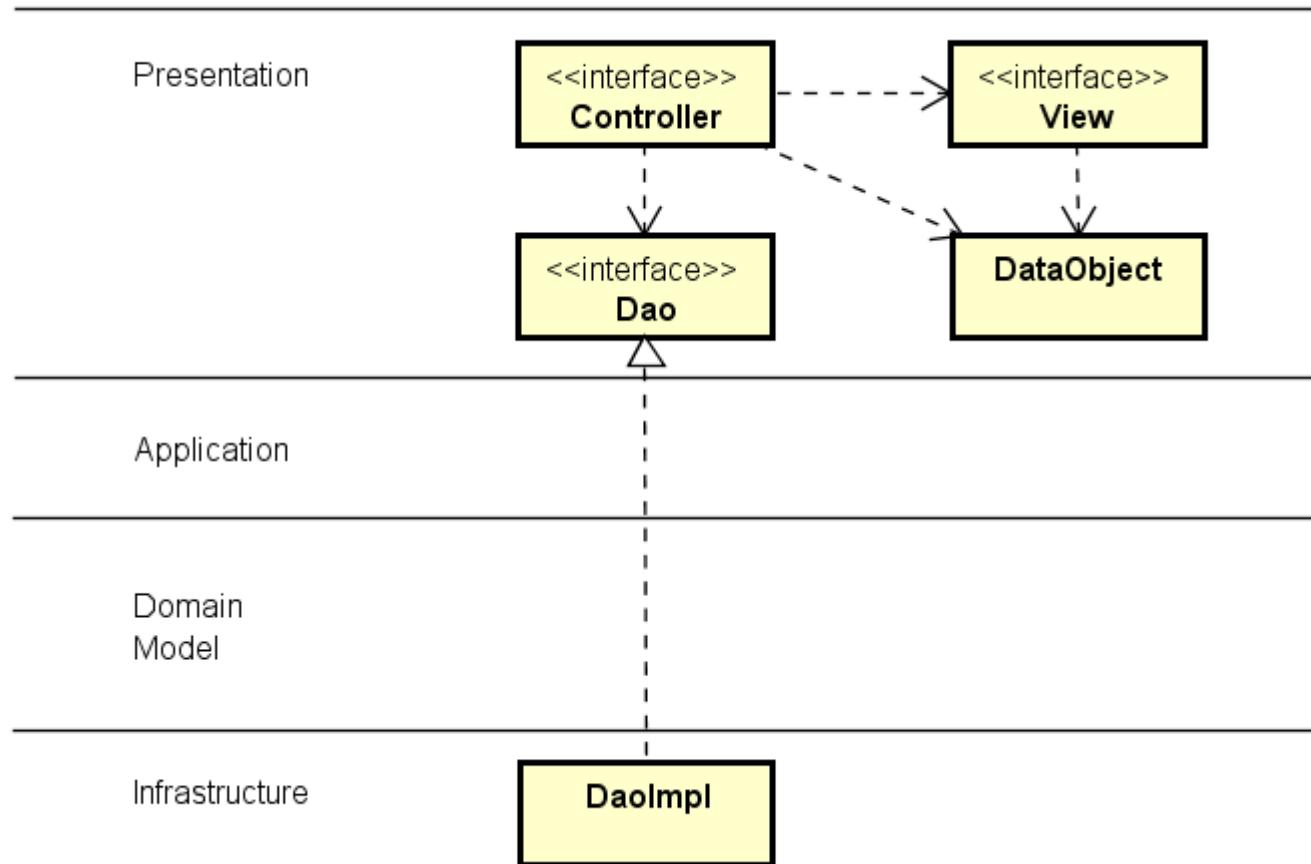
Anemic Domain Model (Anti-pattern)

- Just forms over data (CRUD operation + some validation)



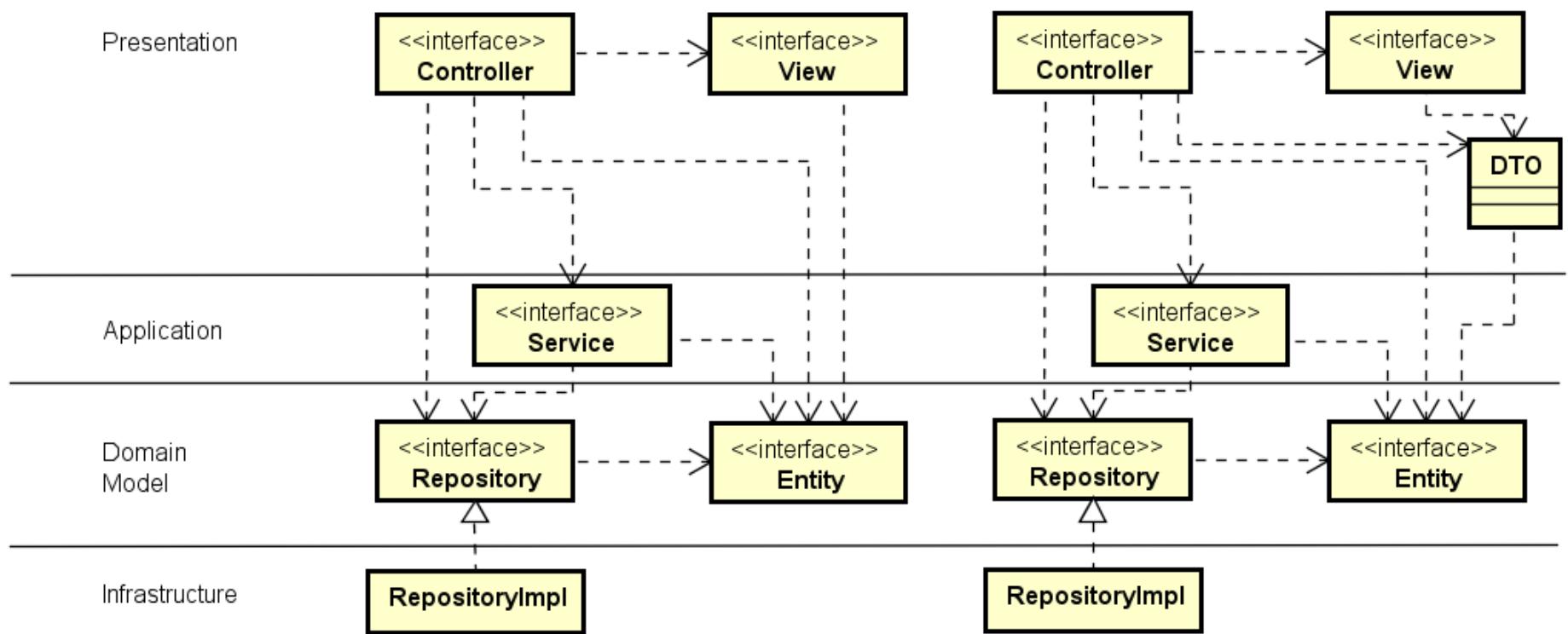
# ALTERNATIVES

## Even less Domain Model



# WHEN TO USE SERVICES?

- Need for driving the workflow, coordination transaction management
- E.g. The controller needs to update more than one entity

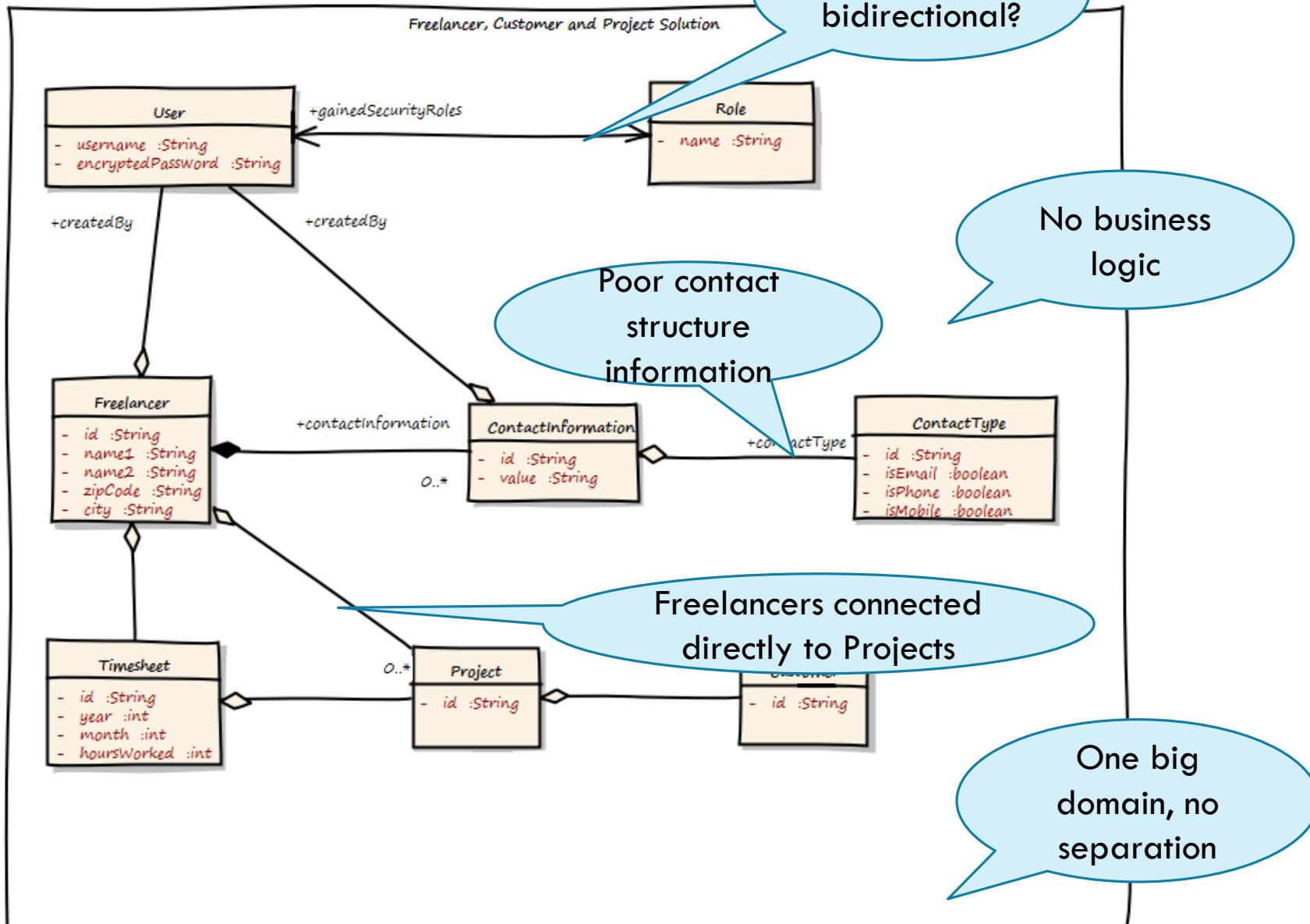


# THE IT BODY LEASING EXAMPLE

A company provides IT Body Leasing. They have some Employees, and also a lot of Freelancers as Subcontractors. Requirements:

- A searchable catalog of Freelancers must be provided
- Allows to store the different Communication Channels available to contact a Freelancer
- A searchable catalog of Projects must be provided
- A searchable catalog of Customers must be provided
- The Timesheets for the Freelancers under contract must be maintained

# FIRST (BAD) DOMAIN MODEL



# SPLITTING THE PROBLEM INTO SUBDOMAINS

Identity and  
Access  
Management  
subdomain

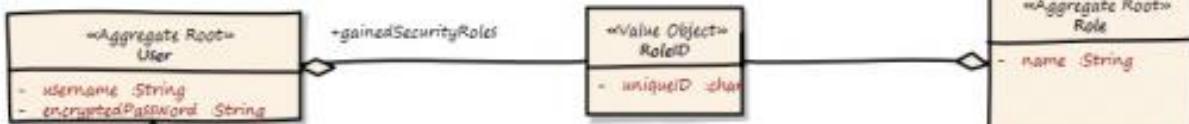
Freelancer  
Management  
subdomain

Customer  
Management  
subdomain

Project  
Management  
subdomain

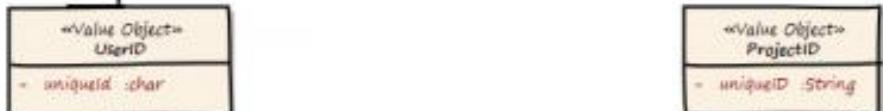
Generic subdomain

Identity and access management bounded context

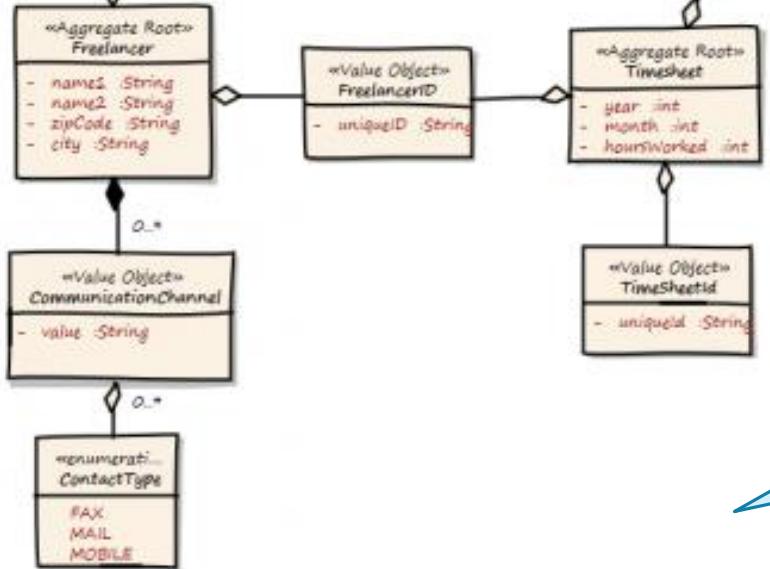


Shared Kernel

Common types

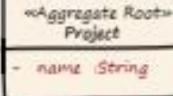


Freelancer management bounded context

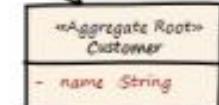


Core Domain

Project management bounded context

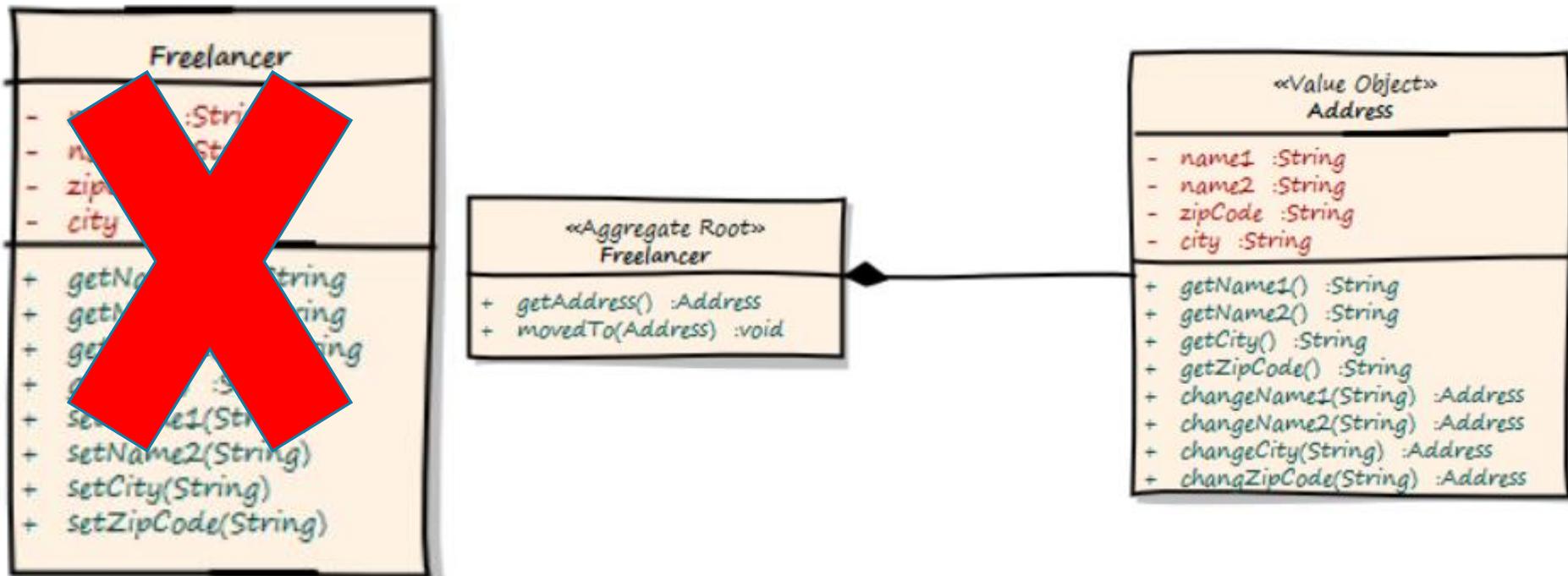


Customer management bounded context



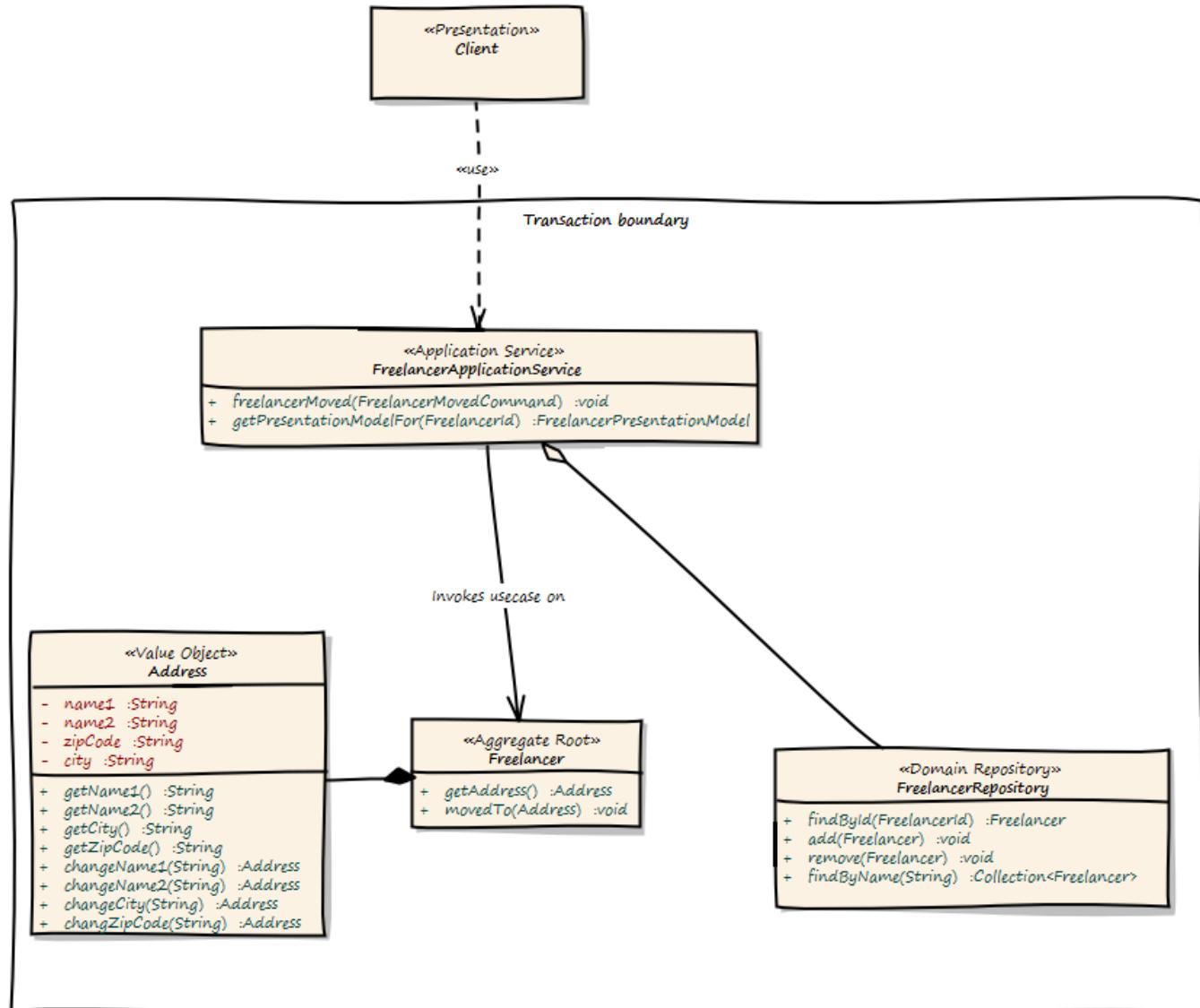
Support Domains

# APPLYING DOMAIN DRIVEN DESIGN

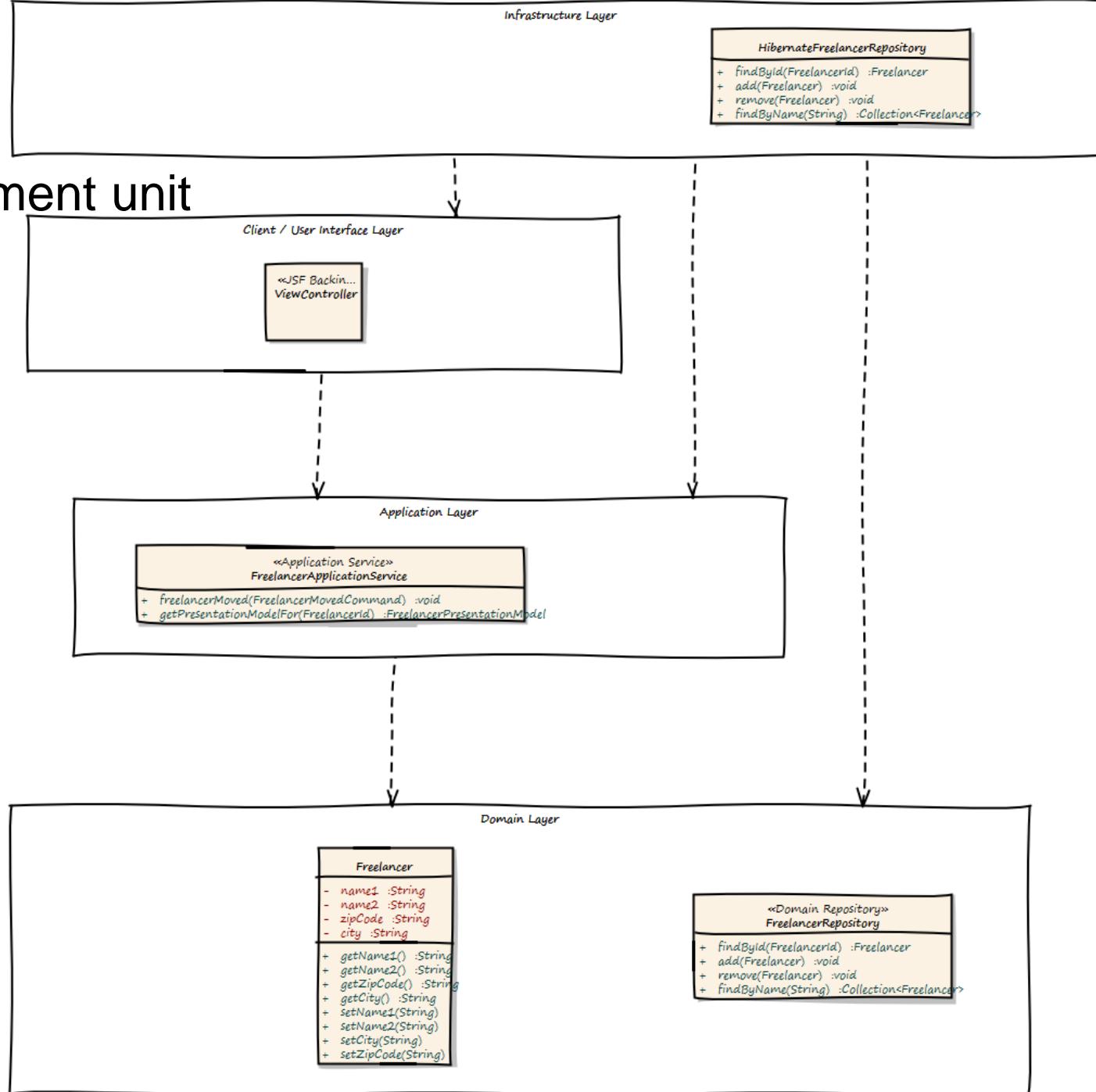


# ADDING BEHAVIOR

Freelancer  
moved to  
new location



# The Deployment unit

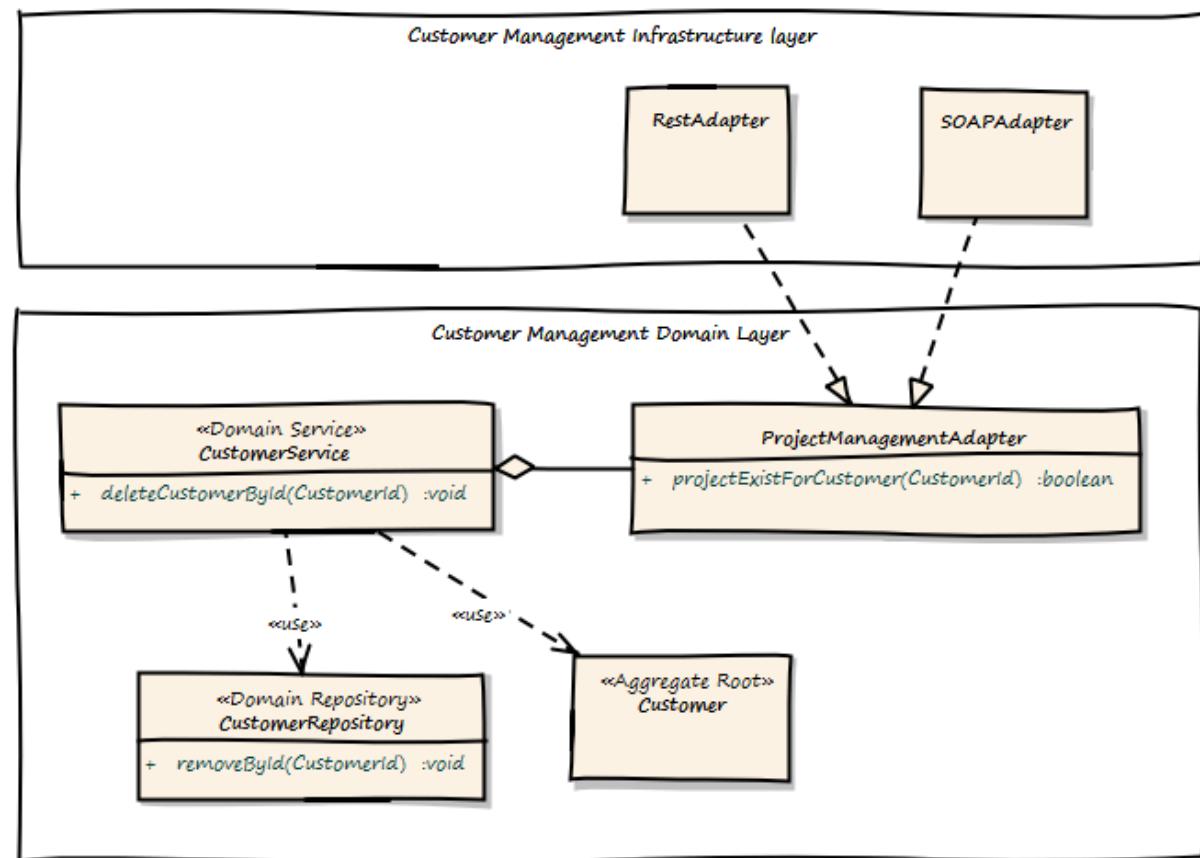


# CONTEXT INTEGRATION

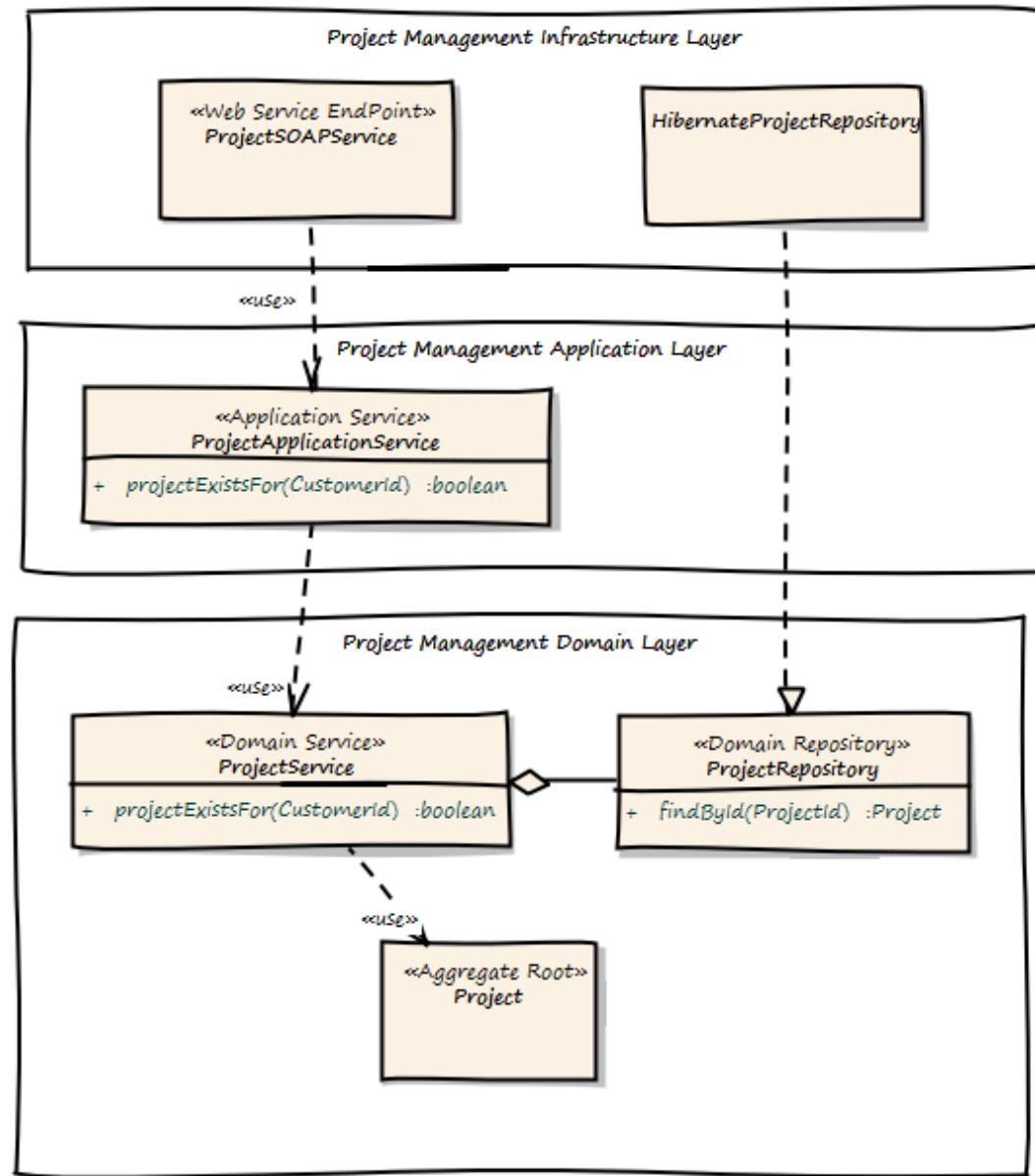
- A Customer can only be deleted if there is no Project assigned

⇒ Domain Service

⇒ Synchronous Integration



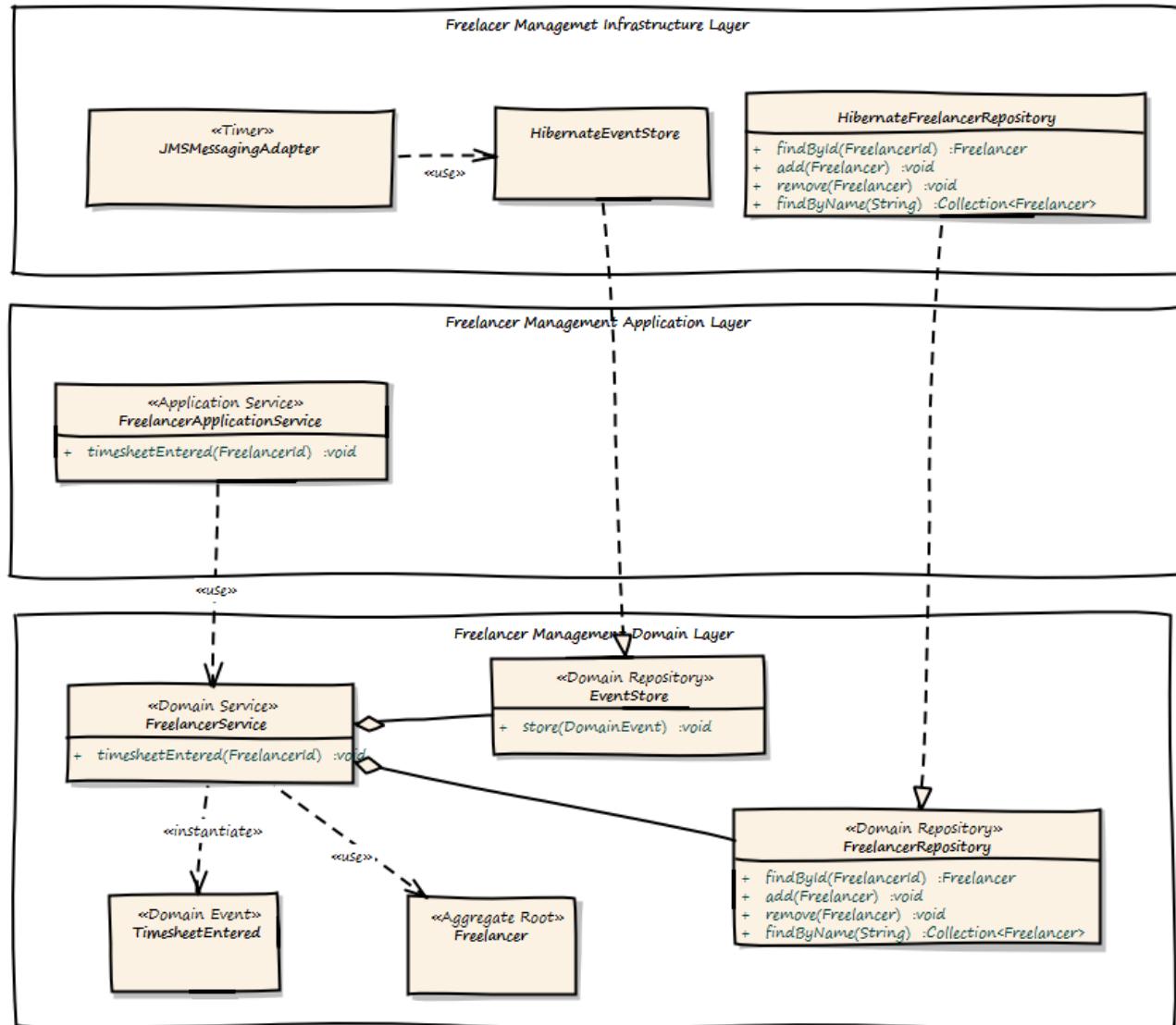
# ON THE PROJECT MANAGEMENT SIDE



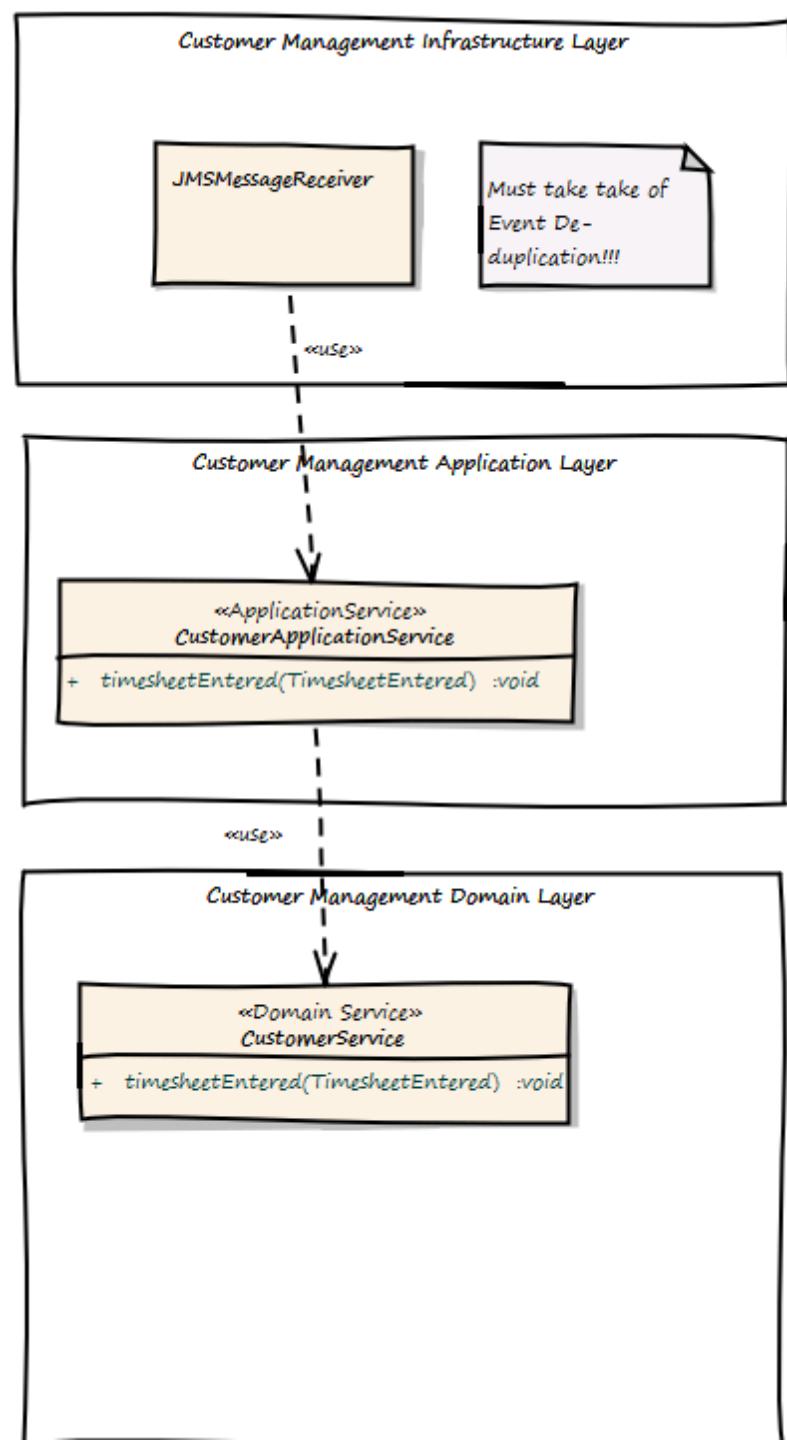
# ASYNCHRONOUS EXAMPLE

Once a Timesheet  
is entered, the  
Customer needs to  
be billed

Use Domain Events



# ON THE CUSTOMER SIDE

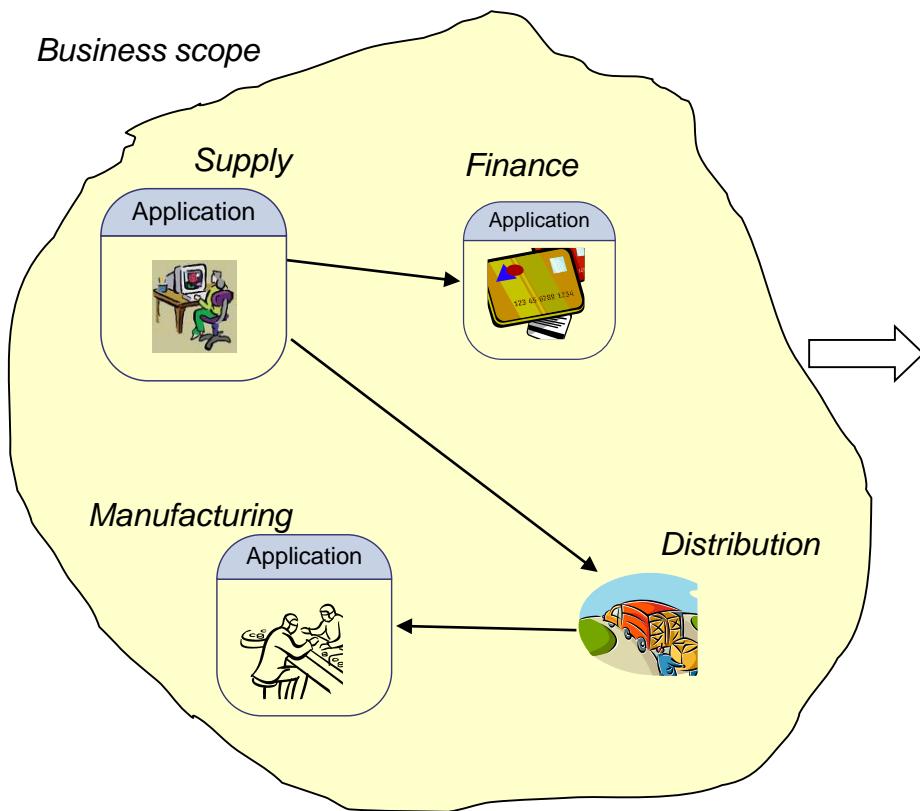


# SERVICE BASED ARCHITECTURES

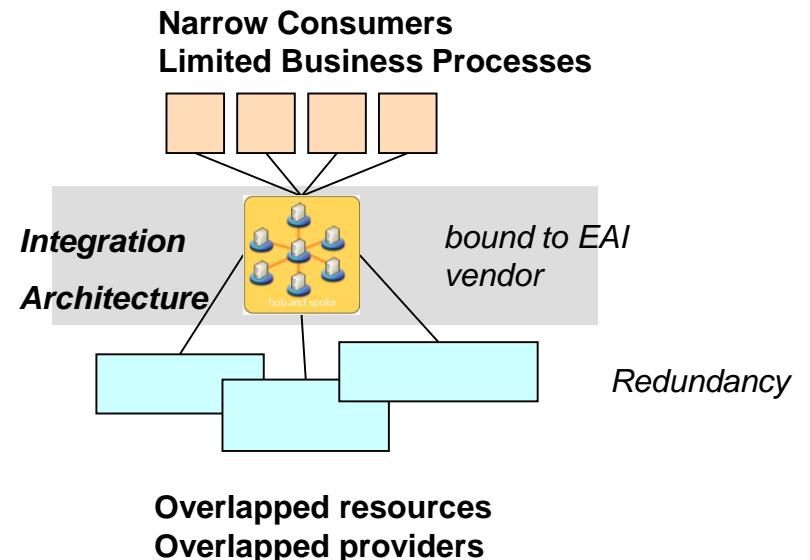
- “1. All teams will henceforth expose their data and functionality through **service interfaces**
- 2. Teams must **communicate** with each other **through** these **interfaces**
- 3. There will be **no other form of interprocess communication** allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The **only** communication allowed is via service interface calls over the network.
- 4. It **doesn't matter** what [API protocol] **technology** you use.
- 5. **Service interfaces**, without exception, must be designed from the ground up to be **externalizable**. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- 6. Anyone who doesn't do this **will be fired**.
- 7. Thank you; **have a nice day!**”

Amazon CEO (supposedly)

# APPLICATION CENTRIC

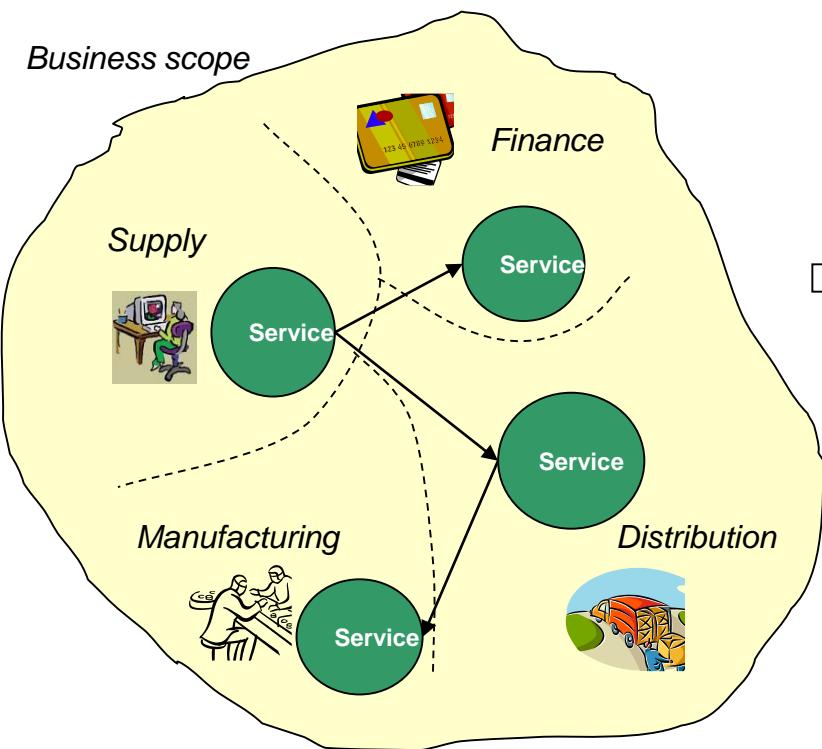


**Business functionality is duplicated in each application that requires it.**



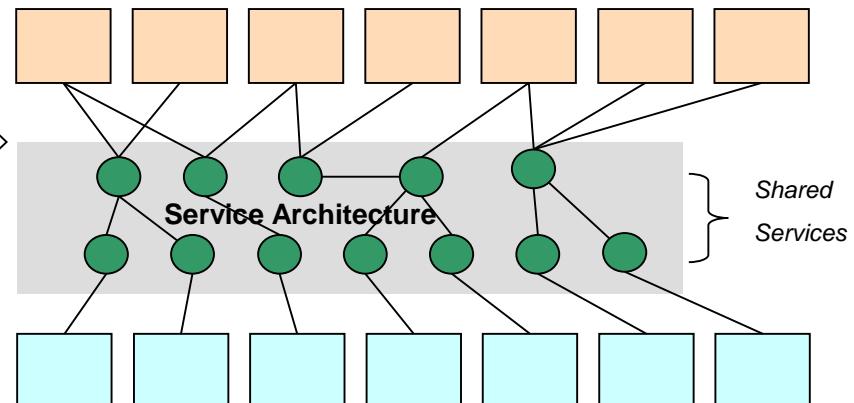
EAI ‘leverage’ application silos with the drawback of data and function redundancy.

# SERVICE CENTRIC



**SOA structures the business and its systems as a set of capabilities that are offered as Services, organized into a Service Architecture**

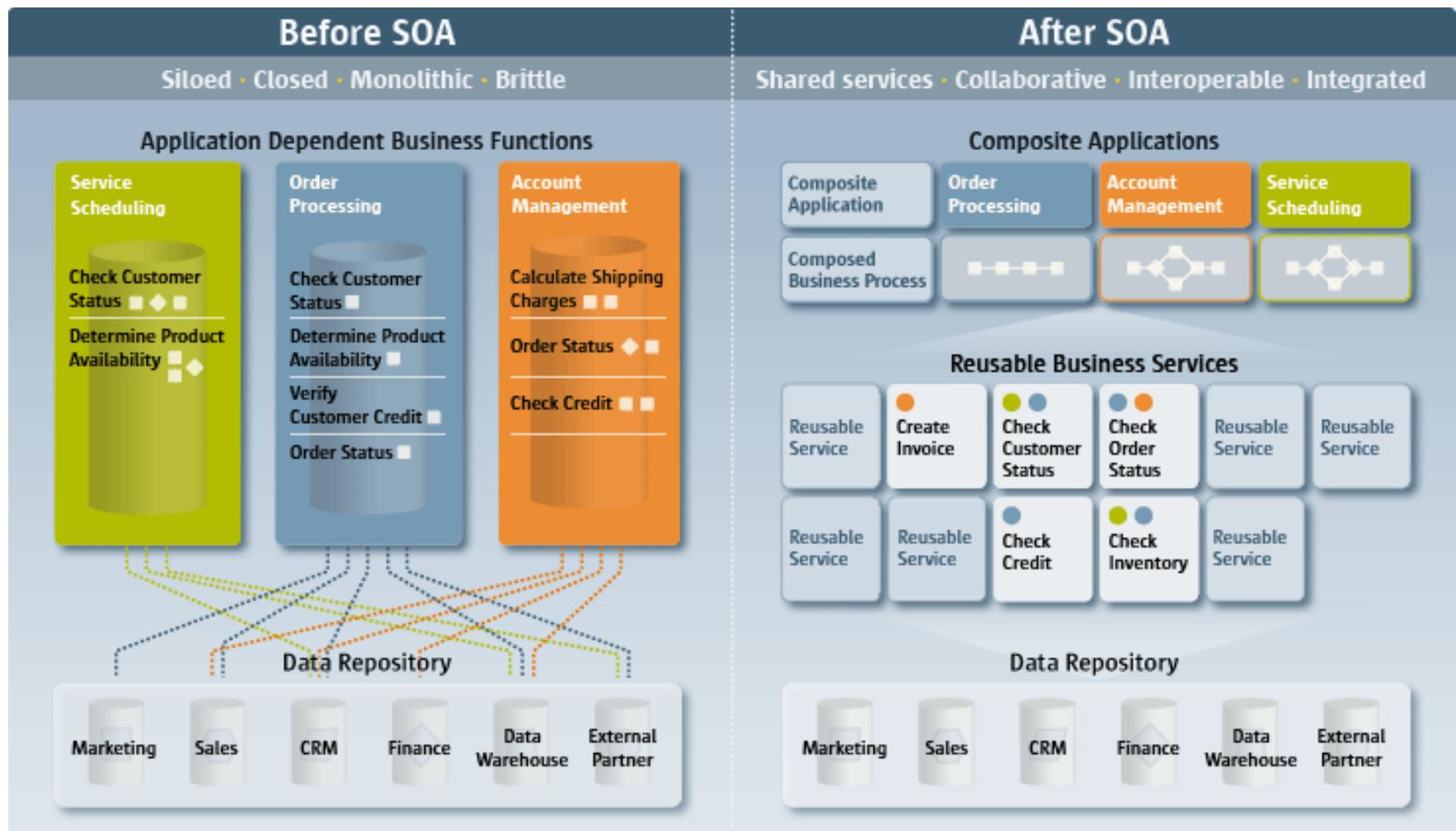
**Multiple Service Consumers  
Multiple Business Processes**



**Multiple Discrete Resources  
Multiple Service Providers**

Service virtualizes how that capability is performed, and where and by whom the resources are provided, enabling multiple providers and consumers to participate together in shared business activities.

# BEFORE SOA – AFTER SOA



# DESIGN PRINCIPLES 1

## **Services are reusable**

- Business functionalities exposed as services are designed with the intention of reuse whenever and where they are required

## **Services share a formal contract**

- Services interact with each through a formal contract which is shared to exchange information and terms of usage

## **Services are loosely coupled**

- Services are designed as loosely coupled entities able to interact while maintaining their state of loose coupling.

## **Services abstract underlying logic**

- The business logic underpinning a service is kept hidden from the outside world. Only the service description and formal contract are visible for the potential consumers of a service

# DESIGN PRINCIPLES 2

## **Services are composable**

- Services may be composed of other services. Hence, a service's logic should be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

## **Services are autonomous**

- A service should be independent of any other service

## **Services are stateless**

- A service shouldn't require to maintain state information rather it should be designed to maximize statelessness

## **Services are discoverable**

- A service should be discoverable through its description, which can be understood by humans and service users.

# DESIGN PRINCIPLES 3

## **Services have a network-addressable interface**

- A service should be invoked from the same computer or remotely – through a local interface or Internet

## **Services are location transparent**

- A service should be discoverable without the knowledge of its real location. A requestor can dynamically discover the location of a service looking up a registry.

The core principles are **autonomy, loose coupling, abstraction, formal contract**

# WHAT IS A WEB SERVICE? (W3C DEFINITION)

A Web service:

- is a software system designed to support **interoperable machine-to-machine interaction** over a network.
- has an **interface** described in a machine-processable format (specifically WSDL).

Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

# WHAT IS A WEB SERVICE: A SIMPLER DEFINITION

A Web Service is a standards-based way for an application to call a function over a network and to do it without having to know:

- the **location** where the function will be executed,
- the **platform** where the function will be run,
- the **programming language** it is written in, or even
- **who** built it.

# WEB SERVICES - SOAP BASED

## Discovery

- Where is the service?

Discovery  
UDDI

## Description

- What service does it offer?
- How do I use it?

Description  
WSDL

## Messaging

- Let's communicate!

Messaging  
SOAP

# STANDARD IS KEY

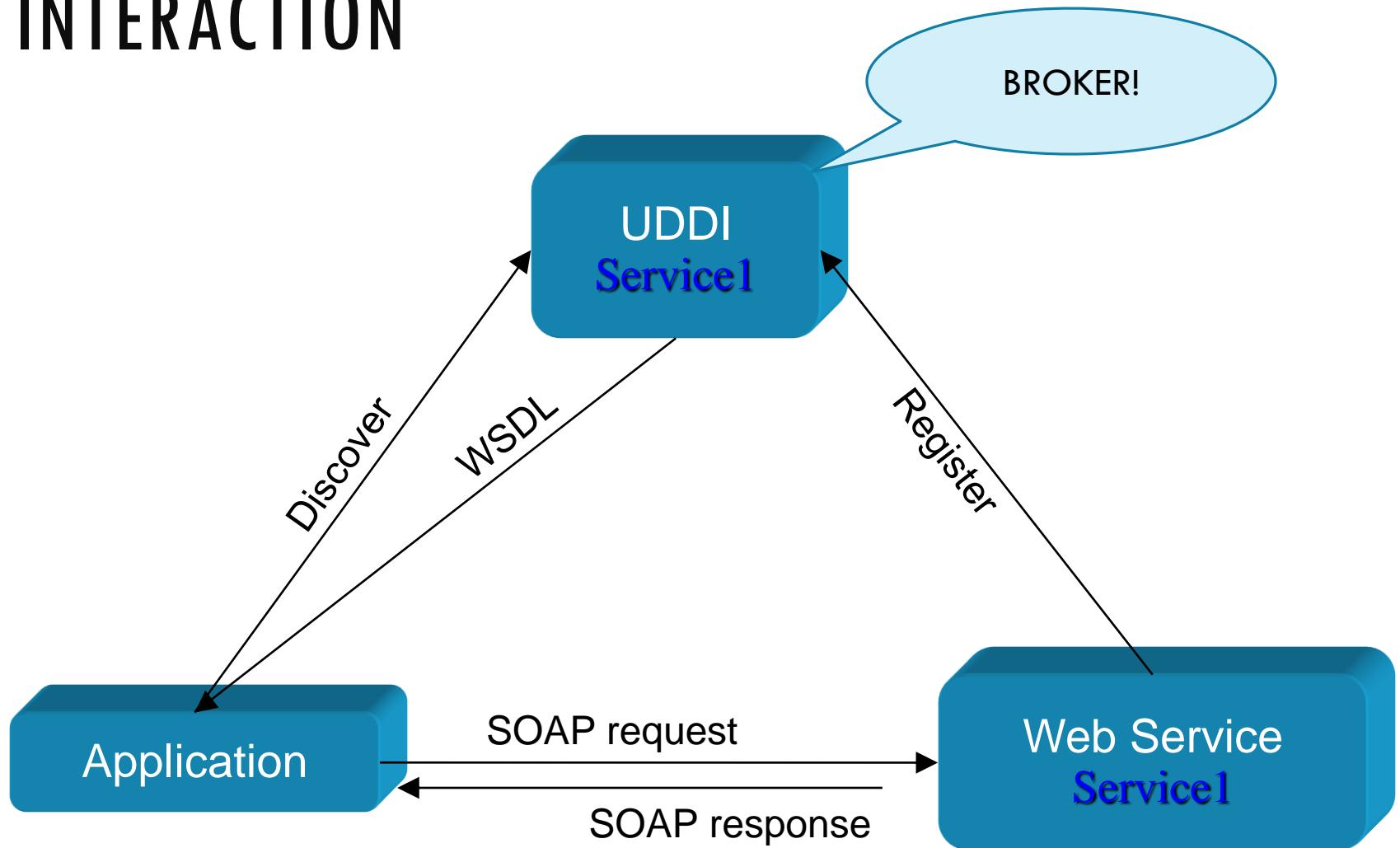
WSDL is used to describe the **function(s)** that an application will be calling documenting in a standard way its entry points, parameters and output

XML is used to carry the values of **parameters** and the **outputs** of the function

SOAP is used as the **messaging protocol** that carries content (XML) over a network transport (typically HTTP)

HTTP is used as the **network transport layer**

# INTERACTION

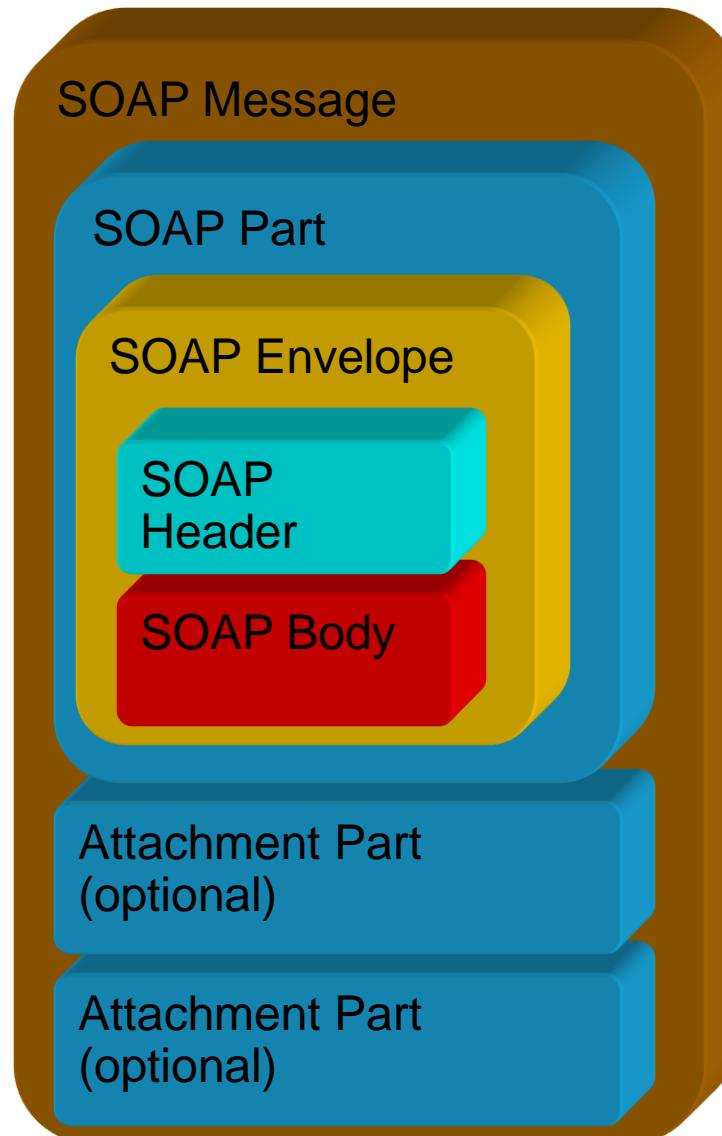


# WSDL: WEB SERVICE DESCRIPTION LANGUAGE

- WSDL (Web Services Description Language) is a public description of the interfaces offered by a web service.
  - Expressed in XML
  - Describes services as a set of endpoints
    - Document-oriented
    - Procedure-oriented
- XML grammar
- <definitions>: root WSDL element
  - <types>: data types transmitted (starts with XML Schema specifications)
  - <message>: messages transmitted
  - <portType>: functions supported
  - <binding>: specifics of transmissions
  - <service>: how to access it

# SOAP: A DESCRIPTION

- Industry standard message format for sending and receiving data between a web services consumer and a web service provider
- SOAP messages are XML documents which have an envelope and:
  - Header (optional): contains information about the message such as date/time it was sent or security information
  - Body: contains the message itself
- SOAP used to stand for Simple Object Access Protocol



# GIVE IT A REST

REST: REpresentation S<sub>tate</sub> Transfer

Rest-ful: Follows the REST principles

Not strictly for web services

Term used loosely as a method of sending information over HTTP without using a messaging envelope

# REPRESENTATIONAL STATE TRANSFER (REST)

- Idea: *Self-contained* requests specify what resource to operate on and what to do to it [Roy Fielding's PhD thesis, 2000]
- A service (in the SOA sense) whose follows the REST principles (next slide) is a RESTful service
- Ideally, RESTful URIs name the operations

# REST PRINCIPLES

[RP1] The key abstraction of information is a **resource**, named by an URL. Any information that can be named can be a resource.

[RP2] The **representation** of a resource is a sequence of bytes, plus representation metadata to describe those bytes.

[RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it (**Stateless**).

# REST PRINCIPLES (CONT'D)

[RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically (**Uniform interface**).

[RP5] Idempotent operations and representation metadata are encouraged in support of **caching** and representation reuse.

[RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server (**Links between resources**).

# RESOURCES

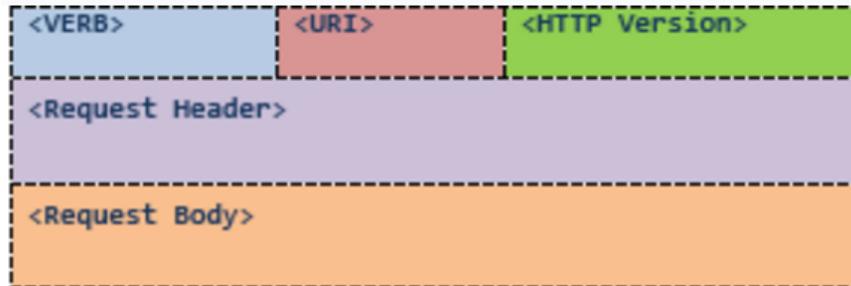
## XML representation

```
<Person>  
  <ID>1</ID>  
  <Name>M Dinso</Name>  
  
<Email>m.dinso@gmail.com</Email>  
  <Country>Romania</Country>  
</Person>
```

## JSON representation

```
{  
  "ID": "1",  
  "Name": "M Dinso",  
  "Email":  
  "m.dinso@gmail.com",  
  "Country": "Romania"  
}
```

# MESSAGES – HTTP REQUEST



<VERB> is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc

<URI> is the URI of the resource on which the operation is going to be performed

<HTTP Version> is the version of HTTP

<Request Header> contains the metadata as a collection of key-value pairs of headers and their values. Ex. client type, the formats client supports, format type of the message body, cache settings for the response, etc.

<Request Body> is the actual message content. In a RESTful service, that's where the representations of resources sit in a message.

# POST REQUEST EXAMPLE

```
POST http://MyService/Person/  
Host: MyService  
Content-Type: text/xml; charset=utf-8  
Content-Length: 123  
<?xml version="1.0" encoding="utf-8"?>  
<Person>  
  <ID>1</ID>  
  <Name>M Dinsø</Name>  
  <Email>m.dinso@gmail.com</Email>  
  <Country>Romania</Country>  
</Person>
```

# HTTP VERBS – UNIFORM INTERFACE

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent
OPTIONS	List the allowed operations on a resource.	Safe
HEAD	Return only the response headers and no response body.	Safe

# STATELESSNESS

Does not maintain the application state for any client.

A request cannot be dependent on a past request and a service treats each request independently.

Example:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

# SOAP VS. REST

## SOAP

- SOAP is still offered by some very prominent tech companies for their APIs (Salesforce, Paypal, DocuSign).
- SOAP is good for applications that require ***formal contracts*** between the API and consumer, since it can enforce the use of formal contracts by using WSDL
- Additionally, SOAP has built in ***WS-Reliable messaging*** to increase security in asynchronous execution and processing.
- SOAP has built-in ***stateful operations***. SOAP is designed support conversational state management.
- Provides support for ***WS\_AtomicTransaction*** and ***WS\_Security***, SOAP can benefit developers when there is a high need for transactional reliability.

# SOAP

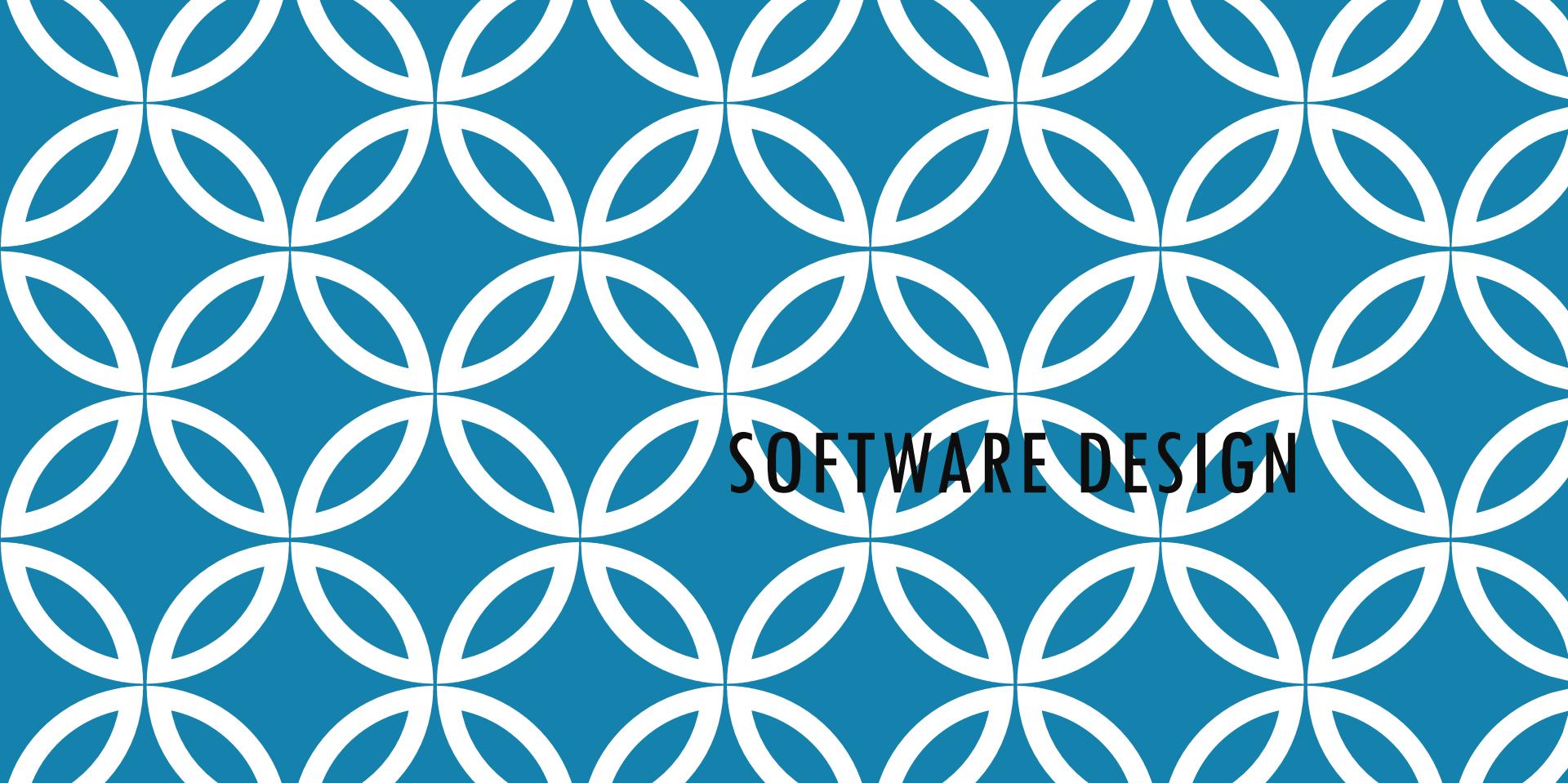
- Language, platform, and transport independent (REST requires use of HTTP)
- Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
- Standardized
- Provides significant pre-build extensibility in the form of the WS\* standards (I.e. WS-Addressing, WS-Policy, WS-Security, WS-Federation, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, and WS-RemotePortlets)
- Built-in error handling
- Automation when used with certain language products

# REST

- Easy to understand: uses HTTP and basic CRUD operations, so it is simple to write and document.
- Makes efficient use of bandwidth, as it's much less verbose than SOAP. Unlike SOAP, REST is designed to be stateless and REST reads can be cached for better performance and scalability.
- Supports many data formats, but the predominant use of JSON means better support for browser clients.
- No expensive tools required to interact with the Web service
- Smaller learning curve
- Efficient (SOAP uses XML for all messages, REST can use smaller message formats)
- Fast (no extensive processing required)
- Closer to other Web technologies in design philosophy

# WRAP-UP

- The business/domain logic layer contains the independent logic
- Several approaches to model the business logic
- Domain driven design helps
  - Dividing a complex domain into subdomains
  - Identify boundary contexts
  - Identify entities and value objects linked into aggregates
  - Better dependency management
  - Better transaction management
- Service orientation



# SOFTWARE DESIGN

Service oriented  
Architectures

# LAST TIME

## Business Logic Layers

- Domain driven
- Services

# CONTENT

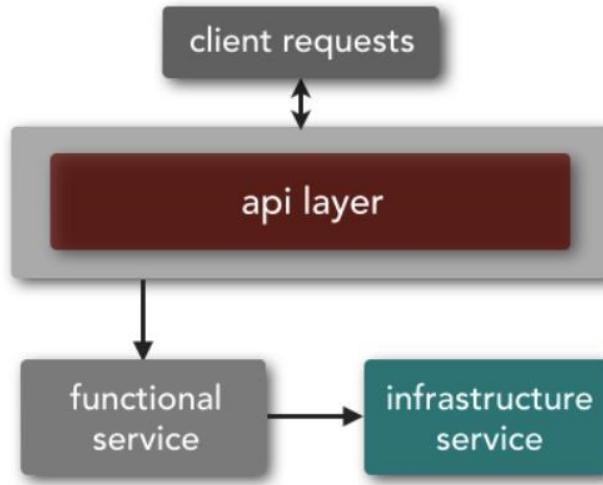
- Services and Microservices
- iDesign Architecture and Method
- Example

# REFERENCES

- **Juval Lowy, Righting software, O'Reilly, 2020**
- **Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]**
- **Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016**
- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- Ian Gorton. 2011. Essential Software Architecture. 2nd Edition, Springer-Verlag [Gorton]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- Armando Fox, David Patterson, and Koushik Sen, SaaS Course Stanford, Spring 2012 [Fox]
- Jacques Roy, SOA and Web Services, IBM
- Mark Bailey, Principles of Service Oriented Architecture, 2008
- Erl, Thomas. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Pearson Education. 2016
- Erl, Thomas. SOA Design Patterns, Prentice Hall, 2009.

<http://soapatterns.org>

# MICROSERVICES TOPOLOGY



## *Functional services*

- support specific business operations or functions
- accessed externally and are generally not shared with any other service

## *Infrastructure services*

- support nonfunctional tasks such as authentication, authorization, auditing, logging, and monitoring.
- not exposed to the outside world but rather are treated as private shared services only available internally to other services

# GRANULARITY

**SOA** – [small application services; very large enterprise services]

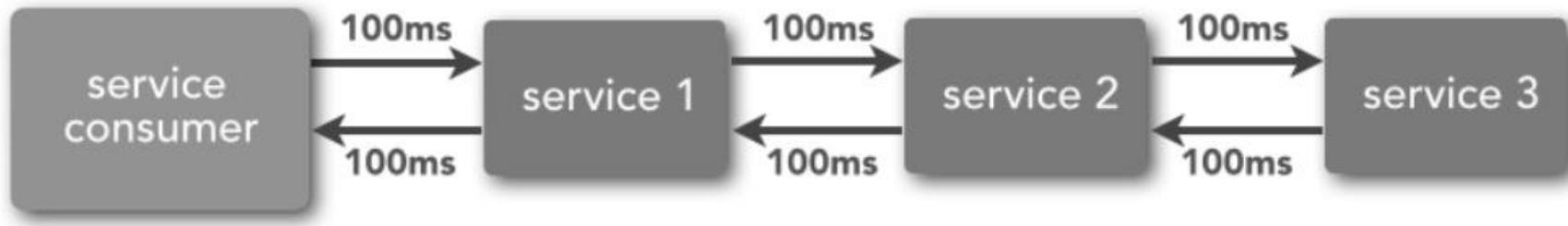
Ex. enterprise *Customer* service handles update and retrieval data views, delegating the lower-level getters and setters to application-level services that were not exposed remotely to the enterprise

**Microservices** - single-purpose services that do one thing really, really well

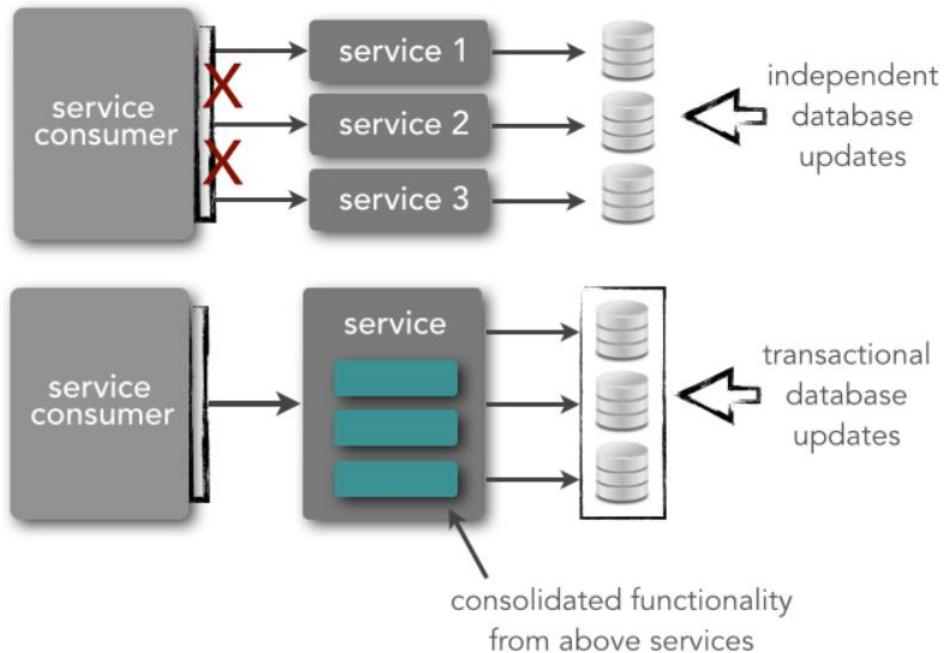
Ex. fine-grained getter and setter services like  
*GetCustomerAddress*, *GetCustomerName*, *UpdateCustomerName*

# IMPACT OF GRANULARITY

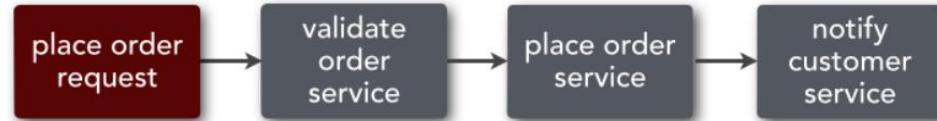
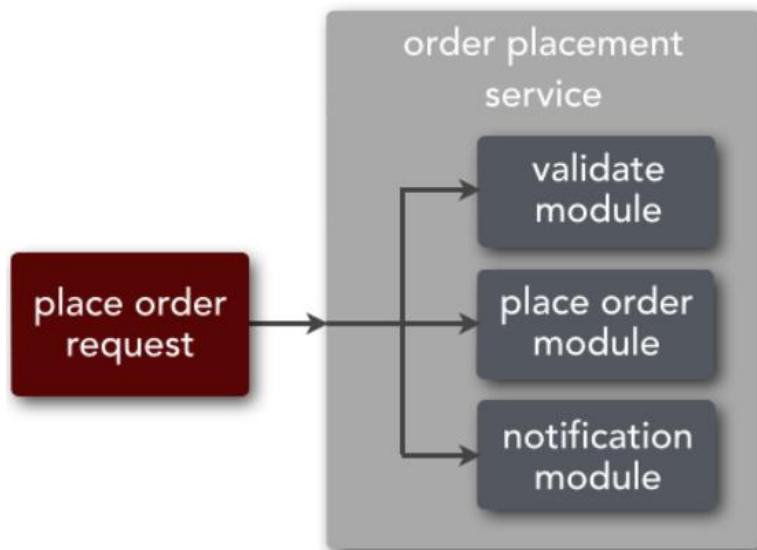
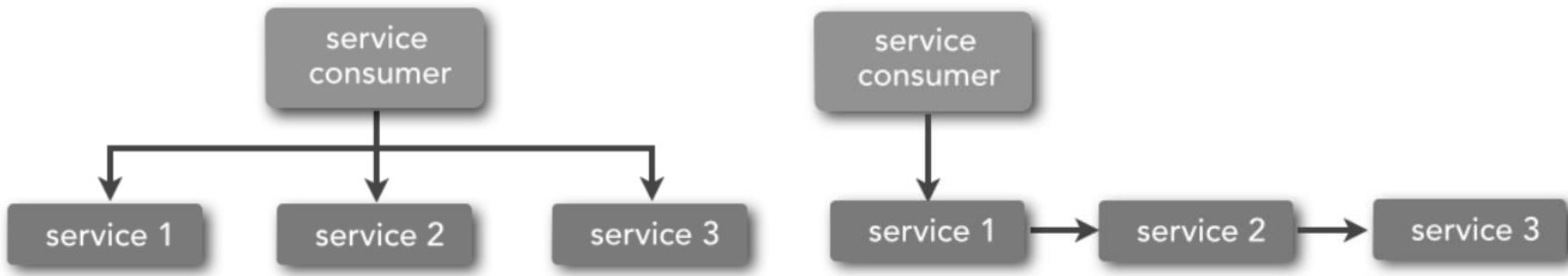
## Performance



## Transaction management

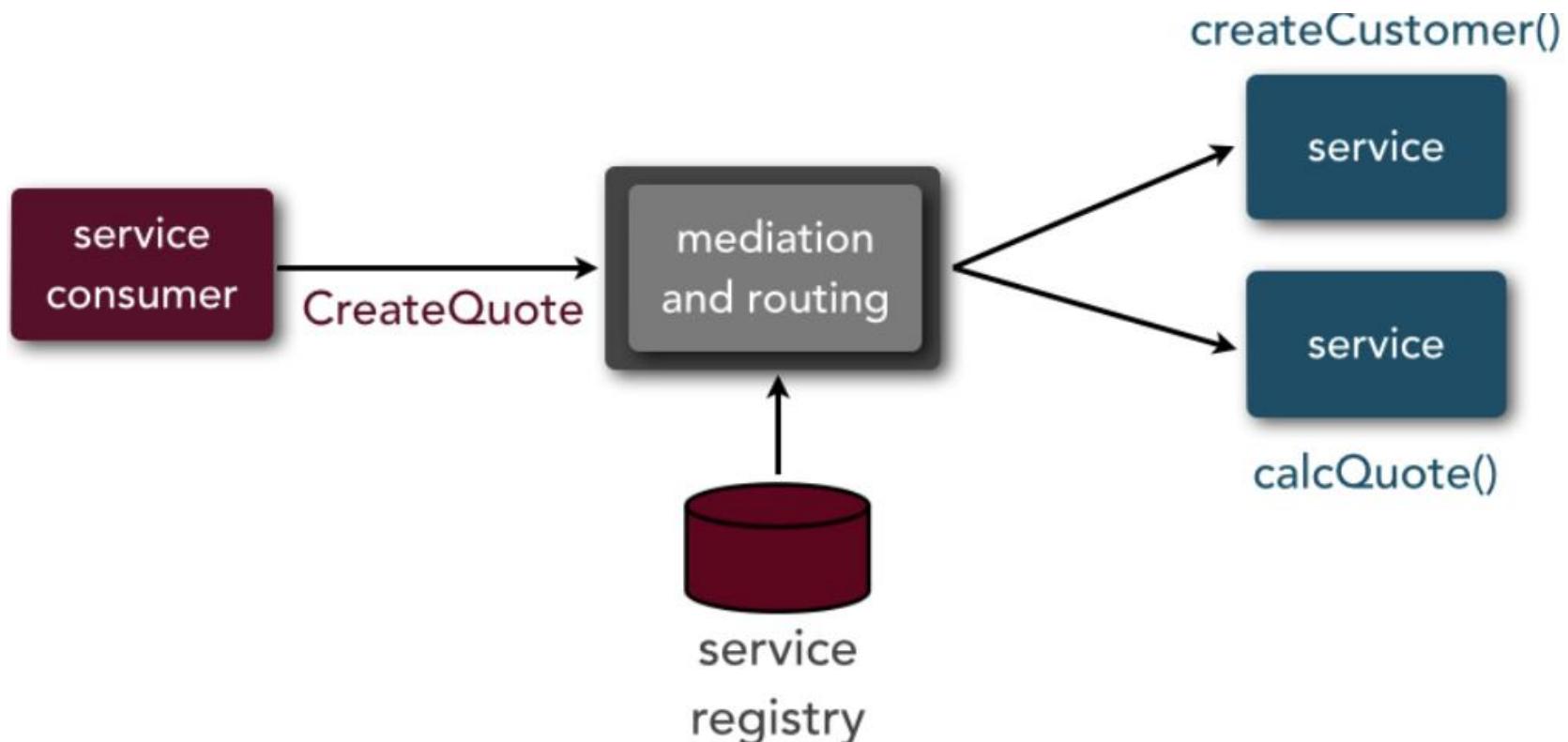


# ORCHESTRATION AND CHOREOGRAPHY



# SOA MIDDLEWARE RESPONSIBILITIES

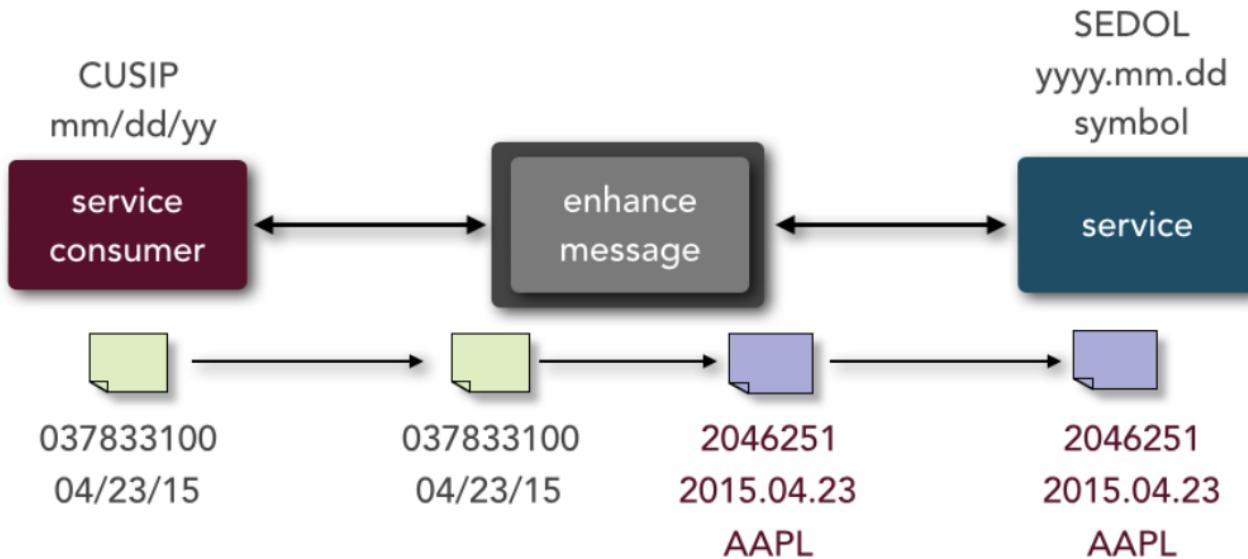
*Mediation and routing* - locate and invoke a service (or services) based on a specific business or user request



# MORE

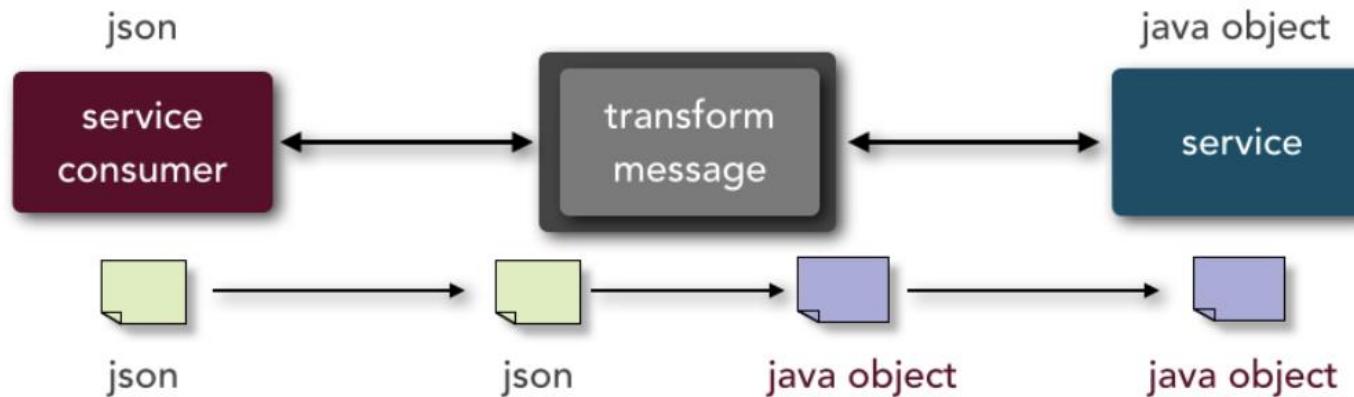
Message *enhancement* - modify, remove, or augment the data portion of a request before it reaches the service.

Ex. changing a date format, adding additional derived or calculated values to the request, and performing a database lookup to transform one value into another



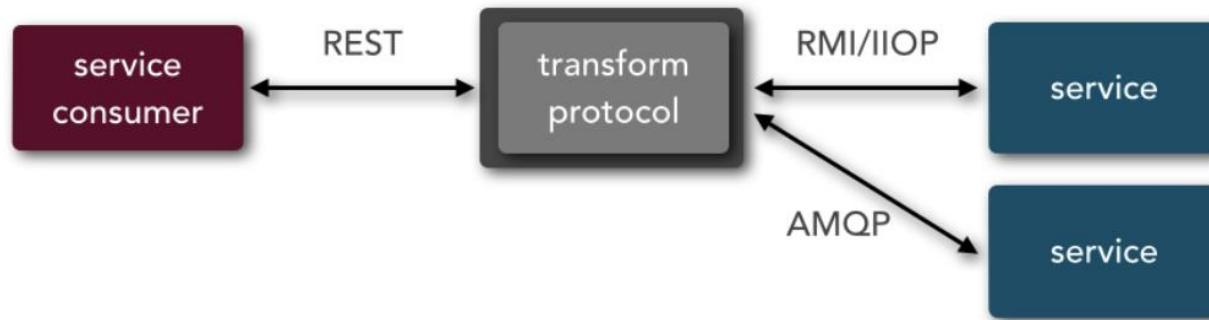
*Message transformation* - modify the format of the data from one type to other.

Ex. the service consumer is calling a service and sending the data in JSON format, whereas the service requires a Java object.



*Protocol transformation* - have a service consumer call a service with a protocol that differs from what the service is expecting.

Ex. the service consumer is communicating through REST, but the services invoked require an RMI/IOP connection and an AMQP connection.



# SOA

# VS.

# MICROSERVICES

- Share-as-much-as-possible
- Uses orchestration and choreography
- Uses Message middleware
- Coarse-grained services
- No pre-described limits as to which remote-access protocols can be used

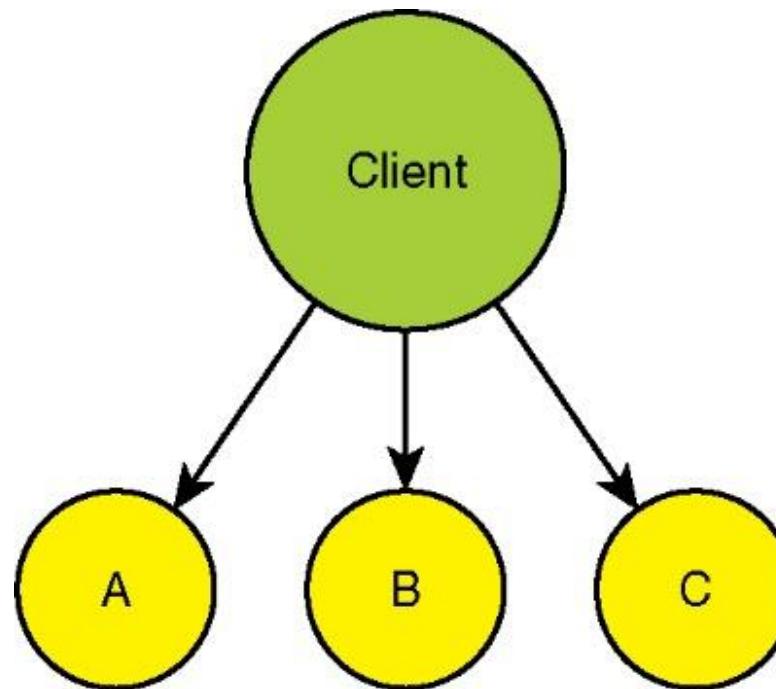
- Share-as-little-as-possible
- Favorizes choreography
- Uses API layer as service access façade
- Fine-grained services
- rely on 2 different remote-access protocols to access services: REST and simple messaging (JMS, MSMQ, AMQP, etc.)

# IDESIGN METHOD FOR SOA

- Decomposition
- Structure
- Composition
- Example

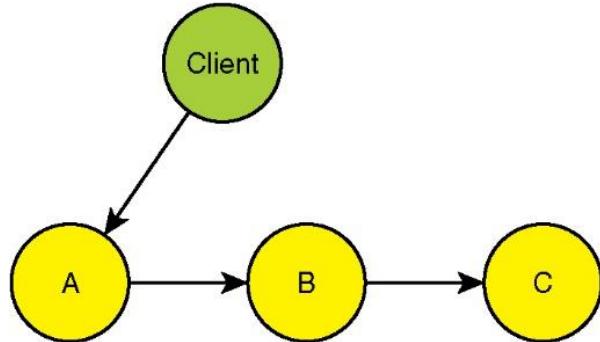
# DECOMPOSITION

- Functional?
- Bloated client orchestrating functionality

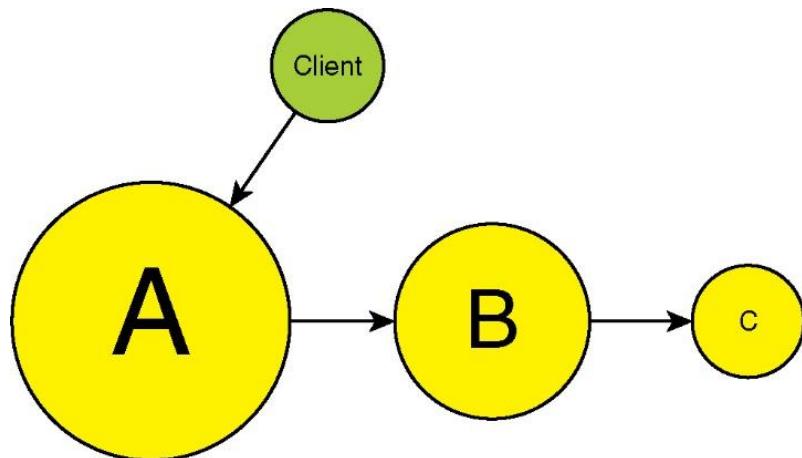


# FUNCTIONAL DECOMPOSITION

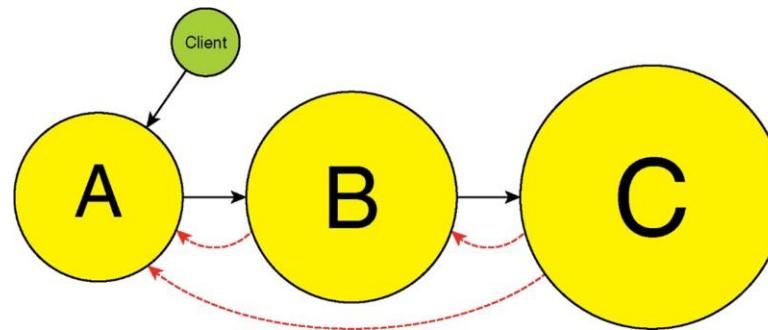
- Chaining functional services (choreography)



- Bloated services



- Additional bloating and coupling



# WHEN TO USE FUNCTIONAL DECOMPOSITION

- requirements discovery technique
- uncover requirements and their relationship,
- structure the requirements in a tree-like manner,
- identify redundancies or mutually exclusive functionalities

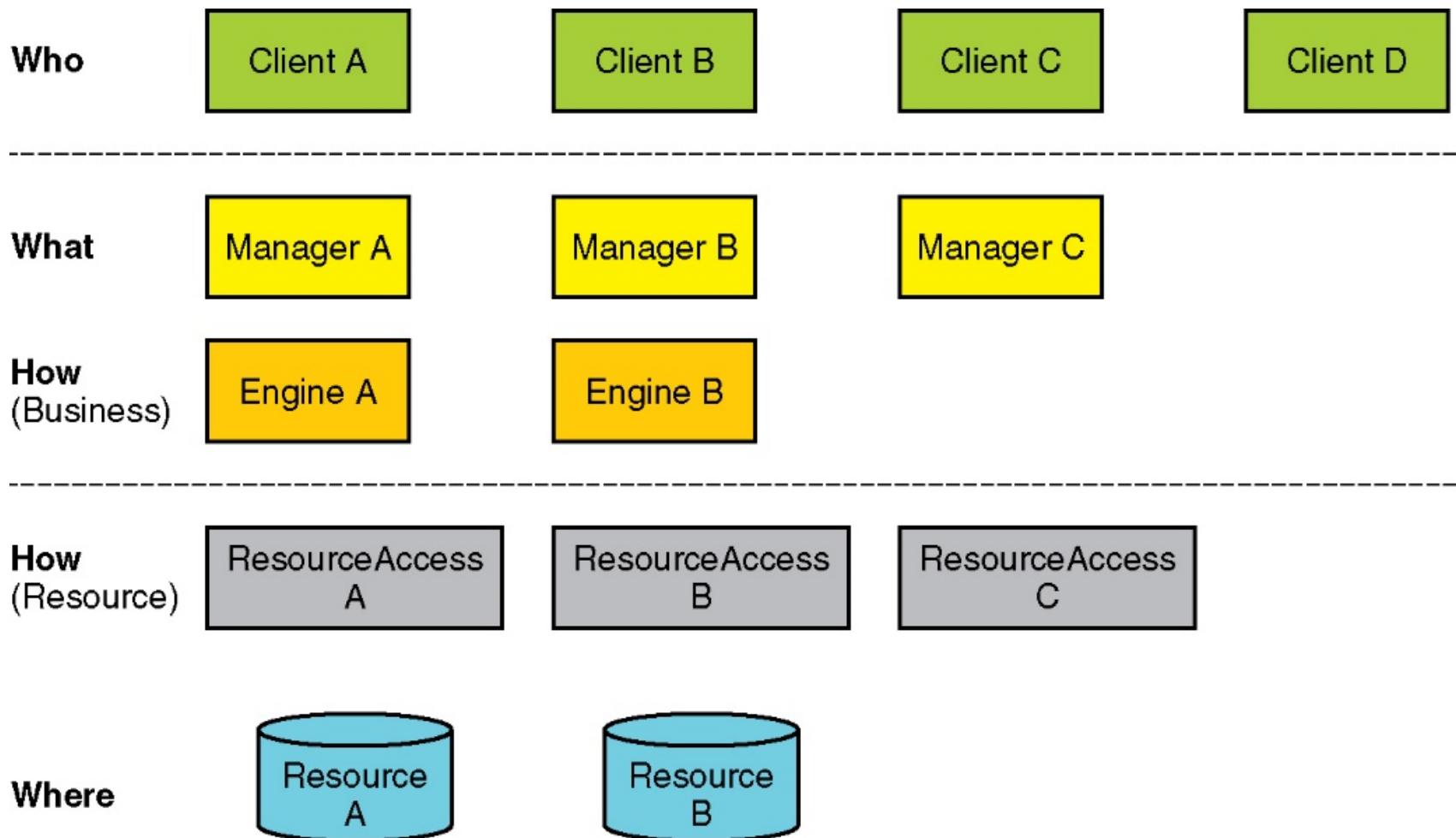
# DECOMPOSITION

- Domain driven ?
- Volatility driven



Change

# IDESIGN STRUCTURE



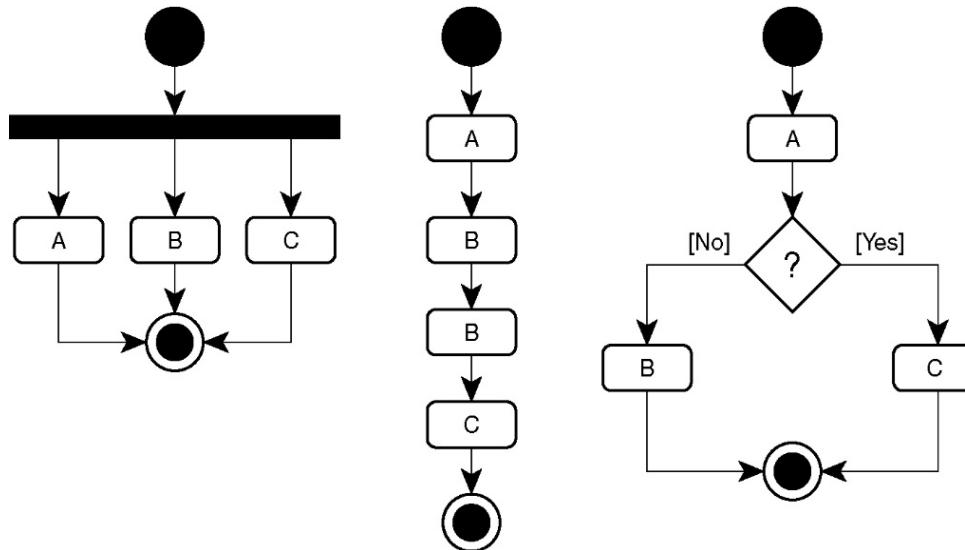
# CLIENT LAYER VOLATILITIES

- Client types: web application, rich desktop application, mobile app, holograms, APIs, etc.
- Different technologies, deployments, versions, life cycles, development teams
- Same entry points to the system, same access security, data types

# BUSINESS LOGIC LAYER VOLATILITIES

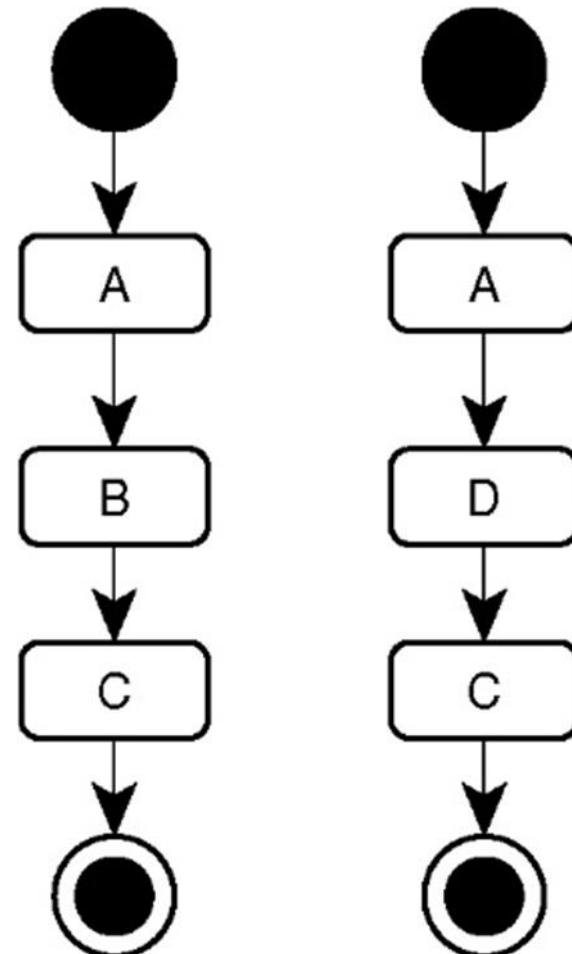
- Axes of volatility
  - Same client over time
  - Several clients at one point in time
- Sequence volatility => **Manager service**
  - ⇒ Encapsulates a family of logically related use-cases

Orchestration



# BLL VOLATILITIES - 2

- Activity volatility
  - ⇒ Engine service
  - ⇒ Encapsulates business rules and activities



# MANAGERS VS ENGINES

- Managers may use zero or more Engines
- Engines may be shared between Managers
- Engines should be highly reusable

# RESOURCE ACCESS AND RESOURCES

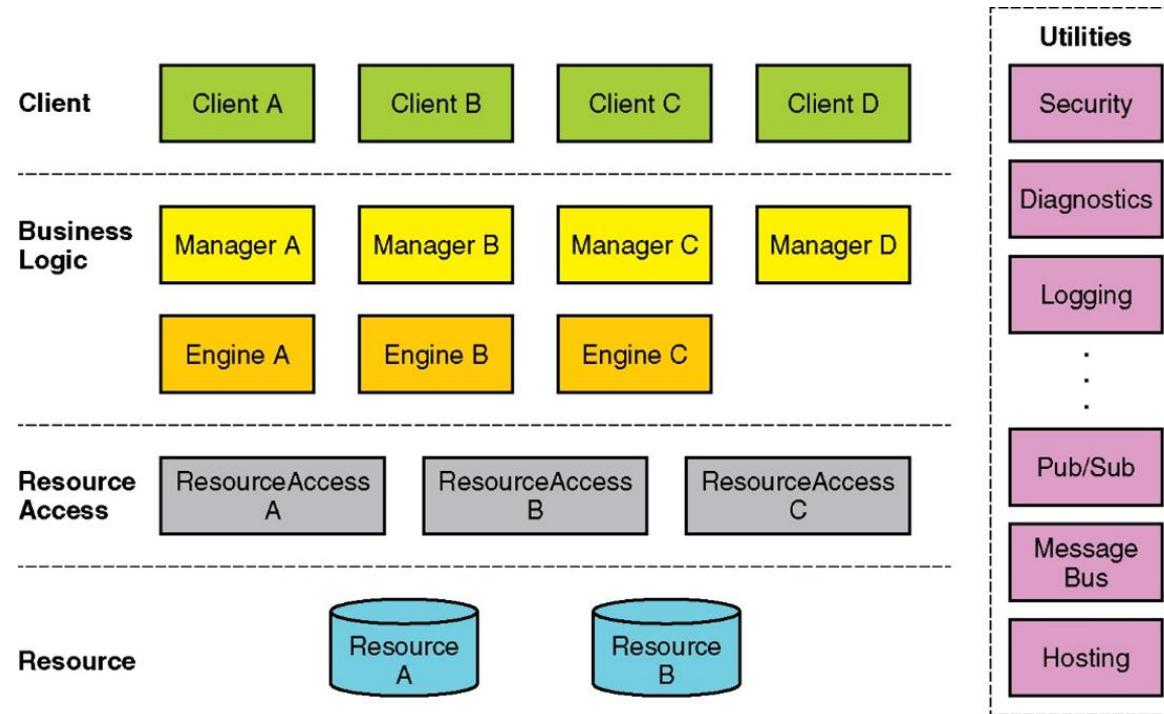
- ResourceAccess services encapsulate the volatility in accessing a Resource
- A Resource can be a relational database, file system, a cache, a message queue, a distributed cloud-based hash-table, etc.
- Well-designed ResourceAccess components expose in their contract the atomic business verbs around a resource.
- Example: in a banking system, atomic business verbs are *debit* and *credit* operations on an Account.
- The ResourceAccess component translates atomic business verbs (i.e. transactions) into CRUD operations on the specific Resource

# RESOURCE ACCESS AND RESOURCES

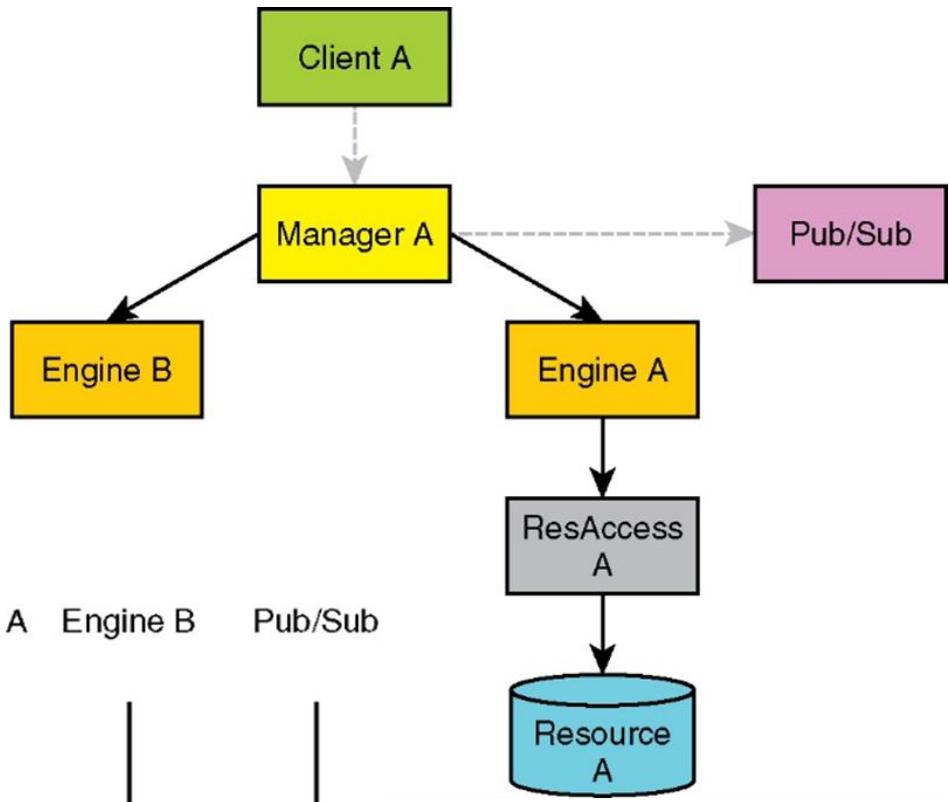
- ResourceAccess services can be shared between Managers and Engines.
- The Resource can be internal to the system or outside the system. Often, the Resource is a whole system in its own right (ex. a SQL DBMS) .
- Resource changes invariably change ResourceAccess as well.

# UTILITIES

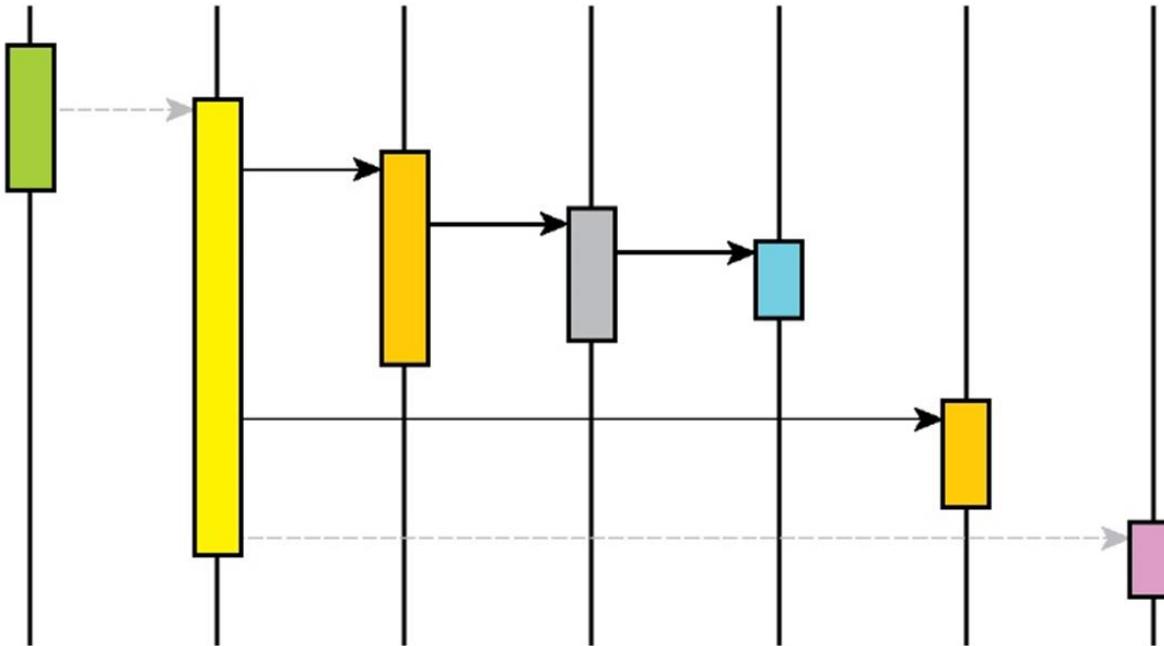
- Services building a common infrastructure that nearly all systems require
- *Utilities* may include Security, Logging, Diagnostics, Instrumentation, Pub/Sub, Message Bus, Hosting, etc.



# INTERACTIONS

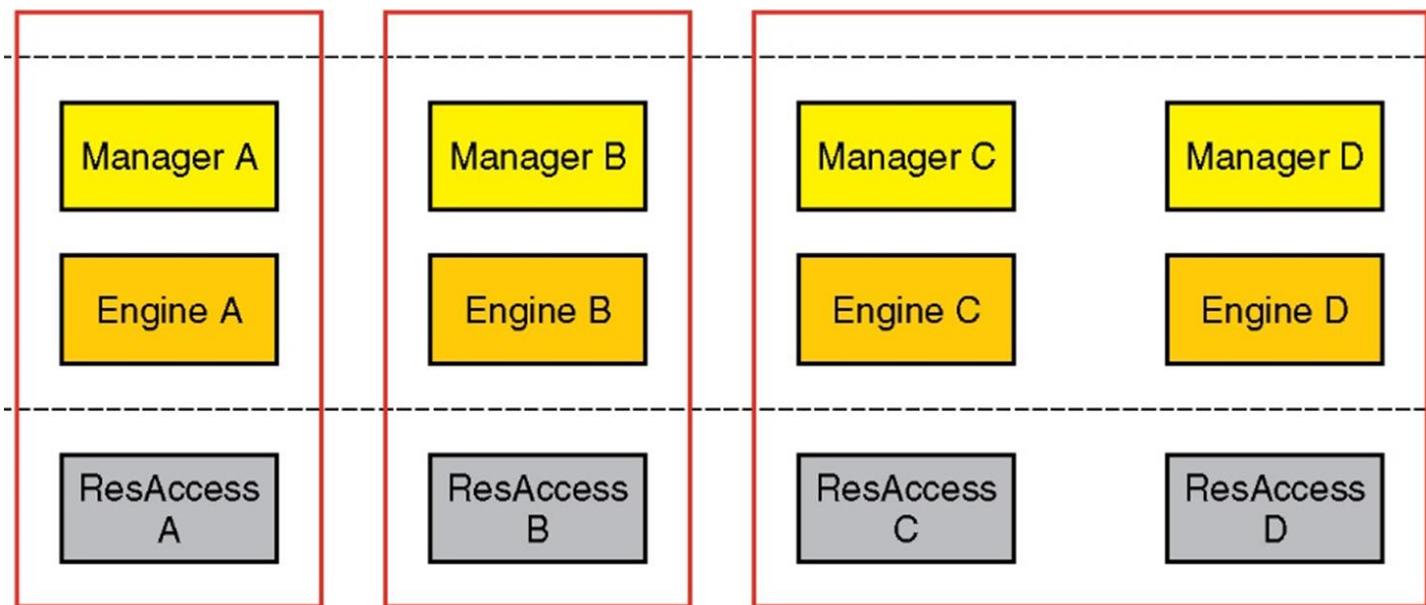


Client    Manager A    Engine A    Resource Access A    Resource A    Engine B    Pub/Sub



# COMMENTS

- Managers to *Engines* ratio
  - $8 > \# \text{Managers} > \# \text{Engines}$
- Volatility decreases top-down
- Reuse increases top-down



# DON'TS!

- *Clients* do not call multiple *Managers* in the same use case
- *Clients* do not call *Engines*
- *Managers* do not queue calls to more than one *Manager* in the same use case
- *Engines* do not receive queued calls
- *ResourceAccess* services do not receive queued calls
- *Clients* do not publish events
- *Engines* do not publish events
- *ResourceAccess* services do not publish events
- Resources do not publish events
- *Engines*, *ResourceAccess*, and *Resources* do not subscribe to events.
- *Engines* never call each other
- *ResourceAccess* services never call each other.

# COMPOSITION

- Objective:
  - Address the current requirements +
  - Withstand future requirements
  - Validate Design

⇒ ***Never design against the requirements***

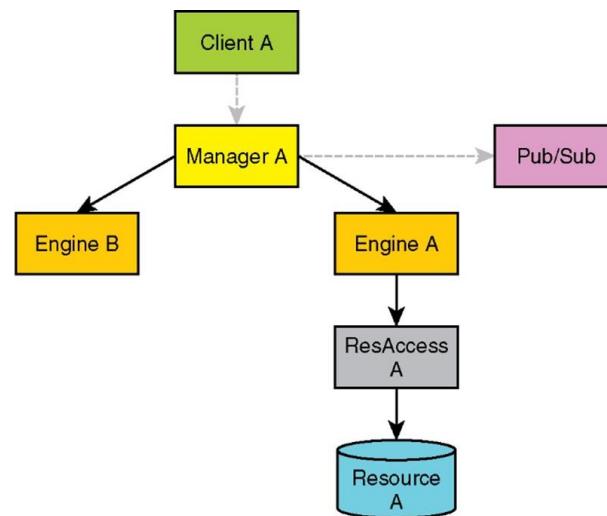
- Use-cases = behavior
- Identify Core Use-cases (usually <= 6)!
- Core use-cases represent the essence of the business
- Core use-cases are not explicitly stated
- Changing use-cases => different interaction between components,  
**not a different decomposition!**

# COMPOSABLE DESIGN

- Architects mission is to identify **the smallest set of components** that he can put together to **satisfy all the core use cases**.
- Example: Requirements specs have 300 use-cases. How many services?
  - 1 service => monolithic architecture 
  - 300 services => difficult to integrate 
  - Order of magnitude = 10 
- Typically: 2 – 5 Managers, 2 -3 Engines, 3 - 8 ResourceAccess and Resources, 2 - 6 Utilities.

# VALID DESIGN

- **Valid design** if you can produce an interaction between your services **for each core use case**
- Call chain diagram shows the interaction between components required to satisfy a particular use case
  - a solid black arrow for synchronous (request/response) calls,
  - a dashed gray arrow for a queued call



# FROM DESIGN TO FEATURES

- ***Features are always and everywhere aspects of integration, not implementation.***
- Containing the change:
- Manager implements the workflow executing the use case. The Manager may be gravely affected by a change.
- The underlying components that the Manager integrates are not affected by the change

# FROM DESIGN TO FEATURES

- Implementing *Engines* is expensive. Each *Engine* represents business activities vital to the system's workflows and encapsulates the associated volatility and complexity.
- Implementing a *ResourceAccess* is nontrivial. It involves:
  - Identifying the atomic business verbs,
  - translating them into the access methodologies for some *Resource*,
  - exposing them as a *Resource-neutral* interface.
- Designing and implementing *Resources* that are scalable, reliable, highly performant, and very reusable is time- and effort-consuming. These tasks may include:
  - designing data contracts, schemas, cache access policies, partitioning, replication, connection management, timeouts, lock management, indexing, normalization, message formats, transactions, delivery failures, poison messages, and much more.

# FROM DESIGN TO FEATURES

- Implementing *Utilities* always requires top skills, and the result must be trustworthy. *Utilities* are the backbone of your system. World-class security, diagnostics, logging, message processing, instrumentation, and hosting do not happen accidentally.
- Designing a superior user experience or a convenient and reusable API for *Clients* is time and labor intensive. *The Clients* also have to interface and integrate with the *Managers*.

# SYSTEM DESIGN EXAMPLE [IDESIGN CHAPTER 5]

- TradeMe, a system for matching tradesmen to contractors and projects.
- Each tradesman has a skill level, and some are certified by regulators to do certain tasks.
- The payment rate for the tradesman varies based on various factors such as discipline, skill level, years of experience, project type, location, and even weather.
- The contractors are general contractors, and they need tradesmen on an ad hoc basis, from as little as a day to as long as a few weeks.
- Tradesmen can come and go on a single project.
- Tradesmen can sign up, list their skills, their general geographic area of availability, and the rate they expect.
- Contractors can sign up, list their projects, the required trades and skills, the location of the projects, the rates they are willing to pay, the duration of engagement, and other attributes of the project. Contractors can even request specific tradesmen with whom they would like to work.

# TRADEME BEHAVIOR

- The system lets market forces set the rate and find equilibrium.
- The projects are construction projects for buildings. The system may also be useful in newly emerging markets, such as oil fields or marine yards.
- The system processes the requests and dispatches the required tradesmen to the work sites. It also keeps track of the hours and wages, and the rest of the reporting to the authorities.
- The system isolates tradesmen from contractors. It collects funds from the contractors and pays the tradesmen. Contractors cannot bypass the system and hire the tradesmen directly
- TradeMe aims to find the best rate for the tradesmen and the most availability for the contractors.
- Both tradesmen and contractors are members in the system and pay an (annual) fee.

# TRADEME LEGACY STATUS

- Presently, nine call centers handle the majority of the assignments. Each call center is specific to a particular locale, regulations, building codes, standards, and labor laws.
- Call centers are staffed with account representatives called reps.
- The legacy system, deployed in European call centers, has full-time users using a two-tier desktop application connected to a database
- Some rudimentary web portals for managing membership bypass the legacy system and work with the database directly
- Users are required to employ as many as five different applications to accomplish their tasks.
- The client applications are independent, each has its own repository, is full of business logic, and the UI and business logic are not separated.
- Vulnerable to security attacks. Was never designed at all, but rather grew organically.

# TRADEME NEW DESIRED FEATURES

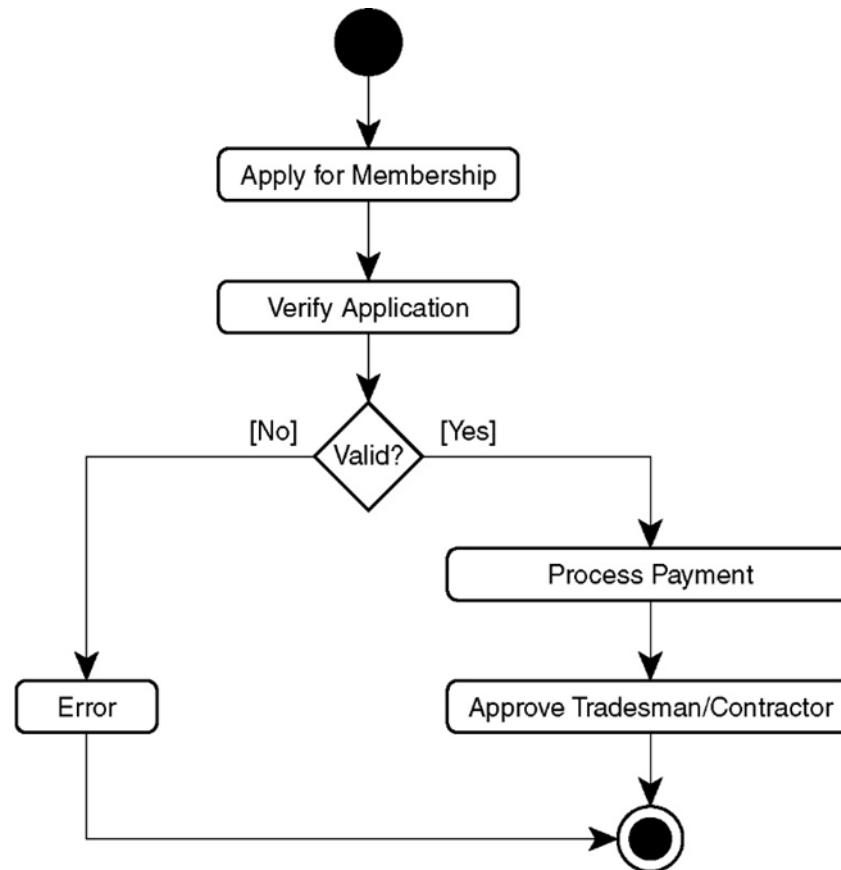
- Mobile device support
- Higher degree of automation of the workflow
- Some connectivity to other systems
- Migration to the cloud
- Fraud detection
- Quality of work surveys, including incorporating the tradesman's safety record in the rate and skill level
- Entering new markets (such as deployment at marine yards)

# TRADEME CHALLENGES

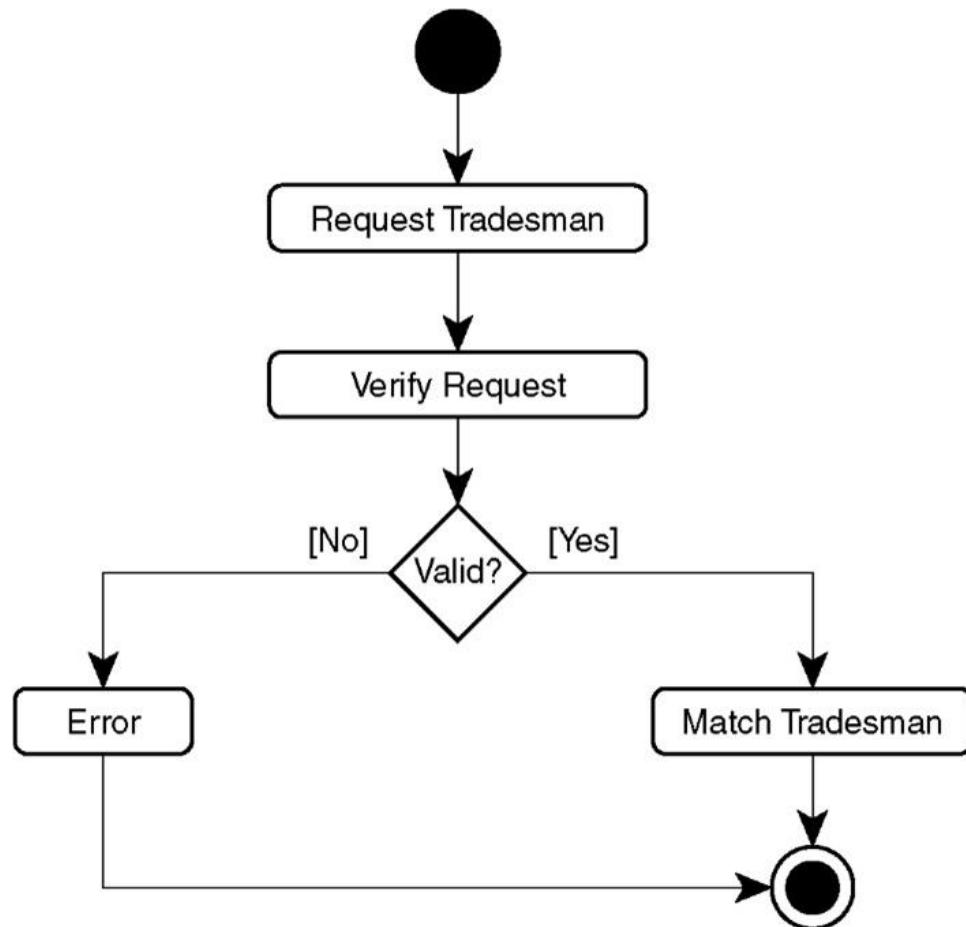
- inability to integrate new features (i.e. integration with external education centres for continuing education of tradesmen)
- adapting to new legislation across locales
- automated processing flows

# TRADEME USE CASES

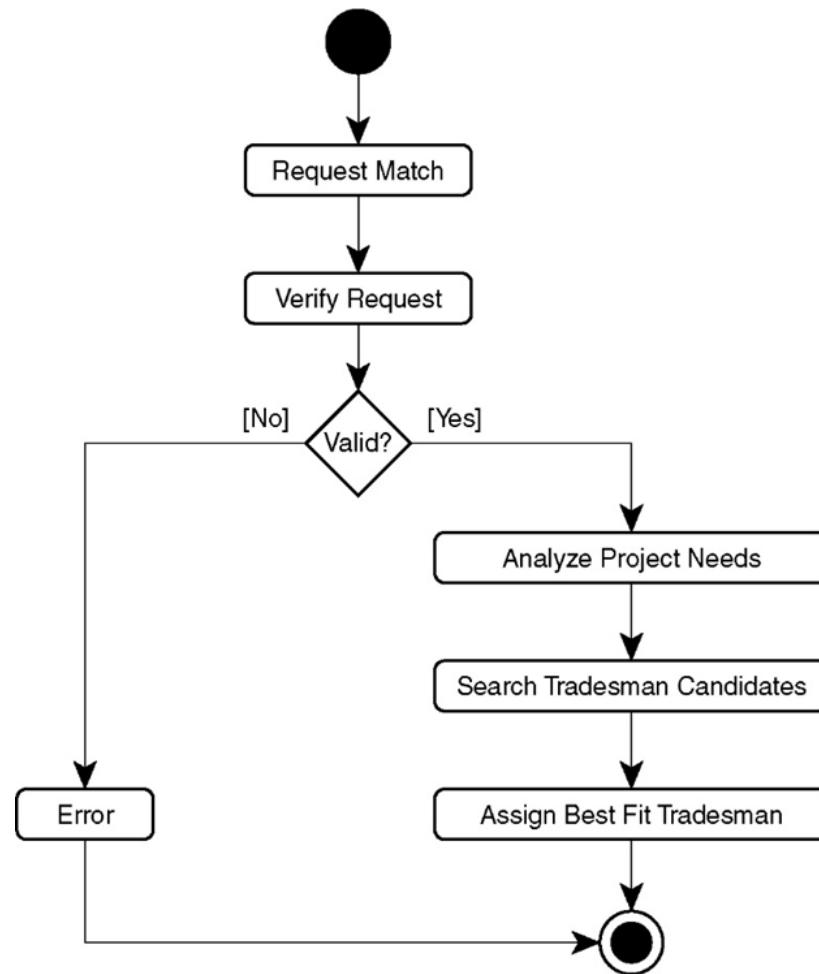
- Add Tradesman or Contractor use case



# REQUEST TRADESMAN OR CONTRACTOR USE CASE



# MATCH TRADESMAN USE CASE



# CORE USE CASE

**TradeMe is a system for matching tradesmen to contractors and projects**

- 3 types of roles:
  - the users (i.e. back-office data entry reps, sysadmins),
  - the market,
  - the members (tradesmen, contractors)

# ANTI-DESIGN APPROACHES

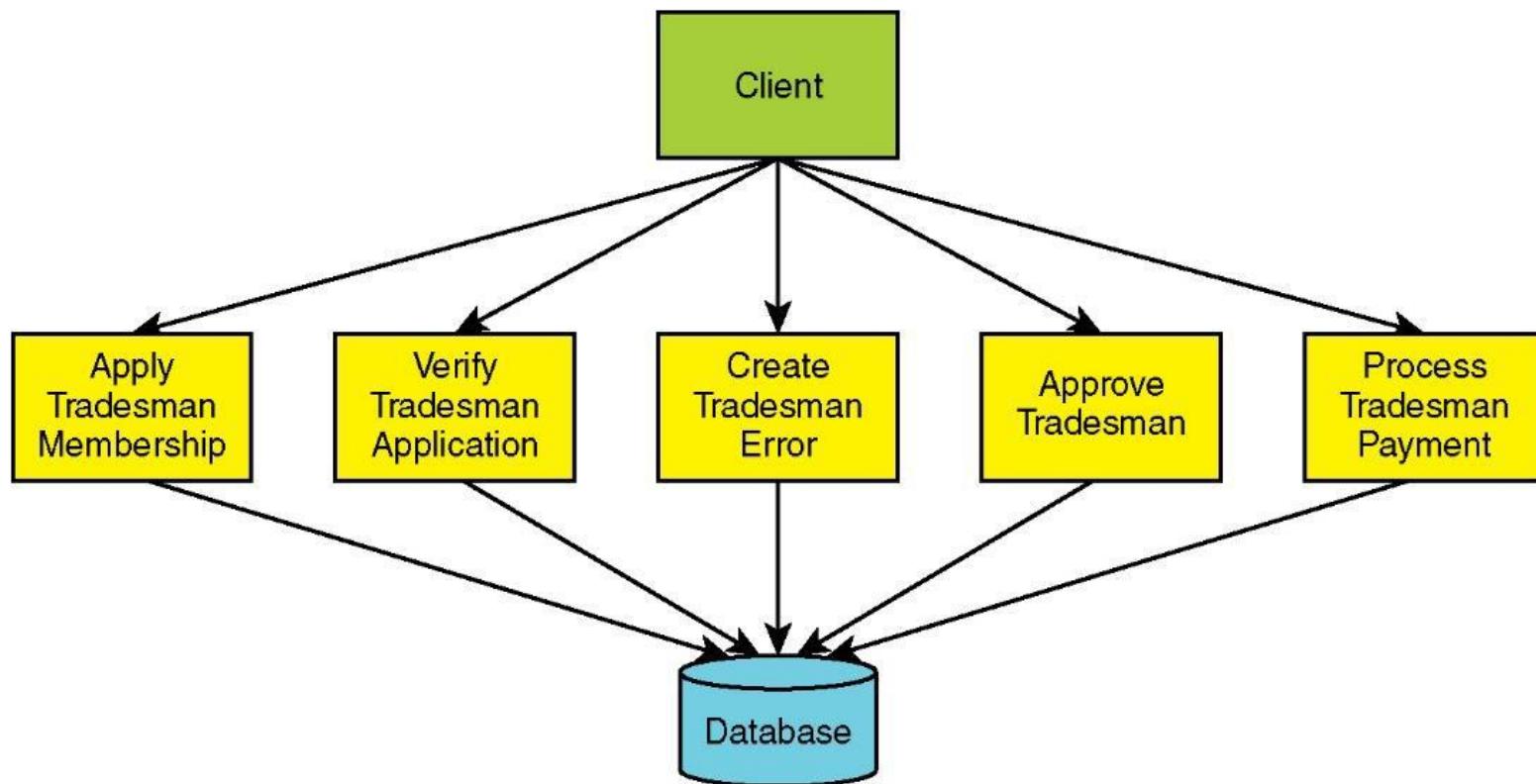
- Monolith – one big service 
- Granular components – one component per activity 

										Contractor Client	Tradesman Client	Admin Client
Apply Tradesman Membership	Apply Contractor Membership	Request Tradesman	Request Match	Request Assign Tradesman	Request Termination Tradesman	Schedule Timer	Request Create Project	Add Project Duration	Request Close Project			
Verify Tradesman Application	Verify Contractor Application	Verify Tradesman Request	Verify Match Request	Verify Assign Request	Verify Termination Request	Find Scheduled Payments	Verify Project Request	Activate Project	Verify Close Project Request			
Process Tradesman Payment	Process Contractor Payment	Match Tradesman	Analyze Project Needs	Verify Availability	Terminate Tradesman From Project	Pay Tradesman	Add Required Trades	Create Project Creation Err.	Release Tradesman			
Approve Tradesman	Approve Contractor	Create Tradesman Request Err.	Search Candidates	Assign Tradesman	Make Tradesman Available		Add Required Skills	Add Bill Rates	Close Project			
Create Tradesman Error	Create Contractor Error	Create Tradesman Match Err.	Assign Best Candidates	Create Availability Error	Create Termination Error		Add Project Location		Create Close Project Error			



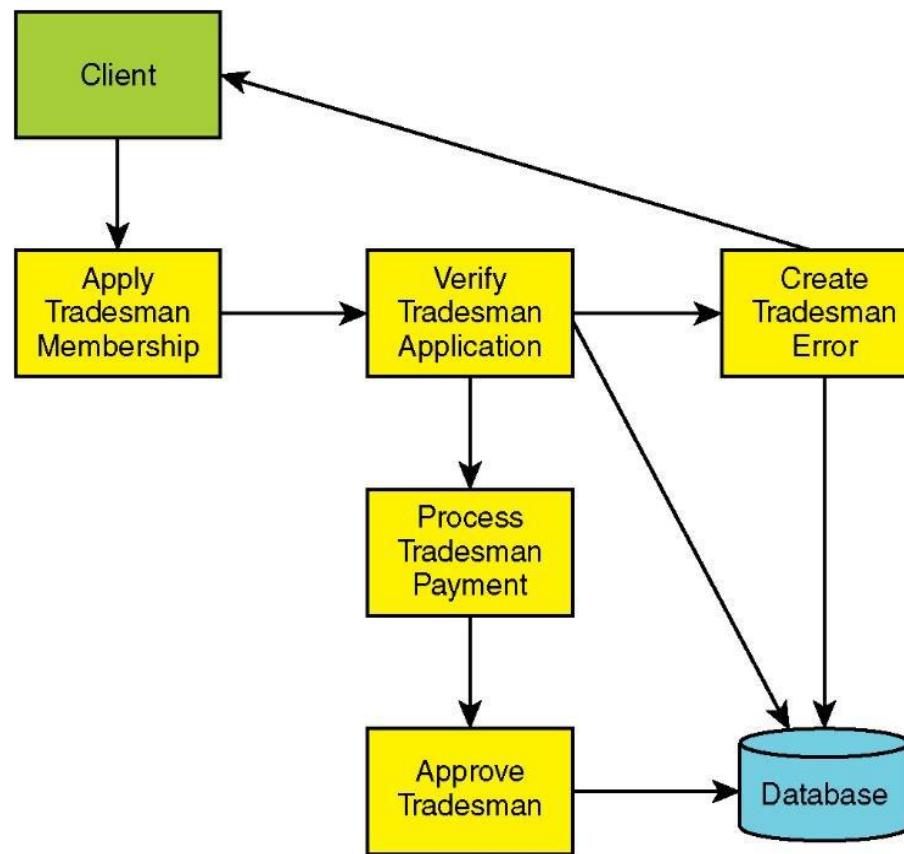
# CLIENT AS ORCHESTRATOR

- Granular components – one component per activity



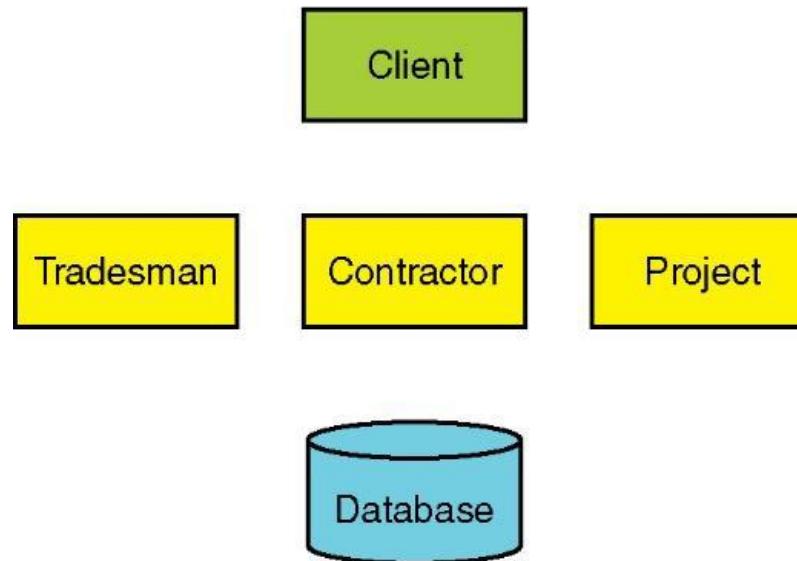
# CHOREOGRAPHY – BASED APPROACH

- Granular components – one component per activity



# DOMAIN DECOMPOSITION

- Large number of decomposition alternatives
- Difficult to validate the design
- Example: who handles a request for a tradesman? The Project? The Tradesman?



# IDESIGN METHOD

- The Who
  - Tradesmen
  - Contractors
  - TradeMe reps
  - Education centers
  - Background processes (i.e., scheduler for payment)
- The What
  - Membership of tradesmen and contractors
  - Marketplace of construction projects
  - Certificates and training for continuing education

# DESIGN METHOD

- The How
  - Searching
  - Complying with regulations
  - Accessing resources
- The Where
  - Local database
  - Cloud
  - Other systems

# AREAS OF VOLATILITY - 1

- Client applications
  - different users (i.e. tradesmen, contractors etc. background processes)
  - different UI technologies, devices, APIs
  - different access (local, net)
- Membership management (Membership Manager)
  - different benefits/discounts
  - different local rules
- Fees (Market Manager)
  - all the ways TradeMe makes money

# AREAS OF VOLATILITY - 2

- Projects (Market manager)
  - different requirements and sizes
- Disputes (Membership Manager)
  - different dispute resolution methods
- Matching and approvals
  - searching criteria and their definition (Market Manager)
  - searching methods for a tradesman (Search Engine)
- Education
  - matching a training class to a tradesman (Education Manager)
  - searching for classes and certifications (Search Engine)
  - compliance with certification regulations (Regulation Engine)

# AREAS OF VOLATILITY - 3

- Regulations (Regulation Engine)
  - different internal, external regulations
- Reports (Regulation Engine)
  - different types of reports
- Localization (Regulation Engine)
  - language and culture
  - local specific regulations
  - may affect the design of Resources
- Resources
  - portals to external systems
  - cloud-based database for tradesmen, projects, contractors
  - local database

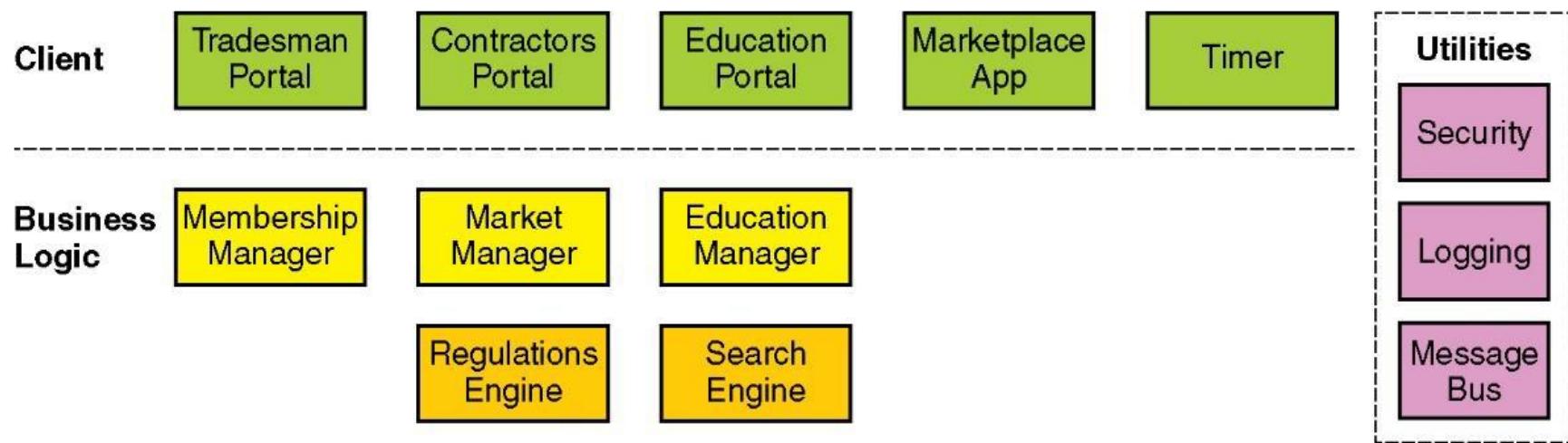
# AREAS OF VOLATILITY - 4

- Resource Access (Regulation Engine)
  - different internal, external regulations
- Deployment model (Message Bus Utility)
  - Sometimes data cannot leave a geographic area, or
  - the company may wish to deploy parts or whole systems in the cloud.
- Authentication and authorization (Security Utility)
  - multiple options for representing credentials and identities.
  - many ways of storing roles or representing claims.

# WEAK VOLATILITIES

- Notification (Regulation Engine)
  - different internal, external regulations
- Analysis (Regulation Engine)
  - different types of reports

# STATIC ARCHITECTURE



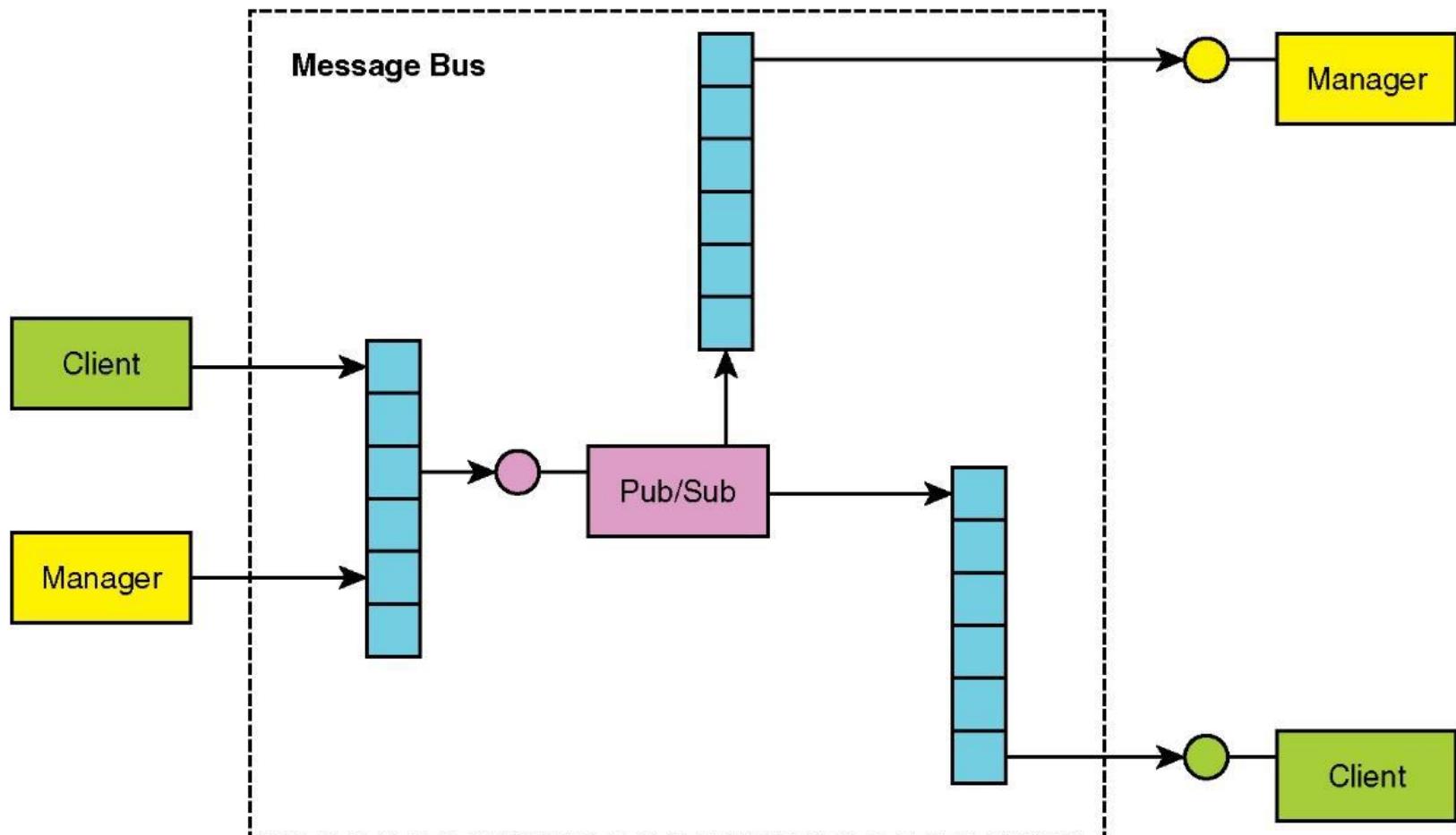
## Resource Access



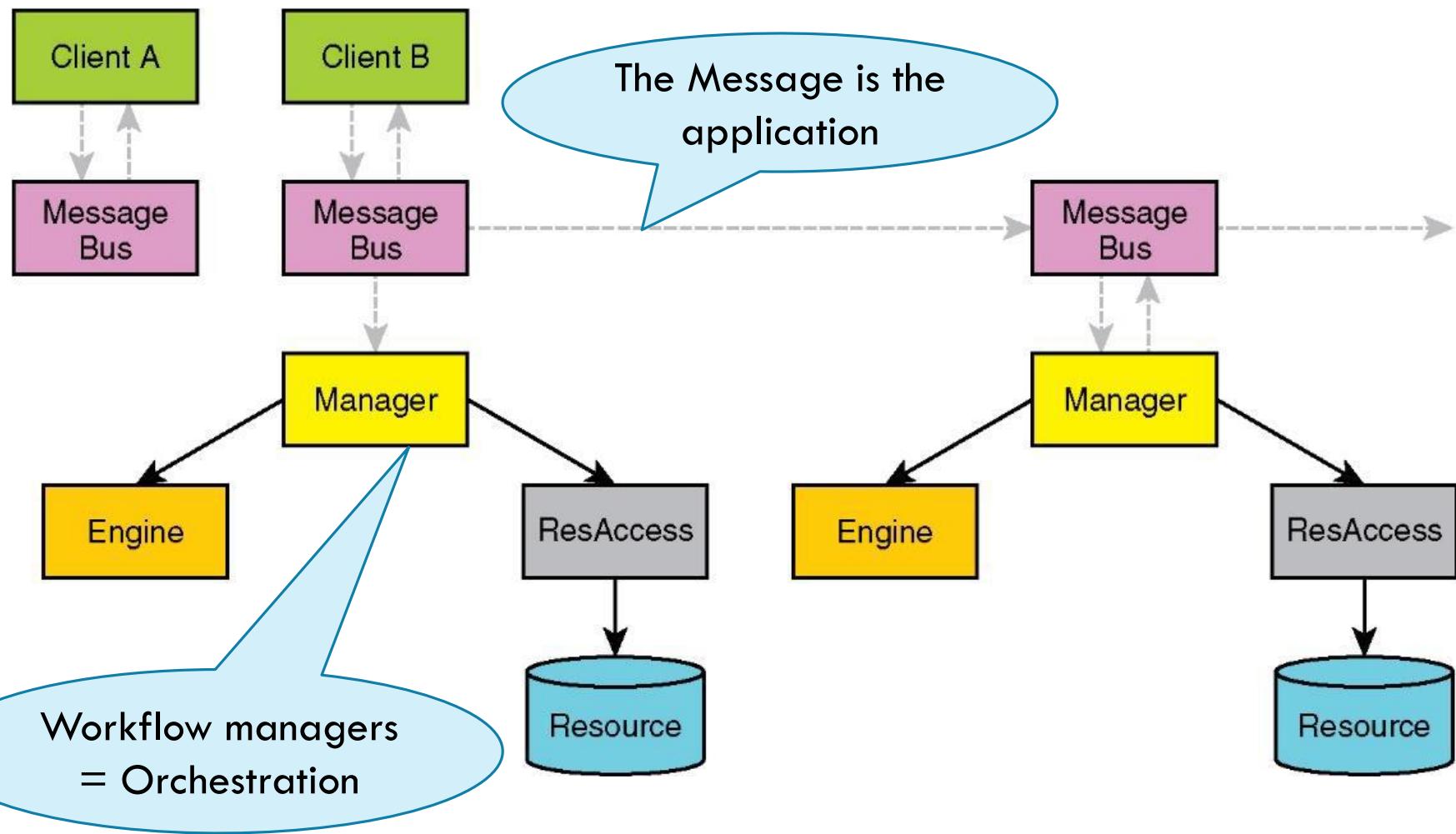
## Resources



# MESSAGE BUS



# OPERATIONAL INTERACTION



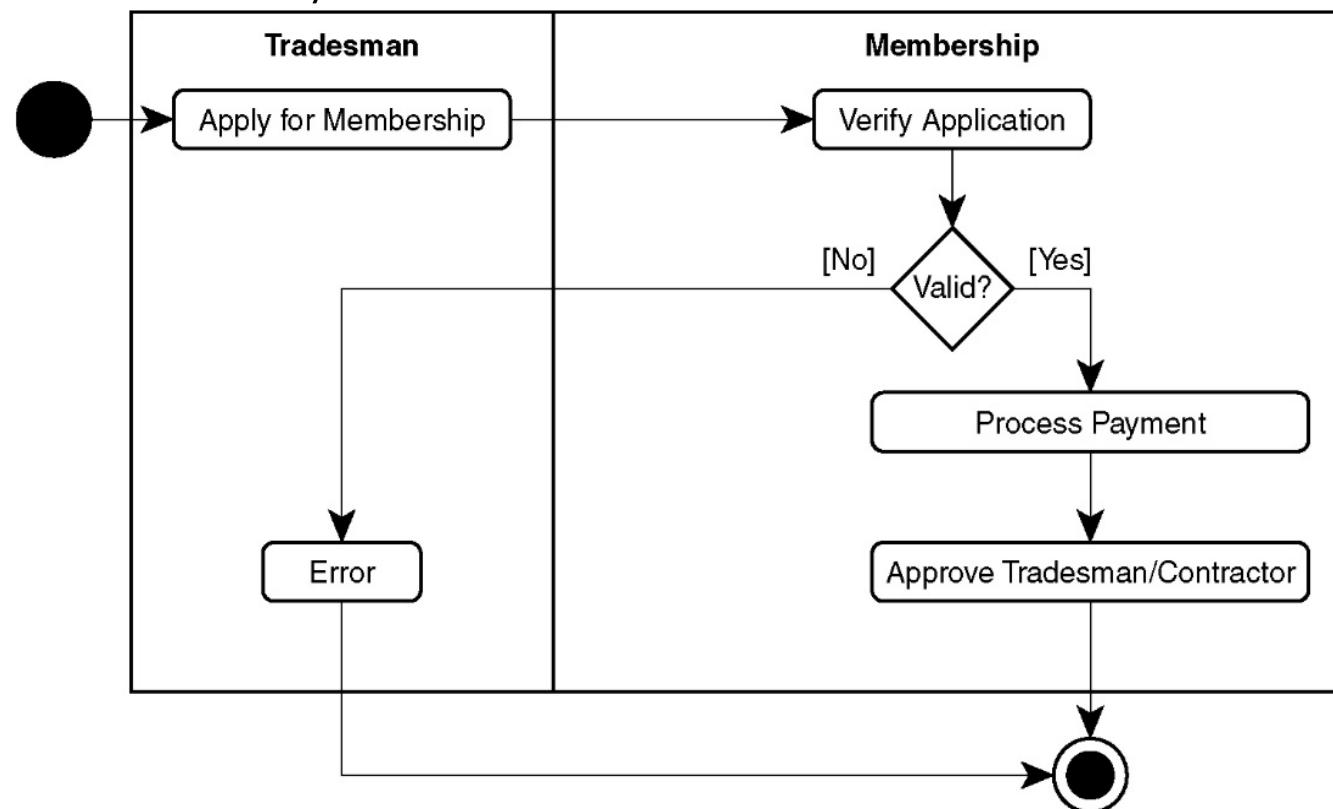
# OPERATIONAL INTERACTION

- The *Clients* and the business logic in the subsystems are **decoupled** from each other by the **Message Bus**
- The **Message Bus encapsulates** the nature of the messages, the location of the parties, and the communication protocol
- No use case initiator (i.e. *Clients*) and use case executioner (i.e. Managers) ever interact directly.
- A multiplicity of **concurrent Clients can interact in the same use case**, with each performing its part of the use case.
- **High throughput** is possible because the queues underneath the Message Bus can accept a very large number of messages per second.

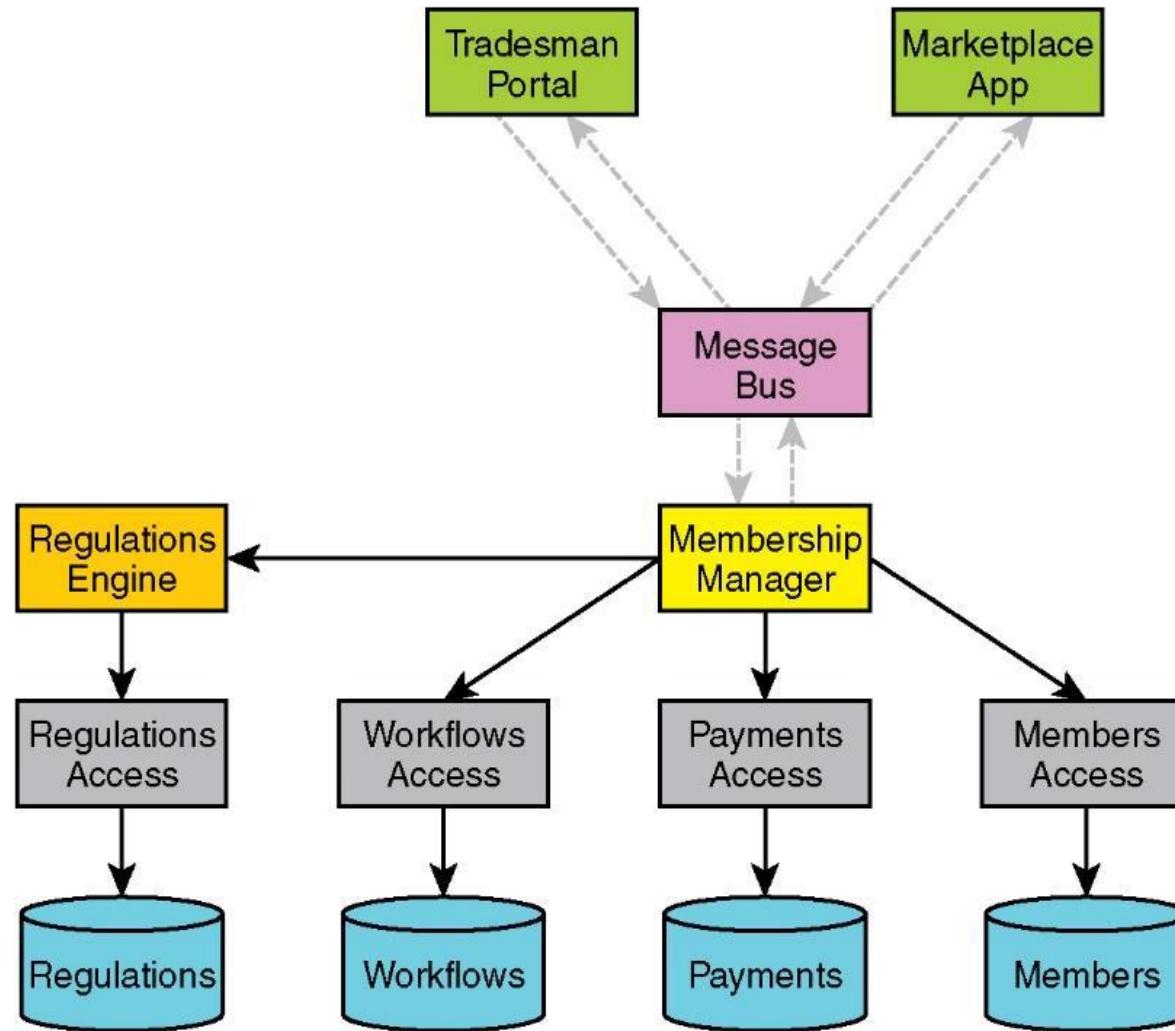
# ARCHITECTURE VALIDATION

Take each use-case and show how the components integrate to support it.

## Add Tradesman/Contractor Use Case



# ADD TRADESMAN/CONTRACTOR USE CASE

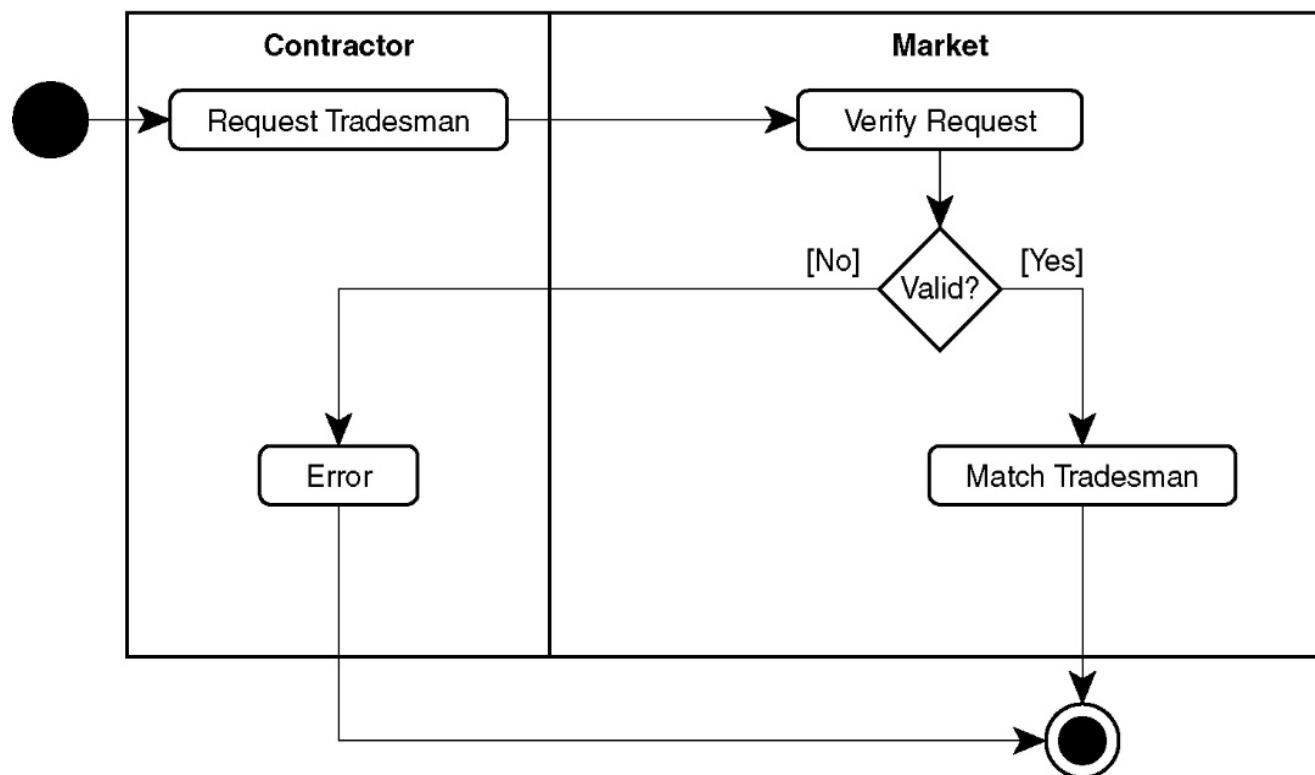


# ADD TRADESMAN/CONTRACTOR USE CASE

- Upon receiving the message, the Membership Manager (which is a workflow Manager) loads the appropriate workflow from the workflow storage.
- This either kicks off a new workflow or rehydrates an existing one to carry on with the workflow execution.
- Once the workflow has finished executing the request, the Membership Manager posts a message back into the Message Bus indicating the new state of the workflow, such as its completion,
- Clients can monitor the Message Bus as well and update the users about their requests.
- The Membership Manager
  - consults the Regulation Engine that is verifying the tradesman or contractor,
  - adds the tradesman or contractor to the Members store,
  - updates the Clients via the Message Bus.

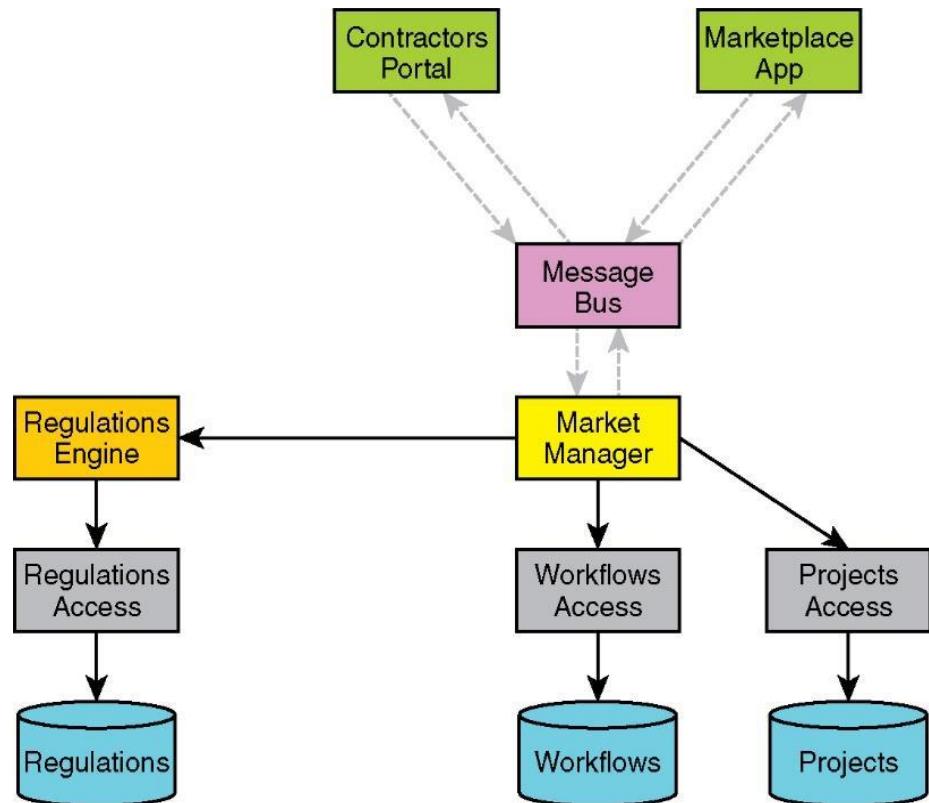
# REQUEST TRADESMAN USE CASE

After initial verification of the request, this use case triggers another use case, Match Tradesman



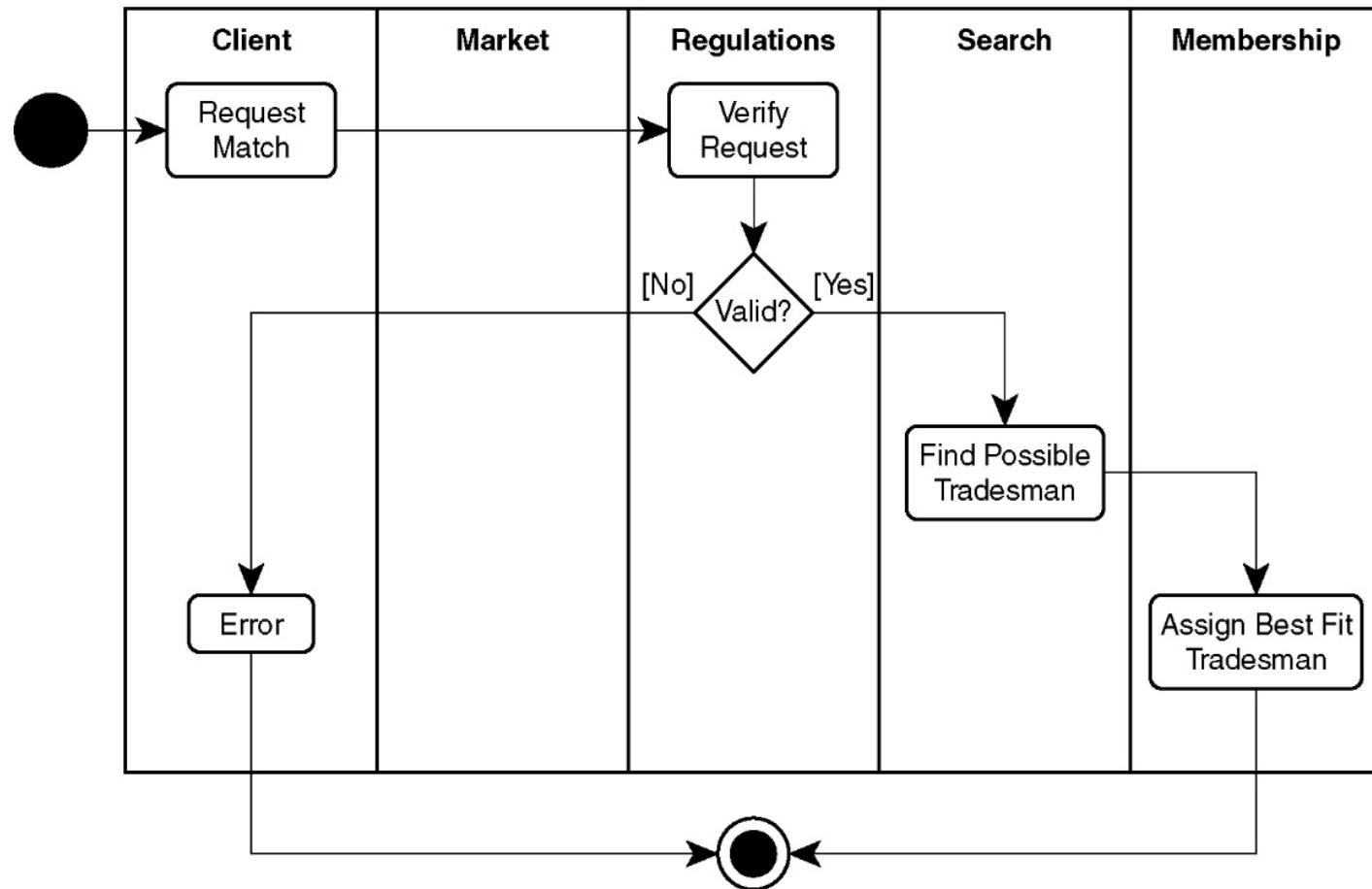
# REQUEST TRADESMAN USE CASE

- Clients post a message to the bus requesting a tradesman.
- Market Manager receives that message and loads the workflow corresponding to this request
- Market Manager consults the Regulation Engine about what may be valid for this request or updates the project with the request for a tradesman.
- The Market Manager can then post back to the Message Bus that someone is requesting a tradesman =>trigger the matching and assignment workflows.



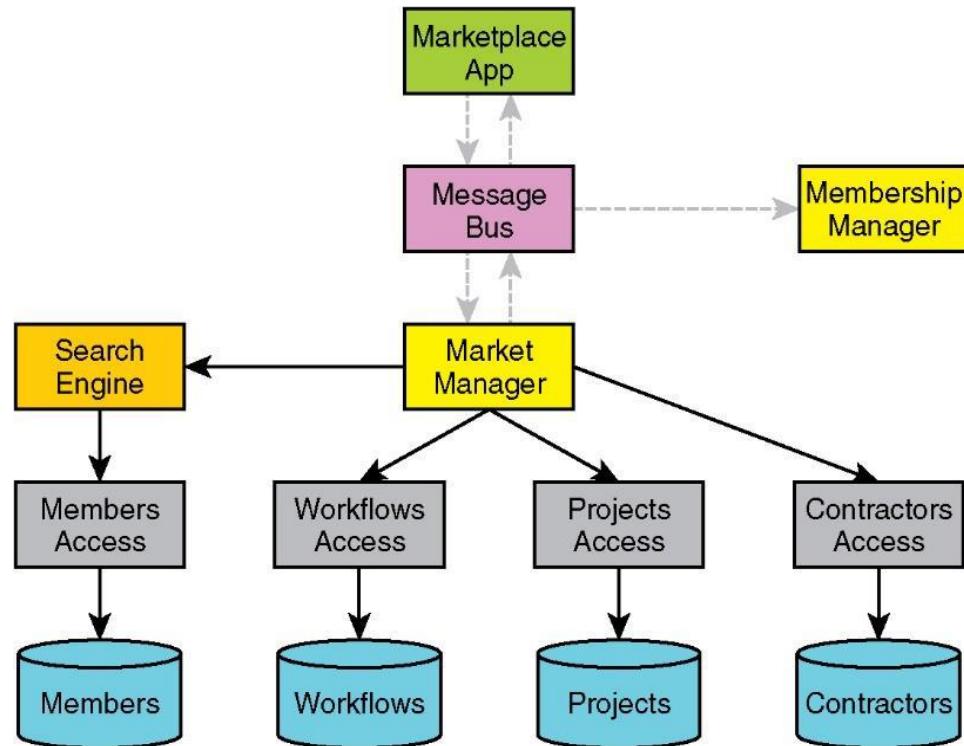
# MATCH TRADESMAN USE CASE

- involves multiple areas of interest



# MATCH TRADESMAN USE CASE

- Market Manager loads the workflow corresponding to this request
- Market Manager executes the workflow ..
- The Market Manager posts back to the Message Bus
- Message Bus sends message to Membership Manager



# IDESIGN DISCUSSION

- A **directive** is a rule that you should never violate, since doing so is certain to cause the project to fail.
- A **guideline** is a piece of advice that you should follow unless you have a strong and unusual justification for going against it. Violating a guideline alone is not certain to cause the project to fail, but too many violations will tip the project into failure.

# DIRECTIVES

## THE PRIME DIRECTIVE

Never design against the requirements.

## DIRECTIVES

Avoid functional decomposition.

Decompose based on volatility.

Provide a composable design.

Offer features as aspects of integration, not implementation.

Design iteratively, build incrementally.

# GUIDELINES - 1

## Requirements

- Capture required behavior, not required functionality.
- Describe required behavior with use cases.
- Document all use cases that contain nested conditions with activity diagrams.
- Eliminate solutions masquerading as requirements.
- Validate the system design by ensuring it supports all core use cases.

## Cardinality

- Avoid more than five Managers in a system without subsystems.
- Avoid more than a handful of subsystems.
- Avoid more than three Managers per subsystem.
- Strive for a golden ratio of Engines to Managers.
- Allow ResourceAccess components to access more than one Resource if necessary.

# GUIDELINES - 2

## Attributes

- Volatility should decrease top-down.
- Reuse should increase top-down.
- Do not encapsulate changes to the nature of the business.
- Managers should be almost expendable.
- Design should be symmetric.
- Never use a public communication channels for internal system interactions.

## Layers

- Avoid open architecture.
- Avoid semi-closed/semi-open architecture.
- Prefer a closed architecture.
  - Do not call up.
  - Do not call sideways (except queued calls between Managers).
  - Do not call more than one layer down.
  - Resolve attempts at opening the architecture by using queued calls or asynchronous event publishing.
- Extend the system by implementing subsystems.

# GUIDELINES - 3

## Interaction rules

- All components can call *Utilities*.
- Managers and *Engines* can call *ResourceAccess*.
- Managers can call *Engines*.
- Managers can queue calls to another Manager.

## Interaction don'ts

- Clients do not call multiple Managers in the same use case.
- Managers do not queue calls to more than one Manager in the same use case.
- Engines do not receive queued calls.
- ResourceAccess components do not receive queued calls.
- Clients do not publish events.
- Engines do not publish events.
- ResourceAccess components do not publish events.
- Resources do not publish events.
- Engines, ResourceAccess, and Resources do not subscribe to events.

# WRAP-UP

- Modern, proven approach for designing service based architectures.
- Needs experience to correctly identify and design services



# SOFTWARE DESIGN

Data Access

# LAST TIME

- Modelling the business logic layer
  - Domain driven design
  - Volatility driven design

# CONTENT

## Hybrid Data Source Pattern

- Active Record
- Table Module

## Data Access

- Gateways (DAO + DTO)
- Data Mapper

## Object-Relational Structural Patterns

## Object-Relational Behavioral Patterns

- Lazy Load
- Identity Map

# REFERENCES

**Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]**

Microsoft Application Architecture Guide, 2009 [MAAG]

Paulo Sousa, Instituto Superior de Engenharia do Porto, Patterns of Enterprise Applications by example

<http://jayurbain.com/msoe/se380/outline.html>

Sun: Core J2EE Pattern Catalog

<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

# ORGANIZING THE BUSINESS LOGIC ACCORDING TO FOWLER

Key architectural decisions, which influence structure of other layers.

## Pure patterns

- Transaction Script
- Domain Model

## Hybrid patterns

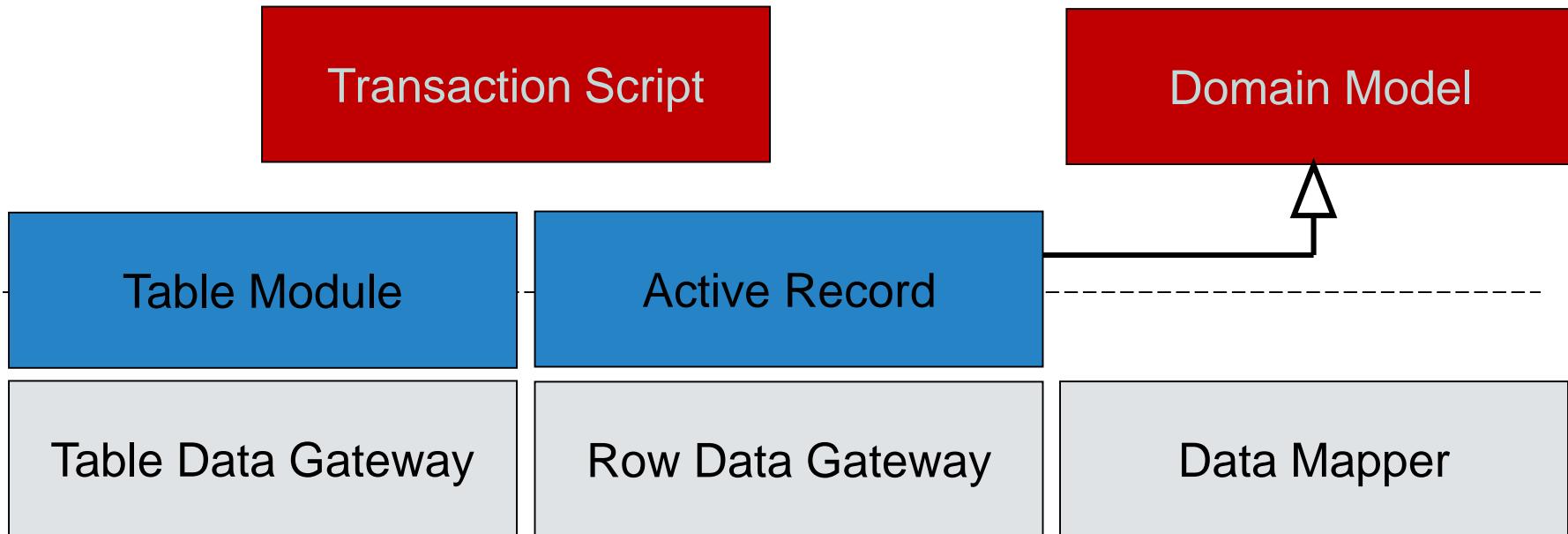
- Active Record
- Table Module

# DOMAIN LOGIC PATTERNS

Presentation



Domain



Data Source

# TRANSACTION SCRIPT

A TS organizes the business logic primarily as a single procedure where each procedure handles a single request from the presentation. [Fowler]

The TS may make calls directly to the DB or through a thin DB wrapper.

Think of a script for a use case or business transaction.

Remember “use-case controller”?

# TRANSACTION SCRIPT

... is essentially a procedure that takes the

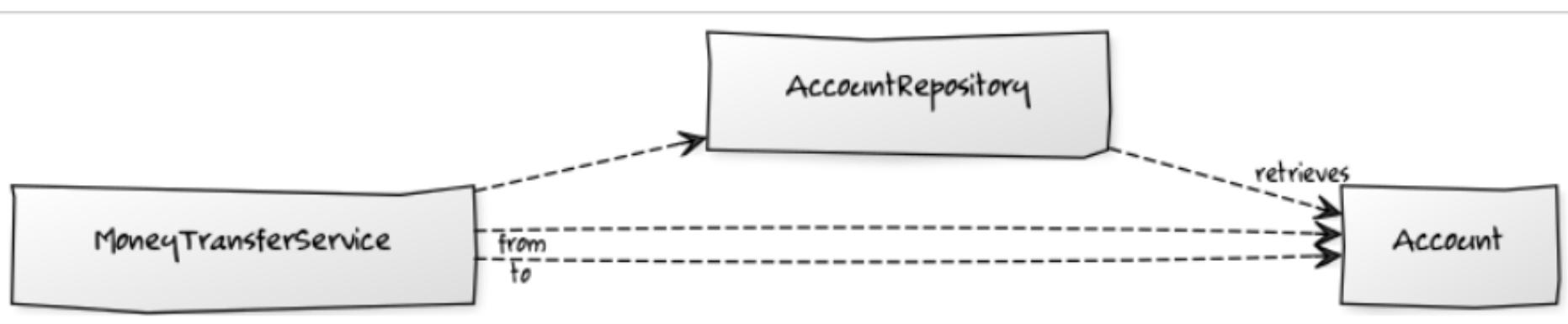
- input from the presentation,
- processes it with validations and calculations,
- stores data in the database,
- (invokes any operations from other systems, and)
- replies with more data to the presentation perhaps doing more calculation to help organize and format the reply data.

[Fowler]

# EXAMPLE

Banking application

Money transfer functionality



```
1 public interface MoneyTransferService {  
2     BankingTransaction transfer(  
3         String fromAccountId, String toAccountId, double amount);  
4 }
```

```
1 public class MoneyTransferServiceTransactionScriptImpl
2     implements MoneyTransferService {
3     private AccountDao accountDao;
4     private BankingTransactionRepository bankingTransactionRepository;
5     . .
6     @Override
7     public BankingTransaction transfer(
8         String fromAccountId, String toAccountId, double amount) {
9         Account fromAccount = accountDao.findById(fromAccountId);
10        Account toAccount = accountDao.findById(toAccountId);
11        . .
12        double newBalance = fromAccount.getBalance() - amount;
13        switch (fromAccount.getOverdraftPolicy()) {
14            case NEVER:
15                if (newBalance < 0) {
16                    throw new DebitException("Insufficient funds");
17                }
18                break;
19            case ALLOWED:
20                if (newBalance < -limit) {
21                    throw new DebitException(
22                        "Overdraft limit (of " + limit + ") exceeded: " + newBalance)
23                }
24                break;
25            }
26        fromAccount.setBalance(newBalance);
27        toAccount.setBalance(toAccount.getBalance() + amount);
28        BankingTransaction moneyTransferTransaction =
29            new MoneyTransferTransaction(fromAccountId, toAccountId, amount);
30        bankingTransactionRepository.addTransaction(moneyTransferTransaction)
31        return moneyTransferTransaction;
32    }
33 }
```

```
1 public enum OverdraftPolicy {  
2     NEVER, ALLOWED  
3 }
```

```
1 // @Entity  
2 public class Account {  
3     // @Id  
4     private String id;  
5     private double balance;  
6     private OverdraftPolicy overdraftPolicy;  
7     . . .  
8     public String getId() { return id; }  
9     public void setId(String id) { this.id = id; }  
10    public double getBalance() { return balance; }  
11    public void setBalance(double balance) { this.balance = balance; }  
12    public OverdraftPolicy getOverdraftPolicy() { return overdraftPolicy; }  
13    public void setOverdraftPolicy(OverdraftPolicy overdraftPolicy) {  
14        this.overdraftPolicy = overdraftPolicy;  
15    }  
16 }
```

# ANALYSIS

## Strengths

- Simplicity

## Weaknesses

- complicated transaction logic
- duplicated logic

# DOMAIN MODEL (EA PATTERN)

An object model of the domain that incorporates both behavior and data. [Fowler]

A DM creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line in an order form.

# DOMAIN MODEL (EA PATTERN)

Realization (via design classes) of UML Domain Model (conceptual classes).

- E.g. person, book, shopping cart, task, sale item, ...

Domain Model classes contain logic for handling validations and calculations.

- E.g. a shipment object calculates the shipping charge for a delivery.

# DM FEATURES

Business logic is organized as an OO model of the domain

- Describes both data and behavior
- Different from database model
  - Process, multi-valued attributes, inheritance, design patterns
  - Harder to map to the database

Risk of bloated domain objects

# MONEY TRANSFER EXAMPLE

```
1 public class MoneyTransferServiceDomainModelImpl
2     implements MoneyTransferService {
3     private AccountRepository accountRepository;
4     private BankingTransactionRepository bankingTransactionRepository;
5     . . .
6     @Override
7     public BankingTransaction transfer(
8         String fromAccountId, String toAccountId, double amount) {
9         Account fromAccount = accountRepository.findById(fromAccountId);
10        Account toAccount = accountRepository.findById(toAccountId);
11        . . .
12        fromAccount.debit(amount);
13        toAccount.credit(amount);
14        BankingTransaction moneyTransferTransaction =
15            new MoneyTransferTransaction(fromAccountId, toAccountId, amount);
16        bankingTransactionRepository.addTransaction(moneyTransferTransaction);
17        return moneyTransferTransaction;
18    }
19 }
```

```
1 // @Entity
2 public class Account {
3     // @Id
4     private String id;
5     private double balance;
6     private OverdraftPolicy overdraftPolicy;
7     ...
8     public double balance() { return balance; }
9     public void debit(double amount) {
10         this.overdraftPolicy.preDebit(this, amount);
11         this.balance = this.balance - amount;
12         this.overdraftPolicy.postDebit(this, amount);
13     }
14     public void credit(double amount) {
15         this.balance = this.balance + amount;
16     }
17 }
```

```
1 public interface OverdraftPolicy {  
2     void preDebit(Account account, double amount);  
3     void postDebit(Account account, double amount);  
4 }
```

```
1 public class NoOverdraftAllowed implements OverdraftPolicy {  
2     public void preDebit(Account account, double amount) {  
3         double newBalance = account.balance() - amount;  
4         if (newBalance < 0) {  
5             throw new DebitException("Insufficient funds");  
6         }  
7     }  
8     public void postDebit(Account account, double amount) {  
9     }  
10 }
```

```
1 public class LimitedOverdraft implements OverdraftPolicy {  
2     private double limit;  
3     . . .  
4     public void preDebit(Account account, double amount) {  
5         double newBalance = account.balance() - amount;  
6         if (newBalance < -limit) {  
7             throw new DebitException(  
8                 "Overdraft limit (of " + limit + ") exceeded: " + newBalance);  
9         }  
10     }  
11     public void postDebit(Account account, double amount) {  
12     }  
13 }
```

# CHOOSING A BUSINESS LOGIC PATTERN

Which one to choose?

- Influenced by the complexity of domain logic.

Application is simple access to data sources

→ Transaction Script, (or Active Record, Table Module)

Significant amount of business logic

→ Domain Model

TS is simpler:

- Easier and quicker to develop and maintain.
- But can lead to duplication in logic / code.

DM – difficult access to relational DB

# DATA ACCESS

- Communication between BLL and Data source:
  - Retrieve data from DB
  - Save data to DB
- Responsibilities:
  - Wrap the access to the DB
    - **Connect** to DB
    - **Execute** queries
    - **Map** data structure to domain model structure
  - **Store** the raw data
    - Set of records
    - Record

# MONOLITHIC APPROACH

**All responsibilities in one class**

- Responsibilities:
  - Wrap the access to the DB
    - Connect to DB
    - Execute queries
  - + **Business Logic**
- Store the raw data
  - Set of records => **Table Module**
  - Record =>**Active Record**

# TABLE MODULE

Provides a single object for all the behavior on a table

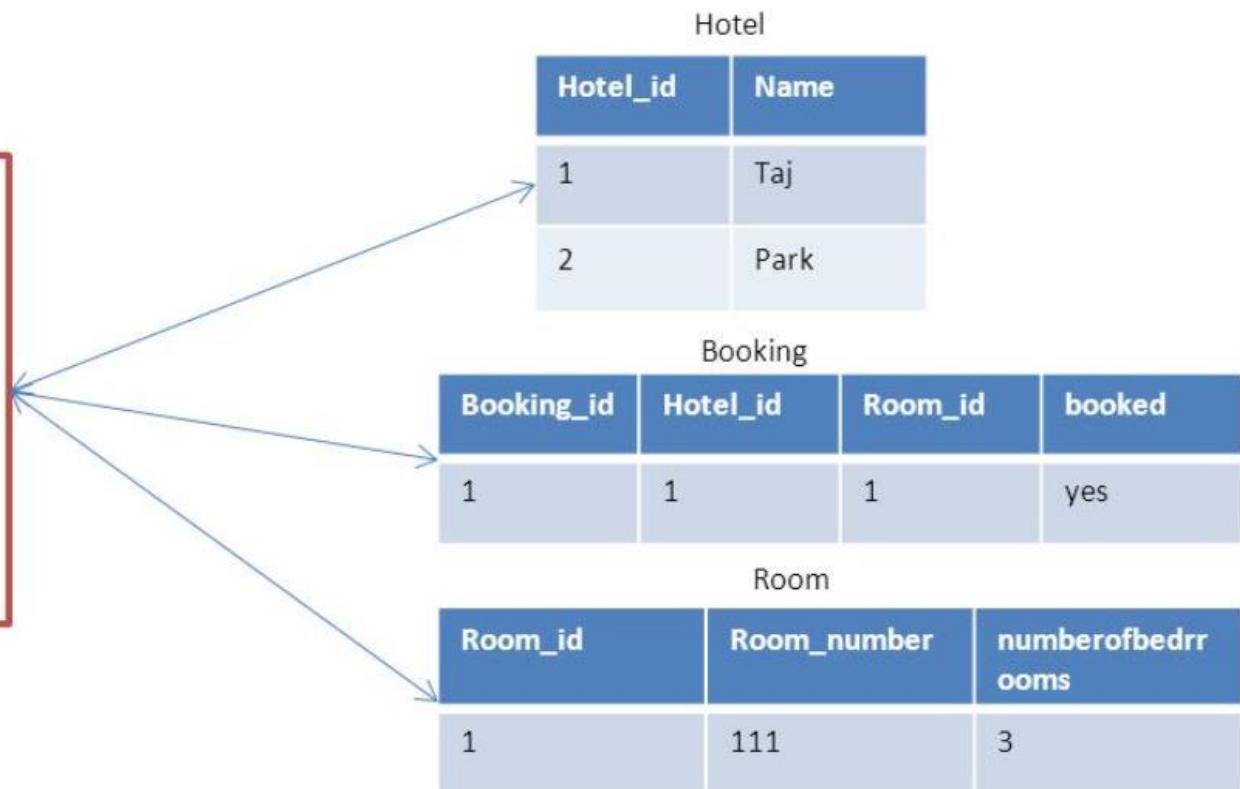
Organizes domain logic with **one class per table**

*Table Module* has no notion of an identity for the objects that it's working with

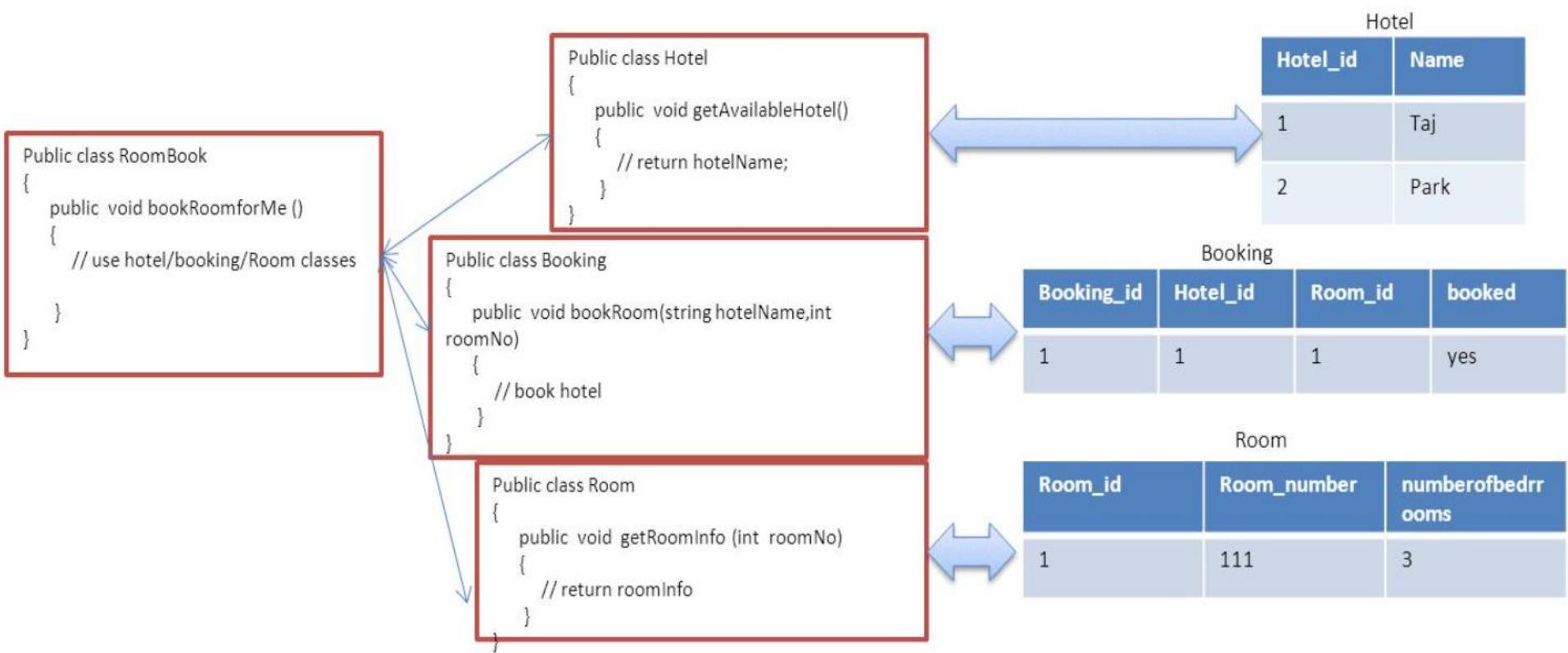
⇒ Id references are necessary

# TRANSACTION SCRIPT VS TABLE MODULE

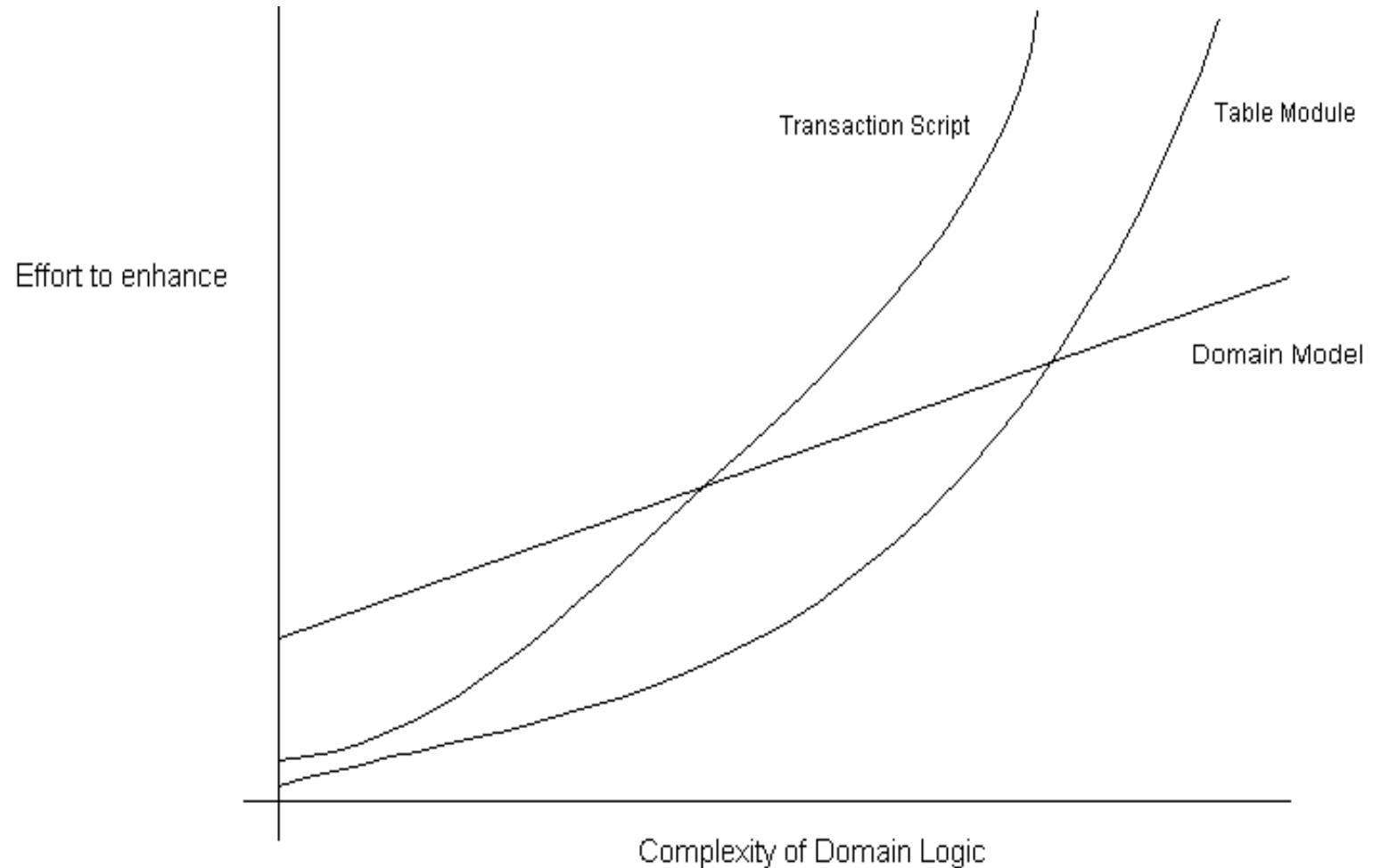
```
Public class Hotel
{
    public void bookRoom(int roomNo)
    {
        // choose hotel
        //check availability
        //calculate price
        // book the room
        // Commit the Transaction
    }
}
```



# TS VS TM



# WHICH ONE TO USE?



# ACTIVE RECORD

Fowler: An object that wraps a record data structure of an external resource, such as a row in a database table, and adds some domain logic to that object.

An AR object carries both data and behavior.

Active Record is a Domain Model in which the classes match very closely the record structure of the underlying database.

# CLASS OPERATIONS

- **construct** an instance of the *Active Record* from a SQL result set row
- **construct** a new instance for later insertion into the table
- static **finder** methods to wrap commonly used SQL queries and return *Active Record* objects
- methods to **update** the database and **insert** into the database with the data in the *Active Record*
- **getting** and **setting** methods for the fields
- methods that implement some pieces of **business logic**

# IMPLEMENTATION

```
class Person{  
    private String lastName;  
    private String firstName;  
    private int numberOfDependents;  
    ...}
```

```
create table people (ID int primary key, lastname  
varchar, firstname varchar, number_of_dependents  
int)
```

# FIND + LOAD AN OBJECT

```
class Person...  
    private final static String findStatementString = "SELECT  
        id, lastname, firstname, number_of_dependents" + " FROM  
        people" + " WHERE id = ?";  
  
    public static Person find(Long id)  
    {  
        Person result = (Person) Registry.getPerson(id);  
        if (result != null) return result;  
        PreparedStatement findStatement = null;  
        ResultSet rs = null;  
        try  
        {  
            findStatement = DB.prepare(findStatementString);  
            findStatement.setLong(1, id.longValue());  
            rs = findStatement.executeQuery();  
            rs.next();  
            result = load(rs);  
            return result;  
        } catch (SQLException e) { throw new  
        ApplicationException(e); }  
        finally { DB.cleanUp(findStatement, rs); }  
    }
```

```
public static Person load(ResultSet rs) throws  
SQLException  
{  
    Long id = new Long(rs.getLong(1));  
    Person result = (Person) Registry.getPerson(id);  
    if (result != null) return result;  
    String lastNameArg = rs.getString(2);  
    String firstNameArg = rs.getString(3);  
    int numDependentsArg = rs.getInt(4);  
    result = new Person(id, lastNameArg, firstNameArg,  
numDependentsArg);  
    Registry.addPerson(result);  
    return result;  
}
```

# SEPARATING RESPONSIBILITIES

Responsibilities:

- Wrap the access to the DB
- Connect to DB
- Execute queries
- Map DB structure to BL structure => **Data Mapper**
- Store the data
  - Set of records => **DTO, RecordSet**
  - Record => **DTO, Row Data Gateway**

**Table Data Gateway, Row Data  
Finder, DAO**

# GATEWAY

- Definition

- An object that encapsulates access to an external system or resource
- Wise to separate SQL (and other forms of data access, query language) from domain logic, and place it in separate classes.
  - Separation of concerns.
- Common technique is to define **a class which maps exactly to a table** in the database => the gateways and the tables are thus *isomorphic*
- Contains **all the database mapping** code for an application, that is all the SQL for CRUD operations
- Contains **no domain logic**

# TABLE DATA GATEWAY

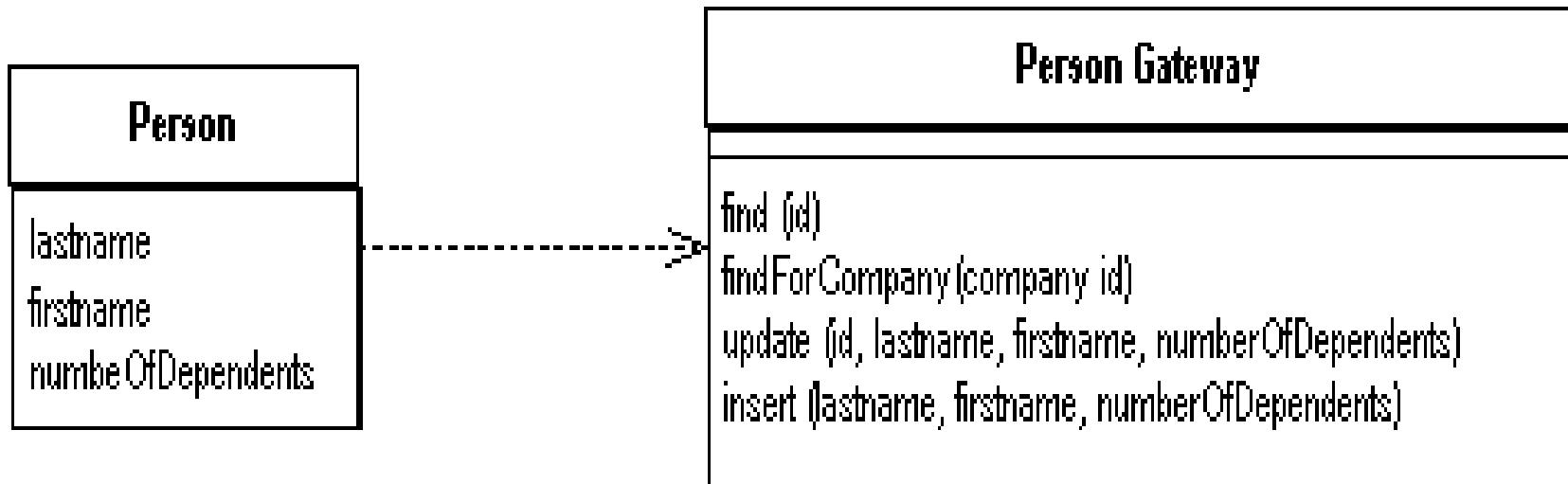
An object that acts as a gateway to a database table. One instance handles all the rows in the table. [Fowler]

Generic data structure of tables and rows that mimics the tabular nature of a DB. Can be used in many parts of application.

- Needs a single instance for **a set of** rows.
- Needs caching strategy, cursors to access database, manage result set.

Common to have GUI tools that work directly with record set

# TDG

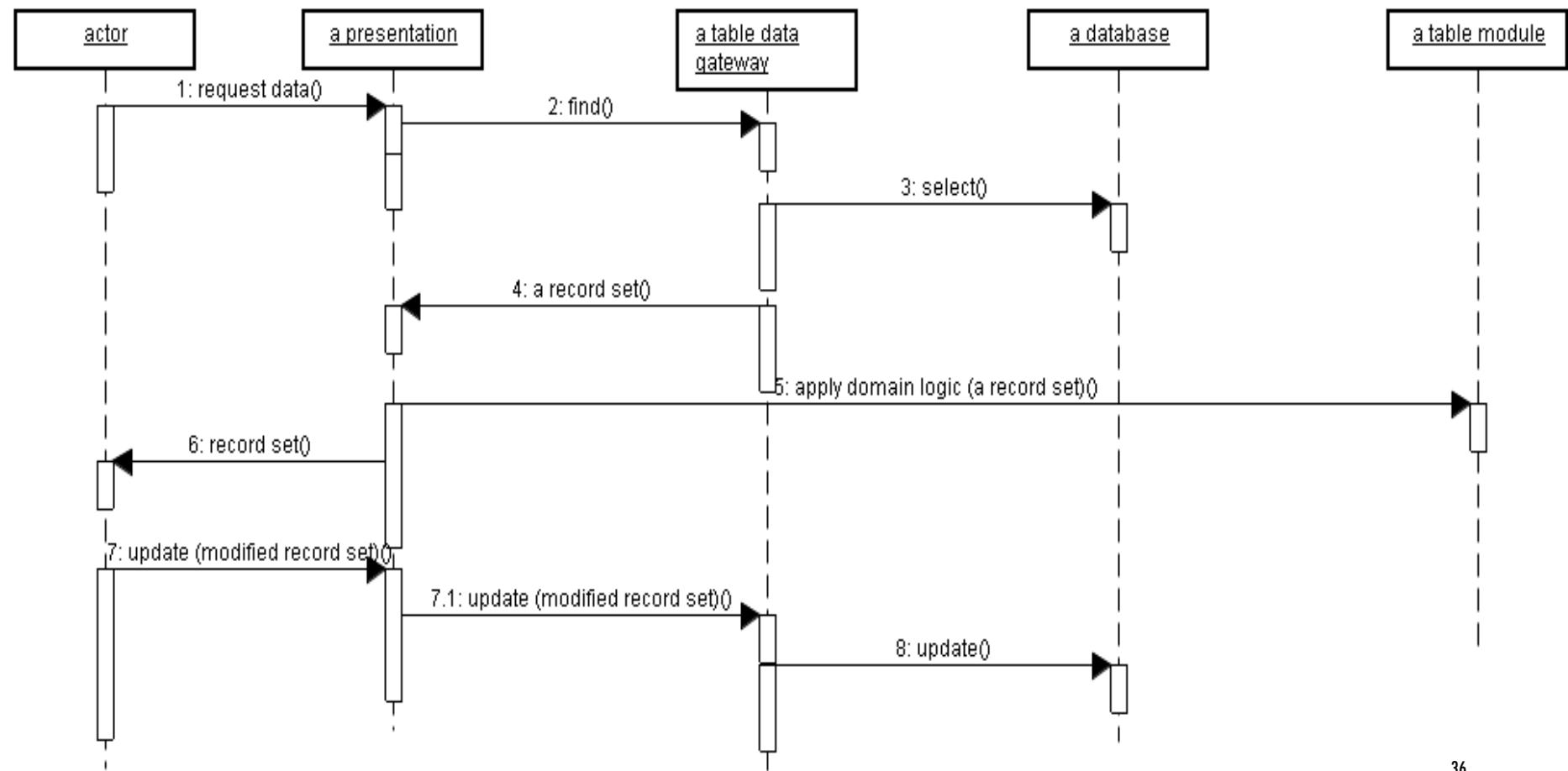


- `find(id) : RecordSet`
- `findWithLastName(String) : RecordSet`
- Iterate through RecordSet
- Update, delete, insert, ...

# FEATURES

- No attributes
- Just CRUD methods
- Challenge: how it returns information from a query ?
  - Data Transfer Object (DTO)
  - RecordSet
- Goes well with Table Module
- Suitable for Transaction Scripts

# SEPARATING THE RESPONSIBILITIES



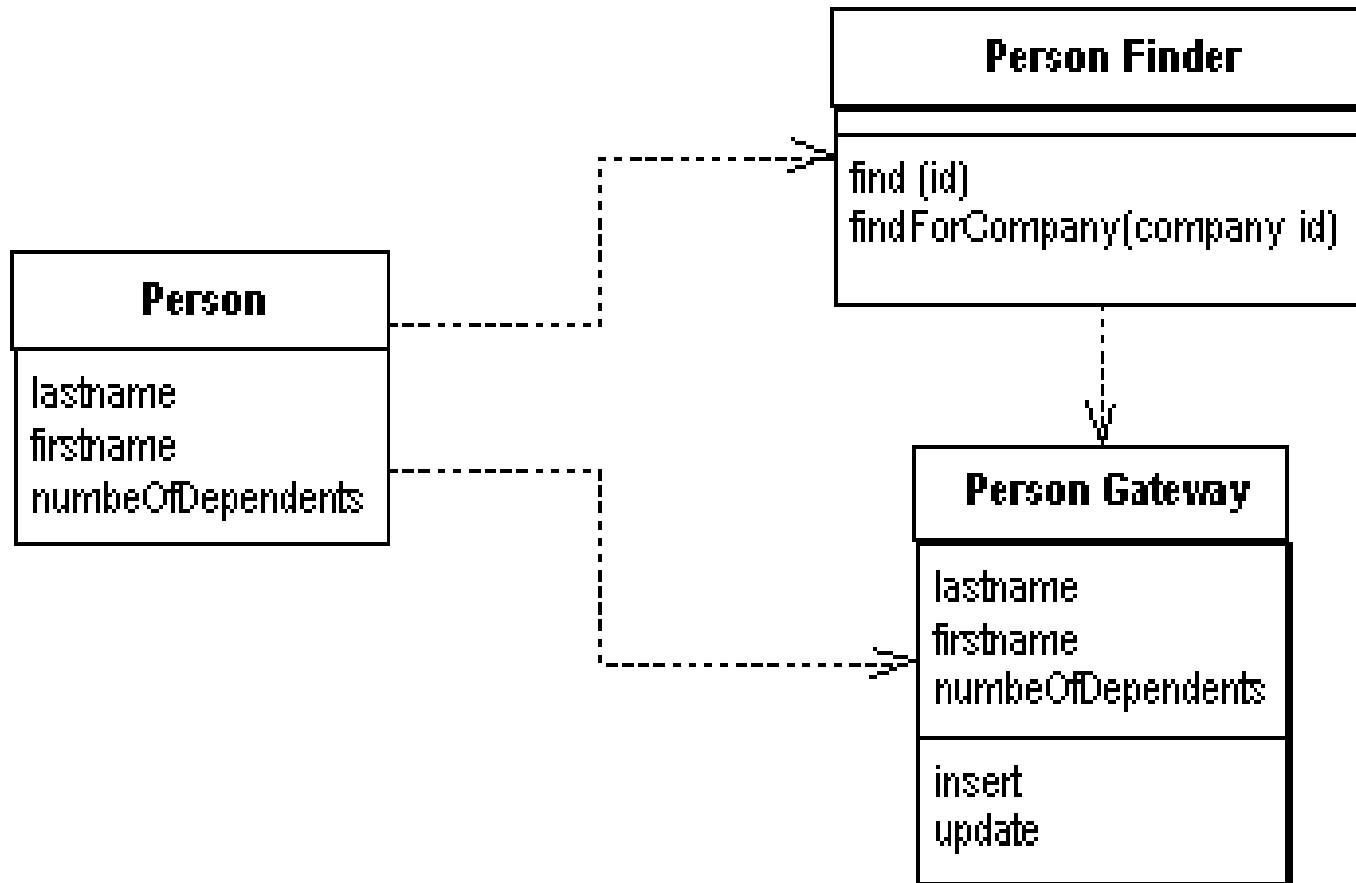
# ROW DATA GATEWAY

- An object that acts as a **single record** in the data source
- Allows in memory storage of object instance without need to access DB.
- Needs an instance of an object for each row.
- Typical approach for *object-relational* mapping tools, e.g., Hibernate.

Fowler RDG combines two responsibilities:

- Class ...Finder with `find(id)` - gateway method which returns the ‘object’ (i.e. SELECT statements)
- Class ...Gateway which stores the ‘object’ data

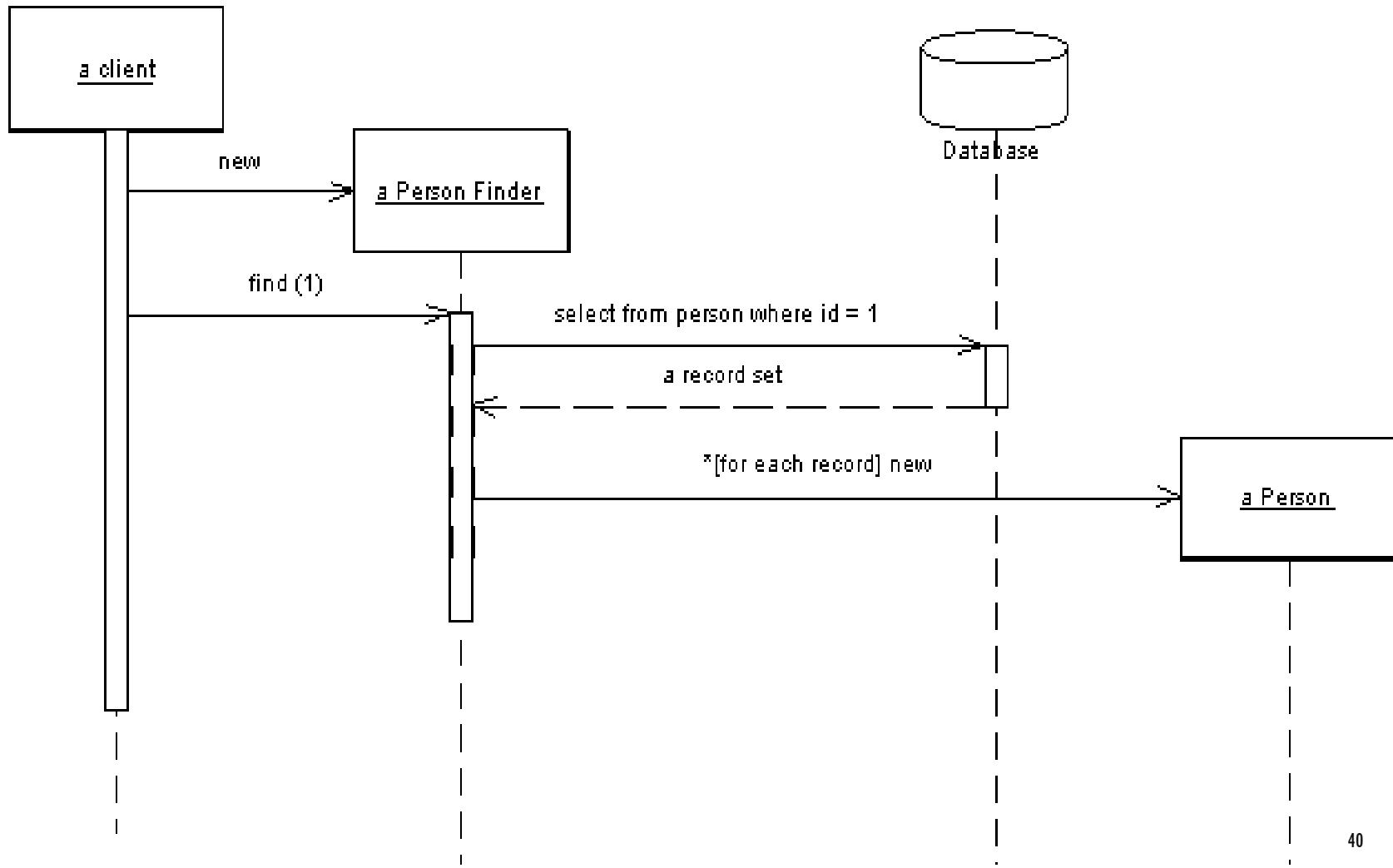
# ROW DATA GATEWAY



# HOW IT WORKS?

- Separate data access code from business logic
- Type conversion from the data source types to the in-memory types
- Works particularly well for Transaction Scripts
- Where to put the find operations that generate the *Row Data* ?
  - separate finder objects
  - each table in a relational database will have:
    - one finder class
    - one gateway class for the results

# RDG BEHAVIOUR



# IMPLEMENTATION

```
class PersonGateway...
{
    private String lastName;
    private String firstName;
    private int
    numberDependents;

    public String getLastName()
    { return lastName; }

    public void setLastName(String
    lastName)
    { this.lastName = lastName; }

    ...
}
```

```
public void insert() {...}
public void update() {...}
public static PersonGateway
load(ResultSet rs) {...}
...
}
```

# PERSONFINDER

```
class PersonFinder...

    private final static String findStatementString = "SELECT id,
lastname, firstname, number_of_dependents " + " from people " +
" WHERE id = ?";

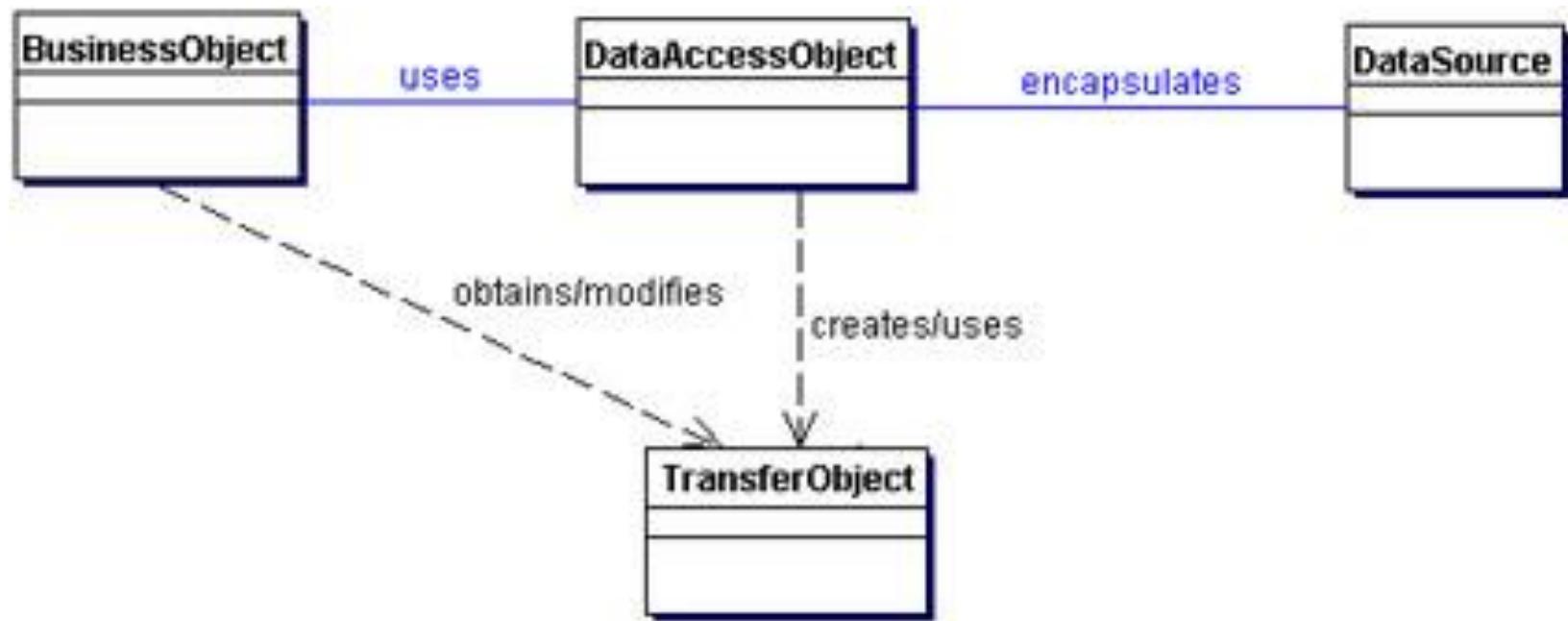
public PersonGateway find(Long id)
{
    PersonGateway result =
PersonGatewayRegistry.getPerson(id);
    if (result != null) return result;
    try
    {
        PreparedStatement findStatement =
DB.prepare(findStatementString);
        findStatement.setLong(1, id.longValue());
        ResultSet rs = findStatement.executeQuery();
        rs.next();
        result = PersonGateway.load(rs);
        return result;
    } catch (SQLException e) { throw new
ApplicationException(e);
}
```

# CLASS PERSONGATEWAY

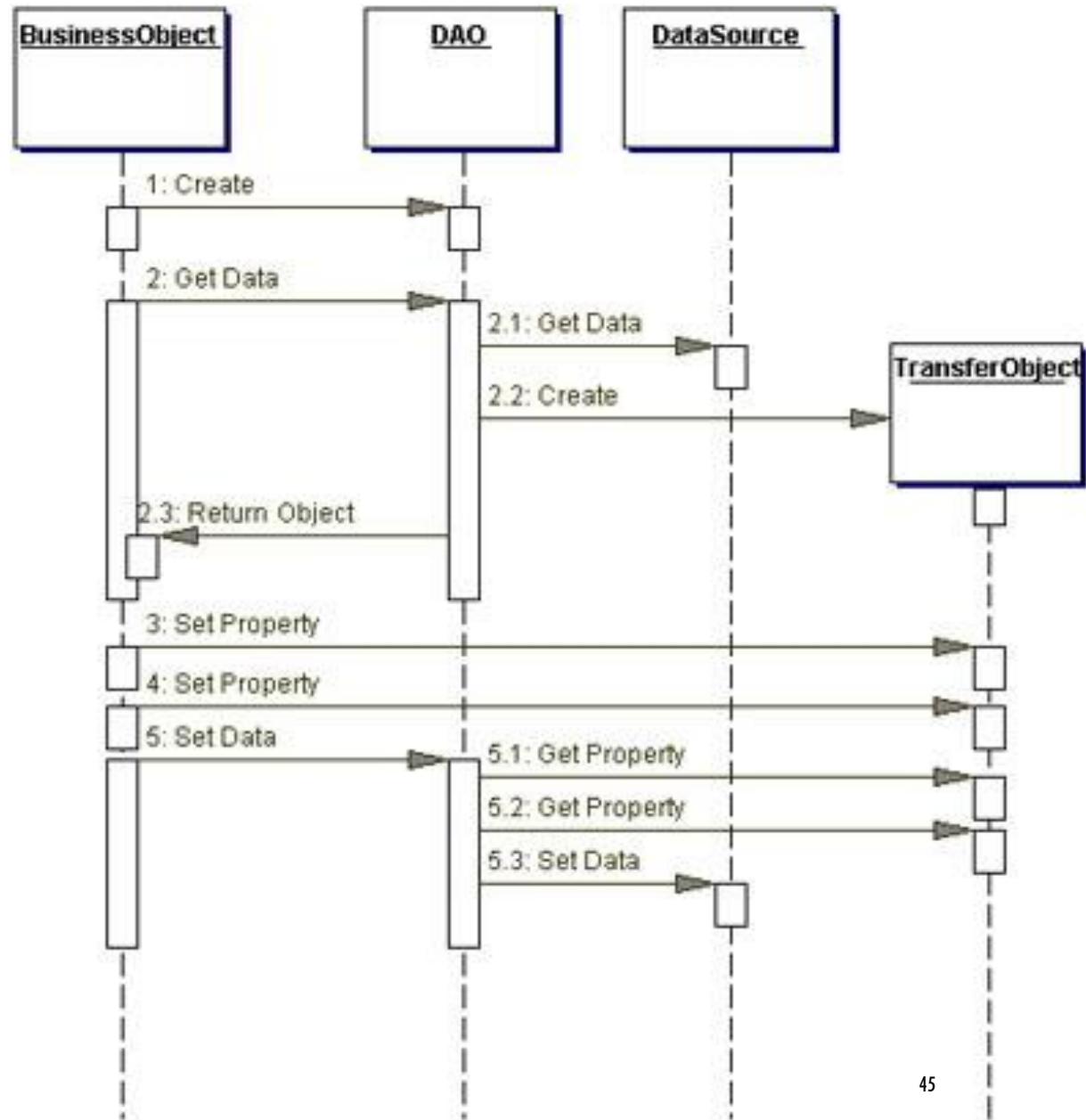
```
private static final String updateStatementString = "UPDATE  
people " + " set lastname = ?, firstname = ?,"  
number_of_dependents = ? " + " where id = ?";  
  
public void update() {  
    PreparedStatement updateStatement = null;  
    try { updateStatement = DB.prepare(updateStatementString);  
        updateStatement.setString(1, lastName);  
        updateStatement.setString(2, firstName);  
        updateStatement.setInt(3, numberOfDependents);  
        updateStatement.setInt(4, getID().intValue());  
        updateStatement.execute(); }  
    catch (Exception e)  
        { throw new ApplicationException(e); }  
    finally  
        {DB.cleanUp(updateStatement); } }  
...
```

# DAO PATTERN

Intent: Abstract and Encapsulate all access to the data source



# HOW IT WORKS

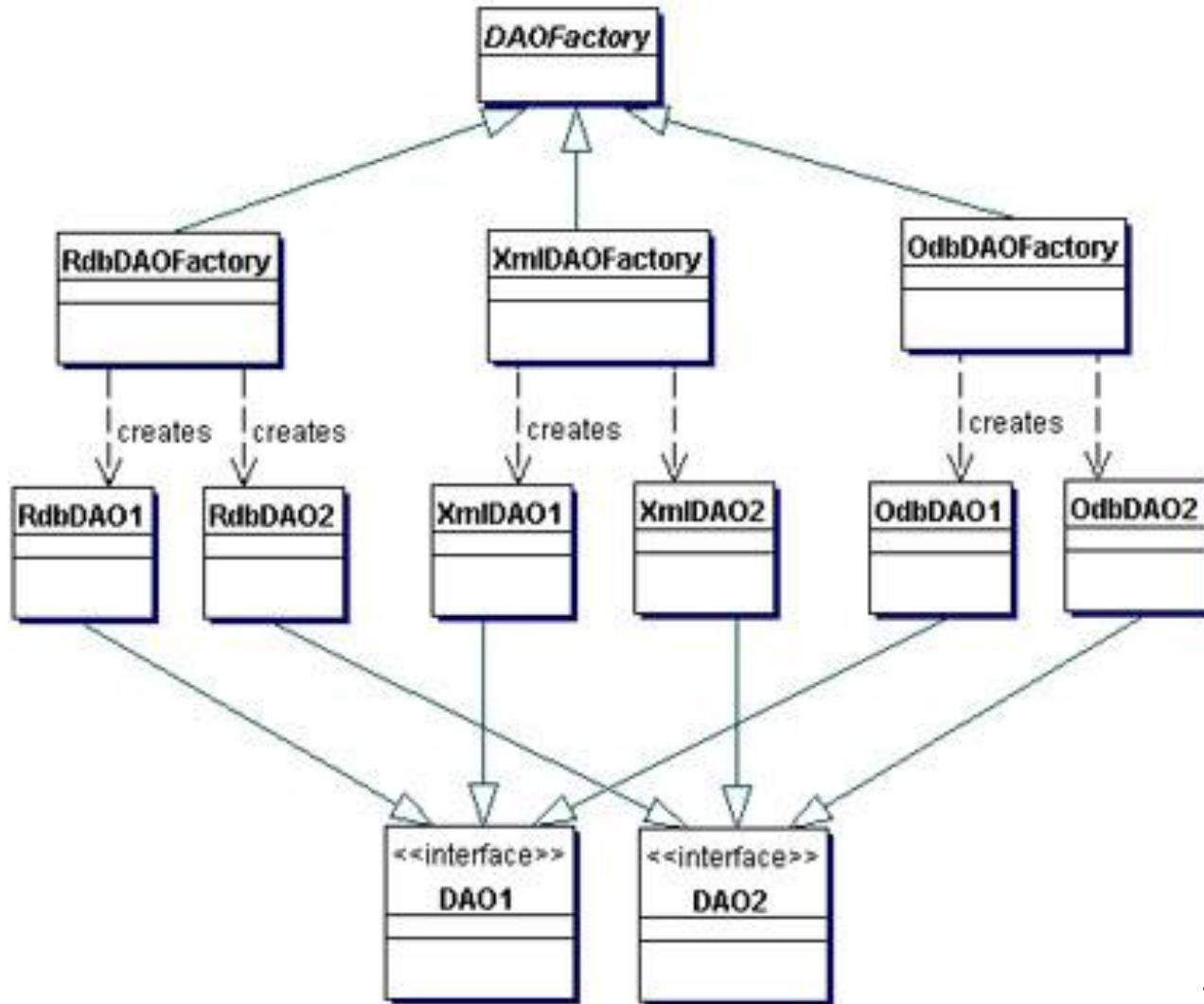


# HOW TO GET THE DAO

Strategies to get DAO's:

- Automatic DAO Code Generation Strategy
  - Metadata Mapping applied
  - JPA Hibernate
- Factory for Data Access Objects Strategy
  - Use **Abstract Factory**: When the underlying storage is subject to change from one implementation to another, this strategy may be implemented using the Abstract Factory pattern.

# ABSTRACT FACTORY



# DATA TRANSFER OBJECT (DTO)

- An object that carries data between processes in order to reduce the number of method calls.
- Usually a set of attributes + setters and getters
- Usually aggregates data from all the server objects that the remote object is likely to want to get data from.
- DTOs are usually structured in tree structures (based on composition) containing primitive types and other DTOs
- Structured around the needs of a particular client (i.e. corresponding to web pages/GUI screens).

# DESIGN DECISIONS

- Keep the DTO structures simple – they have to be serializable
- Use one DTO for a whole interaction vs. different ones for each request?
- Use one DTO for both request and response vs. different ones for each?
- DTO can be a Record Set
- DTOs and domain objects might need mapping

# IMPLEMENTATION DISCUSSION

- Explicit (manual) coding
  - Needs more work and understanding
  - Flexibility related to structure
  - Better performance
- Rely on frameworks
  - Automated
  - Impact on performance
  - Constraints related to structure

# MANUAL IMPLEMENTATION

The class (DTO)

```
public class User {  
    private Integer id;  
    private String name;  
    private String pass;  
    private Integer age;  
    //constructors/getters/setters/}
```

The table

id	int
name	varchar(200)
password	varchar(200)
age	int

# CONNECTING TO THE DATABASE

```
public class ConnectionFactory {  
    public static final String URL =  
"jdbc:mysql://localhost:3306/testdb";  
    public static final String USER = "testuser";  
    public static final String PASS = "testpass";  
  
    /**  
     * Get a connection to database  
     * @return Connection object  
     */  
    public static Connection getConnection()  
{  
        try {  
            DriverManager.registerDriver(new Driver());  
            return DriverManager.getConnection(URL, USER, PASS);  
        } catch (SQLException ex) {  
            throw new RuntimeException("Error connecting to the  
database", ex);  
        }  
    }  
}
```

# DAO IMPLEMENTATION

```
public interface UserDao {  
    User getUser();  
    Set<User> getAllUsers();  
    User getUserByUserNameAndPassword();  
    boolean insertUser();  
    boolean updateUser();  
    boolean deleteUser();  
}  
  
public User getUser(int id) {  
    Connection connection = connectionFactory.getConnection();  
    try {  
        Statement stmt = connection.createStatement();  
        ResultSet rs = stmt.executeQuery("SELECT * FROM user WHERE id=" + id);  
        if(rs.next())  
        {  
            User user = new User();  
            user.setId( rs.getInt("id") );  
            user.setName( rs.getString("name") );  
            user.setPass( rs.getString("pass") );  
            user.setAge( rs.getInt("age") );  
            return user;  
        }  
    } catch (SQLException ex) {  
        ex.printStackTrace();  
    }  
    return null;  
}
```

# A MORE GENERIC DAO

```
public class User {  
    private String name;  
    private String email;  
// constructors / standard setters / getters  
}  
  
public interface Dao<T> {  
    Optional<T> get(long id);  
    List<T> getAll();  
    void save(T t);  
    void update(T t, String[] params);  
    void delete(T t);  
}
```

# IN-MEMORY STORAGE OF USERS

```
public class UserDao implements Dao<User> {
    private List<User> users = new ArrayList<>();
    public UserDao() {
        users.add(new User("John", "john@domain.com"));
        users.add(new User("Susan", "susan@domain.com"));
    }
    @Override
    public Optional<User> get(long id) {
        return Optional.ofNullable(users.get((int) id));
    }
    @Override
    public List<User> getAll() {
        return users;
    }
    @Override
    public void save(User user) {
        users.add(user);
    }
    ...
}
```

# JPA BASED DAO

```
public class JpaUserDao implements Dao<User> {
    private EntityManager entityManager;
    // standard constructors
    @Override
    public Optional<User> get(long id) {
        return Optional.ofNullable(entityManager.find(User.class, id));
    }
    @Override
    public List<User> getAll() {
        Query query = entityManager.createQuery("SELECT e FROM User e");
        return query.getResultList();
    }
    @Override
    public void save(User user) {
        executeInsideTransaction(entityManager ->
            entityManager.persist(user));
    }
}
```

# JPA BASED DAO CONTINUED

```
@Override  
public void update(User user, String[] params) {  
    user.setName(Objects.requireNonNull(params[0], "Name cannot be null"));  
    user.setEmail(Objects.requireNonNull(params[1], "Email cannot be  
null"));  
    executeInsideTransaction(entityManager -> entityManager.merge(user));  
} ...}
```

```
@Entity  
@Table(name = "users")  
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private long id;  
    private String name;  
    private String email;  
    // standard constructors / setters / getters }
```

# BOOTSTRAPPING THE JPA

- Standard approach – `persistence.xml`
- Do your own implementation of the `PersistenceUnitInterface`

```
public class HibernatePersistenceUnitInfo implements PersistenceUnitInfo {

    public static String JPA_VERSION = "2.1";
    private String persistenceUnitName;
    private PersistenceUnitTransactionType transactionType
        = PersistenceUnitTransactionType.RESOURCE_LOCAL;
    private List<String> managedClassNames;
    private List<String> mappingFileNames = new ArrayList<>();
    private Properties properties;
    private DataSource jtaDataSource;
    private DataSource nonjtaDataSource;
    private List<ClassTransformer> transformers = new ArrayList<>();

    public HibernatePersistenceUnitInfo(
        String persistenceUnitName, List<String> managedClassNames, Properties properties) {
        this.persistenceUnitName = persistenceUnitName;
        this.managedClassNames = managedClassNames;
        this.properties = properties;
    }

    // standard setters / getters
}
```

## Create an entity manager factory by wrapping the functionality of *EntityManagerFactoryBuilderImpl*.

```
public class JpaEntityManagerFactory {  
    private String DB_URL = "jdbc:mysql://databaseurl";  
    private String DB_USER_NAME = "username";  
    private String DB_PASSWORD = "password";  
    private Class[] entityClasses;  
  
    public JpaEntityManagerFactory(Class[] entityClasses) {  
        this.entityClasses = entityClasses;  
    }  
  
    public EntityManager getEntityManager() {  
        return getEntityManagerFactory().createEntityManager();  
    }  
  
    protected EntityManagerFactory getEntityManagerFactory() {  
        PersistenceUnitInfo persistenceUnitInfo = getPersistenceUnitInfo(  
            getClass().getSimpleName());  
        Map<String, Object> configuration = new HashMap<>();  
        return new EntityManagerFactoryBuilderImpl(  
            new PersistenceUnitInfoDescriptor(persistenceUnitInfo), configuration)  
            .build();  
    }  
  
    protected HibernatePersistenceUnitInfo getPersistenceUnitInfo(String name) {  
        return new HibernatePersistenceUnitInfo(name, getEntityClassNames(), getProperties());  
    }  
  
    // additional methods  
}
```

```
public class JpaEntityManagerFactory {  
    //...  
  
    protected List<String> getEntityClassNames() {  
        return Arrays.asList(getEntities())  
            .stream()  
            .map(Class::getName)  
            .collect(Collectors.toList());  
    }  
  
    protected Properties getProperties() {  
        Properties properties = new Properties();  
        properties.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");  
        properties.put("hibernate.id.new_generator_mappings", false);  
        properties.put("hibernate.connection.datasource", getMysqlDataSource());  
        return properties;  
    }  
  
    protected Class[] getEntities() {  
        return entityClasses;  
    }  
  
    protected DataSource getMysqlDataSource() {  
        MysqlDataSource mysqlDataSource = new MysqlDataSource();  
        mysqlDataSource.setURL(DB_URL);  
        mysqlDataSource.setUser(DB_USER_NAME);  
        mysqlDataSource.setPassword(DB_PASSWORD);  
        return mysqlDataSource;  
    }  
}
```

## Use the EntityManager to perform CRUD operations

```
public static void main(String[] args) {
    EntityManager entityManager = getJpaEntityManager();
    User user = entityManager.find(User.class, 1);

    entityManager.getTransaction().begin();
    user.setName("John");
    user.setEmail("john@domain.com");
    entityManager.merge(user);
    entityManager.getTransaction().commit();

    entityManager.getTransaction().begin();
    entityManager.persist(new User("Monica", "monica@domain.com"));
    entityManager.getTransaction().commit();

    // additional CRUD operations
}

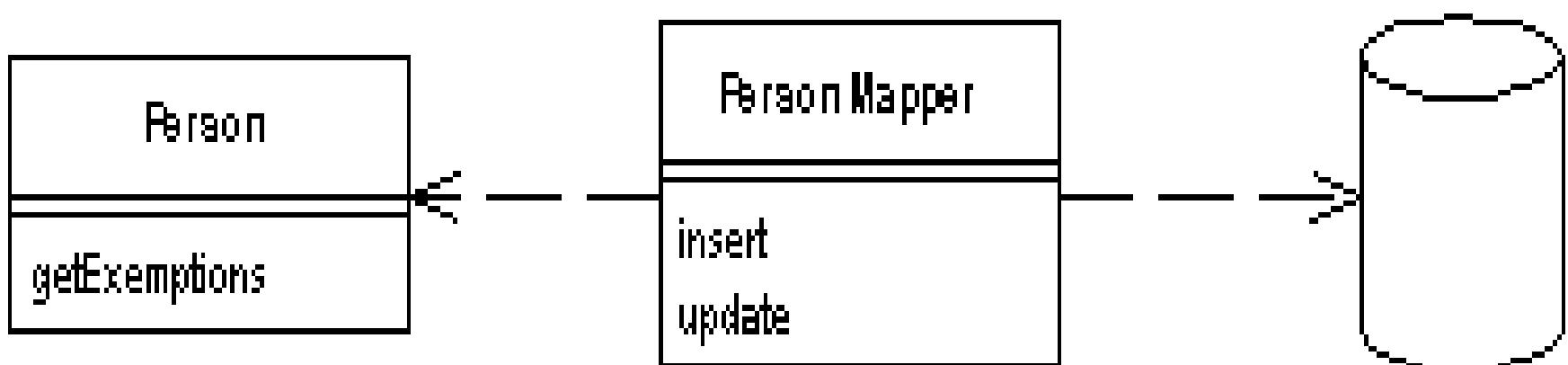
private static class EntityManagerHolder {
    private static final EntityManager ENTITY_MANAGER = new JpaEntityManagerFactory(
        new Class[]{User.class})
        .getEntityManager();
}

public static EntityManager getJpaEntityManager() {
    return EntityManagerHolder.ENTITY_MANAGER;
}
```

# DATA MAPPERS

Acts as an intermediary between Domain Models and the database.

Allows Domain Models and Data Source classes to be independent of each other



# DATA MAPPER LAYER ...

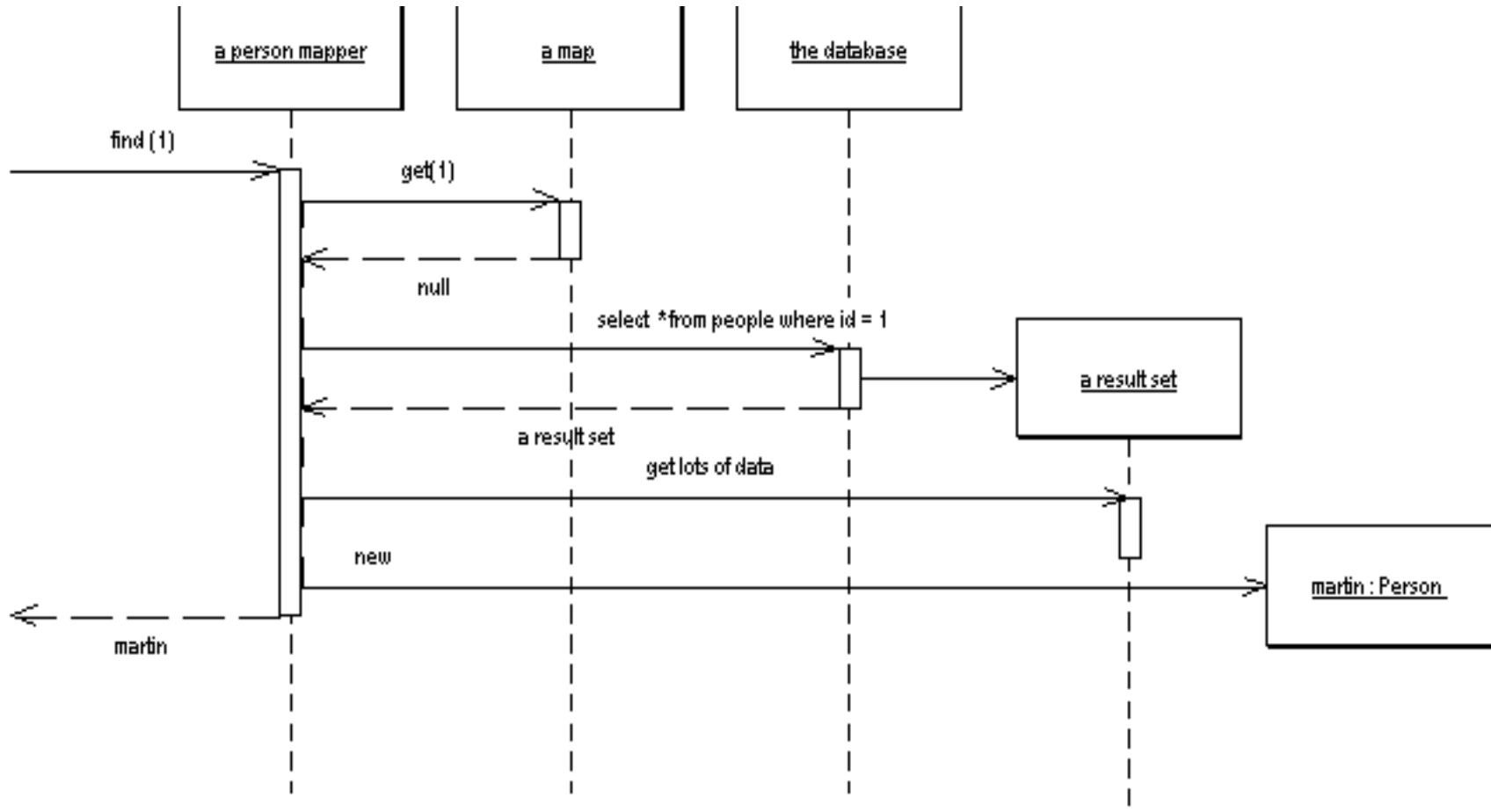
Can either

- Access the database itself, or
- Make use of a Table Data Gateway/DAO.

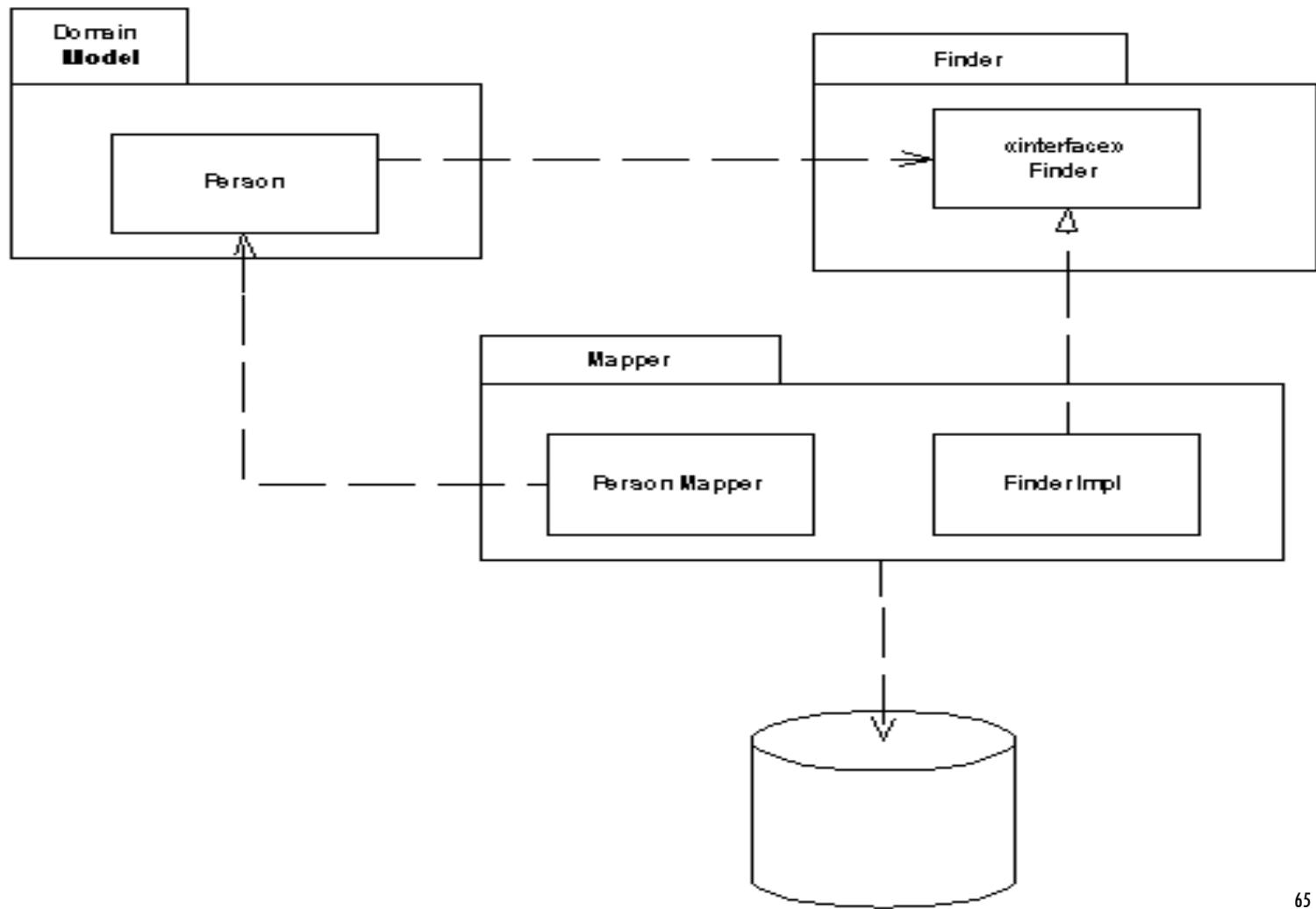
Does not contain Domain Logic.

When it uses a TDG/DAO, the Data Mapper can be placed in the (lower) Domain layer.

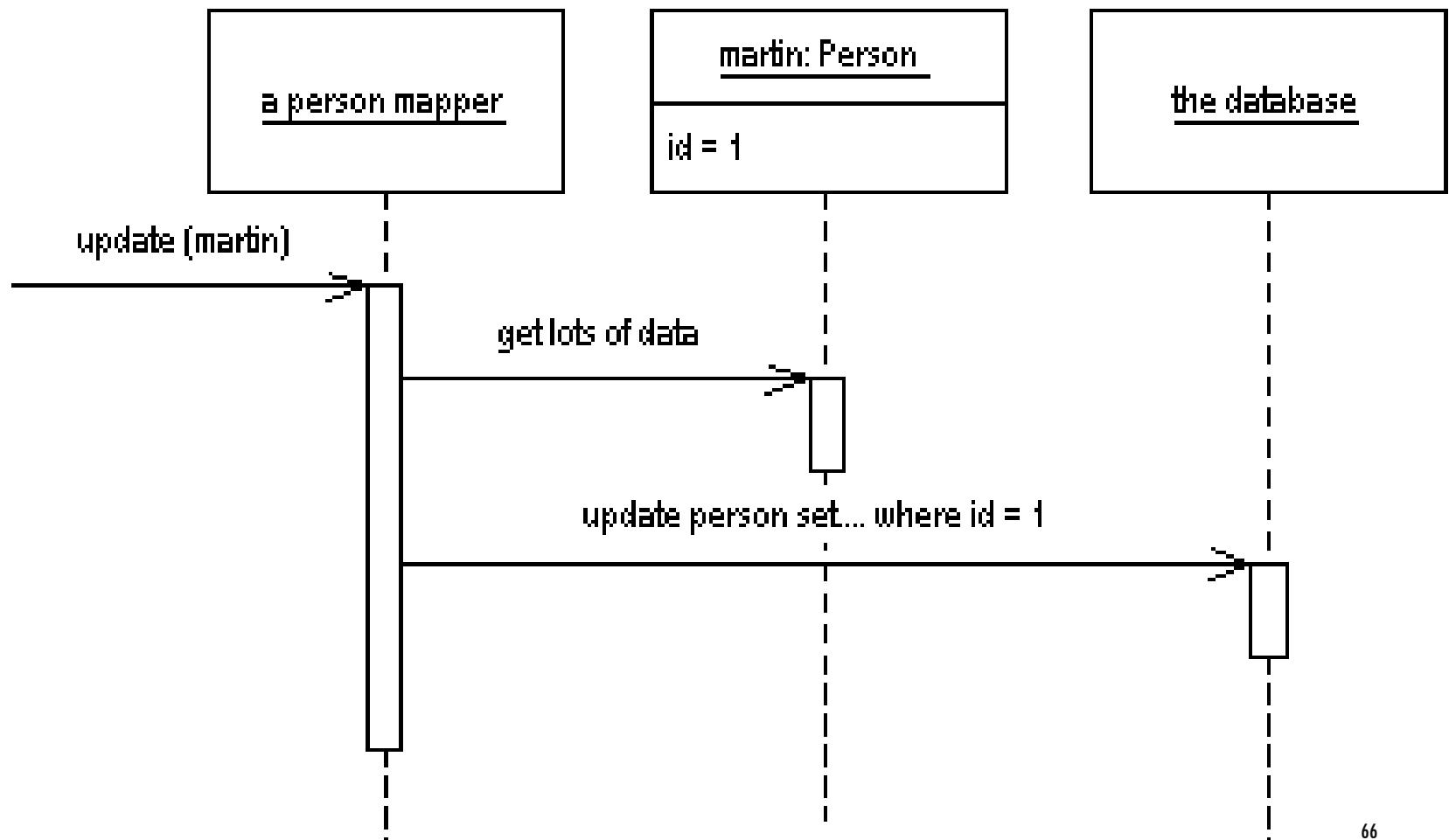
# RETRIEVING DATA



# FINDING OBJECTS



# UPDATING DATA



# FEATURES

- Independent database schema and object model
- Extra layer
- Makes sense with Domain Model

# IMPLEMENTATION

```
class Person {  
    private String name;  
    private int numberOfDependents;  
... }  
  
create table people (ID int primary key, lastname  
    varchar, firstname varchar, number_of_dependents int)
```

# ABSTRACTMAPPER

```
class AbstractMapper...
    protected Map loadedMap = new HashMap();
    abstract protected String findStatement();
    protected DomainObject abstractFind(Long id)
    {
        DomainObject result = (DomainObject) loadedMap.get(id);
        if (result != null) return result;
        PreparedStatement findStatement = null;
        try
        {
            findStatement = DB.prepare(findStatement());
            findStatement.setLong(1, id.longValue());
            ResultSet rs = findStatement.executeQuery();
            rs.next();
            result = load(rs);
            return result;
        } catch (SQLException e)
        {
            throw new ApplicationException(e);
        }
        finally { DB.cleanUp(findStatement); }
    }
```

# LOAD METHOD IN ABSTRACTMAPPER

```
class AbstractMapper...  
protected DomainObject load(ResultSet rs) throws  
SQLException  
{  
    Long id = new Long(rs.getLong(1));  
    if (loadedMap.containsKey(id))  
        return (DomainObject) loadedMap.get(id);  
    DomainObject result = doLoad(id, rs);  
    loadedMap.put(id, result);  
    return result;  
}  
abstract protected DomainObject doLoad(Long id,  
ResultSet rs) throws SQLException;
```

# MAPPER CLASS IMPLEMENTS FINDER

```
class PersonMapper...  
protected String findStatement()  
{  
    return "SELECT " + COLUMNS + " FROM people" + "  
WHERE id = ?";  
}  
public static final String COLUMNS = " id, lastname,  
firstname, number_of_dependents ";  
  
public Person find(Long id)  
{  
    return (Person) abstractFind(id);  
}
```

# DOLOAD IN PERSONMAPPER

```
class PersonMapper...  
protected DomainObject doLoad(Long id,  
ResultSet rs) throws SQLException  
{  
    String name = rs.getString(2) + " " +  
        rs.getString(3);  
    int numDependentsArg = rs.getInt(4);  
    return new Person(id, name,  
numDependentsArg);  
}
```

# STRUCTURAL PROBLEMS

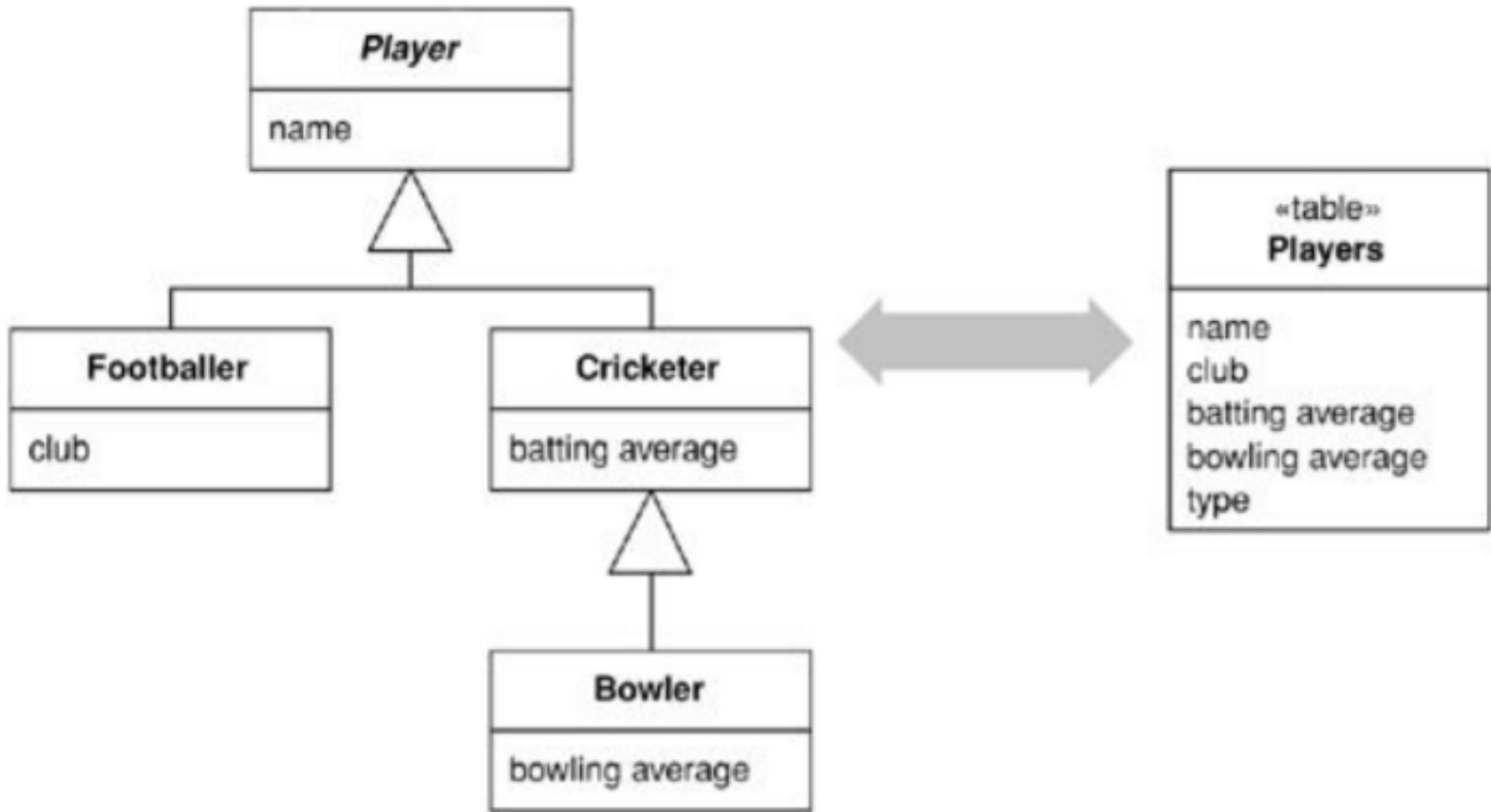
Object Relationships structural mapping problems

- Association/Composition
- **Inheritance**

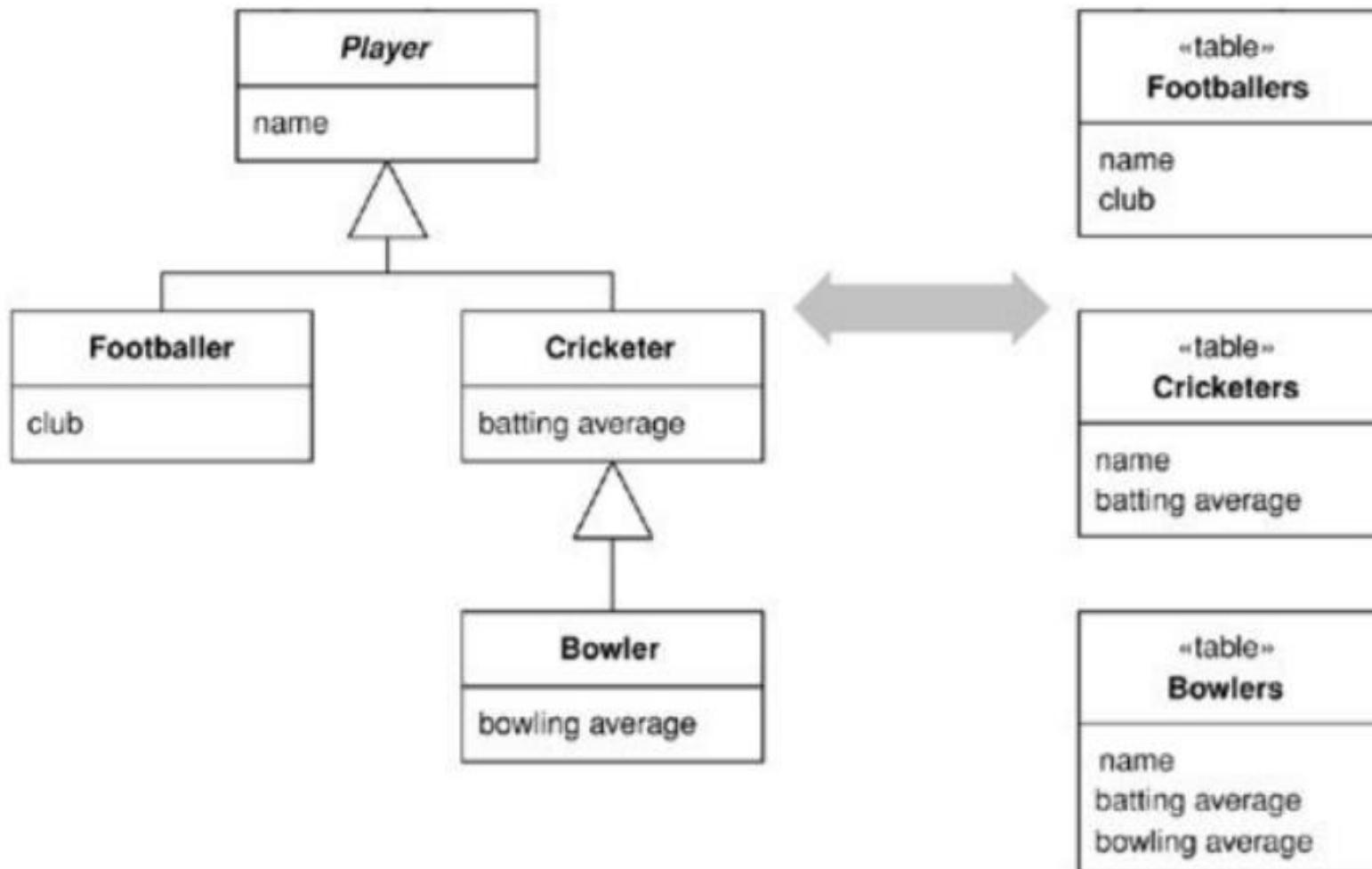
**Object-Relational Structural Patterns**

- **Single Table Inheritance**
- **Class Table Inheritance**
- **Concrete Table Inheritance**

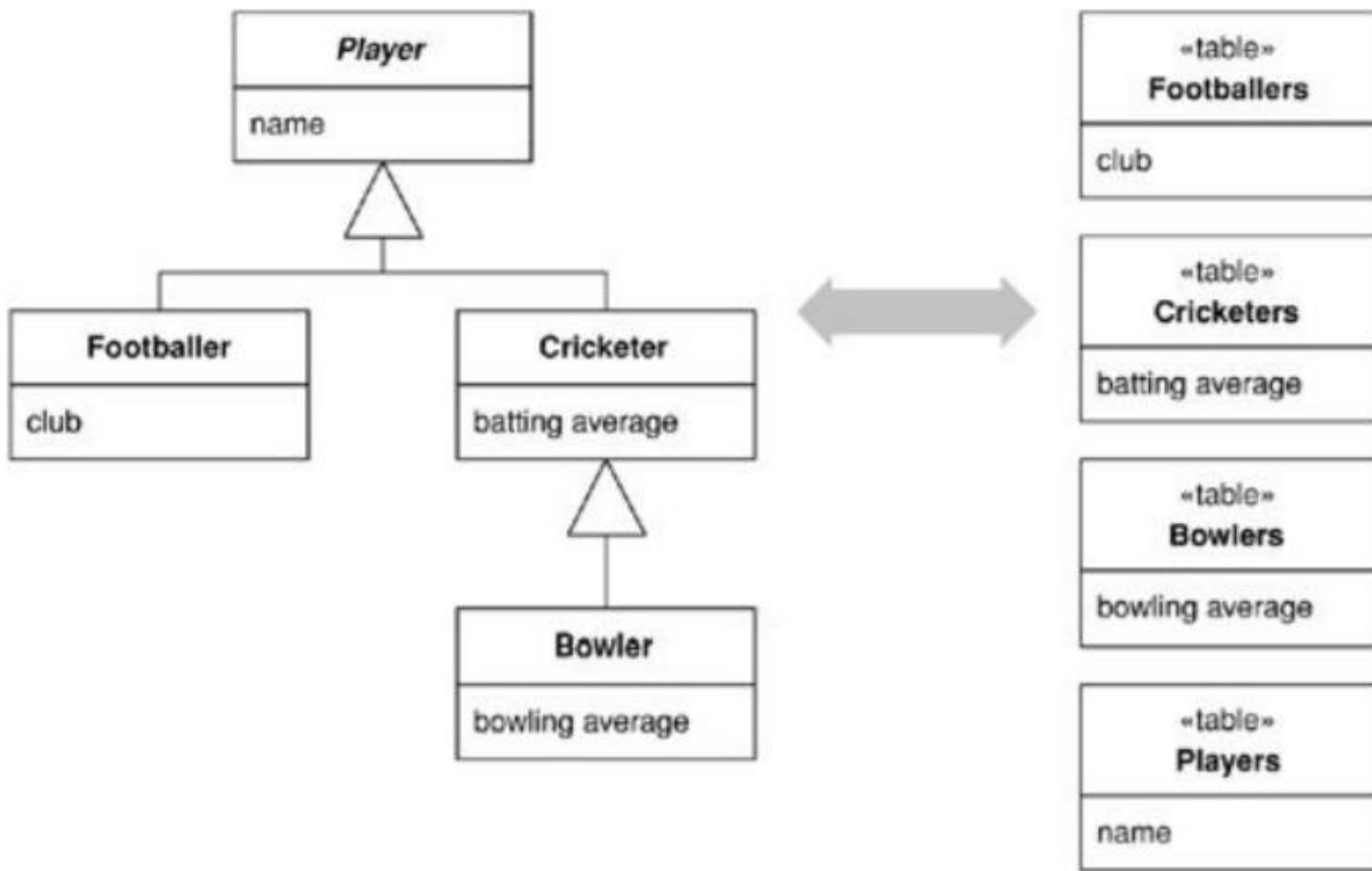
# SINGLE TABLE INHERITANCE



# CONCRETE TABLE INHERITANCE



# CLASS TABLE INHERITANCE



# BEHAVIORAL PROBLEMS

Networks of objects

- E.g. Invoice heading relates to invoice details
- Invoice details refers to Products
- Products refers to Suppliers
- ...

What to do?

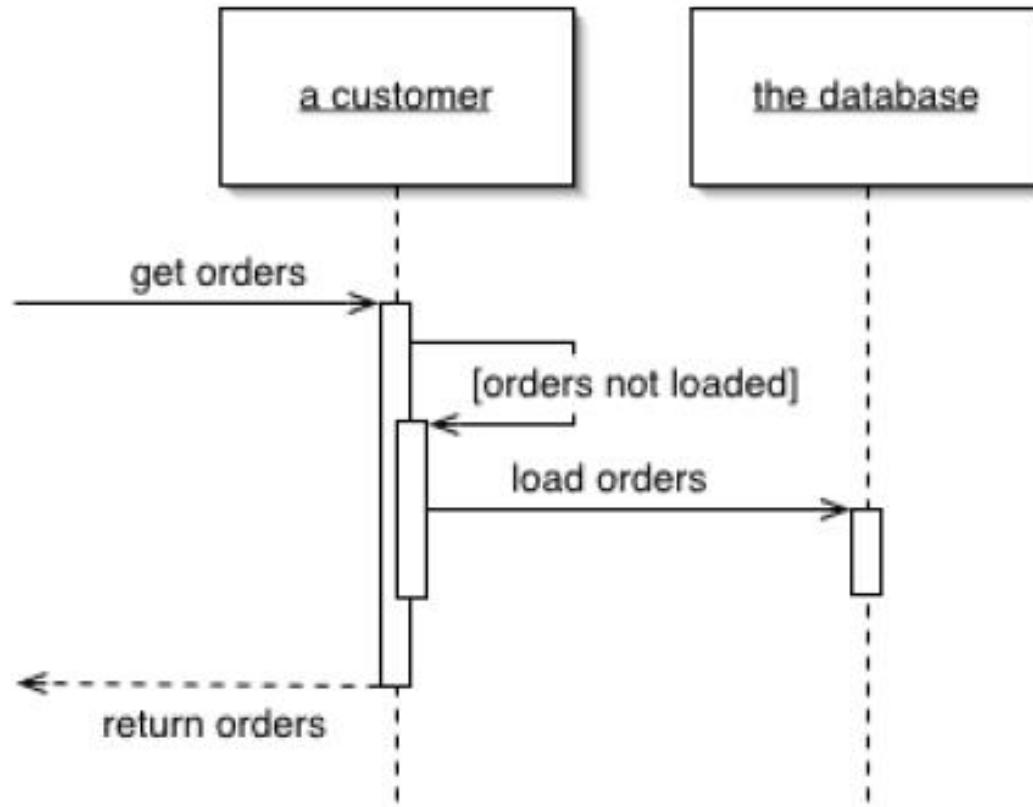
- Load them all into memory?
- How to disallow multiple in-memory copies

## Object-Relational Behavioral Patterns

- **Lazy Load**
- **Identity Map**

# LAZY LOAD

An object that doesn't contain all of the data you need but knows how to get it.



# IMPLEMENTATION OPTIONS

**Lazy initialization:** every access to the field checks first to see if it's null. If so, it calculates the value of the field before returning the field.

```
class Supplier...

    public List getProducts() {
        if (products == null) products = Product.findForSupplier(getID());
        return products;
    }
```

**Virtual proxy:** looks like the object but doesn't contain anything. Only when one of its methods is called does it load the object from the database.

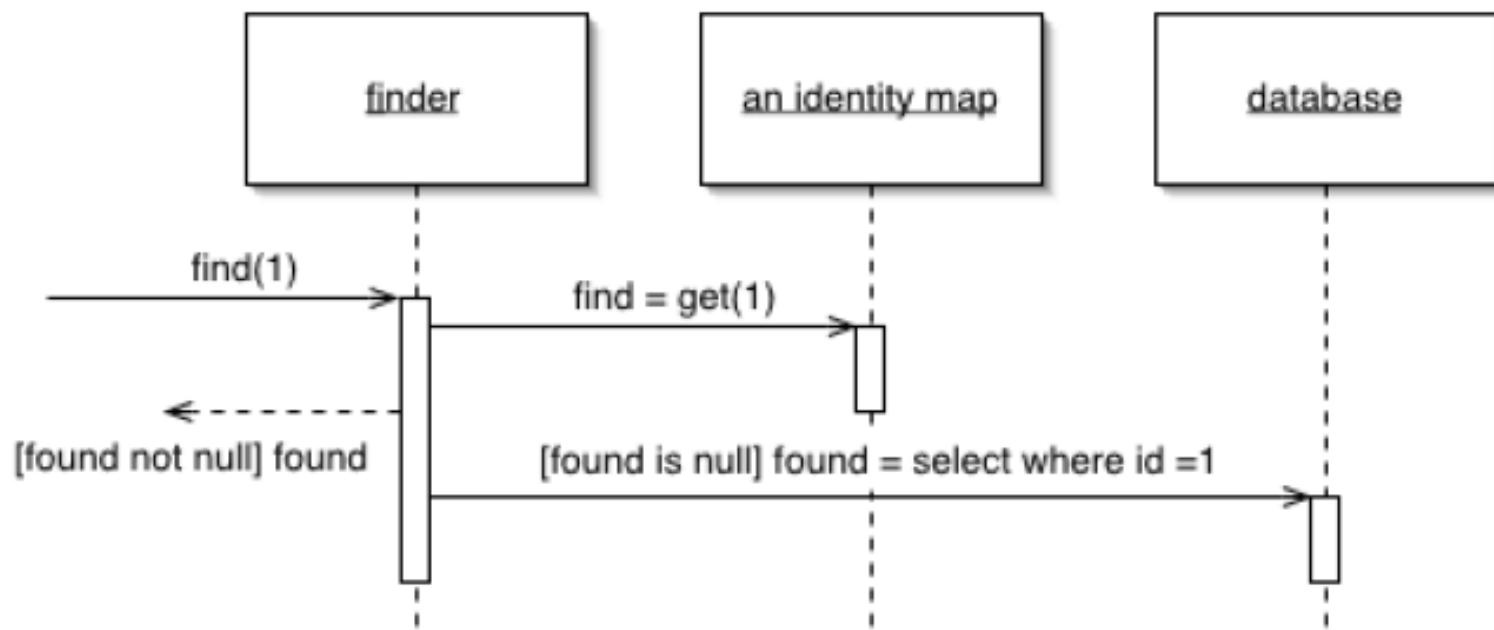
**Ghost:** the real object in a partial state. Ex. when you load the object from the database it contains just its ID. Whenever you try to access a field it loads its full state.

# DISCUSSION

- Inheritance might be a problem with Lazy Load
- Can easily cause more database accesses than you need (ex. fill a collection with Lazy Loads)
- Deciding when to use Lazy Load is all about deciding how much you want to pull back from the database as you load an object, and how many database calls that will require.

# IDENTITY MAP

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them



# HOW IT WORKS

## Map key?

- Primary key in the table (if it is a single column and immutable)

## Explicit vs. generic

- `findPerson(1)`
- `find ("Person", 1)`

## How many?

- One map/session (if you have database-unique keys)
- One map/table
- One map/class
- One map/inheritance tree

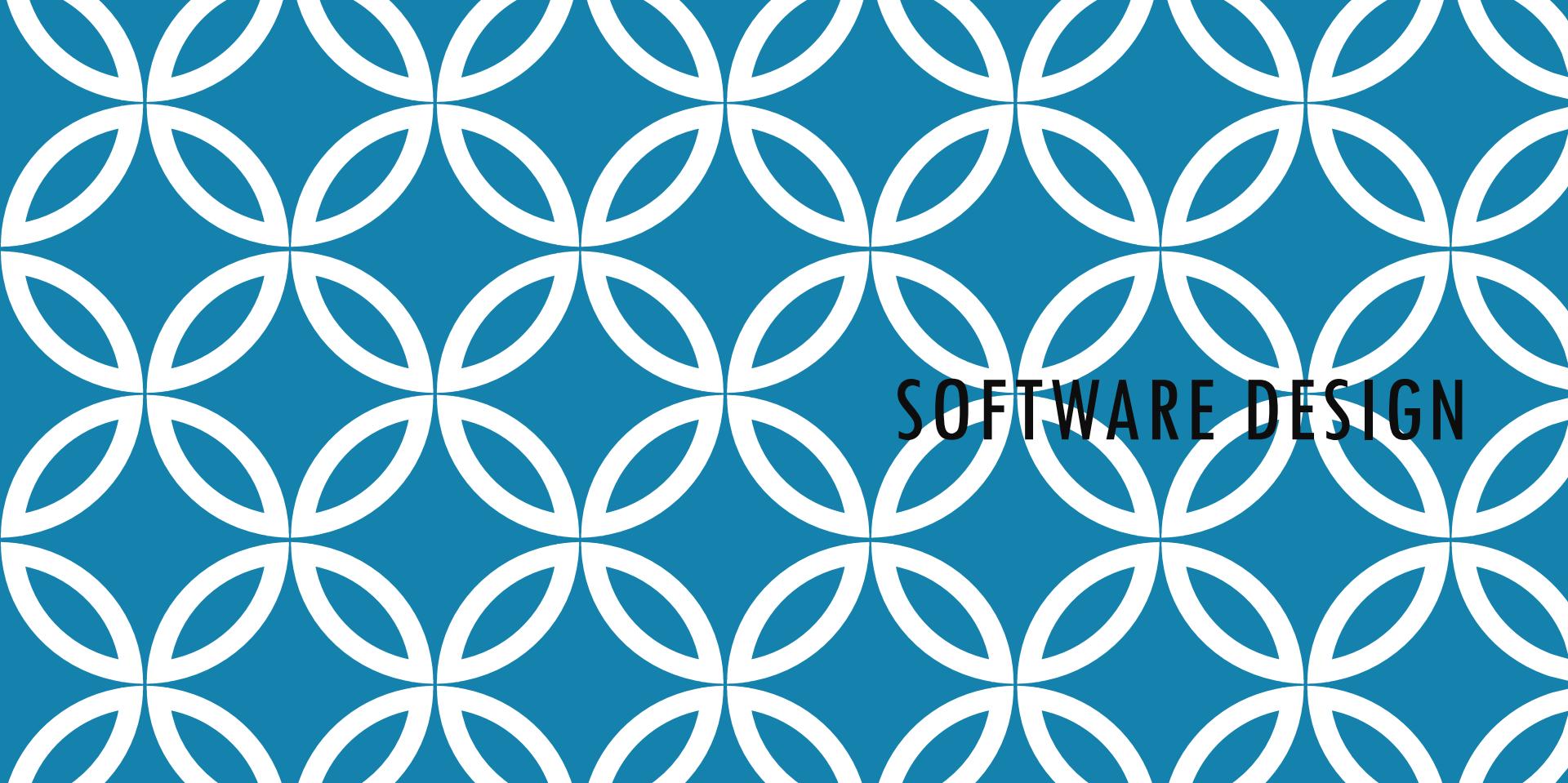
# WHEN TO USE IT

- Use an Identity Map to manage any Entity object brought from a database and modified.
- Acts as a cache for database reads
- May not need an Identity Map for immutable objects
- Helps avoid update conflicts within a single session, but it doesn't do anything to handle conflicts that cross sessions(see the Concurrency topic)

# NEXT TIME

More patterns

- Concurrency
- Presentation



# SOFTWARE DESIGN

Concurrency and  
Presentation

# CONTENT

- Handling concurrency
- Presentation layer

# REFERENCES

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- SaaS Course Stanford
- Ólafur Andri Ragnarsson, Presentation Layer Design, 2014.

# OFFLINE CONCURRENCY PATTERNS

- Multiple threads that manipulate the same data
- A solution -> Transaction managers....as long as all data manipulation is within a transaction.
- What if data manipulation spans transactions?

# BUSINESS TRANSACTIONS

- ACID
- Transactional resource (ex. Database)
- Increase throughput -> short transactions
- Transactions mapped on a single request
- Late transactions -> read data first, start transaction for updates
- Transactions spanning several requests -> long transactions
- Lock escalation (row level -> table level)

# CONCURRENCY PROBLEMS

- Lost updates
- Inconsistent read => Correctness failure
- Liveness – how much concurrency can the system handle?

# EXECUTION CONTEXTS

“A **request** corresponds to a single call from the outside world which the software works on and optionally sends back a response”

“A **session** is a long running interaction between a client and server.”

“A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on.”

“A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process.”

# APPLICATION SERVER CONCURRENCY

## **process-per-session**

- Uses a lot of resources

## **process-per-request**

- Pooled processes
- Sequential requests
- Resources for a request should be released

## **thread-per-request**

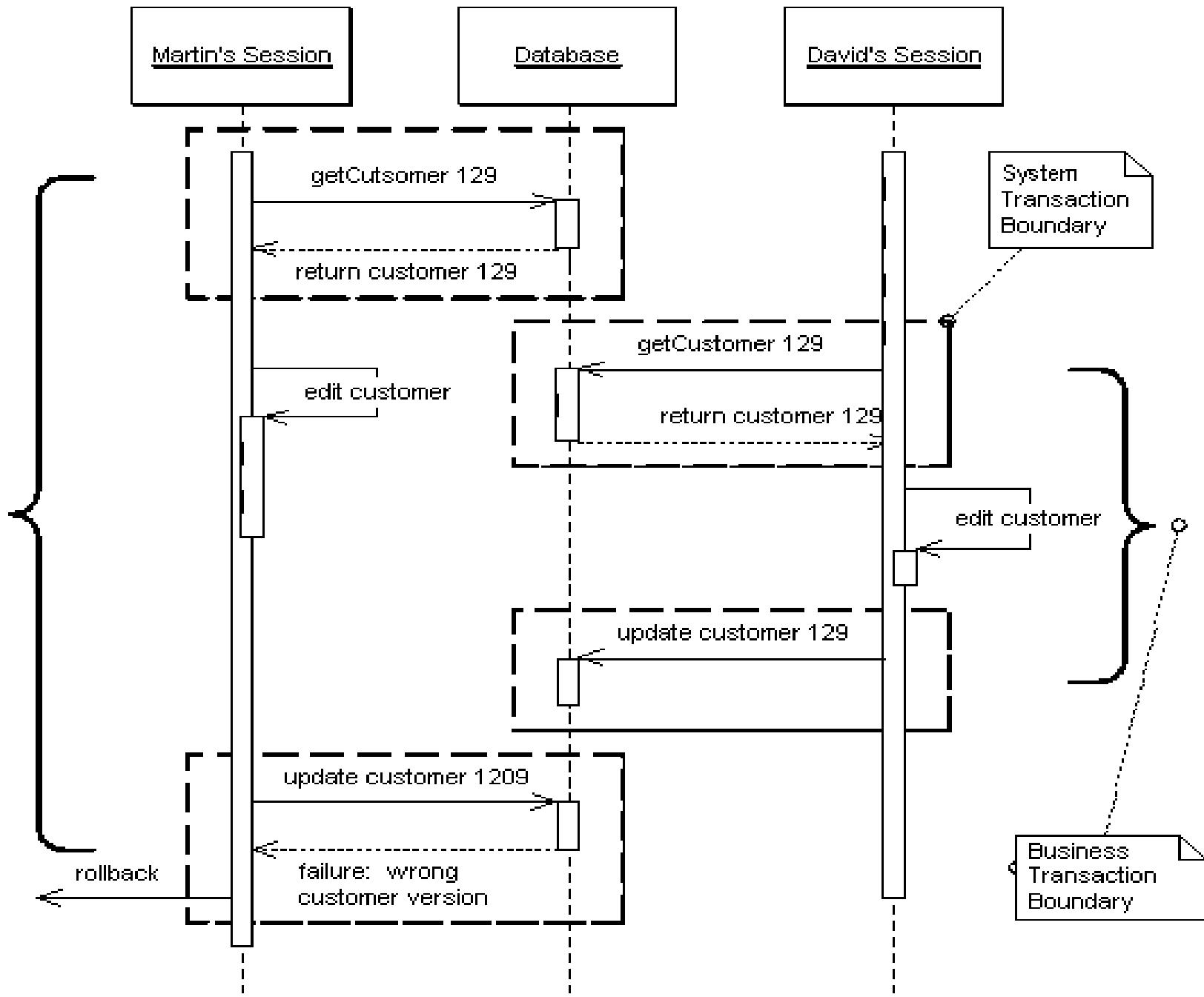
- More efficient
- No isolation

# SOLUTIONS

**isolation:** partition the data so that any piece of data can only be accessed by one active agent.

**immutable data:** separate the data that cannot be modified.

**mutable data than cannot be isolated => Concurrency Control**



# OPTIMISTIC CONCURRENCY CONTROL

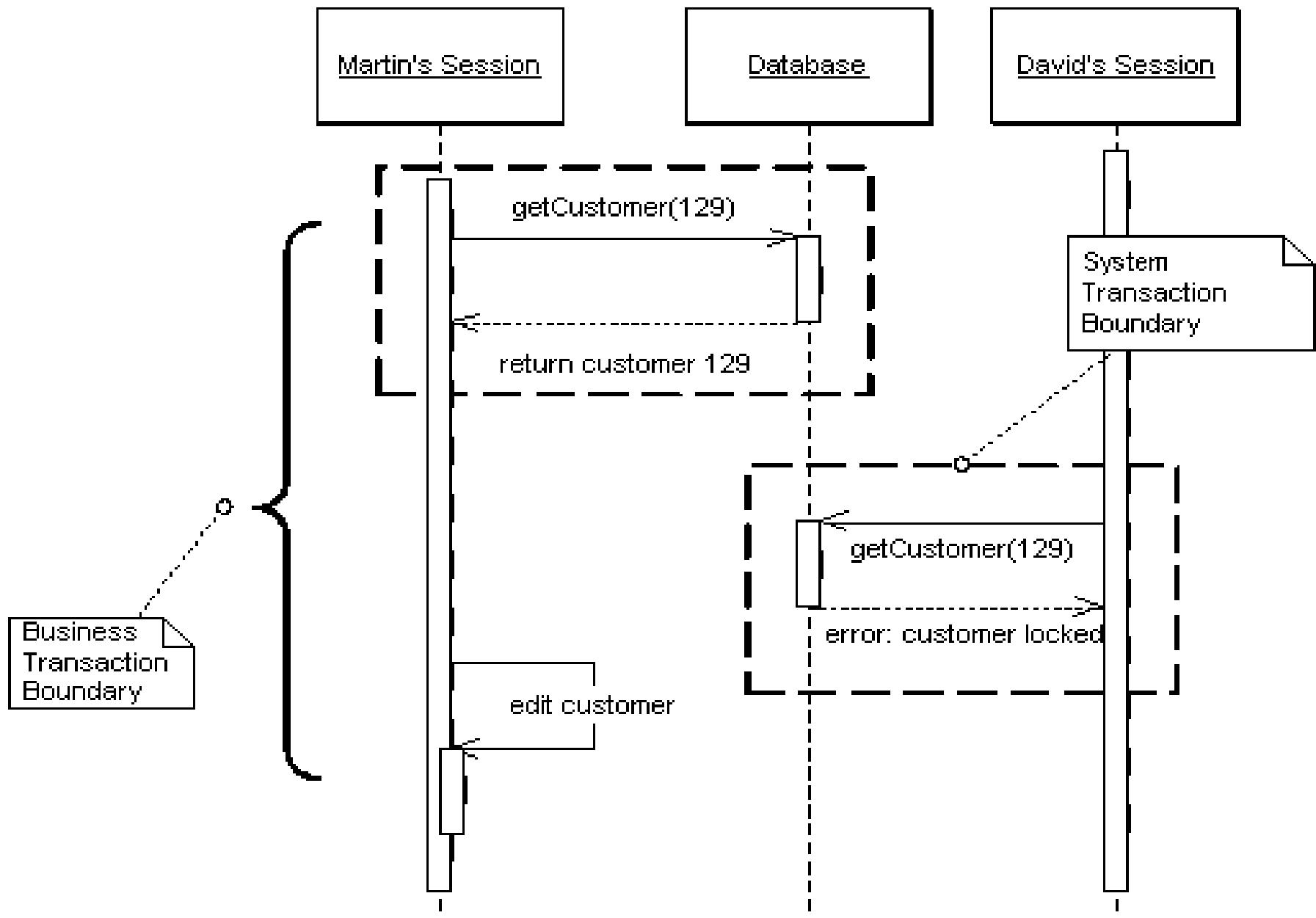
Handles conflicts between concurrent business transactions, by detecting a conflict and rolling back the transaction.

- Conflict detection
- Lock hold during commit
- Supports concurrency
- Suitable for low frequency of conflicts
- Used for not critical consequences

# PESSIMISTIC CONCURRENCY CONTROL

Prevents conflicts between concurrent business transactions by allowing only one business transaction to access data at once.

- Conflict prevention
- Lock hold during the entire transaction
- Does not support concurrency
- Used for critical consequences



# PREVENTING INCONSISTENT READS

Optimistic control

- Versioning

Pessimistic control

- Read -> shared lock
- Write -> exclusive lock

Temporal reads

- Date+time stamps
- Implies full history storage

# DEADLOCKS

- Pick a victim
- Locks with deadlines
- Preventing:
  - Force to acquire all the necessary locks at the beginning
  - Enforce a strategy to grant locks (ex. Alphabetical order of the files)

Combine tactics

# LOCKING

To implement it you need to:

- know what type of locks you need,
- build a lock manager,
- define procedures for a business transaction to use locks

Lock types

- Exclusive write lock
- Exclusive read lock
- Read/write lock
  - Read and write locks are mutually exclusive.
  - Concurrent read locks are acceptable

# LOCK MANAGER

- Responsibility = to grant or deny any request by a business transaction to acquire or release a lock
- A table that maps locks to owners
- Locks should be private to the lock manager.
- Business transactions should access only the lock manager

Protocol of Business transaction to use the lock manager

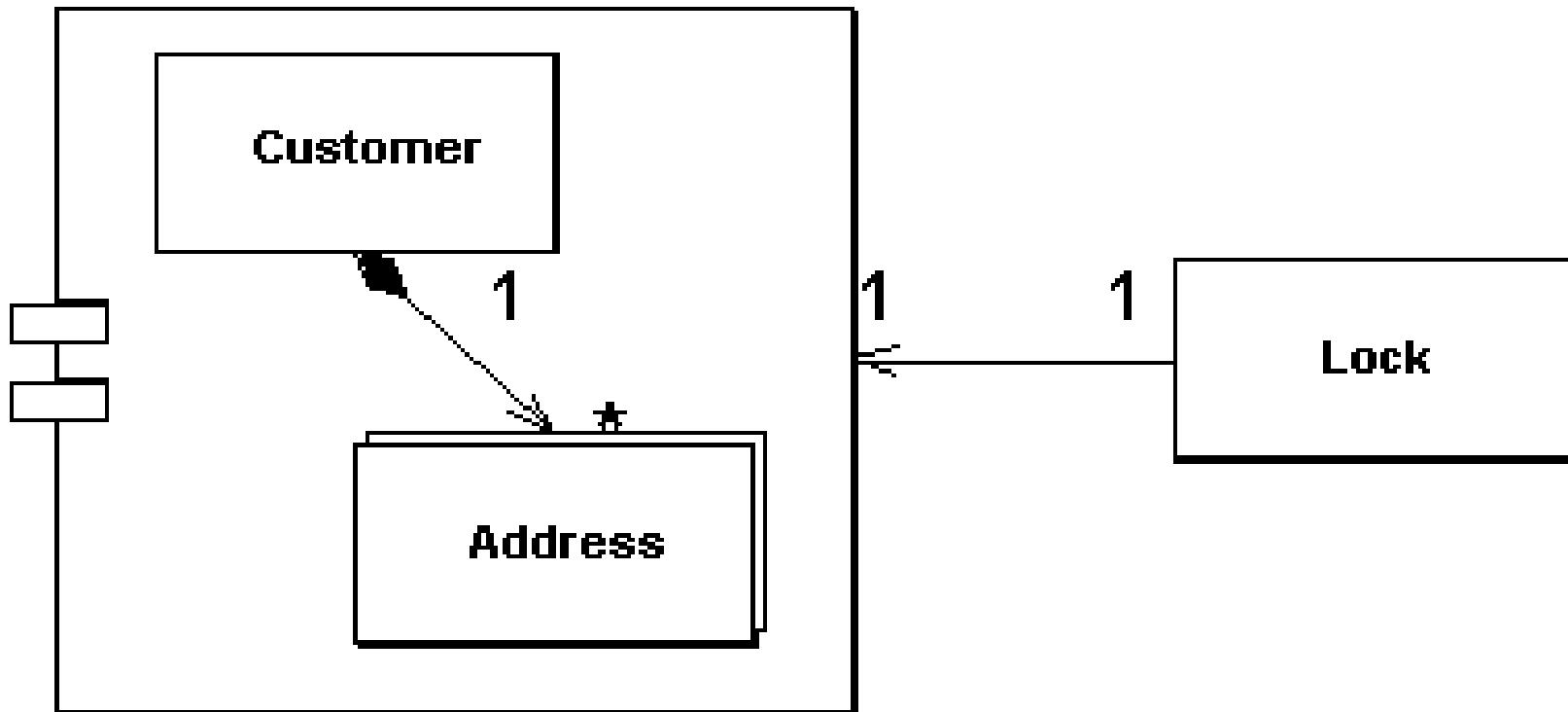
- what to lock (i.e. the ID/primary key)
- when to lock (i.e. lock first, load data next)
- when to release a lock (i.e. after transaction completion),
- how to act when a lock cannot be acquired.

# ANALYSIS

- Access to the lock table must be serialized
- Performance bottleneck
- Consider granularity (Coarse grained lock)
- Possible deadlocks
- Lock timeout for lost sessions

# COARSE-GRAINED LOCK

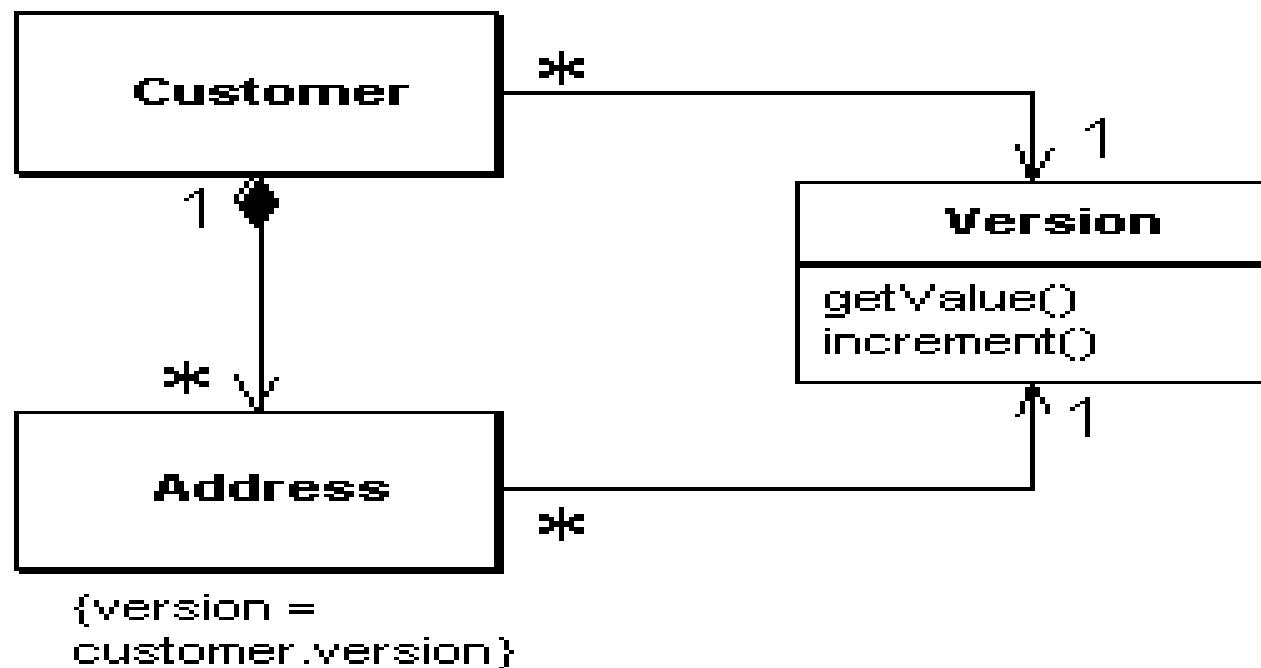
Lock a set of related objects (aggregates) with a single lock



# HOW IT WORKS

A single point of contention for locking a group of objects

Optimistic Lock – shared version



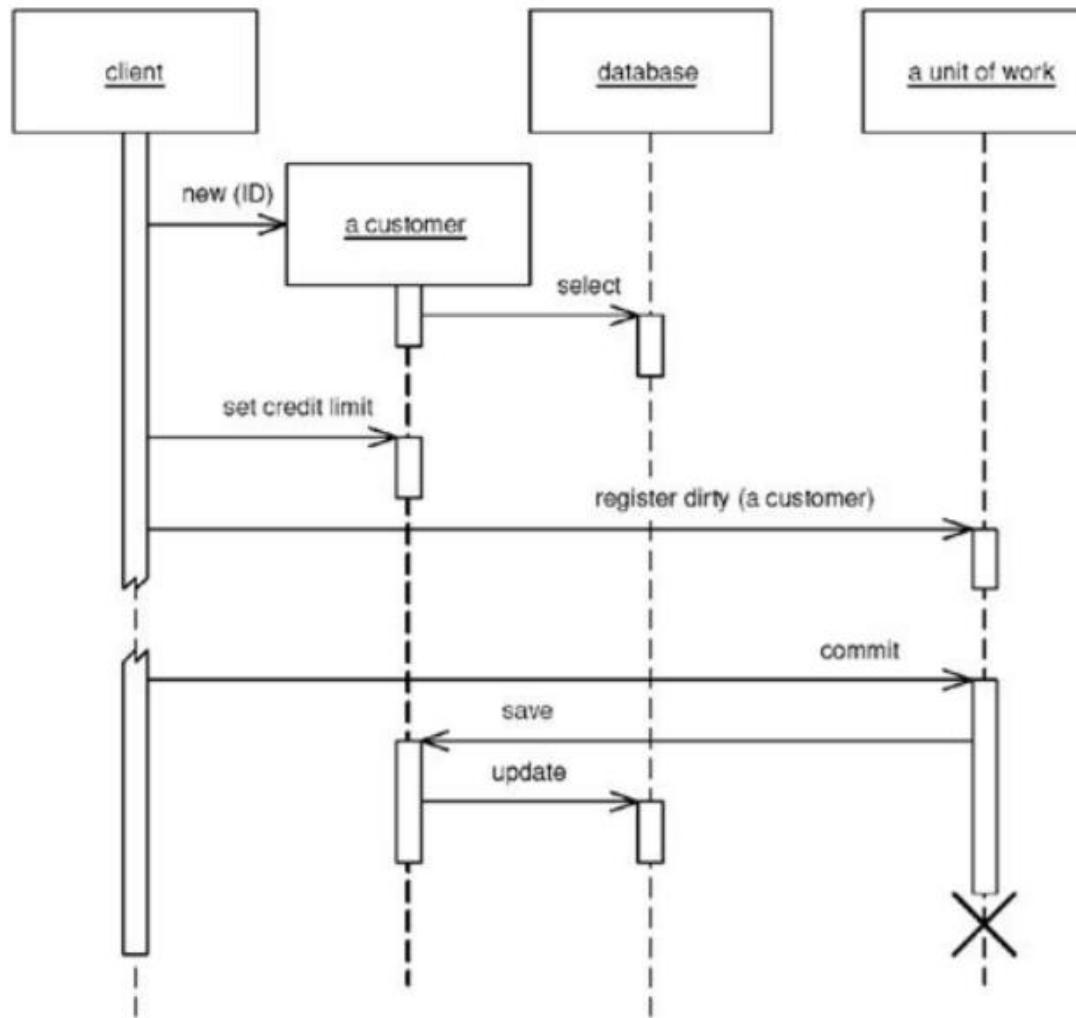
# UNIT OF WORK

- factors the database mapping controller behavior into its own object.
- maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

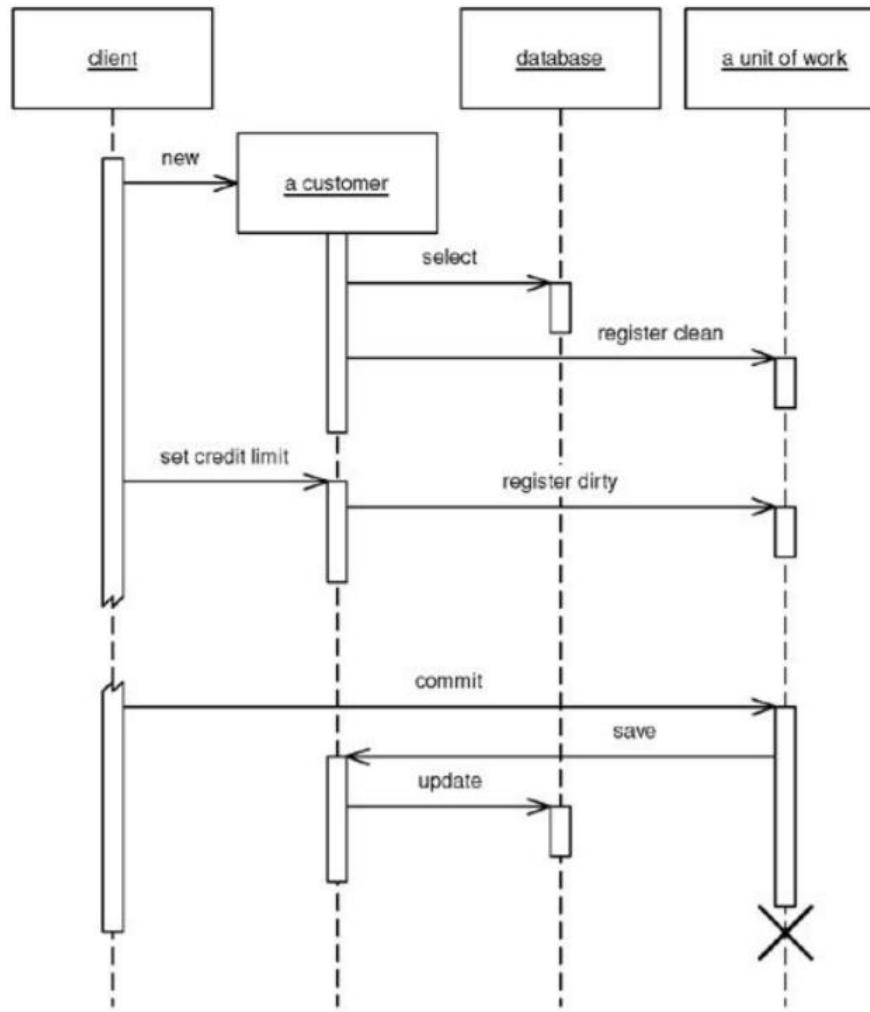
## Unit of Work

```
registerNew(object)  
registerDirty (object)  
registerClean(object)  
registerDeleted(object)  
commit()
```

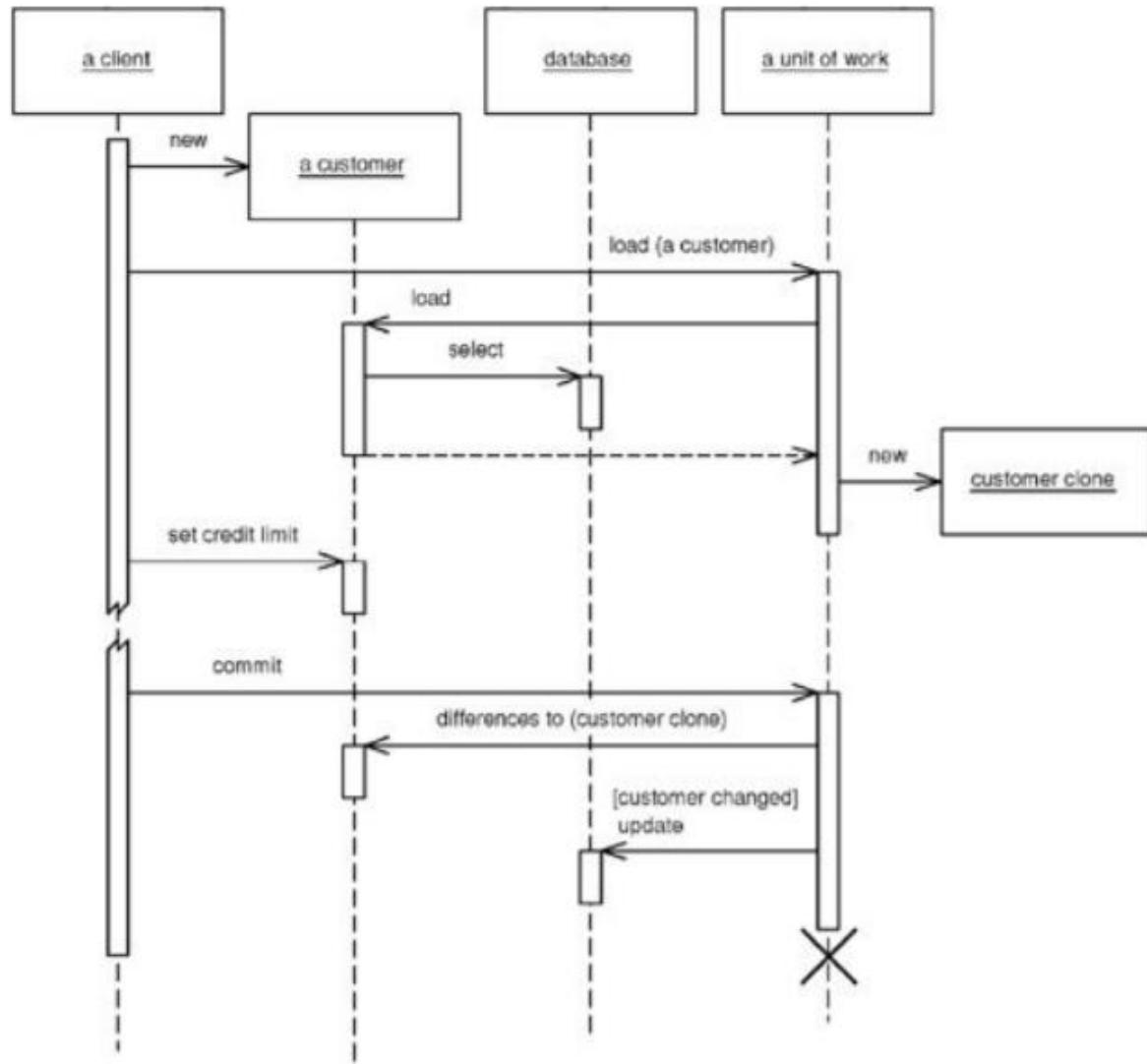
# THE CALLER REGISTERS A CHANGED OBJECT



# THE RECEIVER OBJECT REGISTERS ITSELF

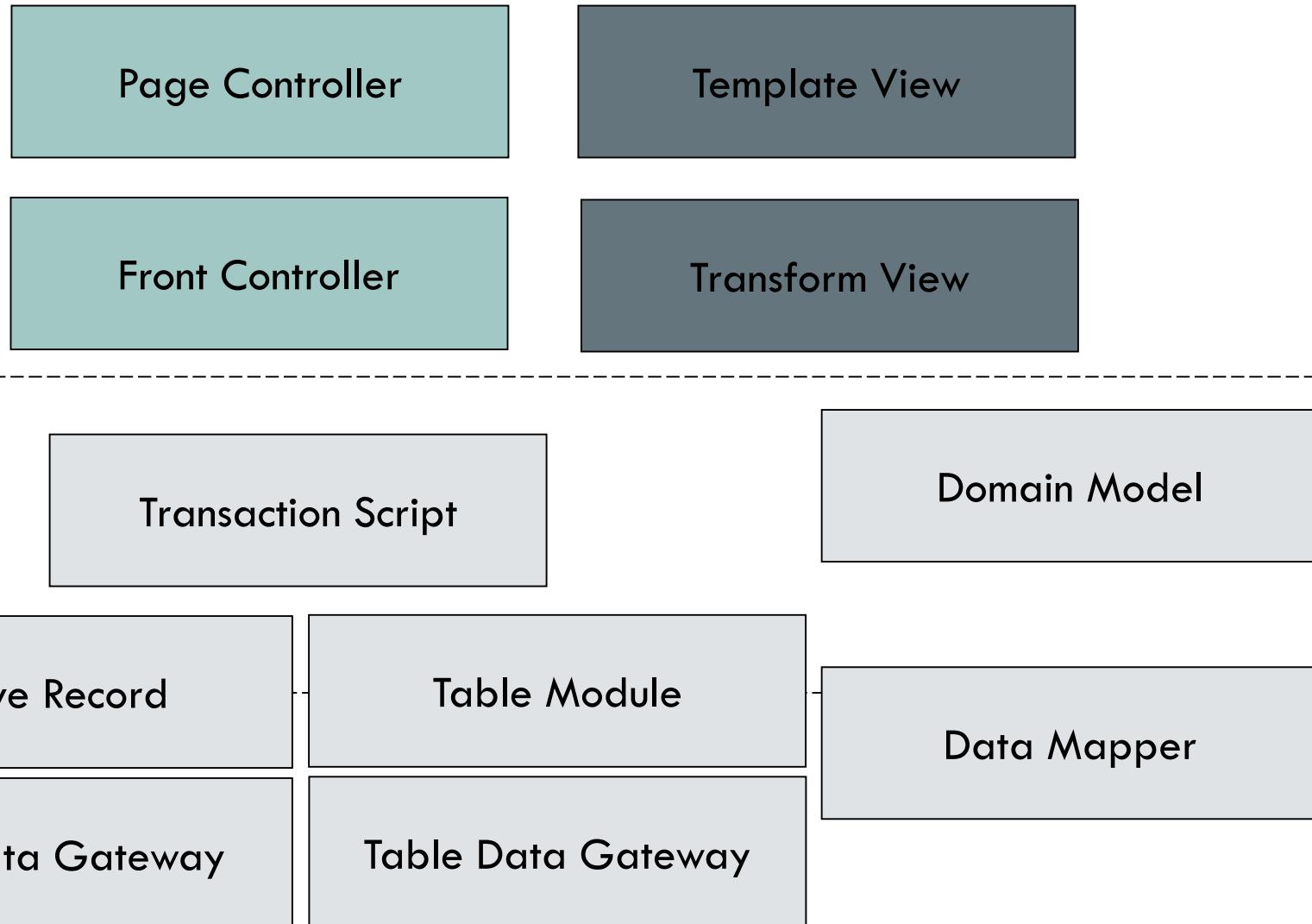


# UNIT OF WORK AS THE CONTROLLER FOR DATABASE ACCESS

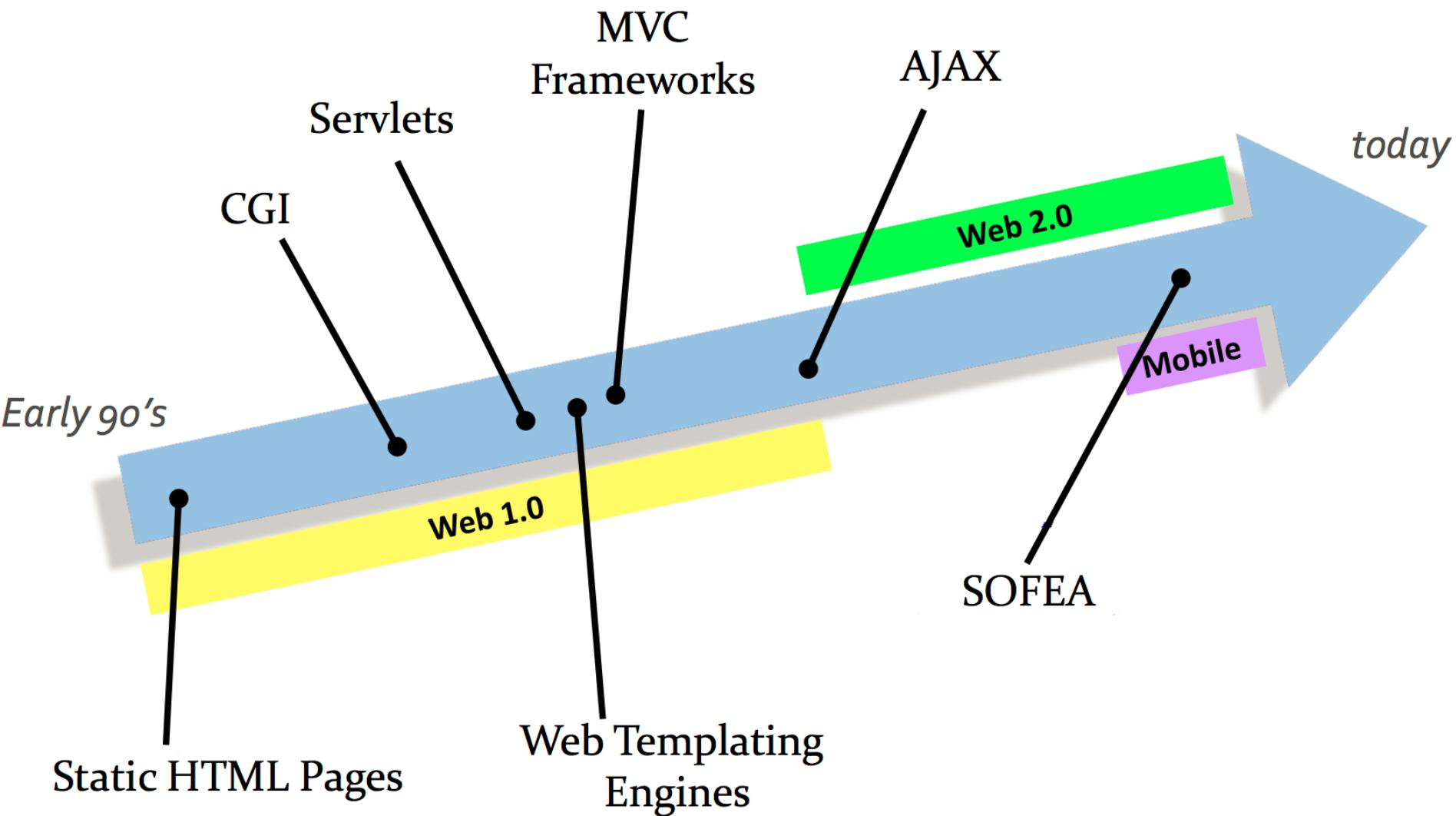


# DISCUSSION

- Unit of Work can be helpful:
  - Controlling the update order (if the database uses referential integrity and checks it with each SQL call)
  - Minimize deadlocks (if transactions use the same sequence of tables to edit, store the order)
  - Handle batch updates



# EVOLUTION OF WEB APPLICATION ARCHITECTURE



# EARLY TECHNOLOGY

HTML (HyperText Markup Language)

- Standard markup language used to create Web pages

CGI (Common Gateway Interfaces)

- Scripts (usually Perl) using common interface between the Web server and programs that generate Web content

Servlet

- Java programming to extend the capabilities of the web server
- Well defined API through run-time environment
- Typically used for dynamic web content generation

# WEB TEMPLATING ENGINE

- Embedded code within static HTML elements
- Mix of static and dynamic HTML
  - “Model 1” Architecture
- Examples
  - Java Server Pages (JSP)
  - PHP
  - Active Server Pages (ASP) .Net

# WEB TEMPLATING ENGINE

Web Template

```
<html> <-- Code  
Hello,  
<b>{$db.name.102}</b> -- Markup  
<html>
```

```
01 Ted  
02 Susan  
. .  
101 Joe  
102 Bob
```



Web Template Engine



Web Browser

Hello, **Bob**

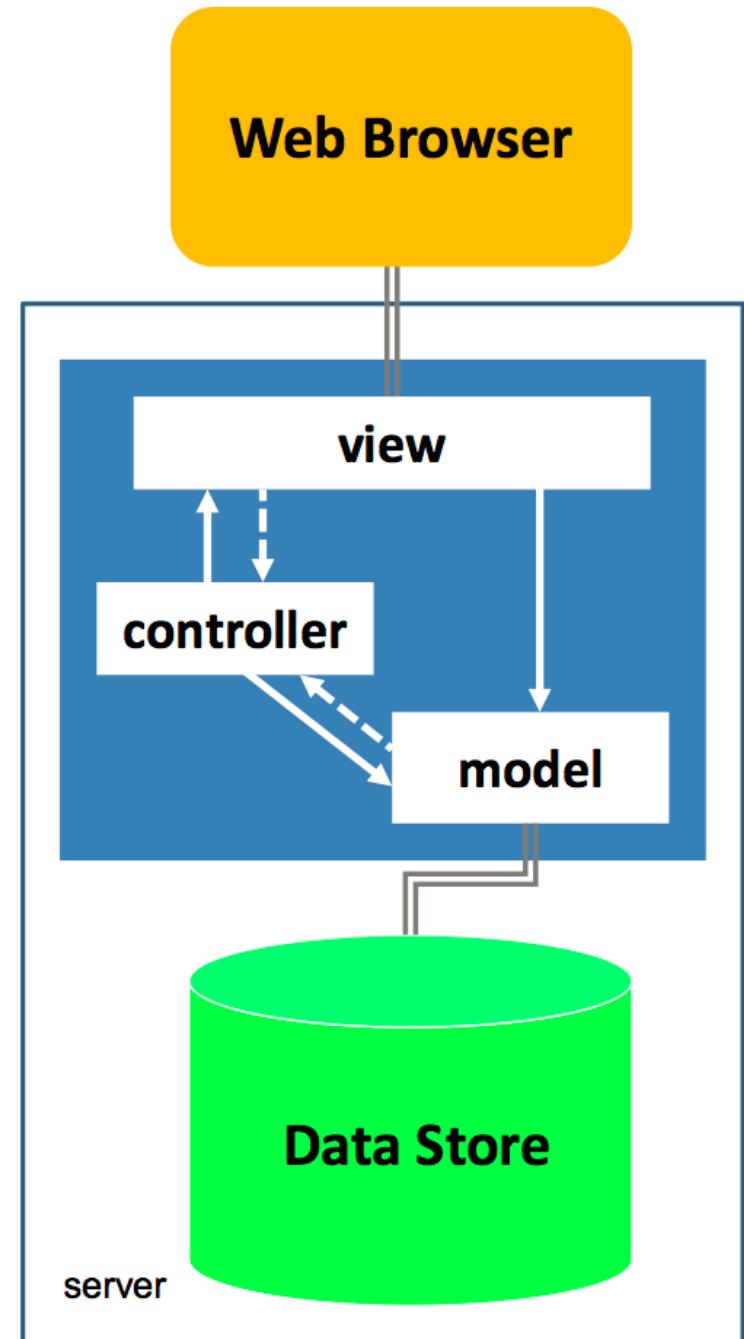
# MVC FRAMEWORKS

## MVC Pattern

- ServerSide Framework
- “Model 2” Architecture

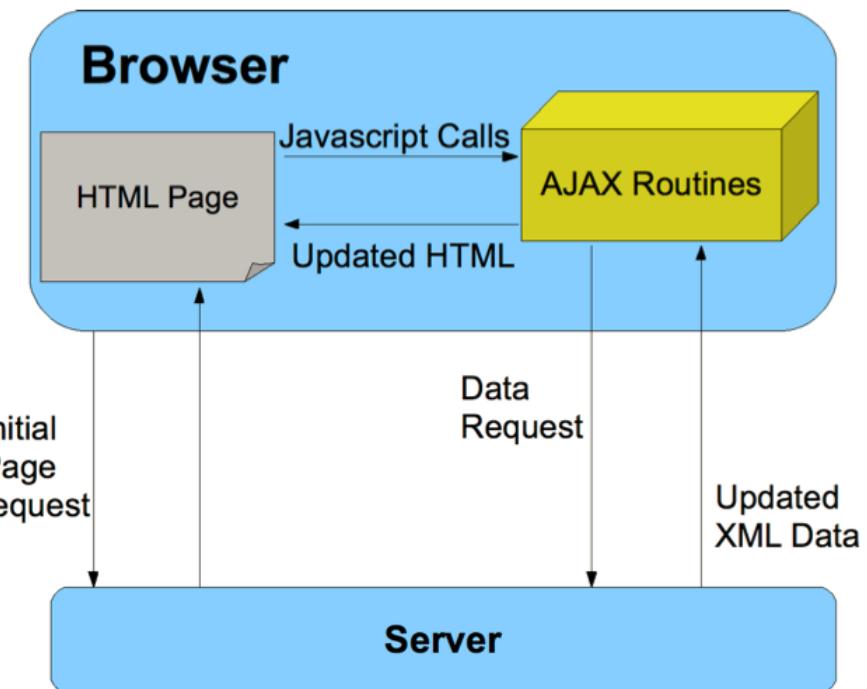
## Examples

- ASP. NET MVC Framework
- Struts, Spring MVC (Java)
- Ruby on Rails (Ruby)
- Django (Python)
- Grails (Groovy)



# AJAX

- Asynchronous JavaScript And XML
- Not a programming language
- Dynamic content changes without reloading the entire page
- HTML/CSS + DOM + XMLHttpRequest Object + JavaScript + JSON/XML



# PROCESS OF WEB APPLICATION

## 1. Application Download

Mobile code (JavaScript, HTML, Applets, Flash) download to the client (web browser)

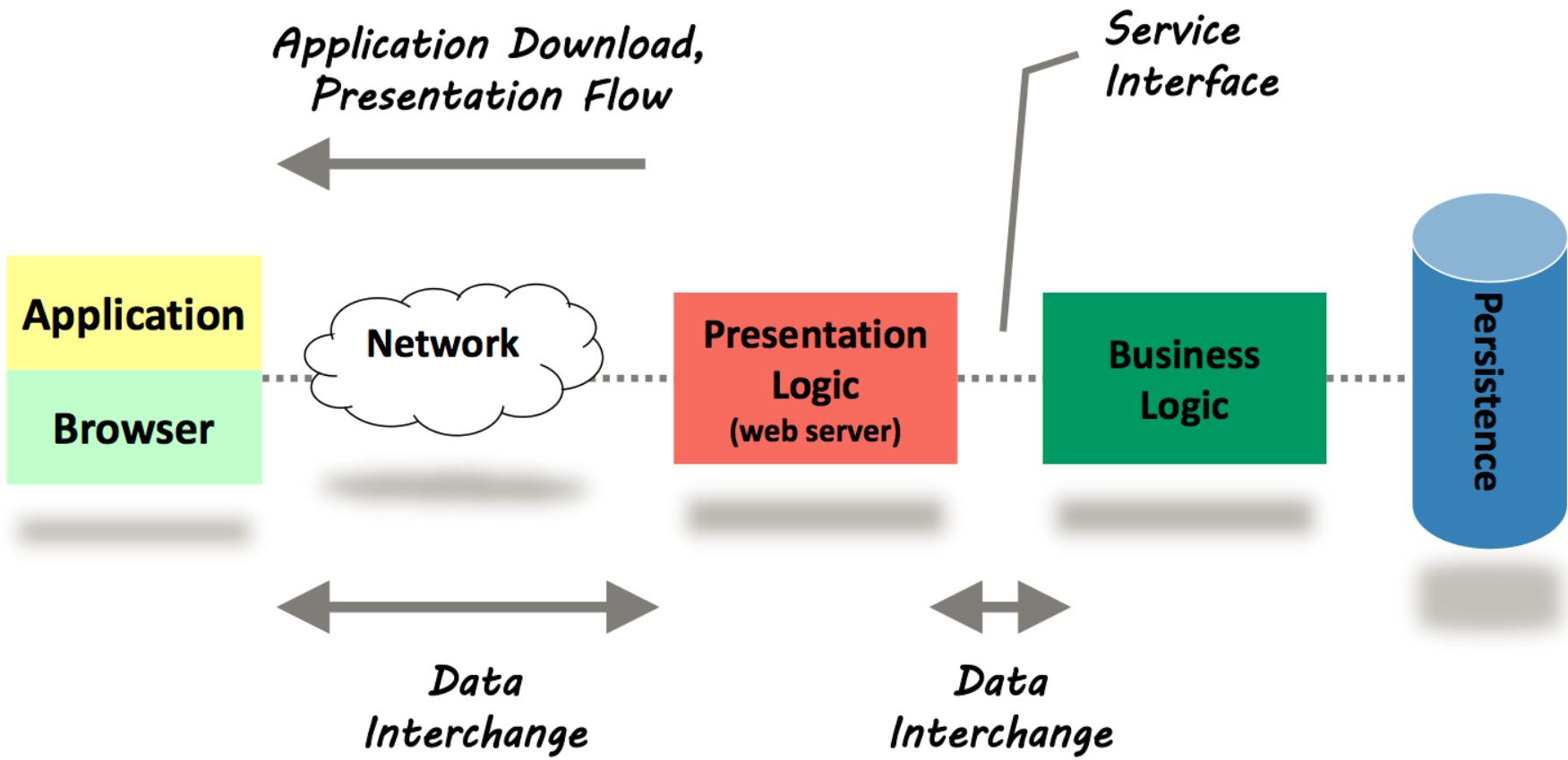
## 2. Presentation Flow

Dynamic visual rendering of the UI (screen changes, new screens, etc.) in response to user input and data state changes

## 3. Data Interchange

The exchange of data between two software components or tiers (search, updates, retrieval, etc.)

# WEB TEMPLATING ENGINE FRAMEWORK



# CHARACTERISTICS

**Tight coupling between Presentation Flow and Data Interchange (both in the web server)**

- Triggering a Presentation Flow (web page update) in a web application initiates a Data Interchange operation
- Every Data Interchange operation results in a Presentation Flow operation

**Presentation Flow and Data Interchange are *orthogonal* concerns that should be decoupled**

- Separate concerns

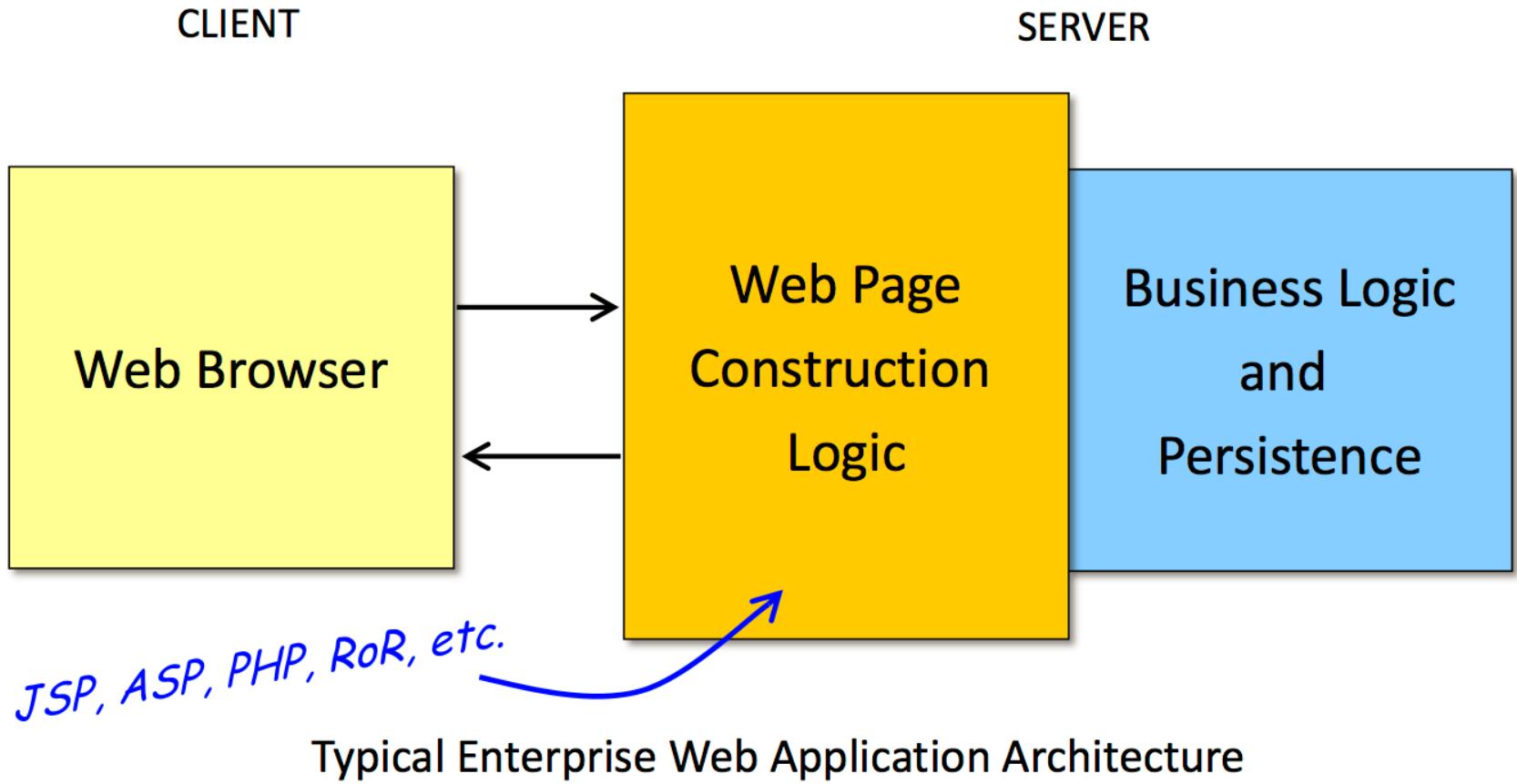
# SERVICE ORIENTED FRONT END ARCHITECTURE (SOFEA)

**Service Oriented Front End Architecture – Synonymous with “Single Page” Web Applications (SPA)**

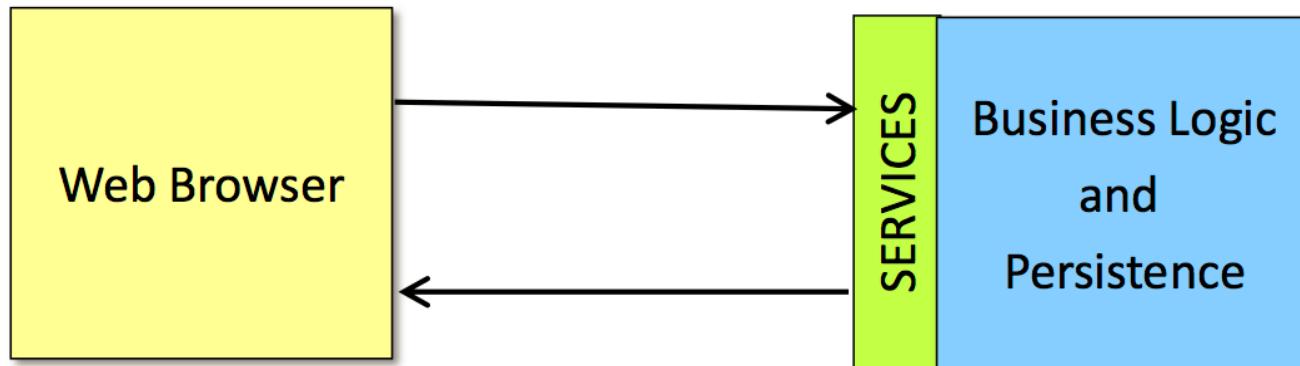
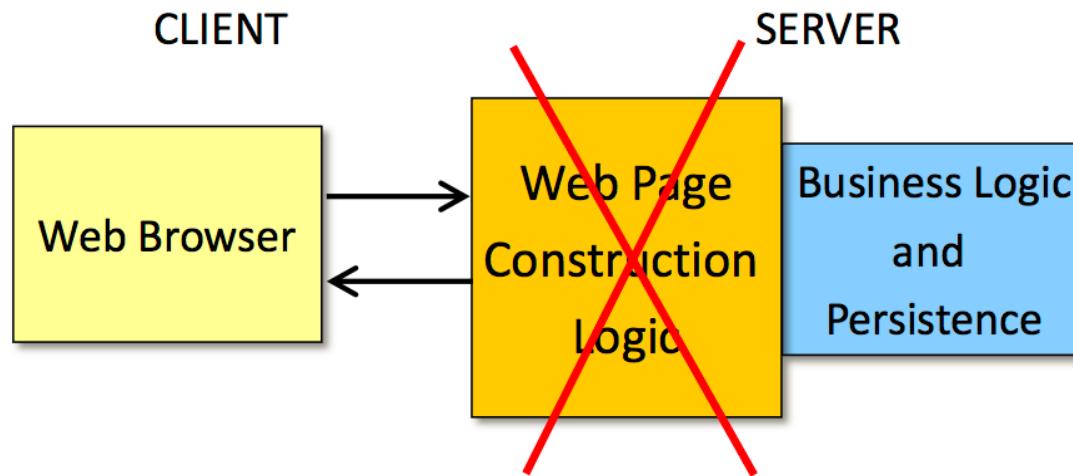
Life above the Service Tier

- How to Build Application Front-ends in a Service-Oriented World (*Ganesh Prasad, Rajat Taneja, Vikrant Todankar*)

# LEGACY ARCHITECTURE



# SOFEA

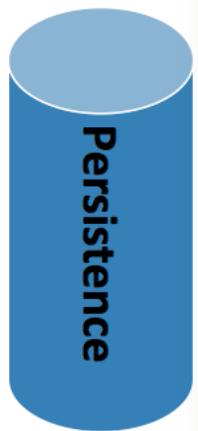
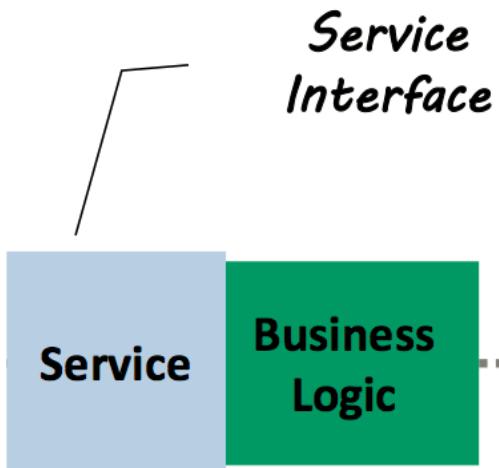
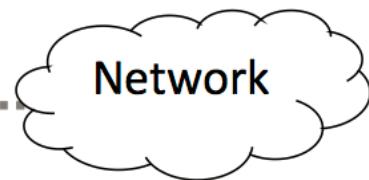


# SOFEA PROCESS ALLOCATION

*Presentation Flow* ↗



*Application Download*



*Data Interchange*



# SOFEA PRINCIPLES

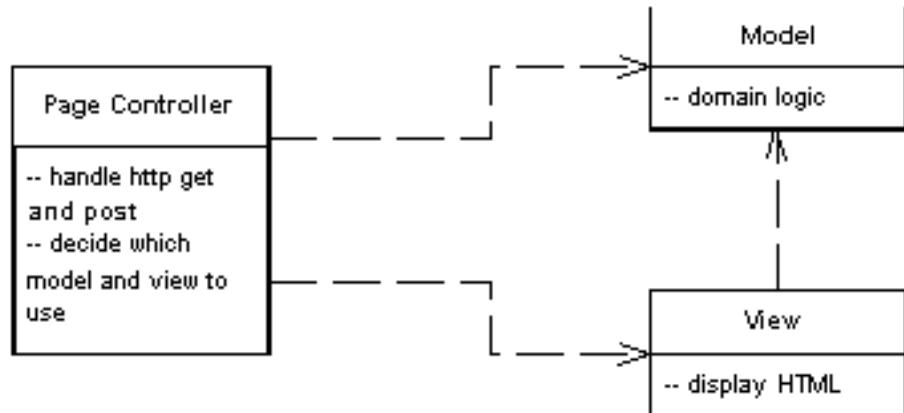
- Application Download, Data Interchange, and Presentation Flow must be decoupled
- No part of the client should be evoked, generated or templated from the server-side.
- Presentation Flow is a *client-side* concern only
- All communication with the application server should be using services (REST, SOAP, etc.)
- The MVC design pattern belongs in the client, not the server



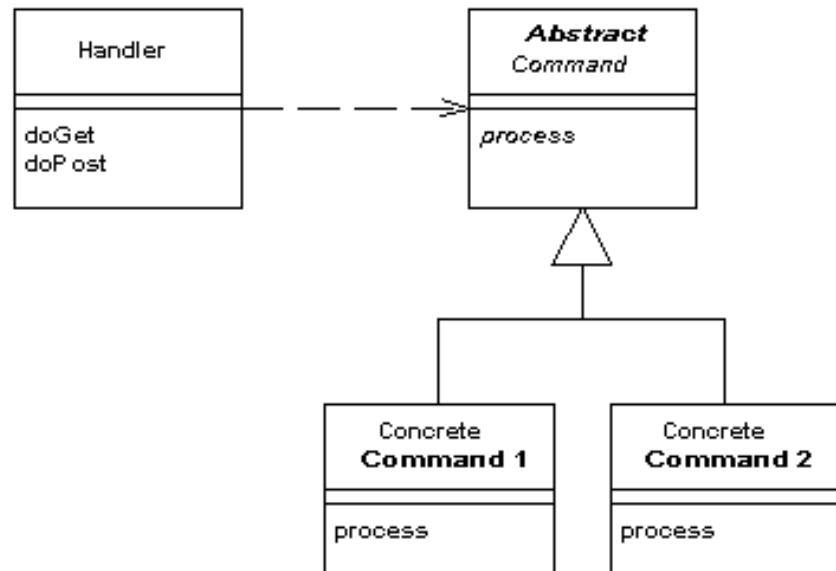
**Going back to MVC frameworks based approach**

# CONTROLLERS

Page controller – an object that handles a request for a specific page or action on a Web site.



Front controller – an object that handles all requests for a web site



# PAGE CONTROLLER

## As Script

- Servlet or CGI program
- Web Applications that need logic and data

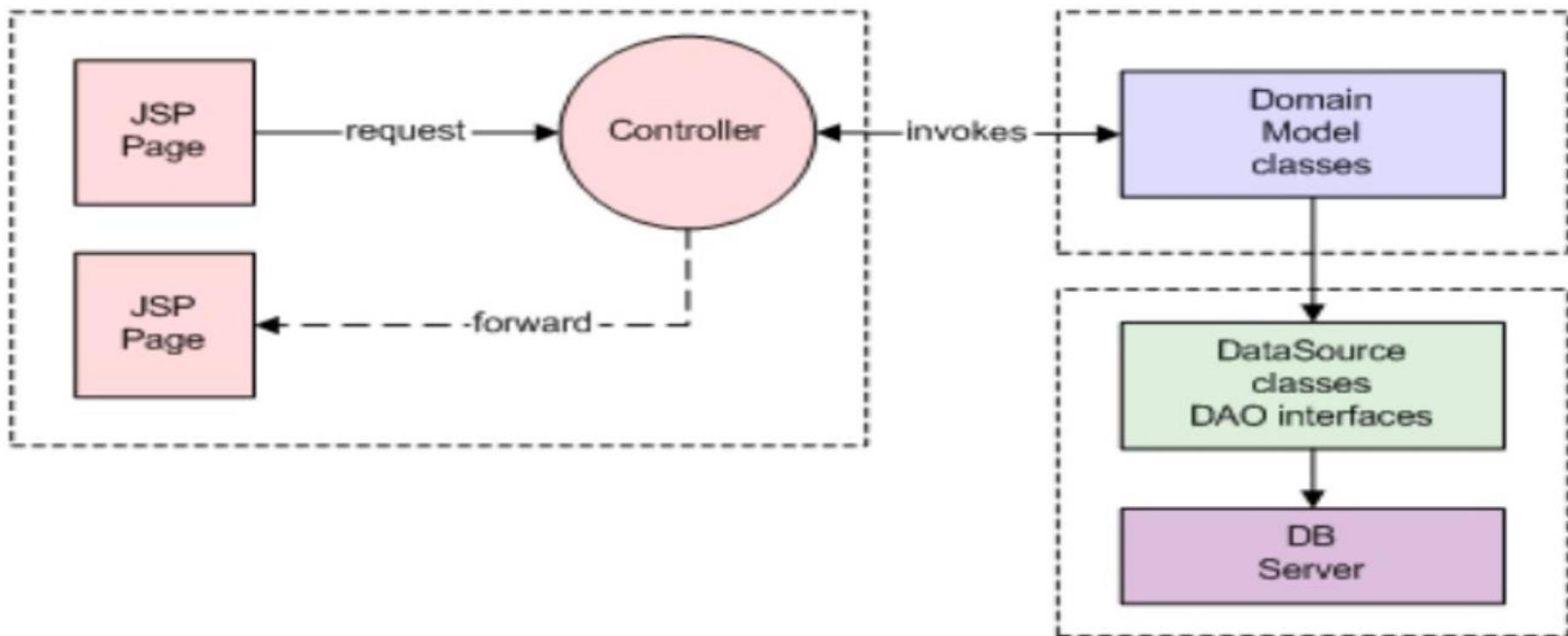
## As Server Page

- ASP, PHP, JSP
- Use helpers to get data from the model
- Logic is simple to none
- Combines Page Controller + Template View

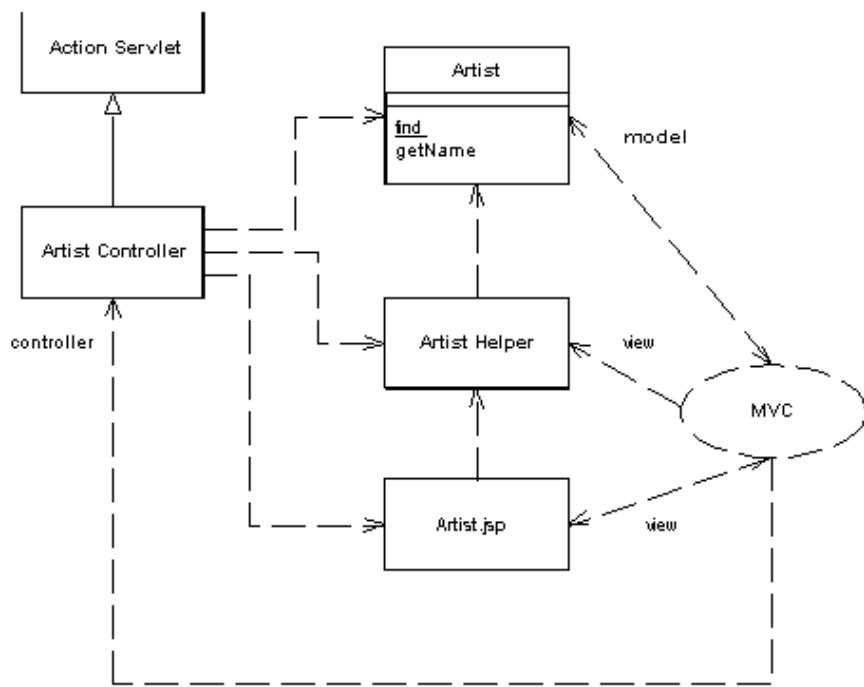
## Basic responsibilities

- **Decode the URL** and extract all data for the action.
- **Create and invoke any model objects** to process the data. All relevant data from the HTML request should be passed to the model so that the model objects don't need any connection to the HTML request.
- **Determine which view** should display the result page and forward the model information to it.

# PAGE CONTROLLER



# SERVLET CONTROLLER AND A JSP VIEW (JAVA)



```
class ArtistController...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        if (artist == null)
            forward("/MissingArtistError.jsp", request, response);
        else {
            request.setAttribute("helper", new ArtistHelper(artist));
            forward("/artist.jsp", request, response);
        }
    }
```

<http://www.thingy.com/recordingApp/artist?name=danielaMercury>.

In web.xml map /artist to a call to ArtistController

```
<servlet>
<servlet-name>artist</servlet-name>
<servlet-class>actionController.ArtistController
</servlet-class>
</servlet>
```

```
<servlet-mapping>
<servlet-name>artist</servlet-name>
<url-pattern>/artist</url-pattern>
</servlet-mapping>
```

# JSP AS REQUEST HANDLER

Delegates control to the helper

The handler JSP is the default view

album.jsp...

```
<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
```

class AlbumConHelper extends HelperController...

```
public void init(HttpServletRequest request, HttpServletResponse response) {
    super.init(request, response);
    if (getAlbum() == null) forward("missingAlbumError.jsp", request, response);
    if (getAlbum() instanceof ClassicalAlbum) {
        request.setAttribute("helper", getAlbum());
        forward("/classicalAlbum.jsp", request, response);
    }
}
```

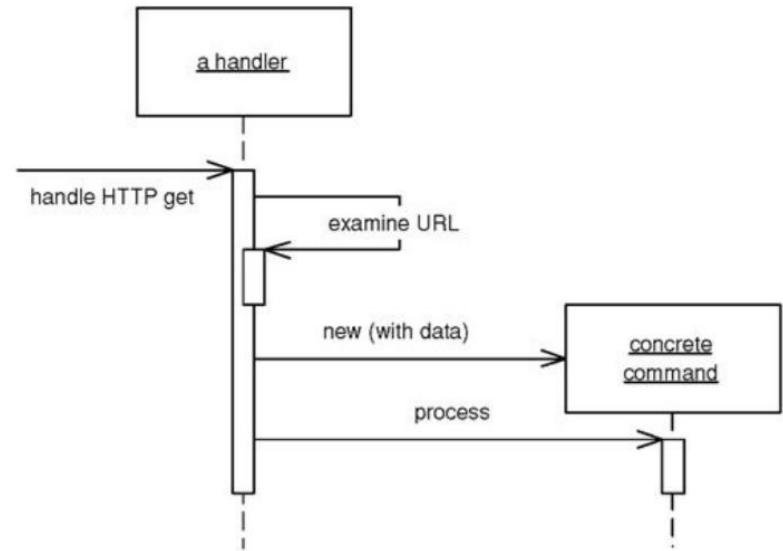
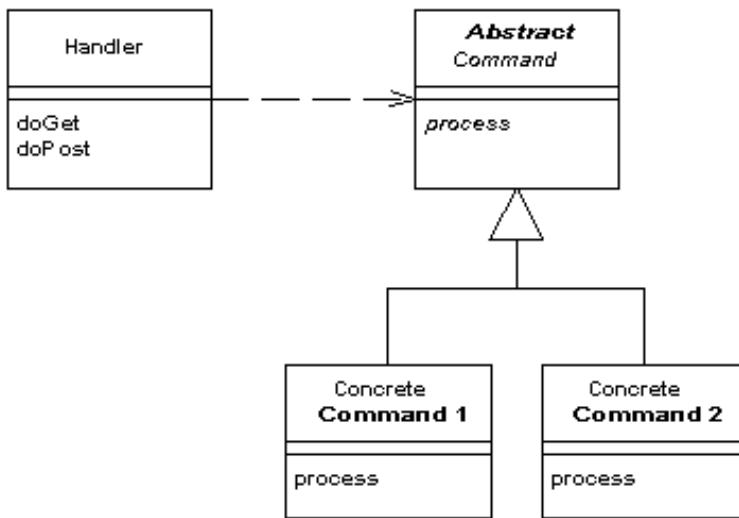
# FRONT CONTROLLER

If many similar things are done when handling a request (i.e. security, internationalization, etc.)

One controller handles all requests

Usually handles in 2 phases:

- Request handling - a web handler (rather a class than a server page)
- Command handling - a hierarchy of commands (classes)



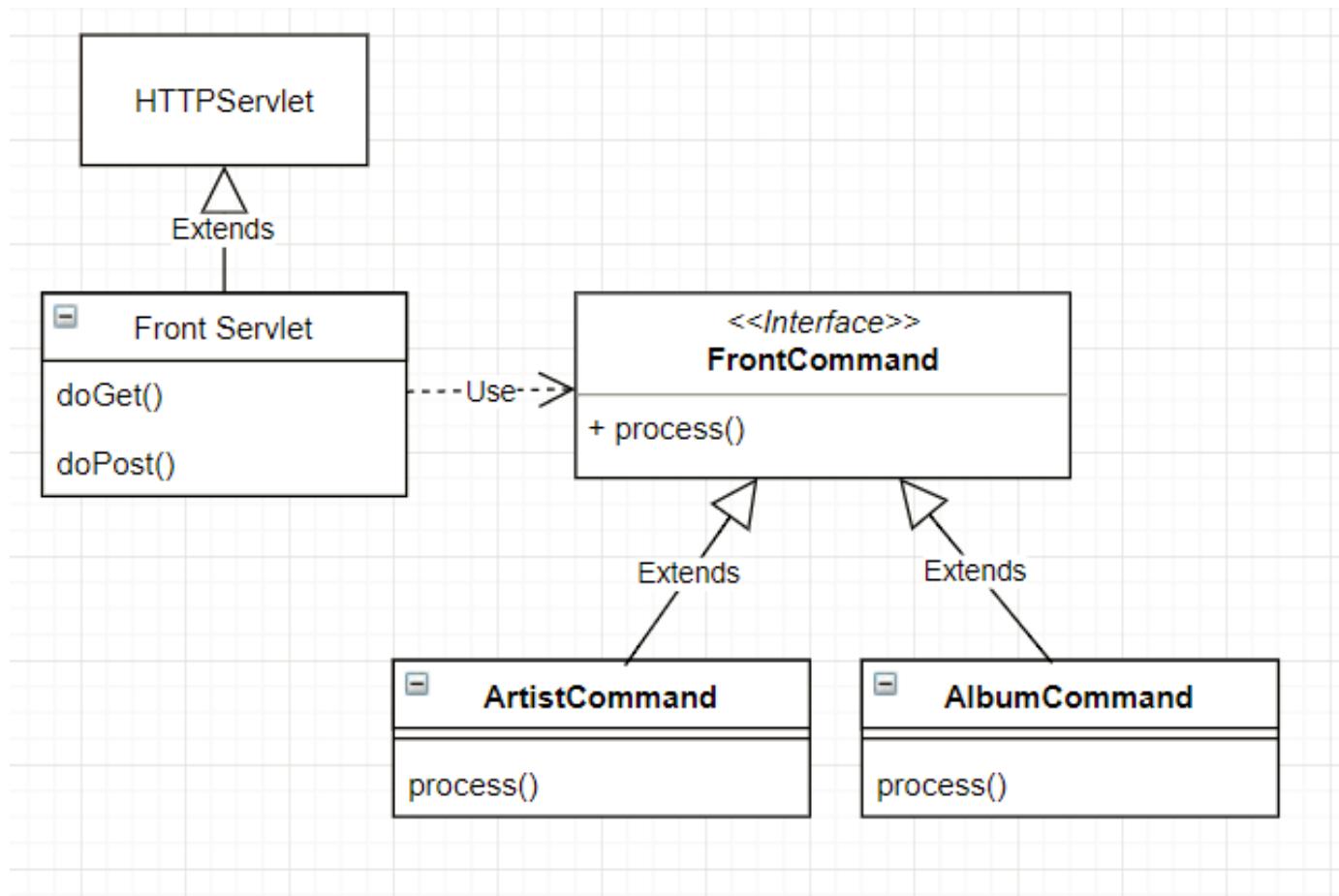
# FRONT CONTROLLER

Handler decides what command:

- **statically**
  - parses the URL and uses conditional logic;
  - advantage of explicit logic,
  - compile time error checking on dispatch,
  - flexibility in URL look-up
- **dynamically**
  - takes a standard piece of the URL and uses dynamic instantiation to create a command class;
  - allows to add new commands without changing the Web handler;
  - can put the name of the command class into the URL or can use a properties file that binds URLs to command class names.

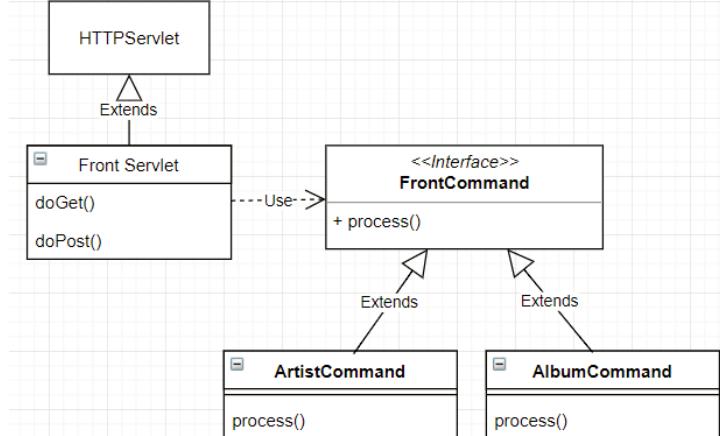
# EXAMPLE

<http://localhost:8080/isa/music?name=astor&command=Artist>



# EXAMPLE

[http://localhost:8080/isa/  
music?name=astor&command=Artist](http://localhost:8080/isa/music?name=astor&command=Artist)



```
class FrontServlet...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        FrontCommand command = getCommand(request);
        command.init(getServletContext(), request, response);
        command.process();
    }

    private FrontCommand getCommand(HttpServletRequest request) {
        try {
            return (FrontCommand) getCommandClass(request).newInstance();
        } catch (Exception e) {
            throw new ApplicationException(e);
        }
    }

    private Class getCommandClass(HttpServletRequest request) {
        Class result;
        final String commandClassName =
            "frontController." + (String) request.getParameter("command") + "Command";
        try {
            result = Class.forName(commandClassName);
        } catch (ClassNotFoundException e) {
            result = UnknownCommand.class;
        }
        return result;
    }
}
```

```
class FrontCommand...
    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;

    public void init(ServletContext context,
                      HttpServletRequest request,
                      HttpServletResponse response)
    {
        this.context = context;
        this.request = request;
        this.response = response;
    }

    abstract public void process() throws ServletException, IOException ;

    protected void forward(String target) throws ServletException, IOException
    {
        RequestDispatcher dispatcher = context.getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
}

class ArtistCommand...
public void process() throws ServletException, IOException {
    Artist artist = Artist.findNamed(request.getParameter("name"));
    request.setAttribute("helper", new ArtistHelper(artist));
    forward("/artist.jsp");
}
```

# DISCUSSION

- Only one Front Controller has to be configured into the Web server
- You can add new commands without changing anything.
- Because new command objects are created with each request, it is thread safe (provided model objects are not shared!).
- Both the handler and the commands are part of the controller. As a result the commands can (and should) choose which view to use for the response. The only responsibility of the handler is in choosing which command to execute.
- Re-factor code better in command hierarchy

# DISCUSSION

## **Page Controller:**

- simple controller logic
- a natural structuring mechanism where particular actions are handled by particular server pages or script classes.

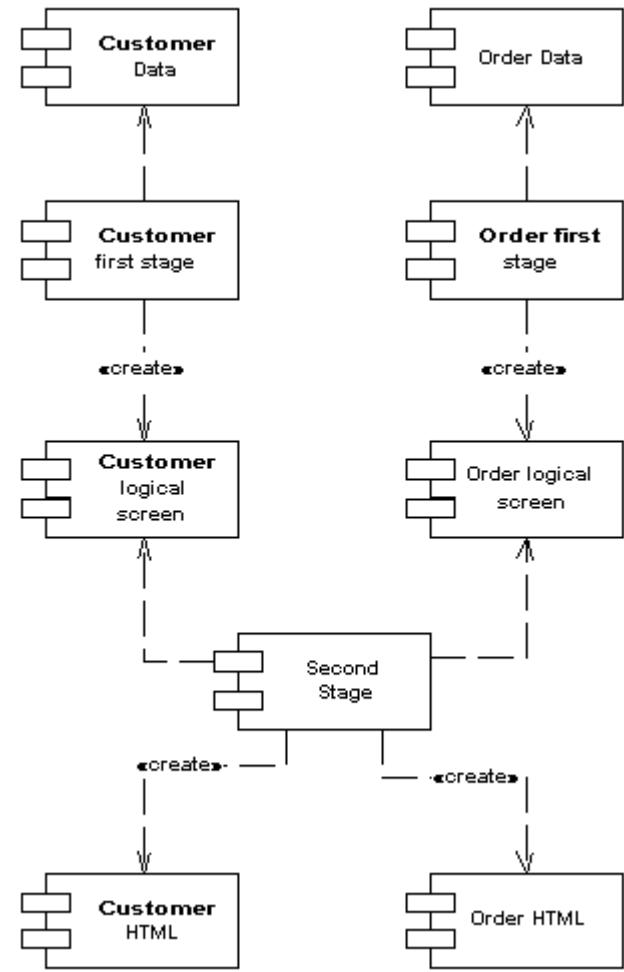
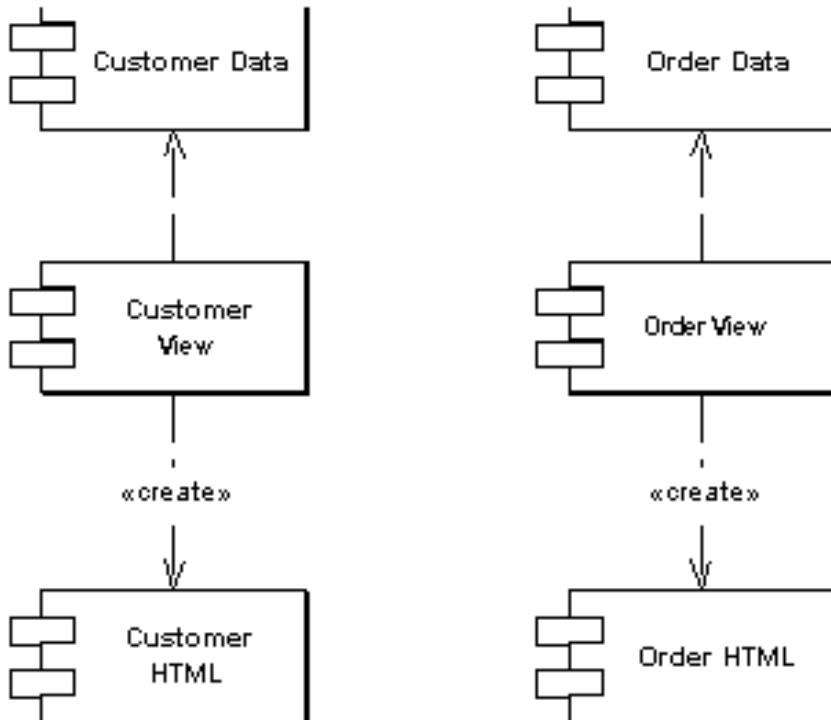
## **Front Controller:**

- greater complexity;
- handles duplicated features (i.e. security, internationalization, providing particular views for certain kinds of users) in one place.
- single point of entry for centralized logic

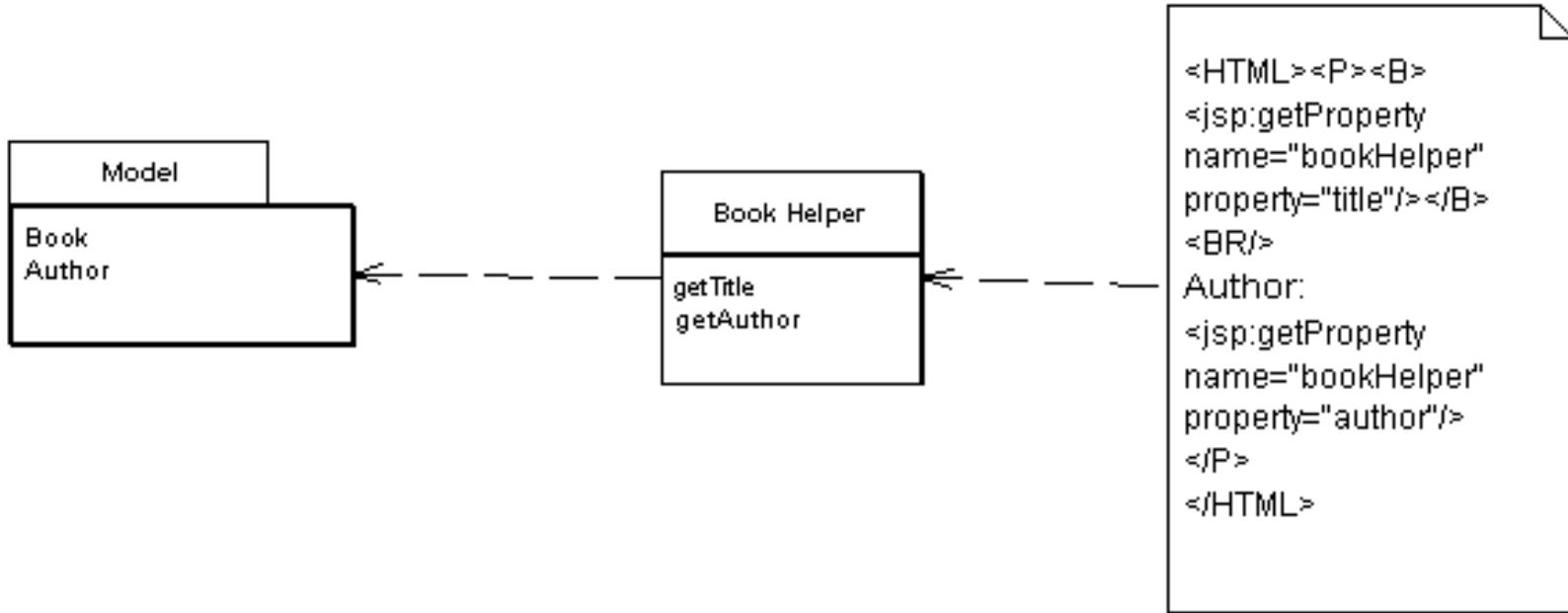
# VIEW

## Two step stage

### Single step stage



# TEMPLATE VIEW



- Embed markers into a static HTML page when it's written
- When the page is used to service requests, the markers are replaced by the results of some computation
- **server pages:**
  - ASP, JSP, or PHP.
  - allow to embed arbitrary programming logic, referred to as **scriptlets**, into the page.

# CONDITIONAL DISPLAY

```
<IF condition = "$pricedrop > 0.1"> ... show some stuff </IF>
```

Templates become programming languages

⇒ Move the condition to the helper to generate the content

⇒ What if the content should be displayed but in different ways?

- Helper generates the markup
- OR use focused tags:

```
<IF expression = "isHighSelling()"><B></IF><property name =  
"price"/><IF expression = "isHighSelling()"></B></IF>
```

replaced by

```
<highlight condition = "isHighSelling" style =  
"bold"><property name = "price"/></highlight>
```

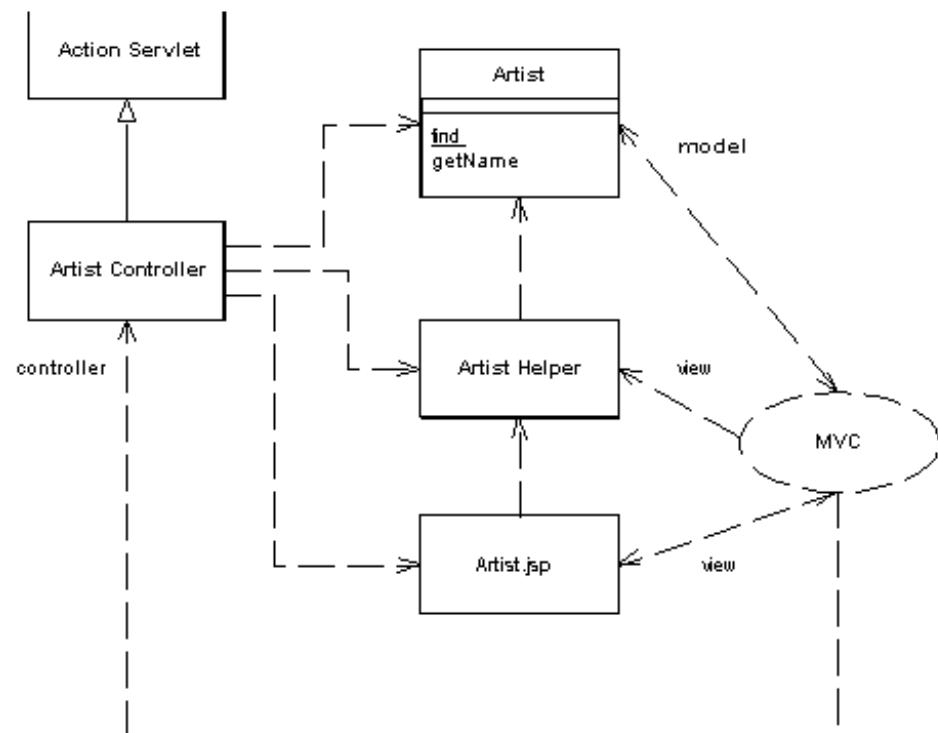
# JSP TEMPLATE VIEW (SEE PAGE CONTROLLER)

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```

```
class ArtistHelper...
    private Artist artist;

    public ArtistHelper(Artist artist) {
        this.artist = artist;
    }

    public String getName() {
        return artist.getName();
    }
```



To access the information from the helper

**<B> <%=helper.getName () %></B>** or

**<B><jsp:getProperty name="helper" property="name"/></B>**

# SHOW A LIST OF ALBUMS FOR AN ARTIST

```
<UL>
<%
    for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
        Album album = (Album) it.next();%>
<LI><%=album.getTitle()%></LI>

<%
    }    %>
</UL>
```

```
class ArtistHelper...
    public String getAlbumList() {
        StringBuffer result = new StringBuffer();
        result.append("<UL>");
        for (Iterator it = getAlbums().iterator(); it.hasNext();) {
            Album album = (Album) it.next();
            result.append("<LI>");
            result.append(album.getTitle());
            result.append("</LI>");
        }
        result.append("</UL>");
        return result.toString();
    }

    public List getAlbums() {
        return artist.getAlbums();
    }
```

```
<UL><tag:forEach host = "helper" collection = "albums" id = "each">
    <LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```



# DISCUSSION

## Benefits:

- Compose the structure of the page based on the template
- Separate design from code (helper)

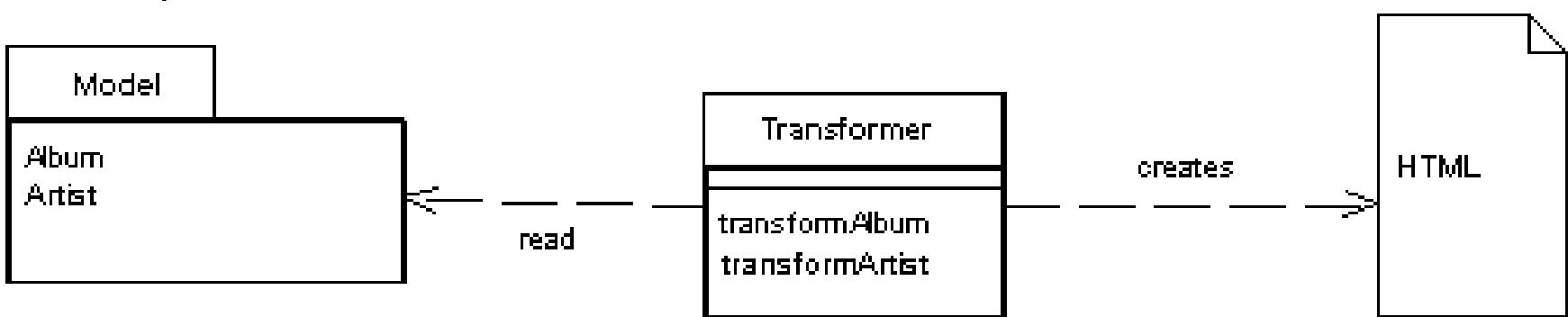
## Liabilities

- Common implementations make it too easy to put complicated logic onto the page => hard to maintain
- Harder to test than Transform View

# TRANSFORM VIEW

Input: Model

Output: HTML



can be written in any language, yet the dominant choice is XSLT (EXtensible Stylesheet Language Transformation).

Input: XML

XML data can be returned as:

- natural return type
- output type which can be transformed to XML automatically
- Data Transfer Object, that can serialize as XML

# TRANSLATED IN CODE

```
class AlbumCommand...
    public void process() {
        try {
            Album album = Album.findNamed(request.getParameter("name"));
            Assert.notNull(album);
            PrintWriter out = response.getWriter();
            XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
            out.print(processor.getTransformation(album.toXmlDocument()));
        } catch (Exception e) {
            throw new A
        }
    }

<album>
    <title>Zero Hour</title>
    <artist>Astor Piazzola</artist>
    <trackList>
        <track><title>Tanguedia III</title><time>4:39</time></track>
        <track><title>Milonga del Angel</title><time>6:30</time></track>
        <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
        <track><title>Milonga Loca</title><time>3:05</time></track>
        <track><title>Michelangelo '70</title><time>2:50</time></track>
        <track><title>Contrabajisimo</title><time>10:18</time></track>
        <track><title>Mumuki</title><time>9:32</time></track>
    </trackList>
</album>
```

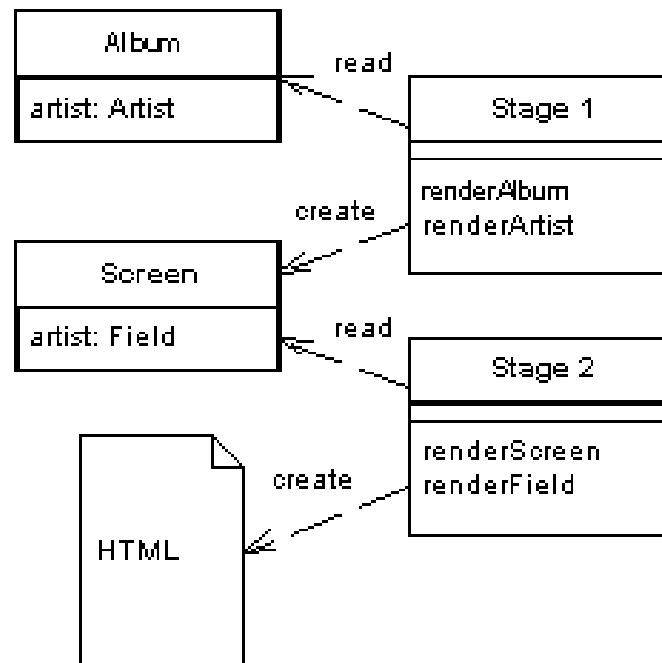
# ADVANTAGES

- Portability: use the same XSLT with XMLs from J2EE or .NET
- Avoid too much logic in view, hence focus on the HTML rendering
- Easier to test: run the Transform View and capture the output for testing.
- Easier to change the appearance of a Web site: change the common transforms.

# TWO STEP VIEW

Multi-page application:

- transforms the model data into a logical presentation without any specific formatting
- converts that logical presentation with the actual formatting needed.



# HOW TO DO IT

two-step XSLT:

- domain-oriented XML => presentation-oriented XML,
- presentation-oriented XML => HTML.

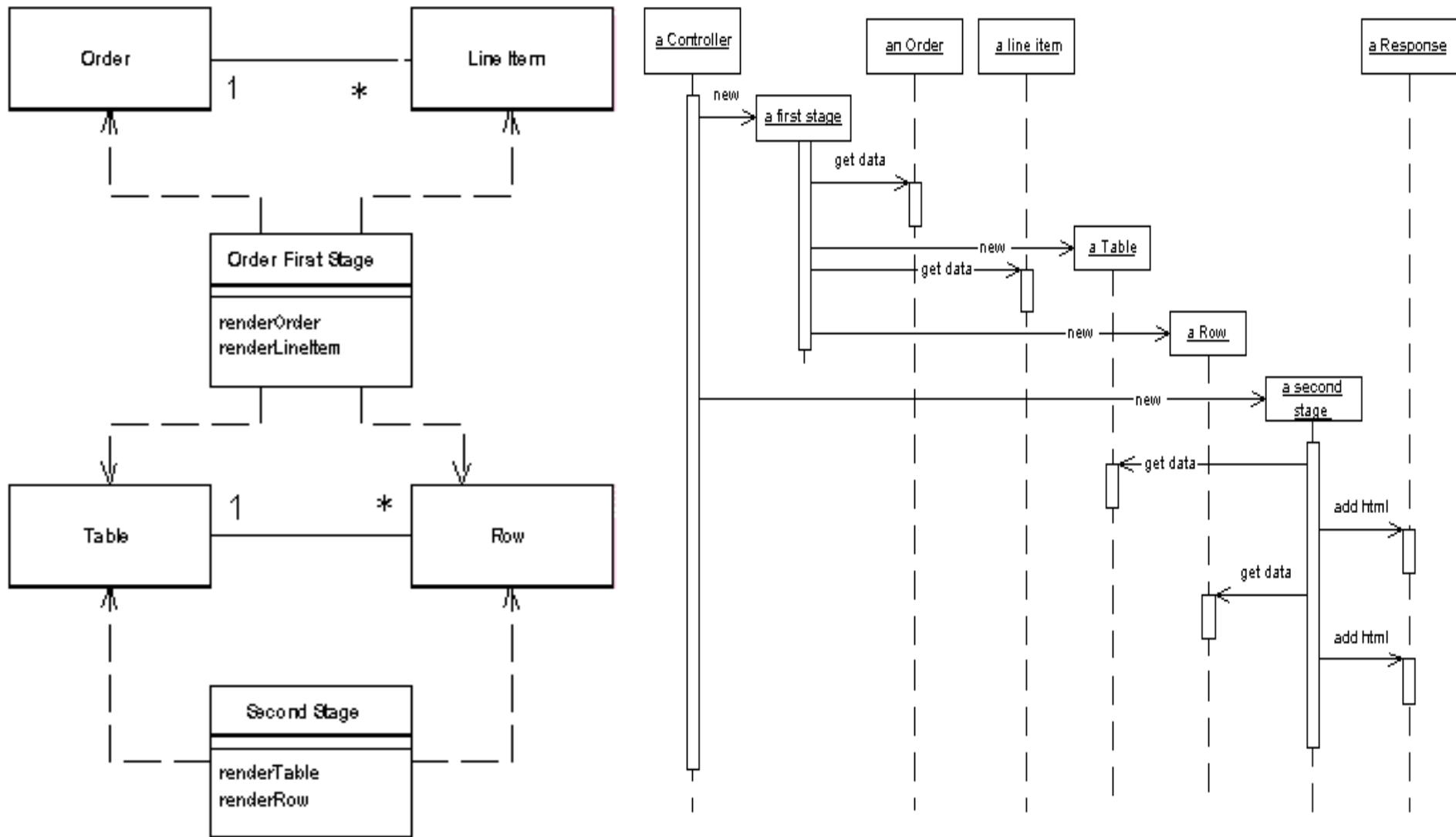
presentation-oriented structure as a set of classes (table/row class):

- domain information instantiates T/R classes.
- renders the T/R classes into HTML
  - each presentation-oriented class generates HTML for itself or
  - having a separate HTML renderer class

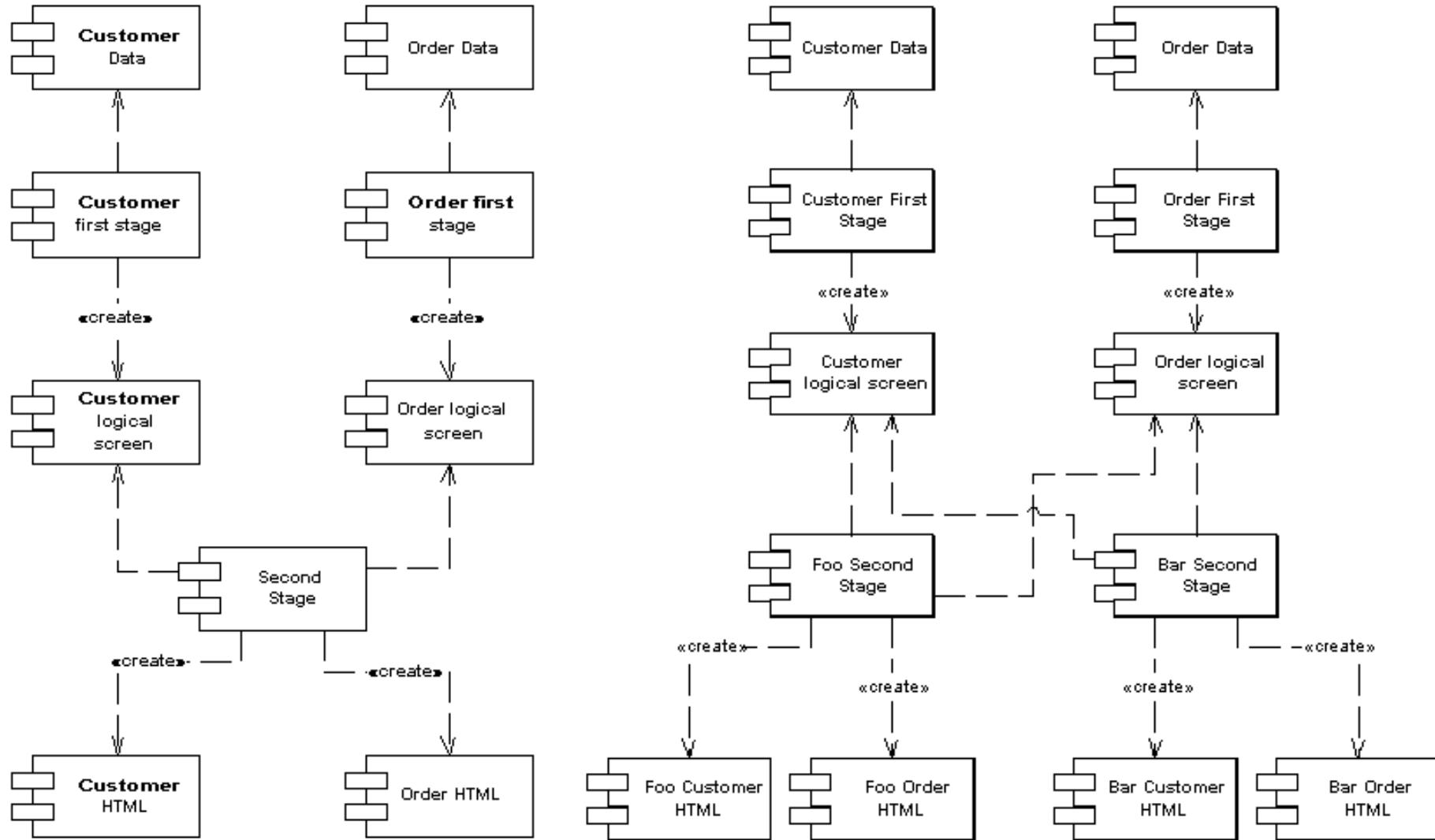
Template View based approach

- The template system converts the logical tags into HTML.

# EXAMPLE



# ONE VS. TWO APPEARANCES



# DISCUSSION

## Advantages:

- Two step view solves the difficulty with Transform view w.r.t. **multiple transforms** module & global changes.
- If the website has **multiple appearances/themes**, the complexity is higher. With two step view, the issue is resolved and the advantage is compounded with multiple pages/themes.

## Liabilities:

- **Hard to find enough commonality** between the screens to get a simple enough presentation-oriented structure
- **Not for designers/non-programmers.** Programmers have to write code for different rendering.
- **Harder** programming model **to learn**
- Complexity increases if **multiple devices** have to be supported.

# NEXT TIME

- Design Patterns
- What about the Midterm exam?
  - Me: post some example questions
  - You: solve at home -> post solutions in Moodle
  - We: discuss the solutions/questions
- When?
  - Next week (no Easter break)
  - After Easter break



# SOFTWARE DESIGN

Creational Design  
Patterns

# CONTENT (OF THE NEXT LECTURES)

## Design Patterns

- Creational Patterns
  - Factory Method
  - Prototype
  - Abstract Factory
- Structural Patterns
  - Adapter
  - Composite
  - Decorator
  - Proxy
  - Bridge
- Behavioral Patterns

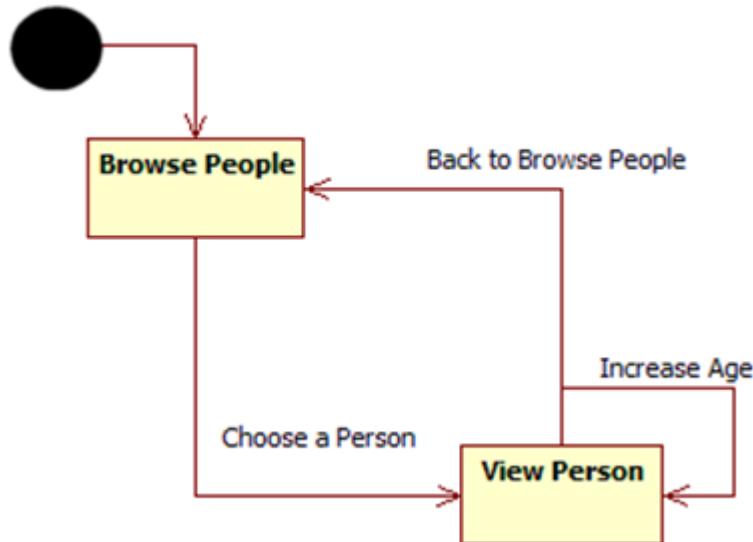
# REFERENCES

- **Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.**
- Univ. of Timisoara Course materials
- Stuart Thiel, Enterprise Application Design Patterns: Improved and Applied, MSc Thesis, Concordia University Montreal, Quebec, Canada [Thiel]

# RECAP LAYERED SOLUTIONS [THIEL]

People management web app

- Browse people
- View person
- Change person data (i.e. increase age)

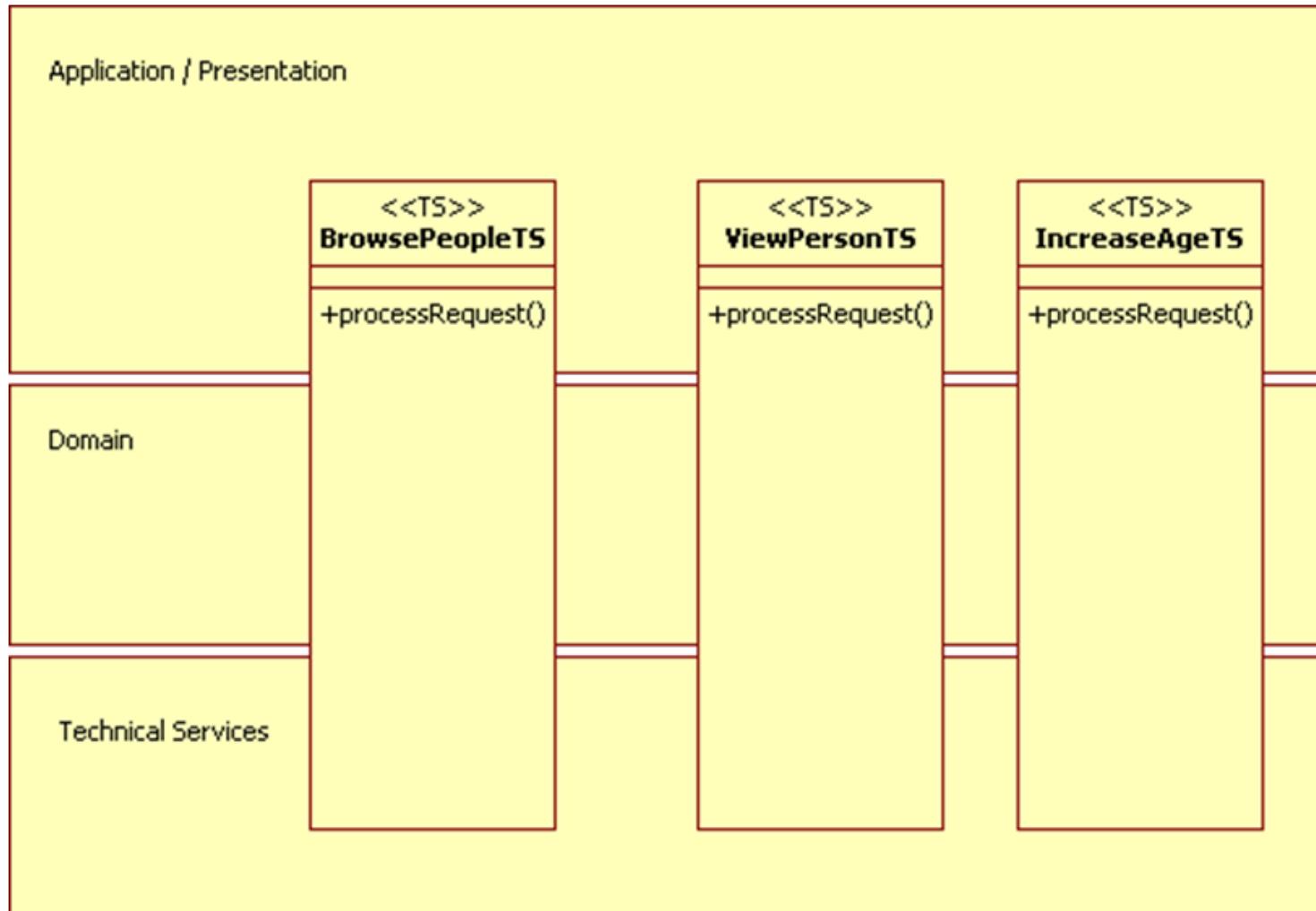


Please choose a person to see their age

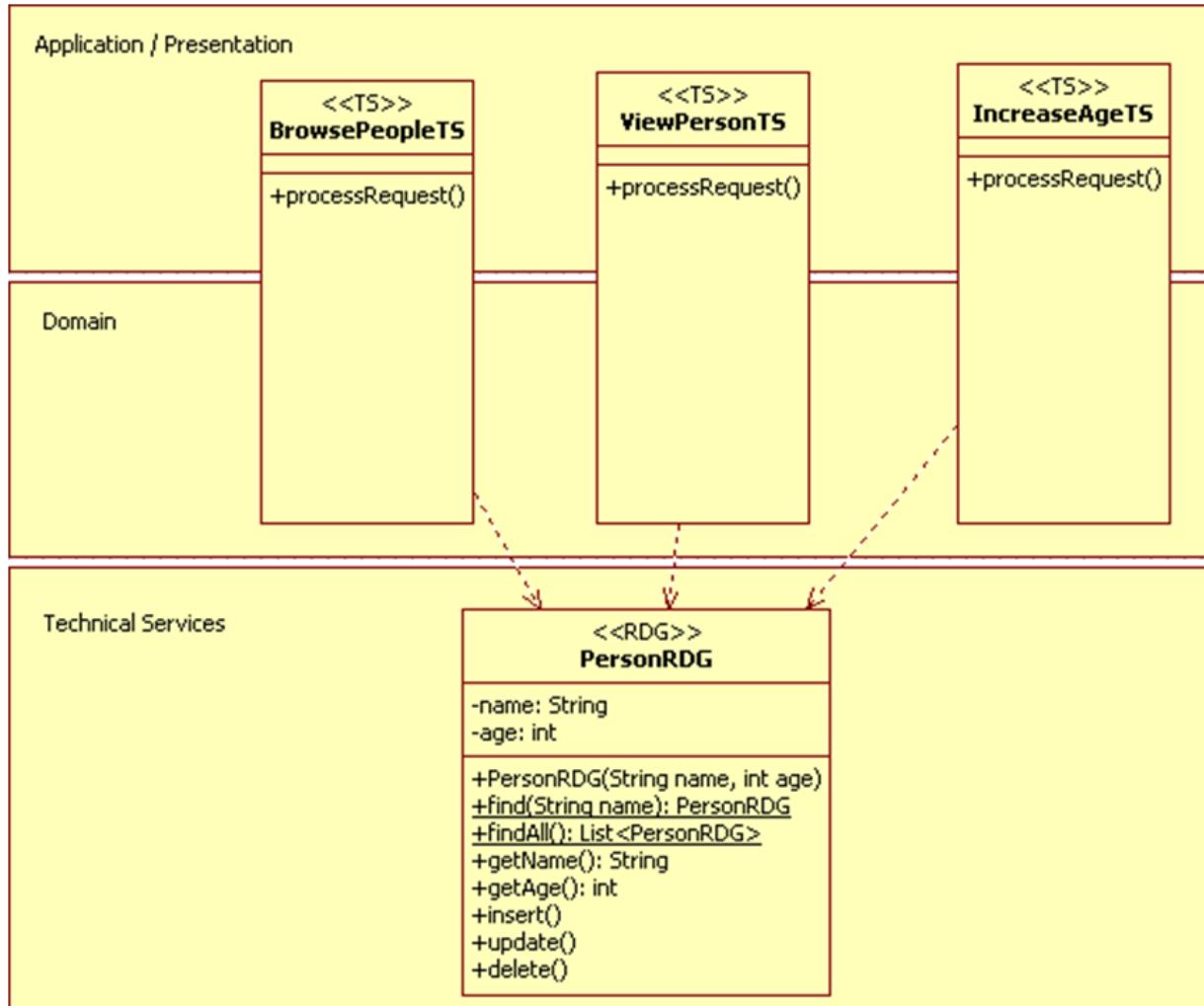
Alice  
 Bob  
 Chuck  
 Dave  
 Edith

**Choose This Person!**

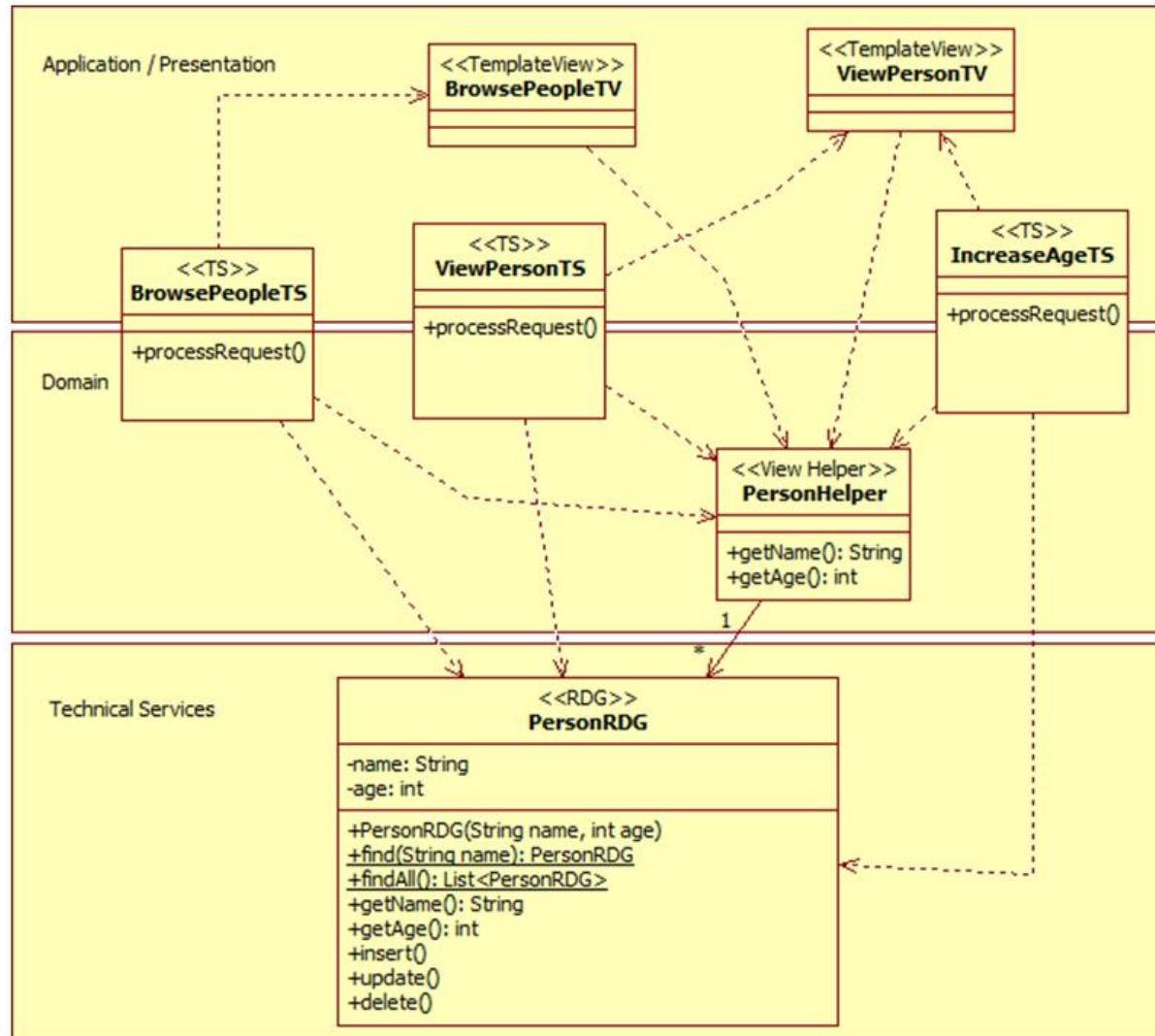
# JUST TRANSACTION SCRIPT (TS)



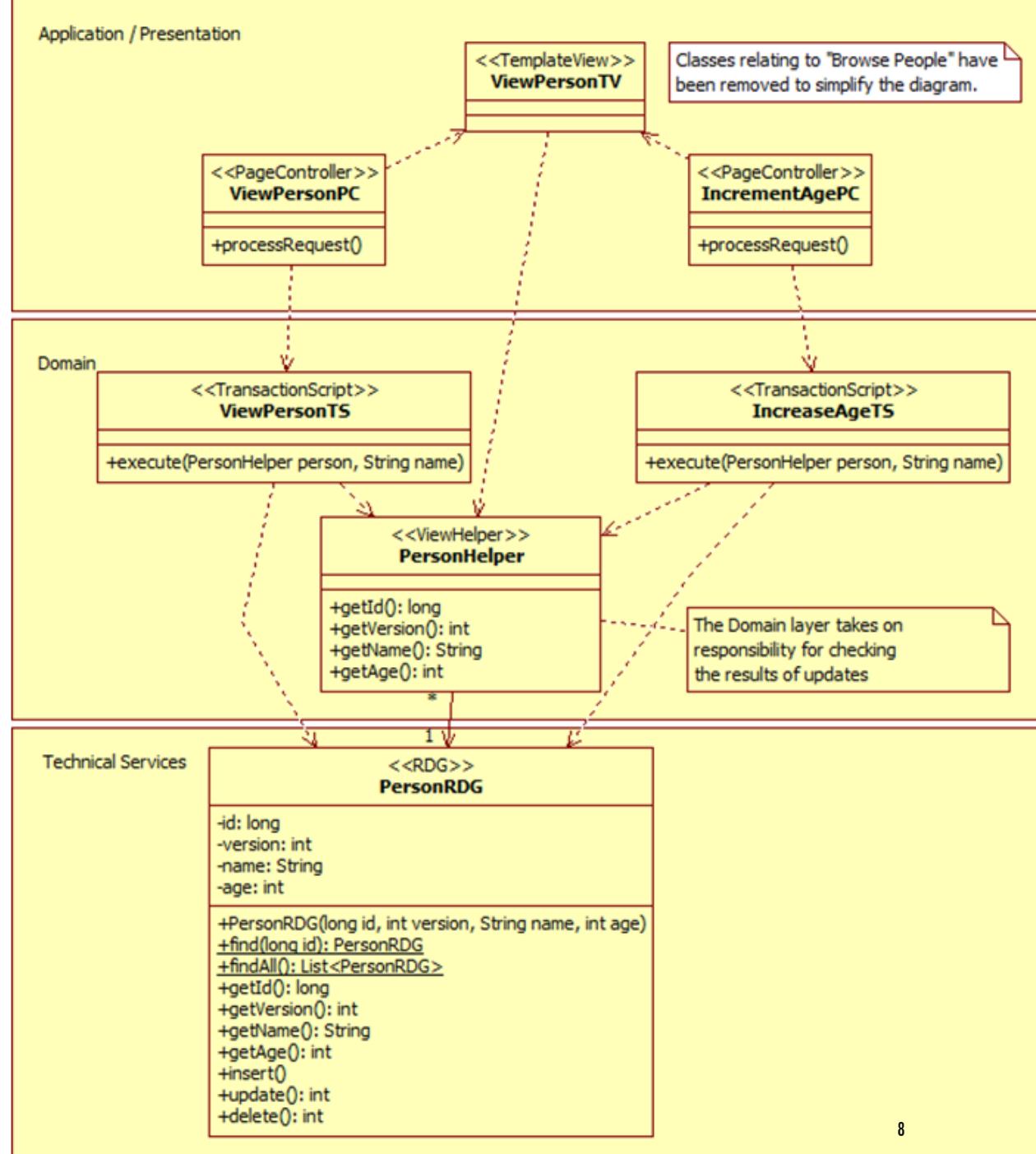
# TS + RDG (DAO)



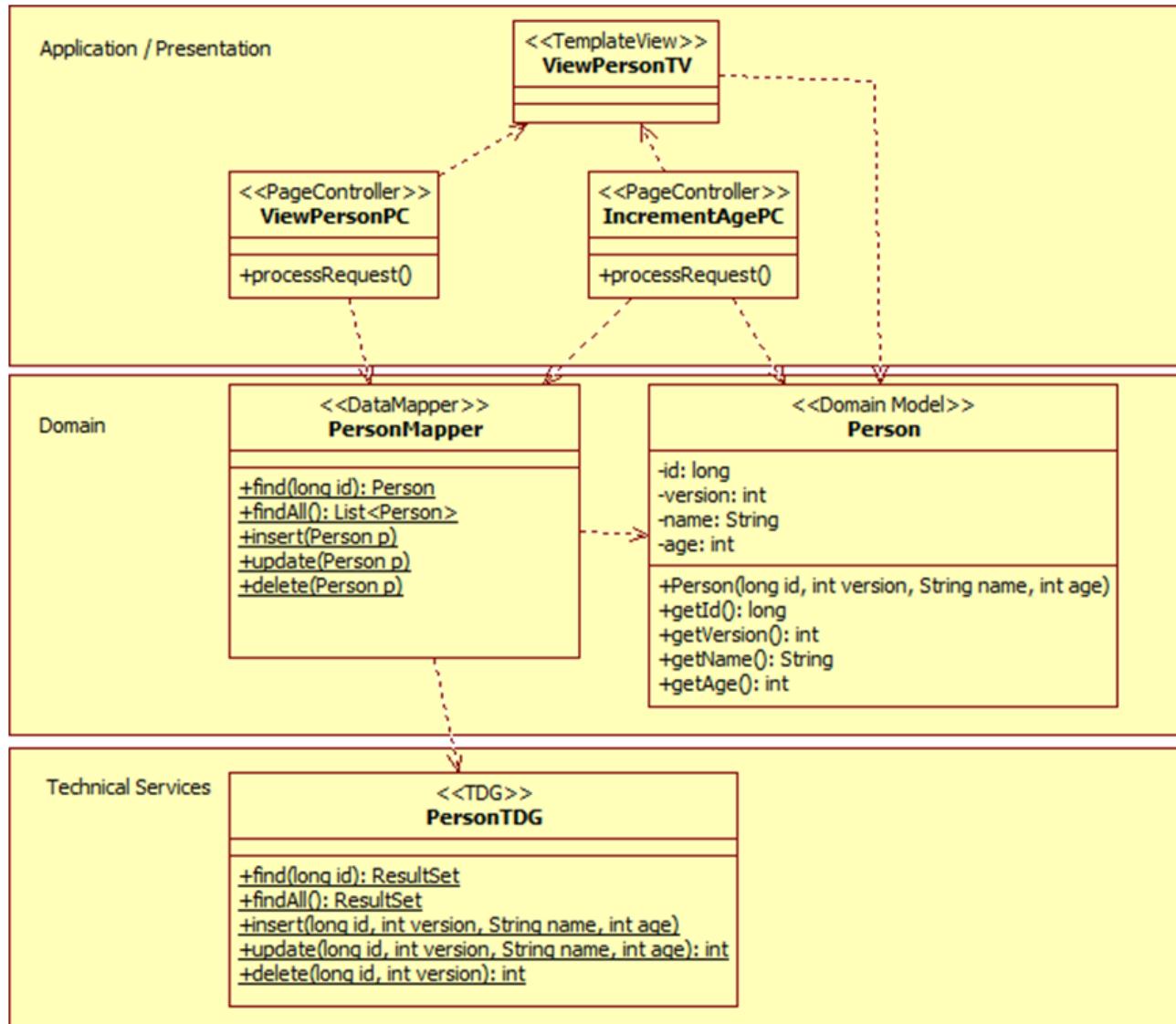
# TS + RDG + TV



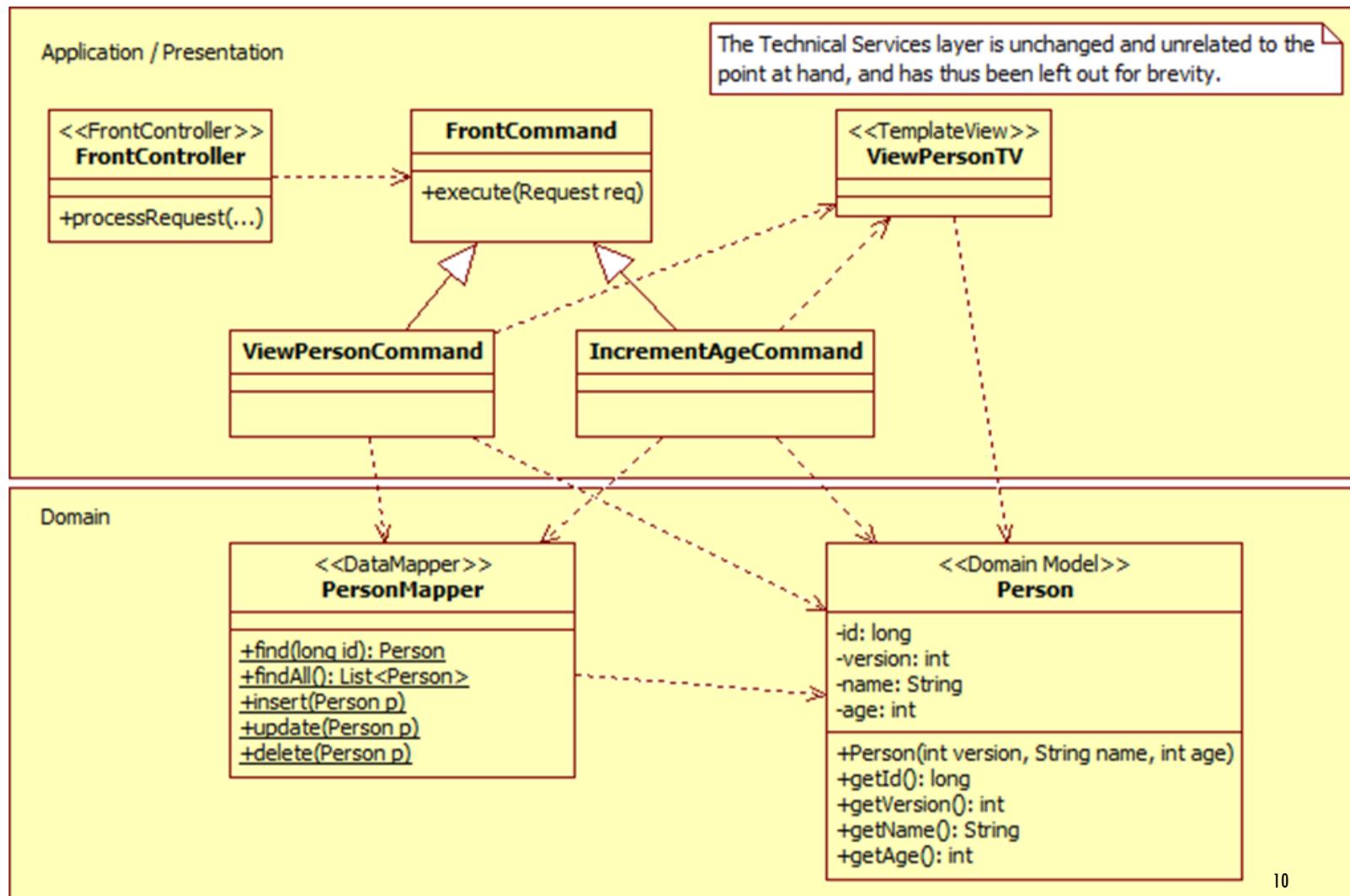
# TS + RDG + TV + PC



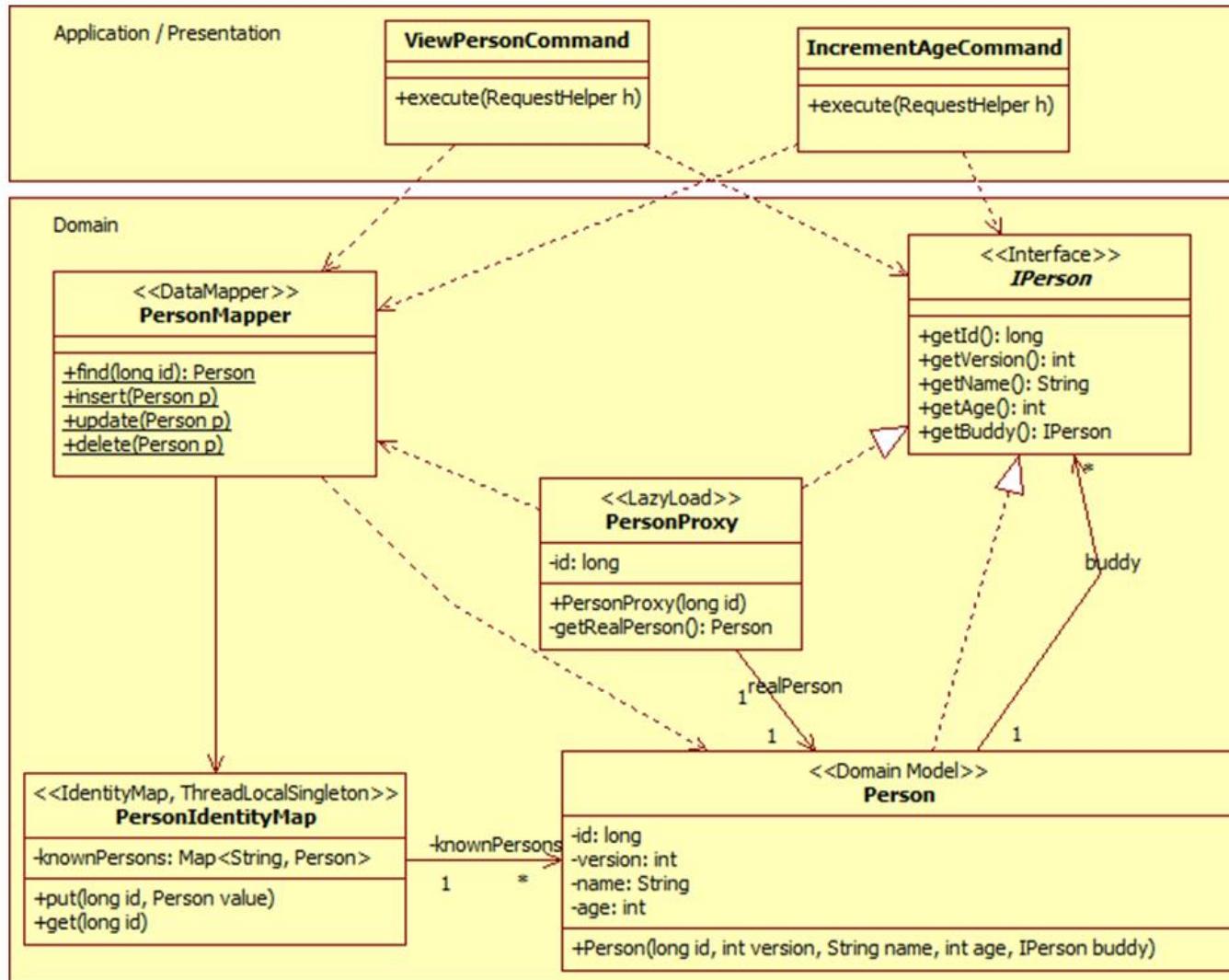
# DM+TDG+DMAPPER+TV+PC



# ...OR DM+...+FC



# ... + LAZY LOAD + IM



# THE WEB

§2.1 100,000 feet  
• Client-server (vs. P2P)

§2.2 50,000 feet  
• HTTP & URIs

§2.3 10,000 feet  
• XHTML & CSS

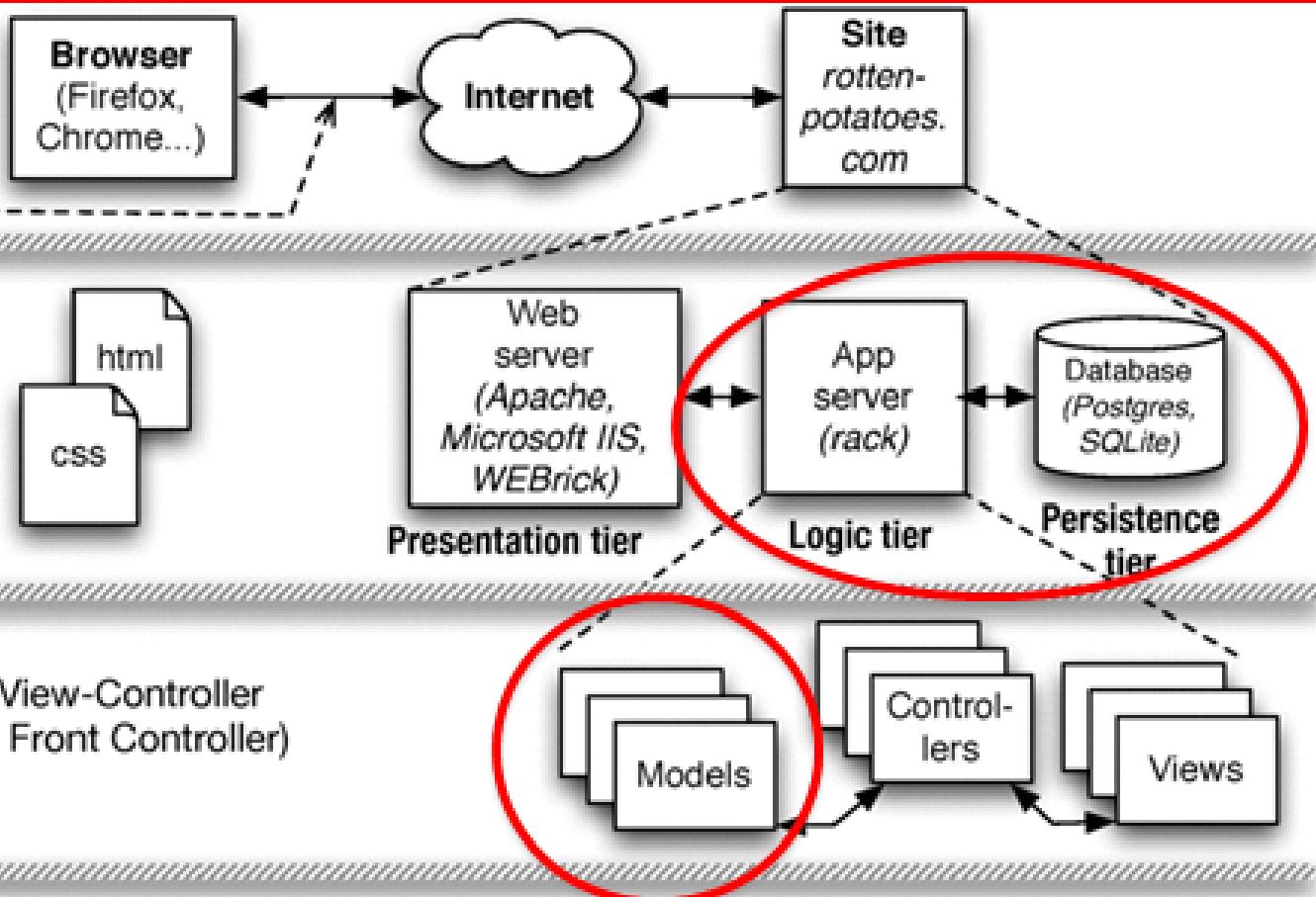
§2.4 5,000 feet  
• 3-tier architecture  
• Horizontal scaling

§2.5 1,000 feet—Model-View-Controller  
(vs. Page Controller, Front Controller)

§2.6 500 feet: Active Record models (vs. Data Mapper)

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

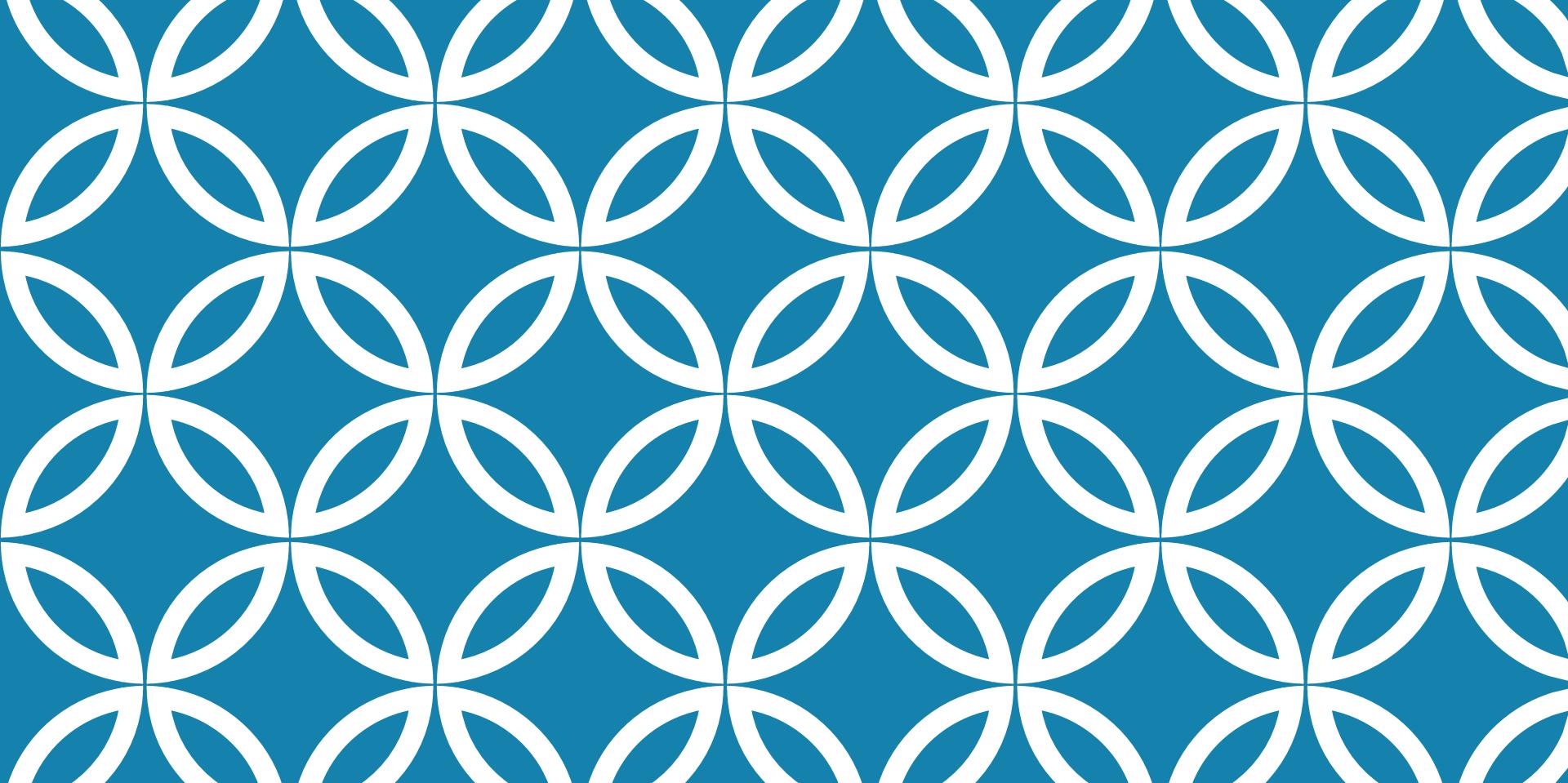
§2.8 500 feet: Template View (vs. Transform View)



- Active Record
- Data Mapper

- REST

- Template View
- Transform View



# DESIGN PATTERNS

# CLASSIFICATION OF DESIGN PATTERNS

## Creational Patterns

- deal with initializing and configuring classes and objects
- *how am I going to create my objects?*

## Structural Patterns

- deal with decoupling the interface and implementation of classes and objects
- *how classes and objects are composed to build larger structures*

## Behavioral Patterns

- deal with dynamic interactions among societies of classes and objects
- *how to manage complex control flows (communications)*

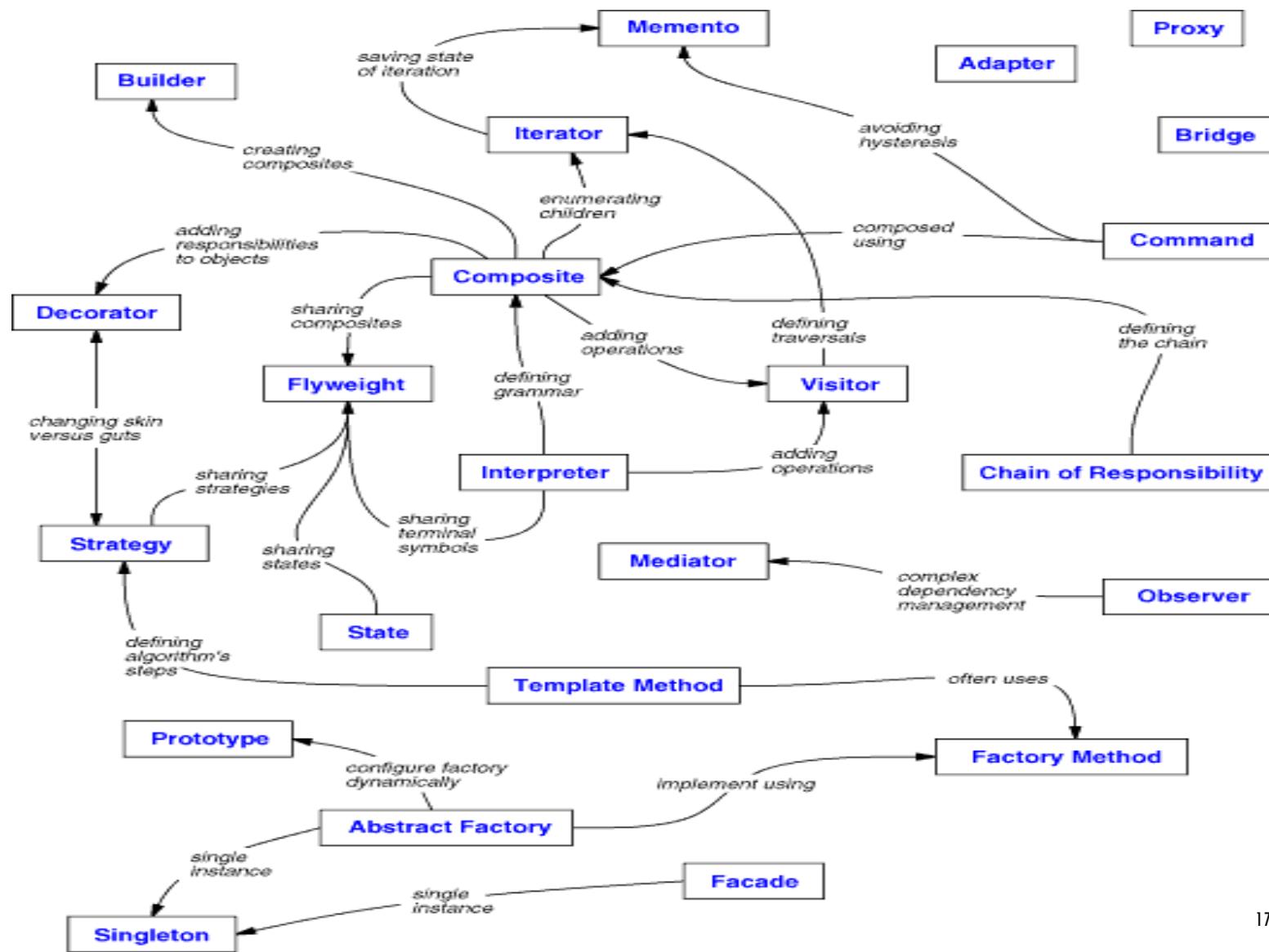
# DRAWBACKS OF DESIGN PATTERNS

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from patterns overload
- Integrating patterns into a software development process is a human-intensive activity

# DESIGN PATTERN SPACE

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Singleton Prototype	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Command Chain of Responsibility Strategy Visitor Iterator Mediator Memento Observer State

# DP RELATIONSHIPS



# CREATIONAL DP

- Abstract instantiation process
- System independent of how objects are created, composed, and represented
- **Class** creational patterns use inheritance to vary class instantiation
- **Object** creational patterns delegate instantiation to another object
- Focus on defining small behaviors and combining into more complex ones

# RECURRING THEMES

- Encapsulate knowledge about which concrete classes the system uses
- Hide how instances of these classes are created and put together

# ADVANTAGES

- Flexibility in what gets created
  - Who created it
  - How it was created and when
- Configure system with “product” objects that vary in structure and functionality
- Configuration can be static or dynamic
- Create standard or uniform structure

# LET'S START SIMPLE...

**Widget**

widgMethod1()  
widgMethod2()  
etc.

**ApplicationClass**

appMethod1()  
appMethod2()  
etc.

```
class ApplicationClass {  
    public appMethod1() { ...  
        Widget w = new Widget(); ...  
    } ... }
```

We can modify the internal **Widget** code without modifying **ApplicationClass**

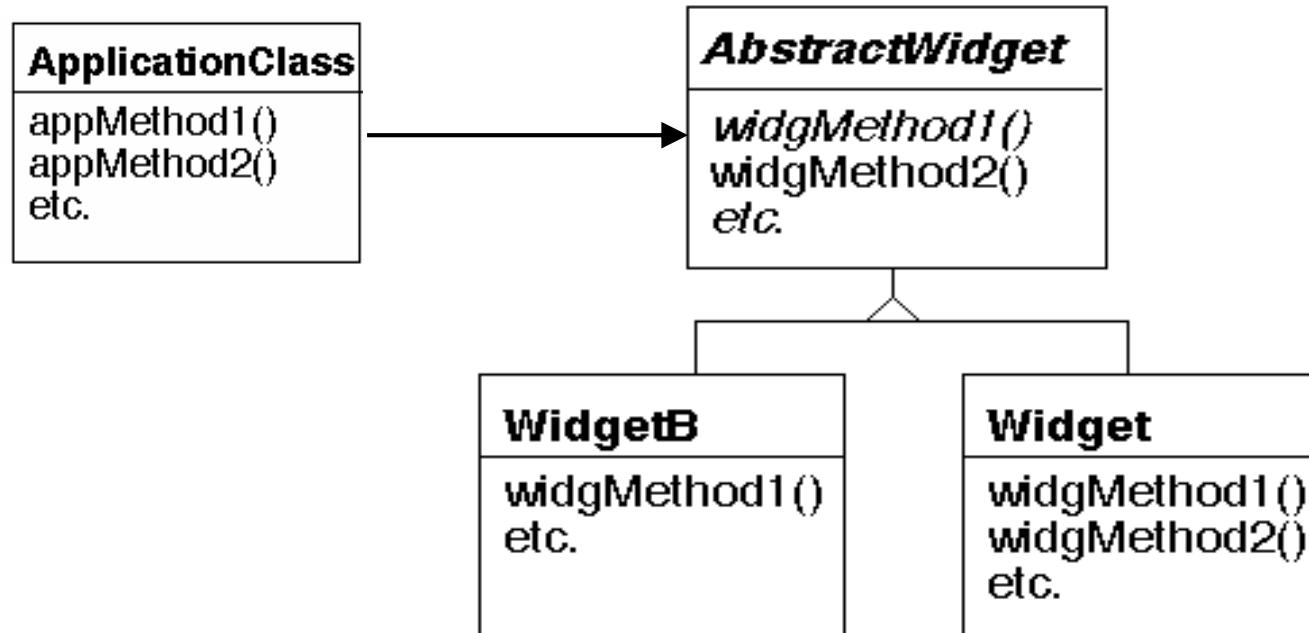
# PROBLEMS WITH CHANGES

What happens when we discover a new **widget** and would like to use in the **ApplicationClass**?

Multiple coupling between **Widget** and **ApplicationClass**

- **ApplicationClass** knows the interface of **Widget**
- **ApplicationClass** explicitly uses the **Widget** type
- hard to change because **Widget** is a concrete class
- **ApplicationClass** explicitly creates new Widgets in many places
- if we want to use the new **Widget** instead of the initial one, changes are spread all over the code

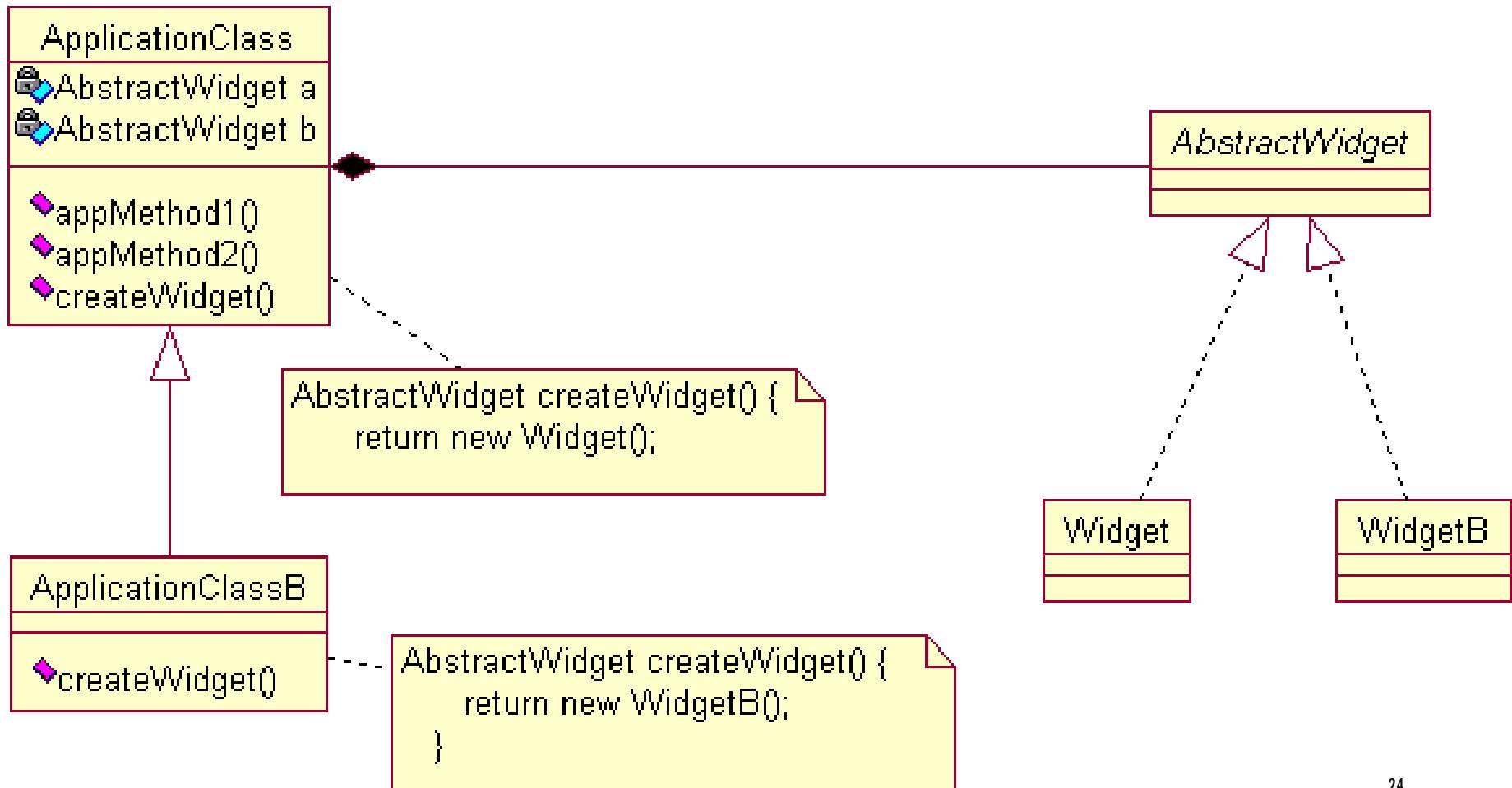
# APPLY “DEPEND ON ABSTRACTIONS”



**ApplicationClass** depends now on an (abstract) interface

But we still have hard coded which widget to create

# USE A FACTORY METHOD



# EVALUATION OF FACTORY METHOD SOLUTION

Explicit creation of **Widget** objects is not anymore dispersed

- easier to change

Functional methods in **ApplicationClass** are decoupled from various concrete implementations of widgets

Avoid ugly code duplication in **ApplicationClassB**

- subclasses reuse the functional methods, just implementing the concrete *Factory Method* needed

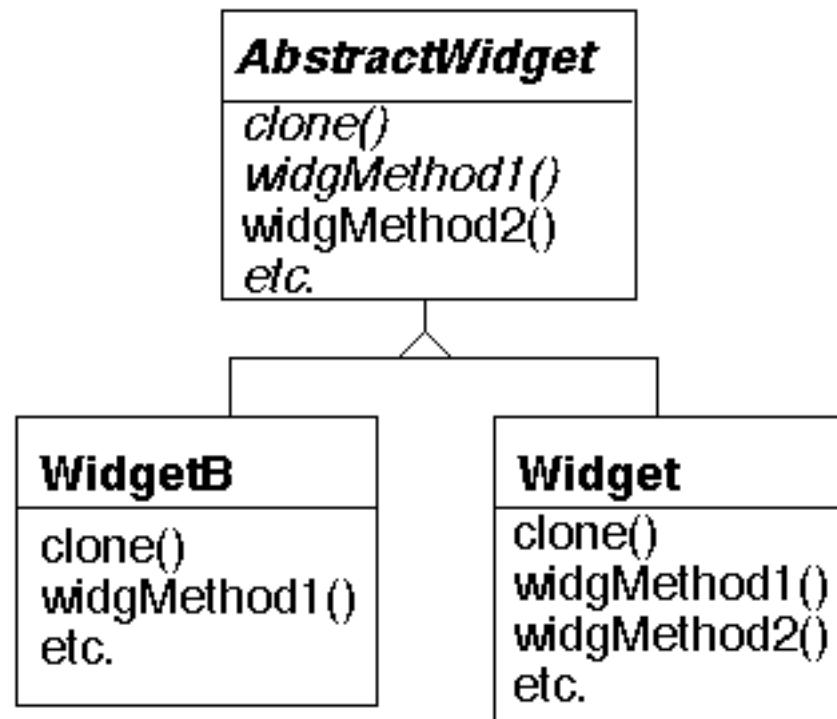
## Disadvantages

- create a subclass only to override the factory-method
- can't change the **Widget** at run-time

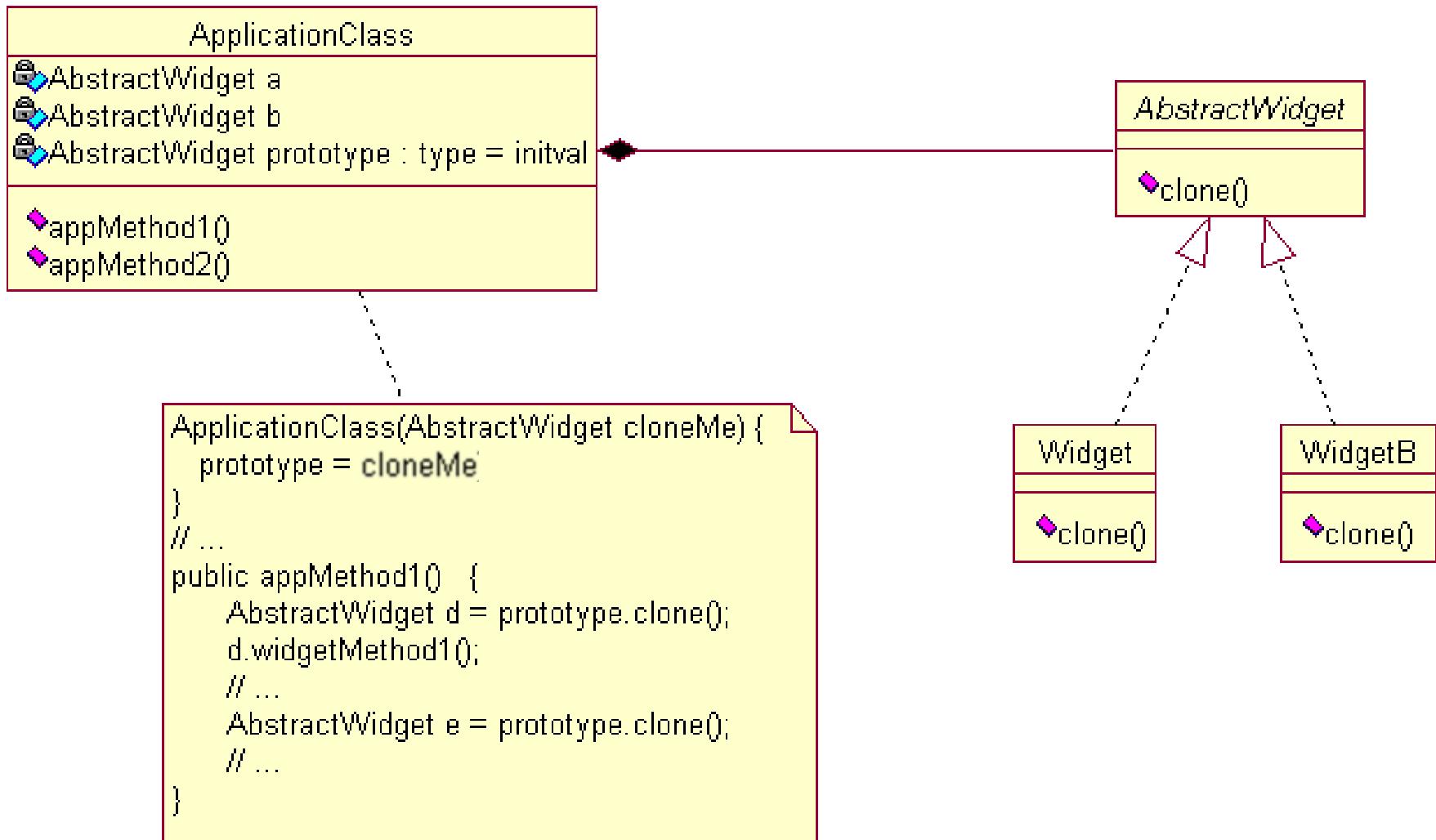
# SOLUTION 2: CLONE A PROTOTYPE

Provide the **Widgets** with a clone method

- make a copy of an existing Widget object



# USING THE CLONE



# ADVANTAGES

Classes to instantiate may be specified dynamically

- client can install and remove prototypes at run-time

We avoided subclassing of **ApplicationClass**

- Remember: *Favor Composition over Inheritance!*

Totally hides concrete product classes from clients

- Reduces implementation dependencies

# MORE CHANGES

What if **ApplicationClass** uses other "products" too...

- e.g. Wheels, etc.

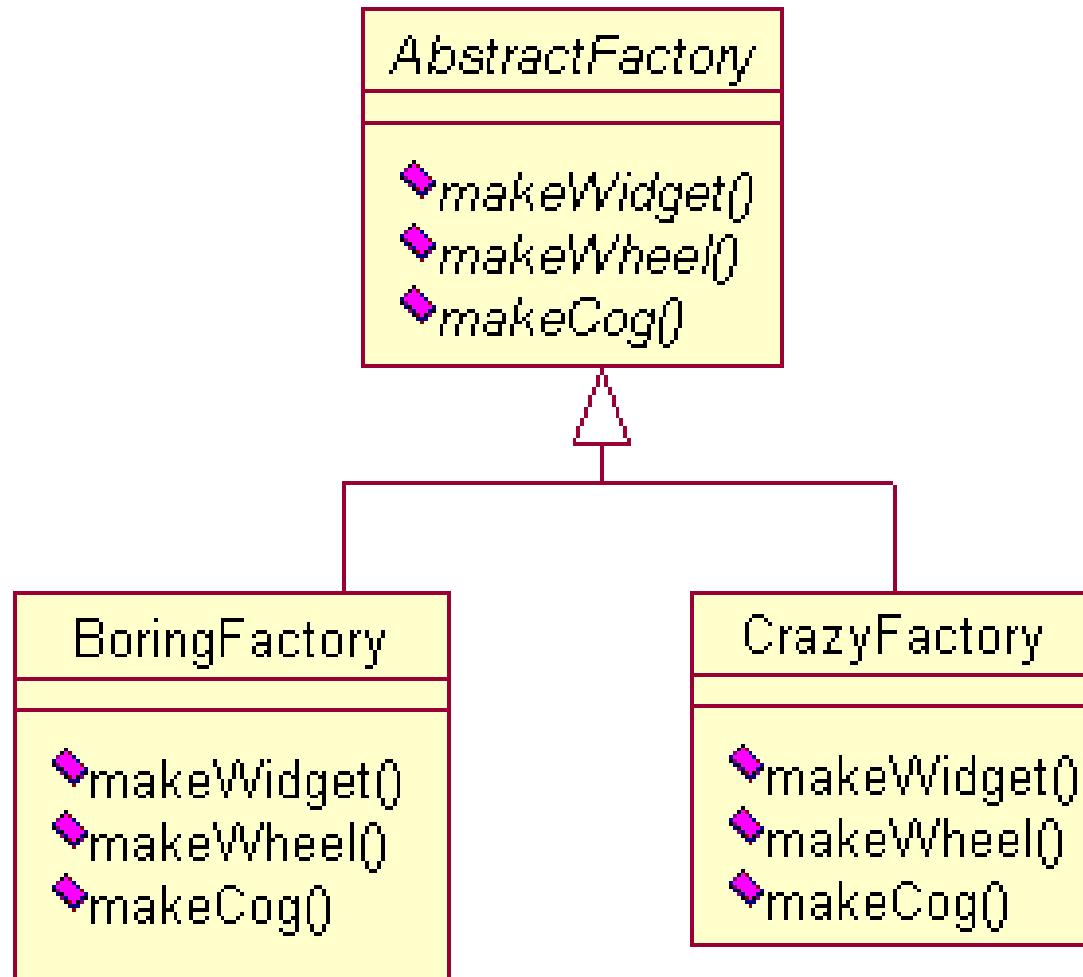
Each one of these stays for an object family

- i.e. all of these have subclasses

Assume that there are restrictions on what type of Widget can be used with which type of Wheel or Cog

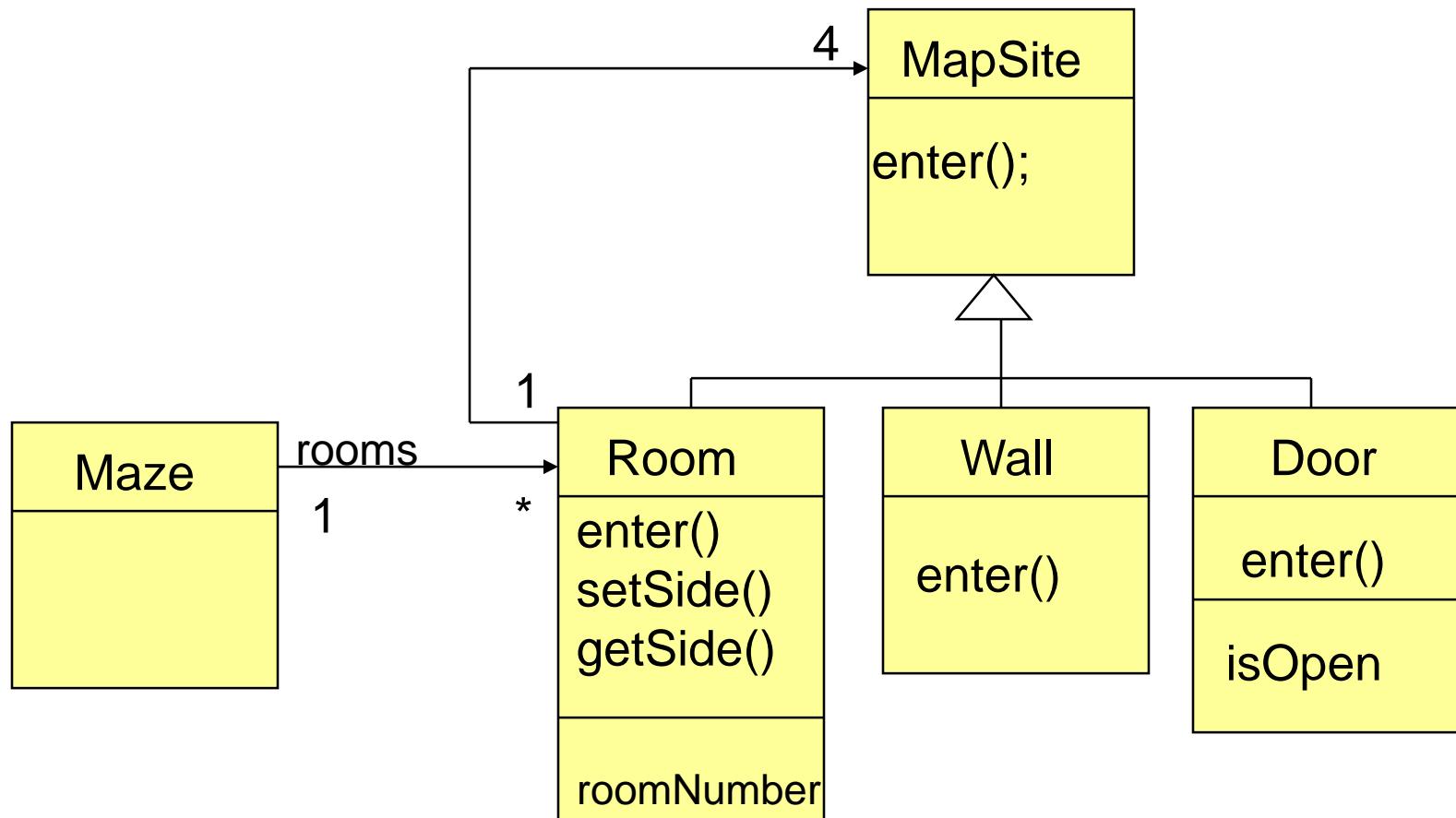
Factory Methods or Prototypes can handle each type of product but it gets hard to insure the wrong types of items are not used together

# SOLUTION: CREATE AN ABSTRACT FACTORY



# CREATIONAL DP IN ACTION

## Maze Game



# COMMON ABSTRACT CLASS FOR ALL MAZE COMPONENTS

```
public enum Direction {North, South, East, West};  
  
class MapSite {  
    public abstract void enter();  
};
```

Meaning of enter() depends on *what* you are entering.

- room → location changes
- door → if door is open, go in

# COMPONENTS OF THE MAZE – MAZE

```
class Maze {  
    public void addRoom(Room r) {...};  
    Room getRoom(int) {...};  
};
```

A maze is a collection of rooms. Maze can find a particular room given the room number.

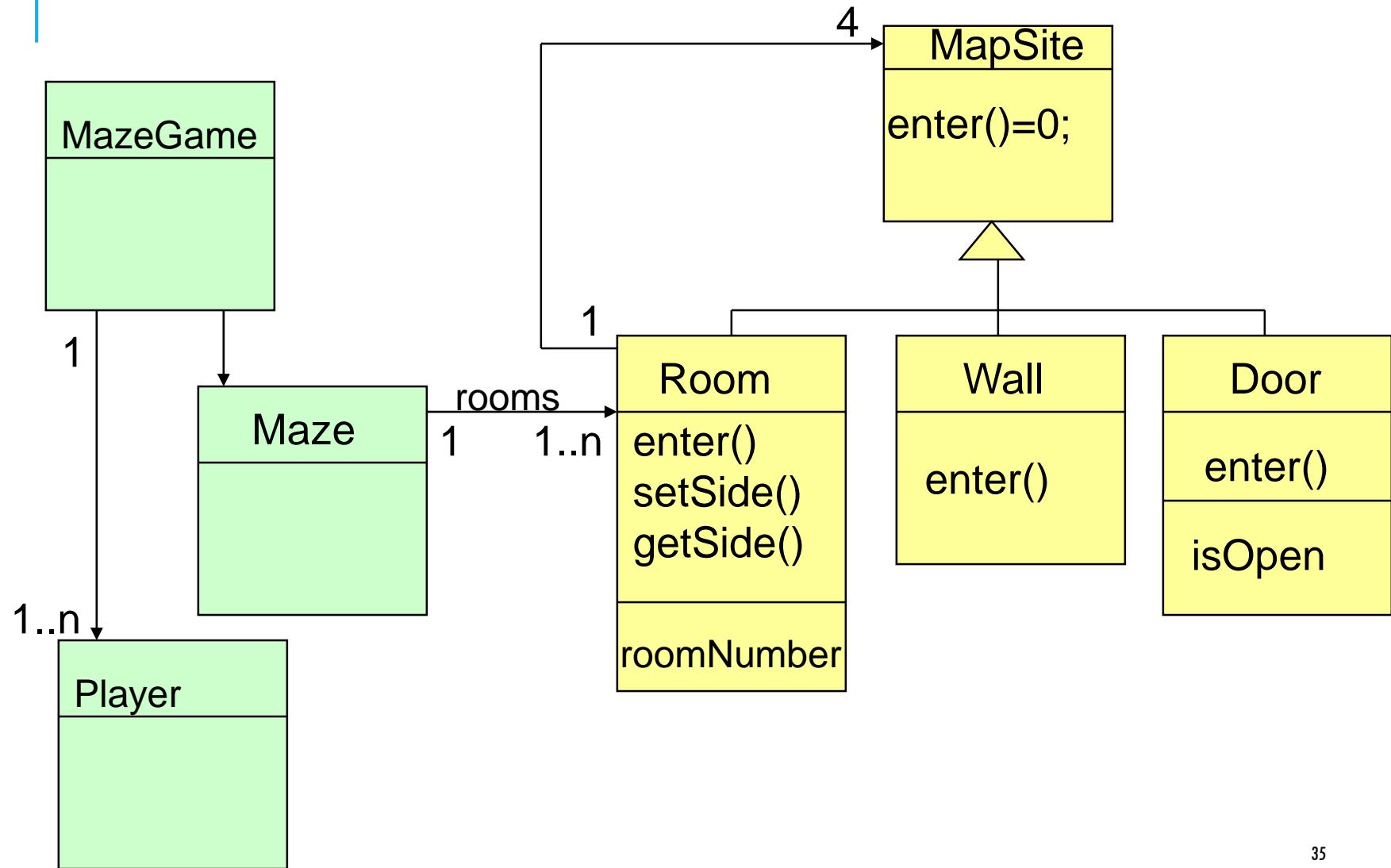
roomNo() could do a lookup using a linear search or a hash table or a simple array.

# COMPONENTS OF THE MAZE – WALL & DOOR & ROOM

```
public class Room extends MapSite {  
    int roomNumber;  
    MapSite sides[4];  
  
    public Room(int roomNo) {...};  
    public MapSite getSide(Direction d) {...};  
    public void setSide(Direction d,  
MapSite m) {...};  
  
    public void enter() {...};  
}
```

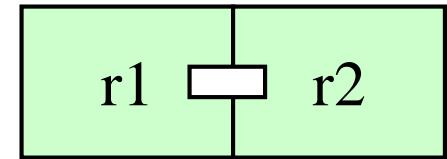
```
public class Wall extends MapSite  
{  
    public Wall();  
    public void enter();  
};  
  
public class Door extends MapSite  
{  
    Room room1, room2;  
    bool isOpen;  
  
    public Door(Room r1, r2){...};  
    public void enter() {...};  
};
```

# WE WANT TO PLAY A GAME!



# CREATING THE MAZE

```
public class MazeGame {  
    public Maze 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());  
    ... }  
}
```



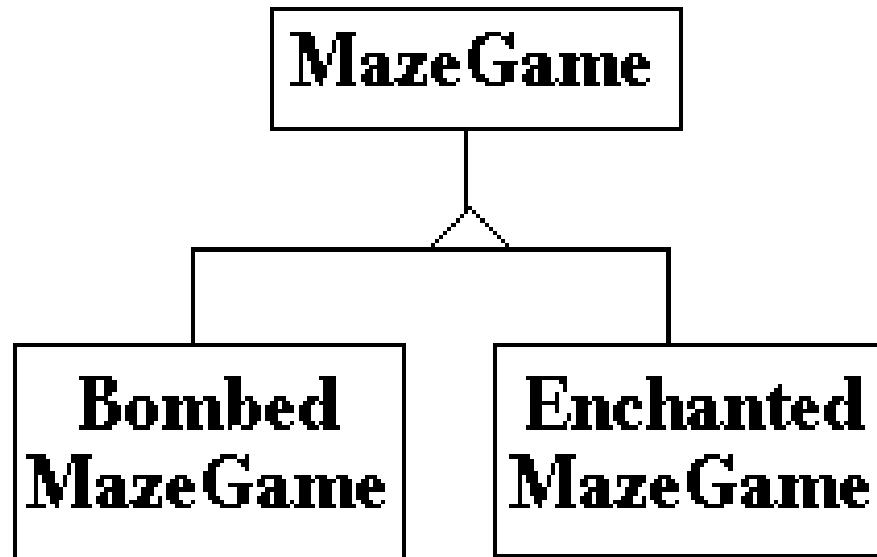
The problem is **inflexibility** due to hard-coding of maze layout

Pattern can make game creation more flexible... *not* smaller!

# WE WANT FLEXIBILITY IN MAZE CREATION

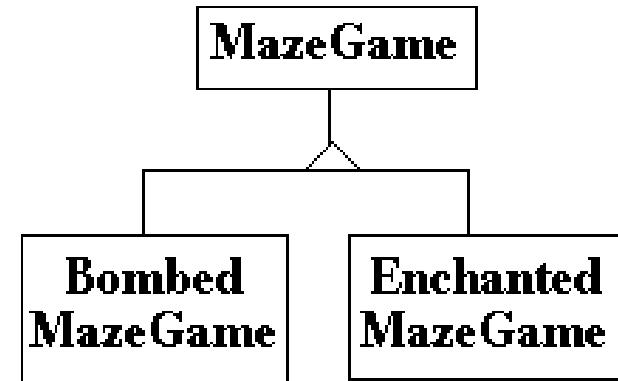
Be able to vary the kinds of mazes

- Rooms with bombs
- Walls that have been bombed
- Enchanted rooms
- Need a spell to enter the door!



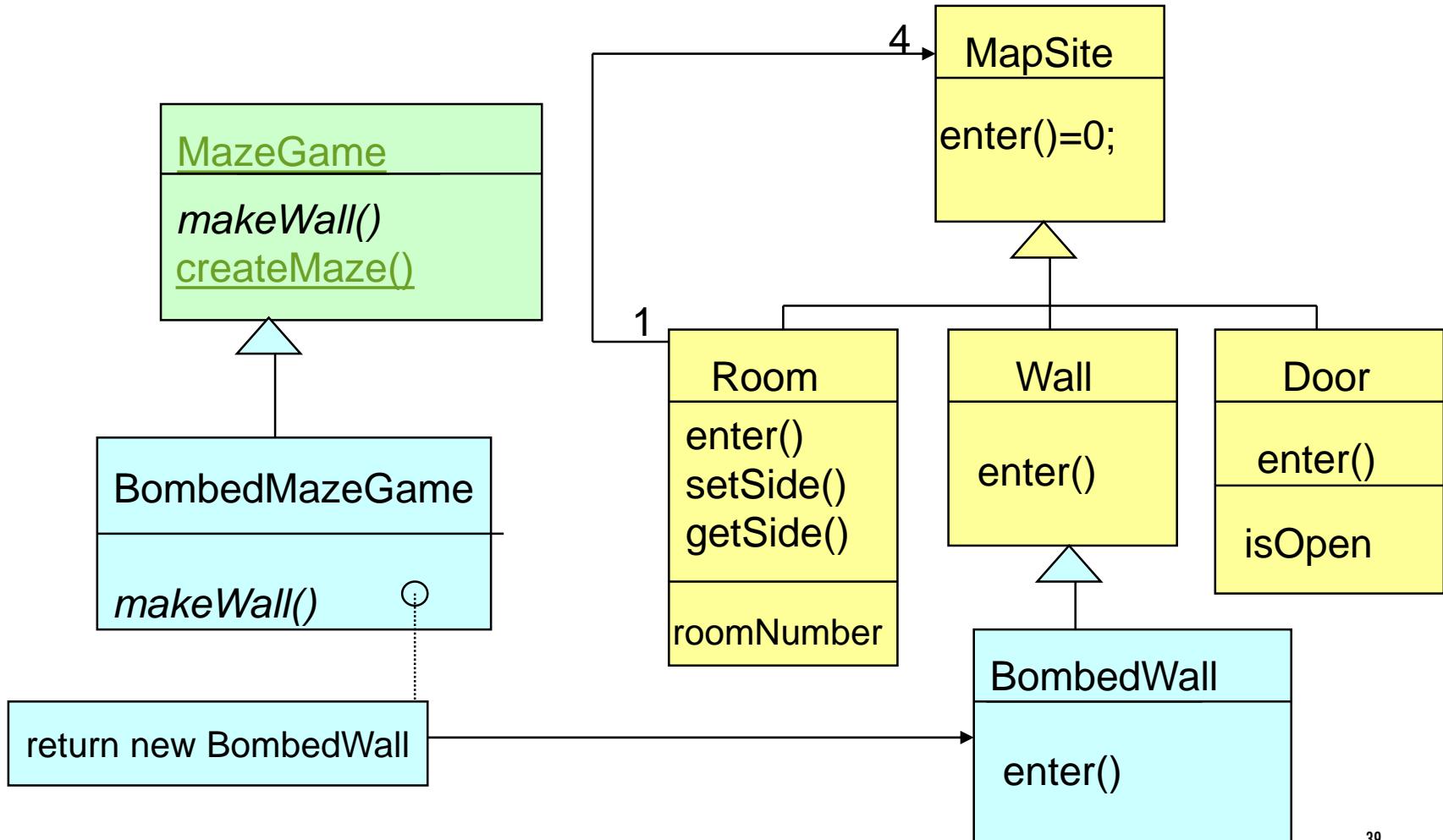
# IDEA 1: SUBCLASS MAZEGAME, OVERRIDE CREATEMAZE

```
public class BombedMazeGame {  
    public Maze createMaze() {  
        Maze aMaze = new Maze;  
        Room r1 = new RoomWithABomb(1);  
        Room r2 = new RoomWithABomb(2);  
        Door theDoor = new Door(r1, r2);  
        aMaze.addRoom(r1);  
        aMaze.addRoom(r2);  
  
        r1.setSide(North, new BombedWall());  
        r1.setSide(East, theDoor);  
        r1.setSide(South, new BombedWall());  
        r1.setSide(West, new BombedWall());  
    }  
}
```



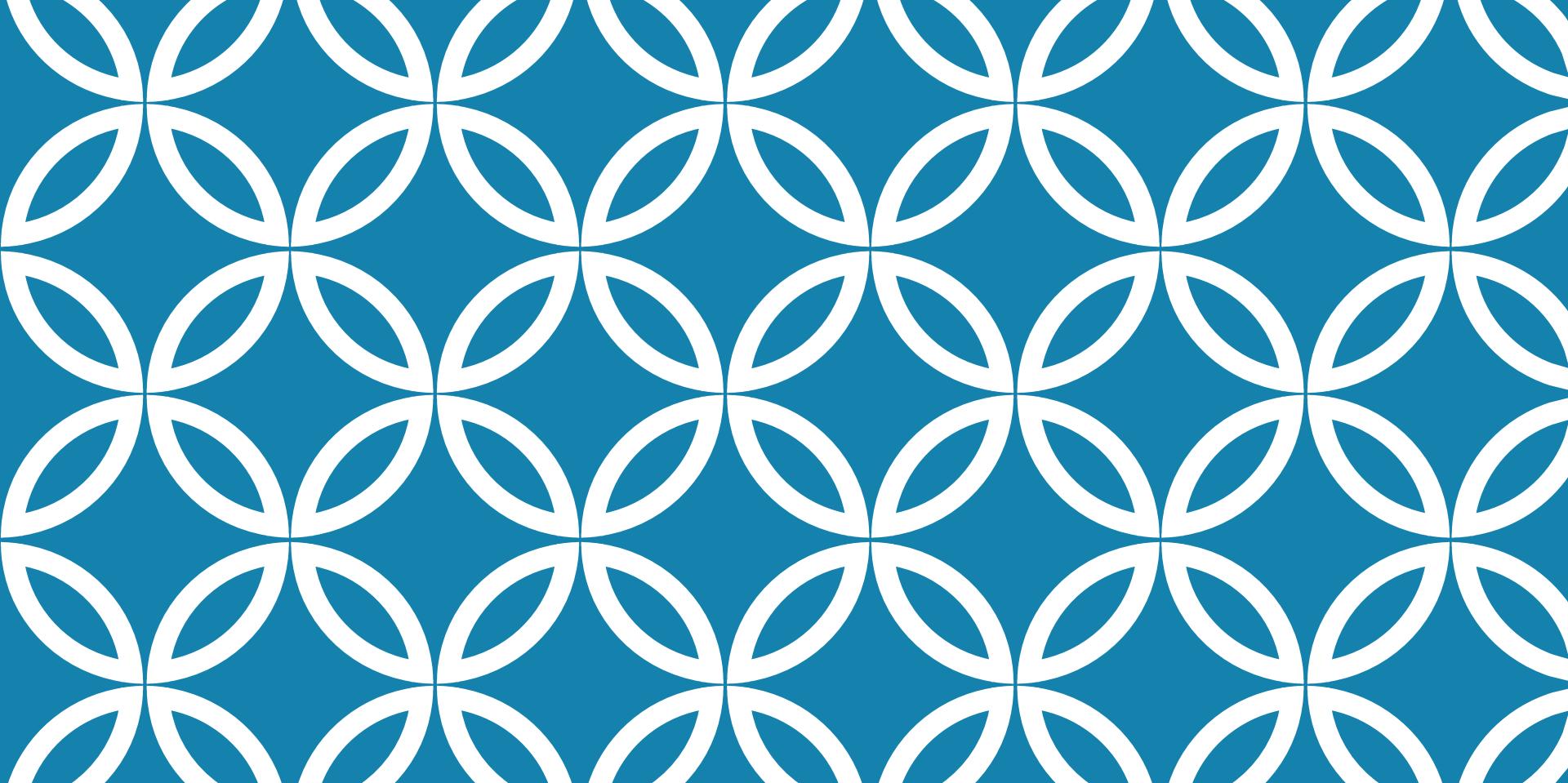
Lots of code duplication... :(

# IDEA 2: USE A FACTORY METHOD



# APPLYING FACTORY METHOD

```
public class MaseGame {  
    public Maze 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;  
    } }
```



# FACTORY METHOD

---

# BASIC ASPECTS

## Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses

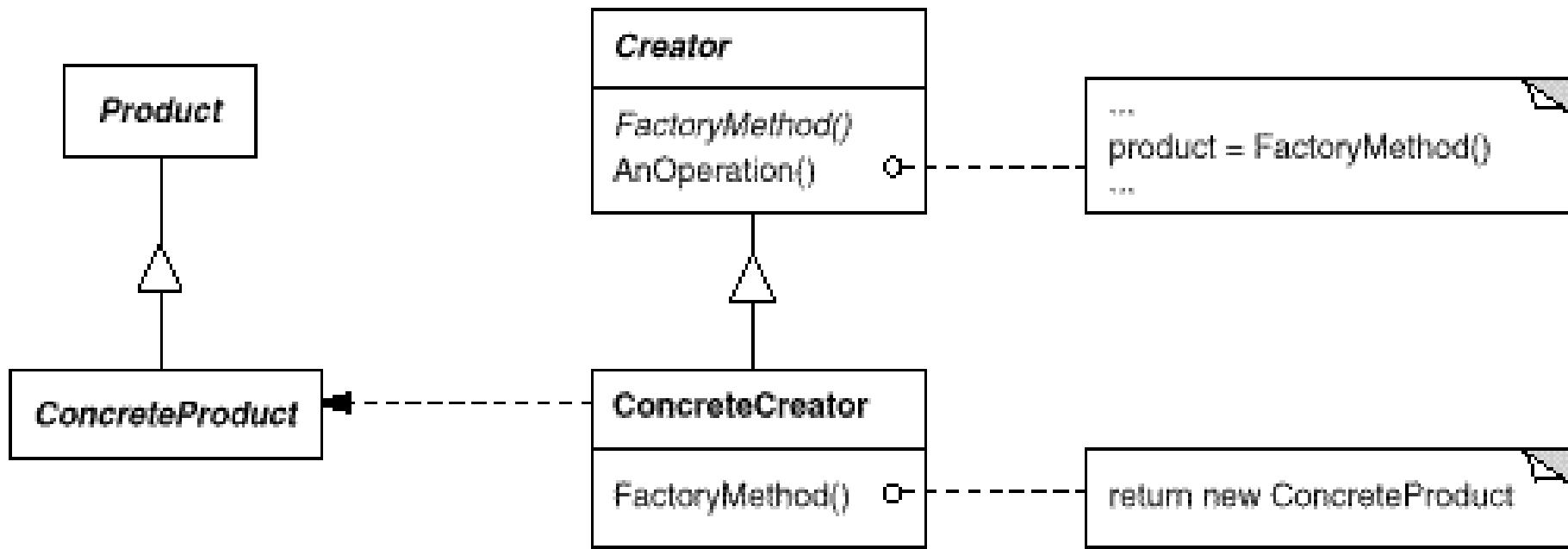
## Also Known As

- Virtual Constructor

## Applicability

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsibility to one of several helper subclasses

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## Product

- defines the interface of objects that will be created by the FM
- Concrete Product implements the interface

## Creator

- declares the FM, which returns a product of type Product.
- may define a default implementation of the FM

## ConcreteCreator

- overrides FM to provide an instance of ConcreteProduct

*Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct*

# CONSEQUENCES

Eliminates binding of application specific classes into your code.

- creational code only deals with the Product interface

Provides hooks for subclassing

- subclasses can change this way the product that is created

Clients might have to subclass the Creator just to create a particular ConcreteProduct object.

# IMPLEMENTATION ISSUES

## Varieties of Factory Methods

- Creator class is **abstract**
  - does not provide an implementation for the FM it declares
  - requires subclasses
- Creator is a **concrete** class
  - provides default implementation
  - FM used for flexibility
  - create objects in a separate operation so that subclasses can override it

## Parametrization of Factory Methods

- A variation on the pattern lets the factory method create multiple kinds of products
- a **parameter** identifies the type of Product to create
- all created objects share the Product interface

# PARAMETERIZING THE FACTORY

```
class Creator {  
    public Product create(productId id)  
    {  
        if (id == MINE) return new MyProduct();  
        if (id == YOURS) return new YourProduct();  
        ...  
    }  
};  
  
Class MyCreator extends Creator {  
    public Product create(productId id) {  
        if (id == MINE) return new YourProduct();  
        if (id == YOURS) return new MyProduct();  
        if (id == THEIRS) return new TheirProduct();  
        return super.create(id); // called if others fail  
    }  
}
```

selectively *extend* or *change* products that get created

# IDEA 3: FACTORY METHOD IN PRODUCT

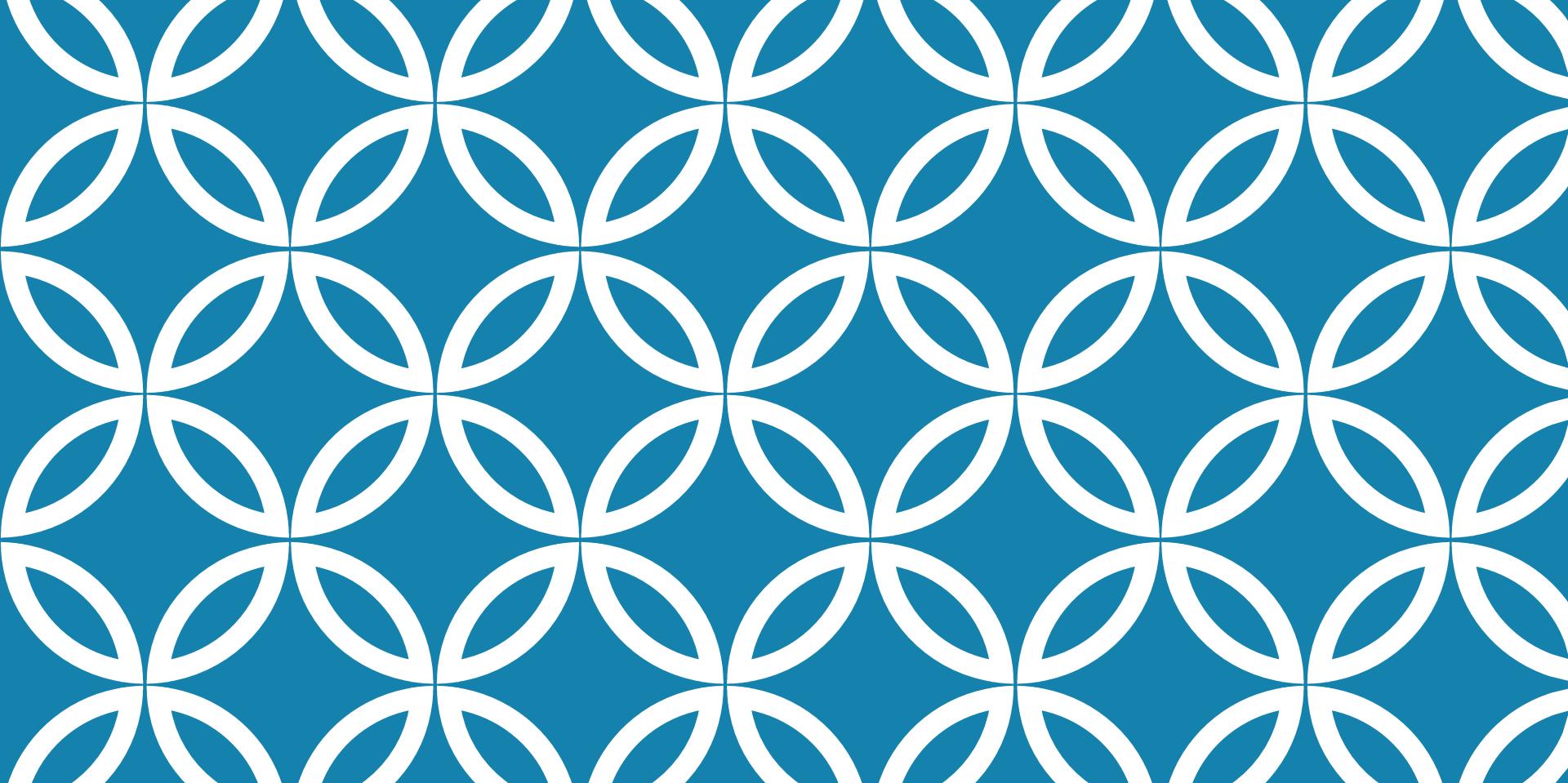
Make the product responsible for creating itself

- e.g. let the Door know how to construct an instance of it rather than the MazeGame

The client of the product needs a reference to the "creator"

- specified in the constructor

```
class Room extends MapSite {  
    public Room makeRoom(int no) {  
        return new Room(no);  
    }  
    // ...;  
  
class RoomWithBomb extends Room {  
    public Room makeRoom(int no) {  
        return new RoomWithBomb();  
    }  
    // ...;  
  
class MazeGame {  
    private Room roomMaker;  
    // ...  
    public MazeGame(Room rfactory) {  
        roomMaker = rfactory;  
    }  
    public Maze createMaze() {  
        Maze aMaze = new  
Maze();  
        Room r1 = roomMaker.makeRoom(1);  
        // ...};
```



# THE PROTOTYPE PATTERN

# BASIC ASPECTS

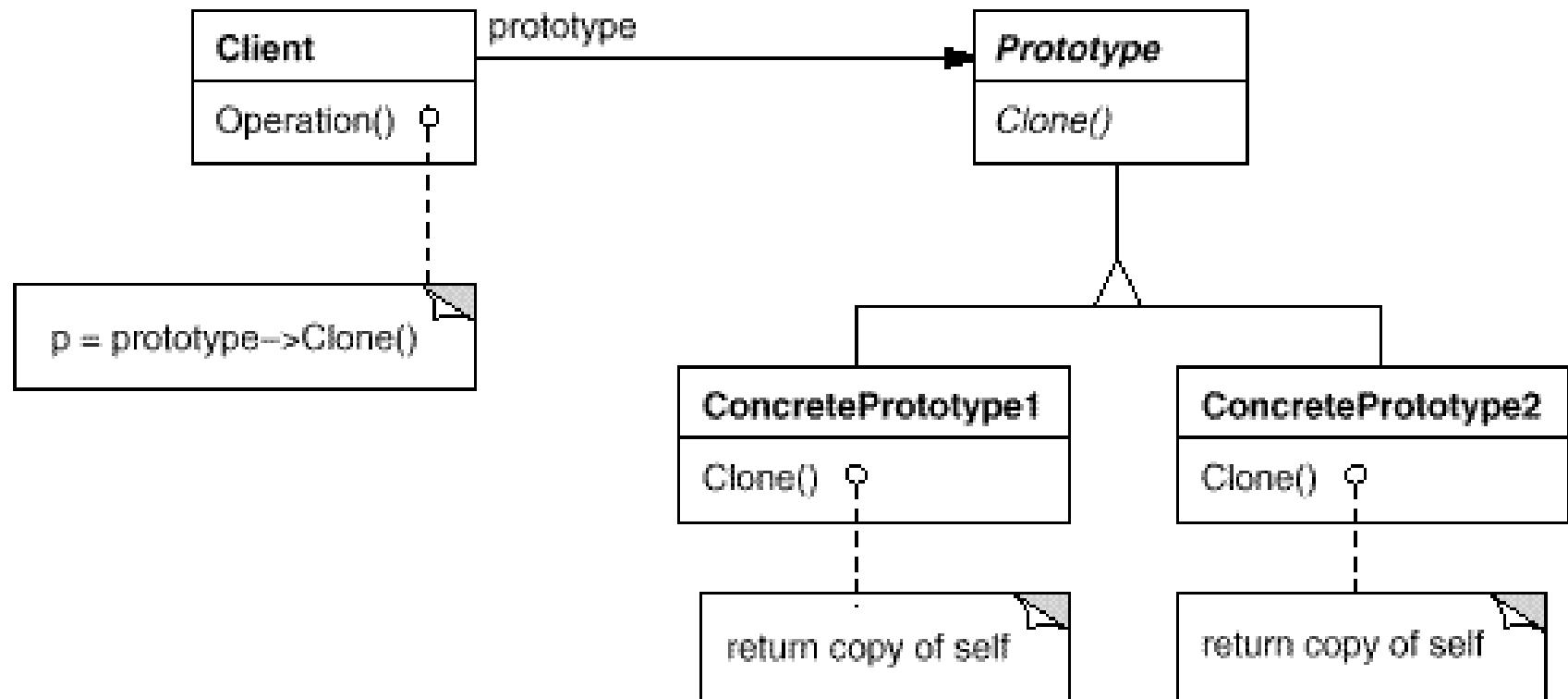
## Intent

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

## Applicability

- when a client class should be independent of how its products are created, composed, and represented **and**
- when the classes to instantiate are *specified at run-time*

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## Prototype

- declares an interface for cloning itself.

## ConcretePrototype

- implements an operation for cloning itself.

## Client

- creates a new object by asking a prototype to clone itself.

*A client asks a prototype to clone itself.*

*The client class must initialize itself in the constructor with the proper concrete prototype.*

# CONSEQUENCES

Adding and removing products at run-time

Reduced subclassing

- avoid parallel hierarchy for creators

Specifying new objects by varying **values of prototypes**

- client exhibits new behavior by delegation to prototype

Each subclass of Prototype must implement **clone**

- difficult when classes already exist or
- internal objects don't support copying or have circular references

# IMPLEMENTATION ISSUES

## Using a Prototype manager

- number of prototypes isn't fixed
  - keep a registry → **prototype manager**
- clients instead of knowing the prototype know a manager

## Initializing clones

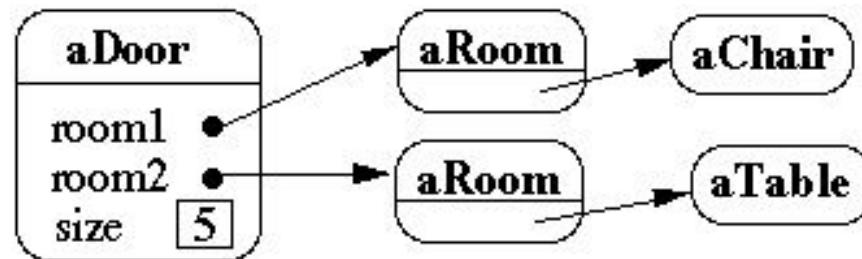
- heterogeneity of initialization methods
- write an **Initialize** method

## Implementing the **clone** operation

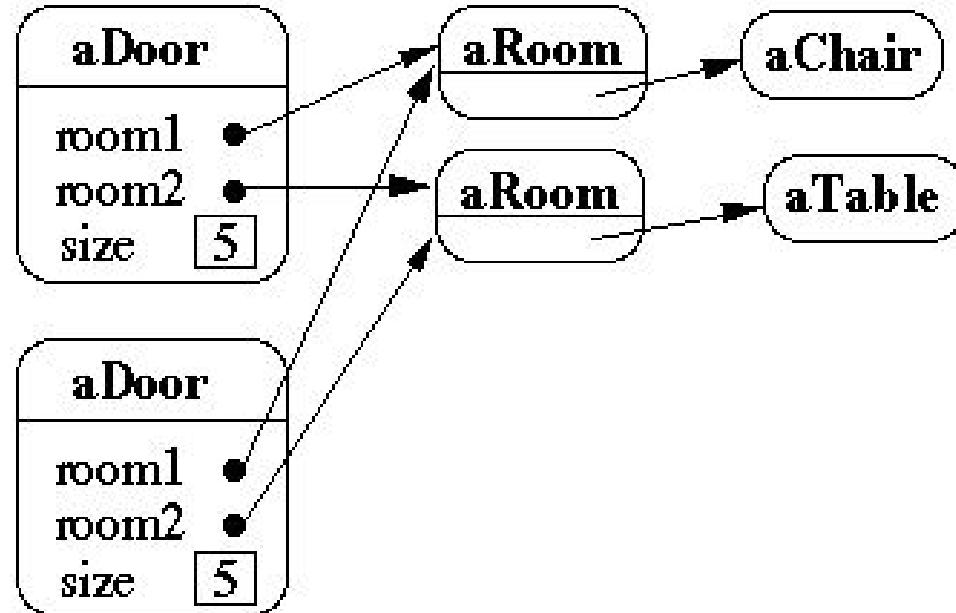
- shallow vs. deep copy

# SHALLOW COPY VS. DEEP COPY

Original

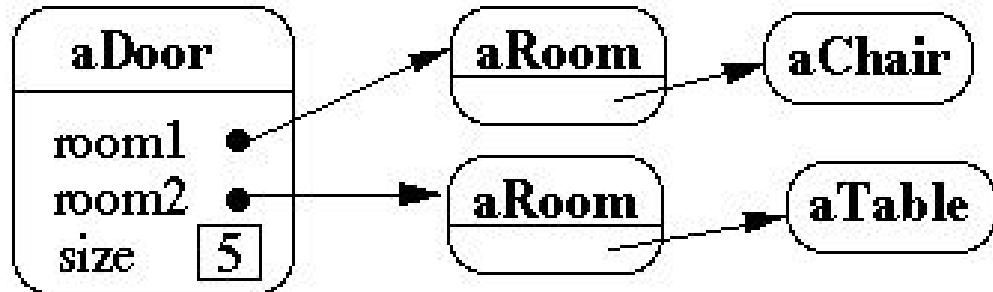


Shallow Copy

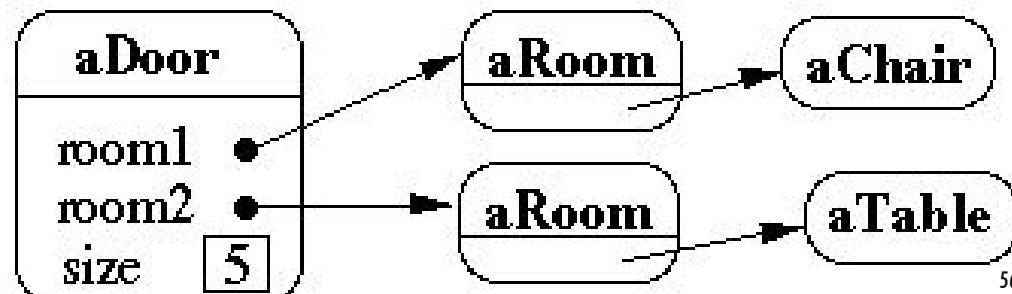
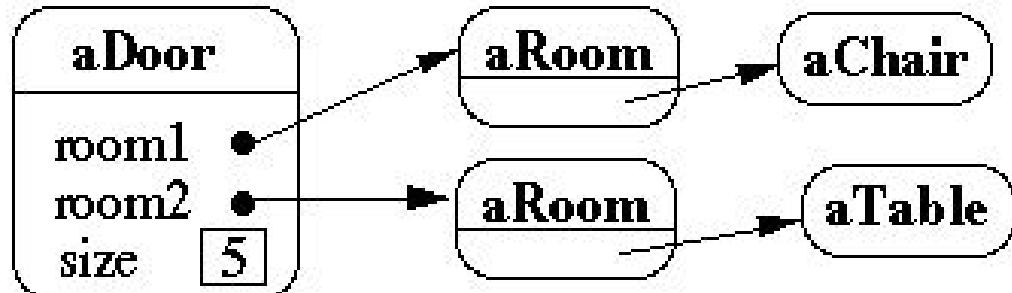


# SHALLOW COPY VS. DEEP COPY (2)

Original



Deep Copy



# CLONING IN C++ – COPY CONSTRUCTORS

```
class Door {  
public:  
    Door();  
    Door( const Door& );  
    virtual Door* clone() const;  
    virtual void Initialize( Room*, Room* );  
private:  
    Room* room1; Room* room2;  
};  
  
//Copy constructor  
Door::Door ( const Door& other )  {  
    room1 = other.room1; room2 = other.room2;  
}  
  
Door* Door::clone() {  
    return new Door( *this );  
}
```

# CLONING IN JAVA – OBJECT CLONE()

```
protected Object clone() throws  
CloneNotSupportedException
```

Creates a clone of the object

- allocate a new instance and,
- place a *bitwise clone* of the current object in the new object.

```
class Door implements Cloneable {  
    public void Initialize( Room a, Room b) {  
        room1 = a; room2 = b;  
    }  
  
    public Object clone() throws  
CloneNotSupportedException {  
        return super.clone();  
    }  
    Room room1, room2;  
}
```

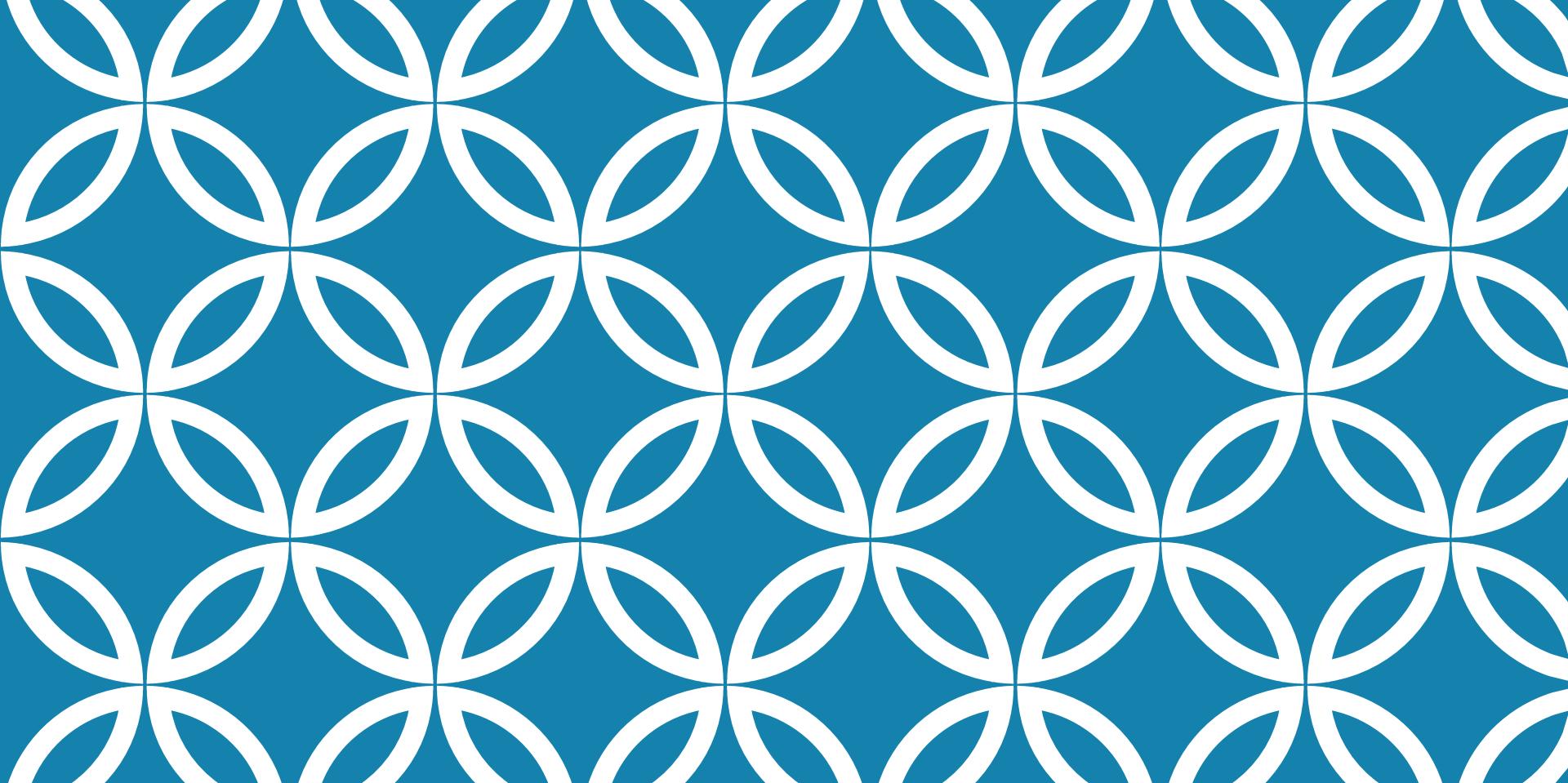
# SOLVING THE MAZE PROBLEM

```
class MazePrototypeFactory {  
    private Maze _prototypeMaze;  
    private Room _prototypeRoom;  
    private Wall _prototypeWall;  
    private Door _prototypeDoor;  
  
    public MazePrototypeFactory(Maze m,  
        Wall w, Room r, Door d) {  
        _prototypeMaze = m; _prototypeWall  
        = w;  
        _prototypeRoom = r; _prototypeDoor  
        = d;  
    }  
  
    public Wall makeWall()  
    {  
        return _prototypeWall.clone();  
    }  
  
    public Maze makeMaze() {...}  
    public Room makeRoom(int) {...}
```

```
public Door makeDoor(Room r1, r2) {  
    Door door =  
    _prototypeDoor.clone();  
    door.initialize(r1, r2);  
    return door;  
}
```

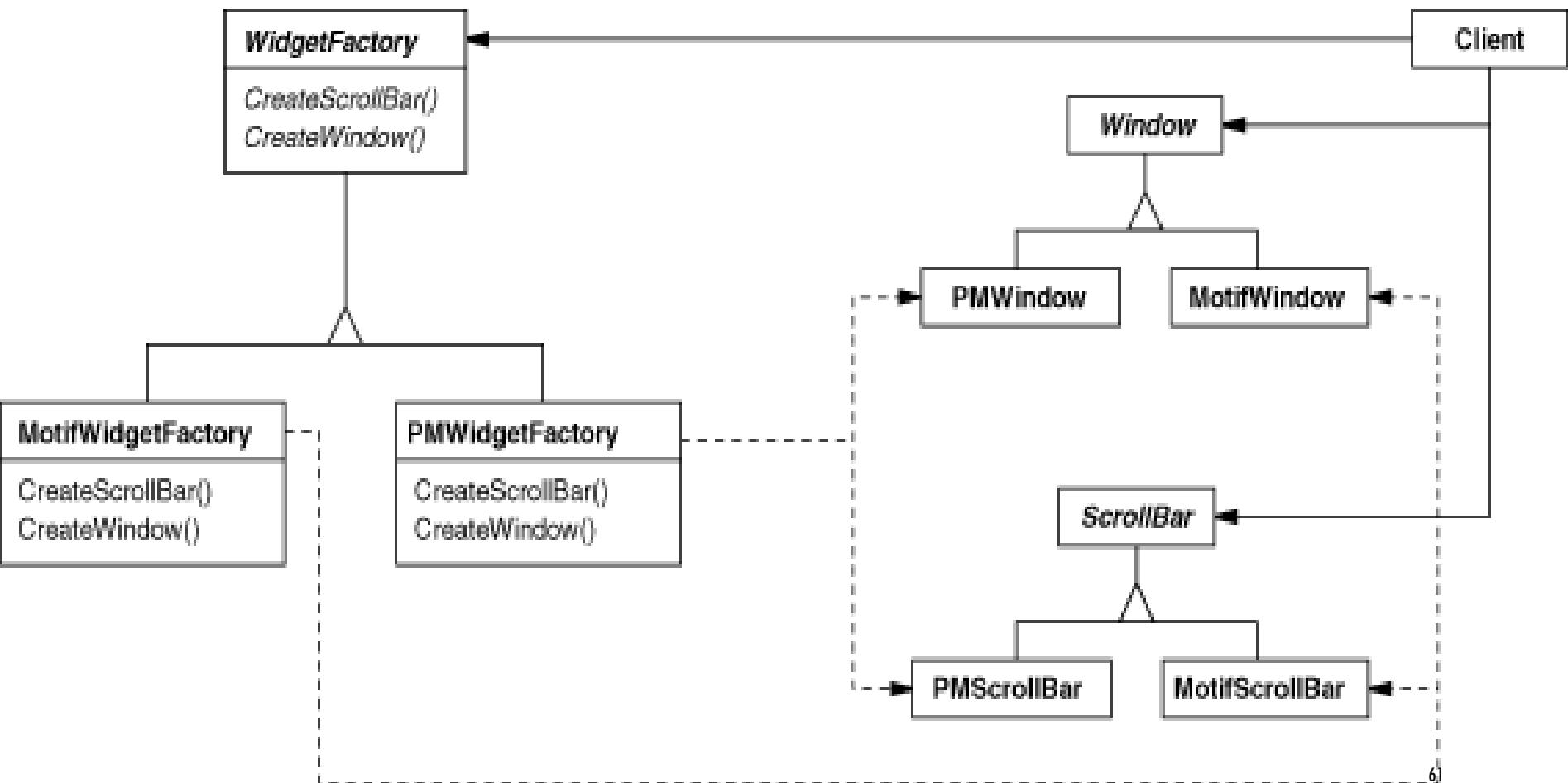
## Creating a maze for a game.....

```
MazePrototypeFactory simpleMazeFactory  
= new MazePrototypeFactory (new Maze(),  
new Wall(), new Room(), new Door());  
  
MazeGame game;  
  
Maze maze =  
game.createMaze(simpleMazeFactory);
```



# ABSTRACT FACTORY

# INTRODUCTIVE EXAMPLE



# BASIC ASPECTS

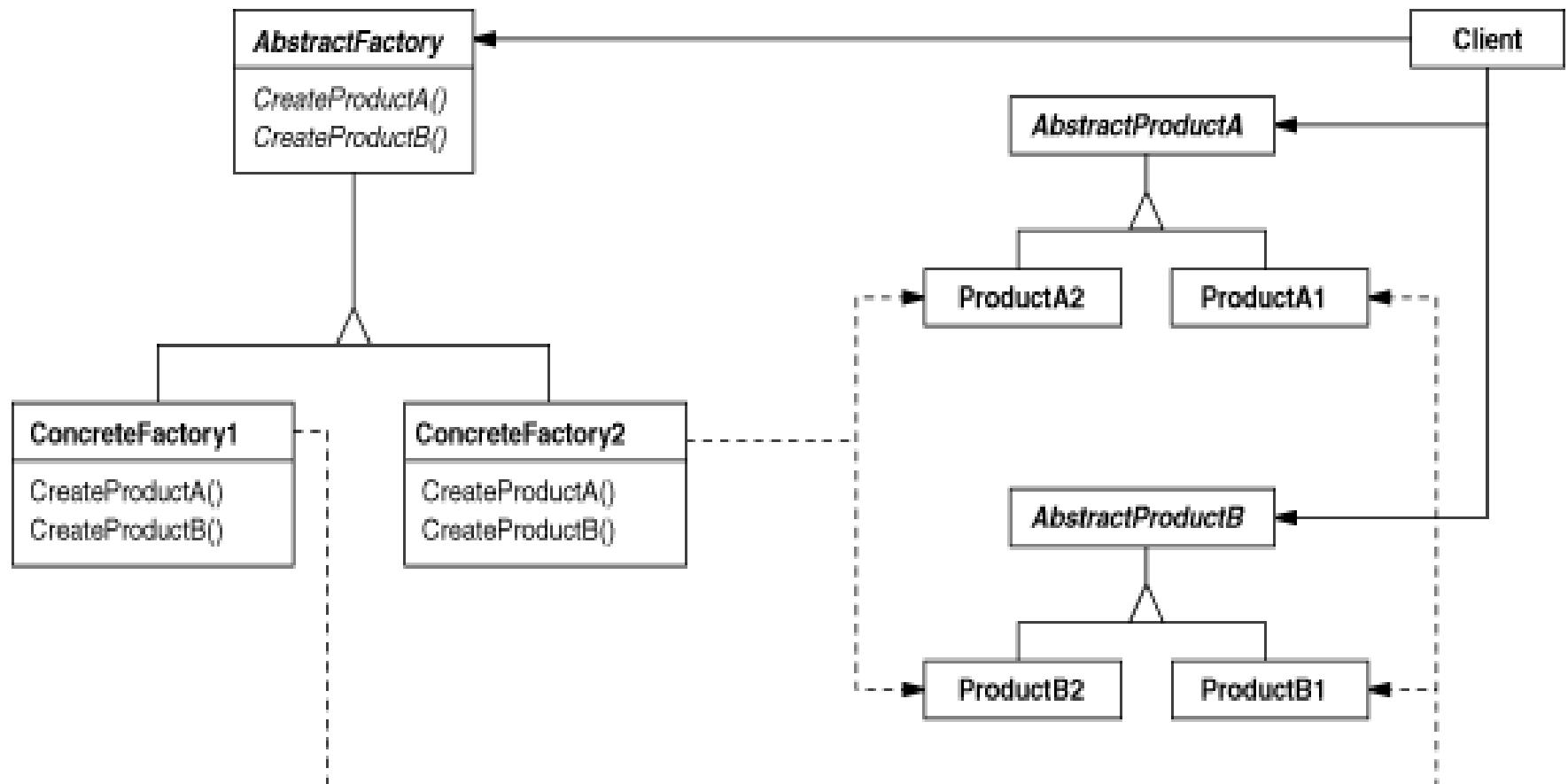
## Intent

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

## Applicability

- System should be independent of how its products are created, composed and represented
- System should be configured with one of multiple families of products
- Need to **enforce** that a family of product objects is used together

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## Abstract Factory

- declares an interface for operations to create abstract products

## ConcreteFactory

- implements the operations to create products

## AbstractProduct

- declares an interface for a type of product objects

## ConcreteProduct

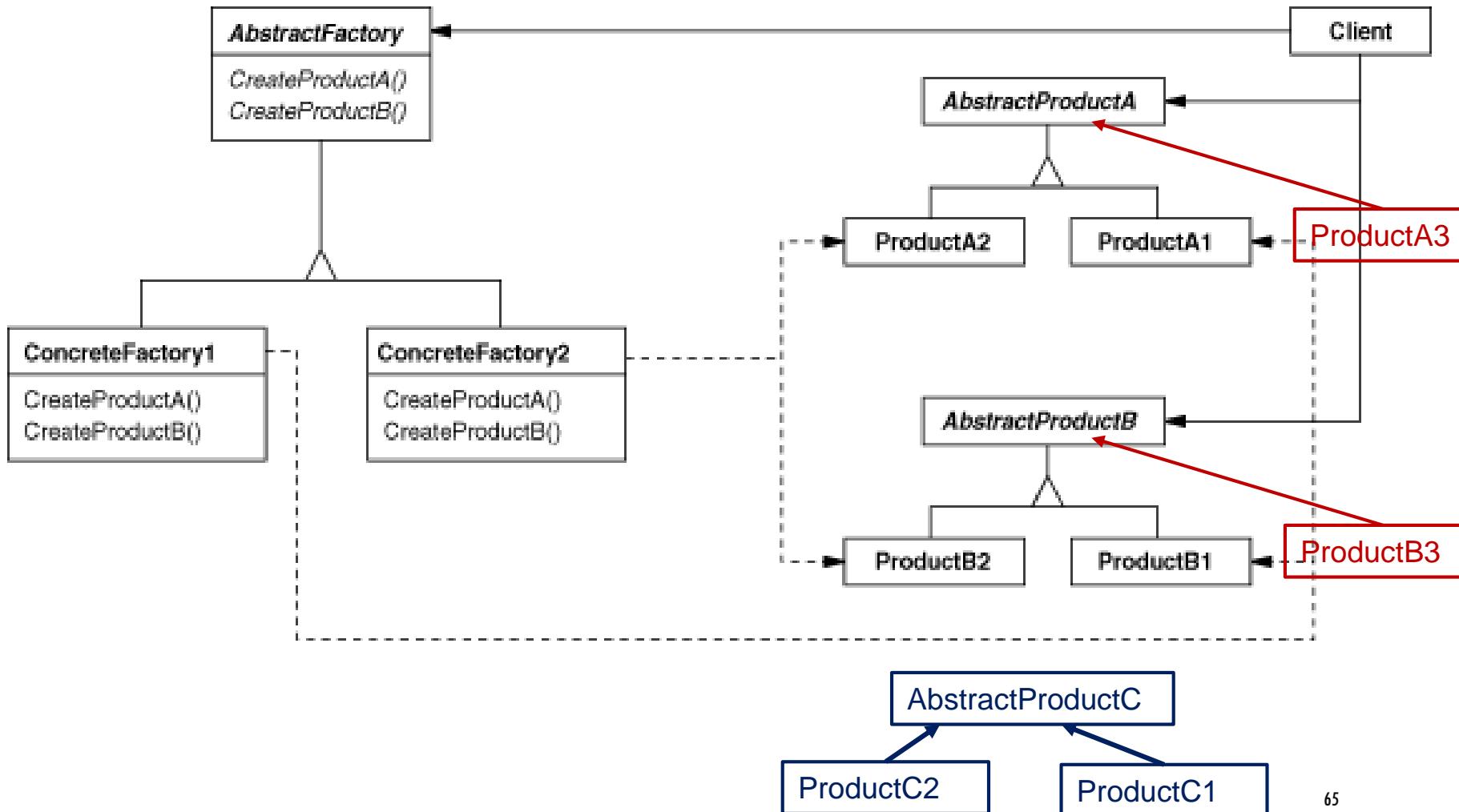
- declares an interface for a type of product objects

## Client

- uses only interfaces decl. by AbstractFactory and AbstractProduct

*A single instance of a ConcreteFactory created.*

# WHAT CHANGES ARE SUPPORTED?



# CONSEQUENCES

Isolation of concrete classes

- appear in **ConcreteFactories** not in client's code

Exchanging of product families becomes easy

- a **ConcreteFactory** appears only in one place

Promotes consistency among products

- all products in a family change **at once**, and change **together**

Supporting new kinds of products is difficult

- requires a change in the interface of **AbstractFactory**
- ... and consequently all subclasses

# IMPLEMENTATION ISSUES

## Factories as Singletons

- to assure that only one `ConcreteFactory` per product family is created

## Creating the Products

- collection of *Factory Methods*
- can be also implemented using *Prototype*
  - define a prototypical instance for each product in `ConcreteFactory`

## Defining Extensible Factories

- a single factory method with parameters
- more flexible, less safe!

# CREATING PRODUCTS

...using own factory methods

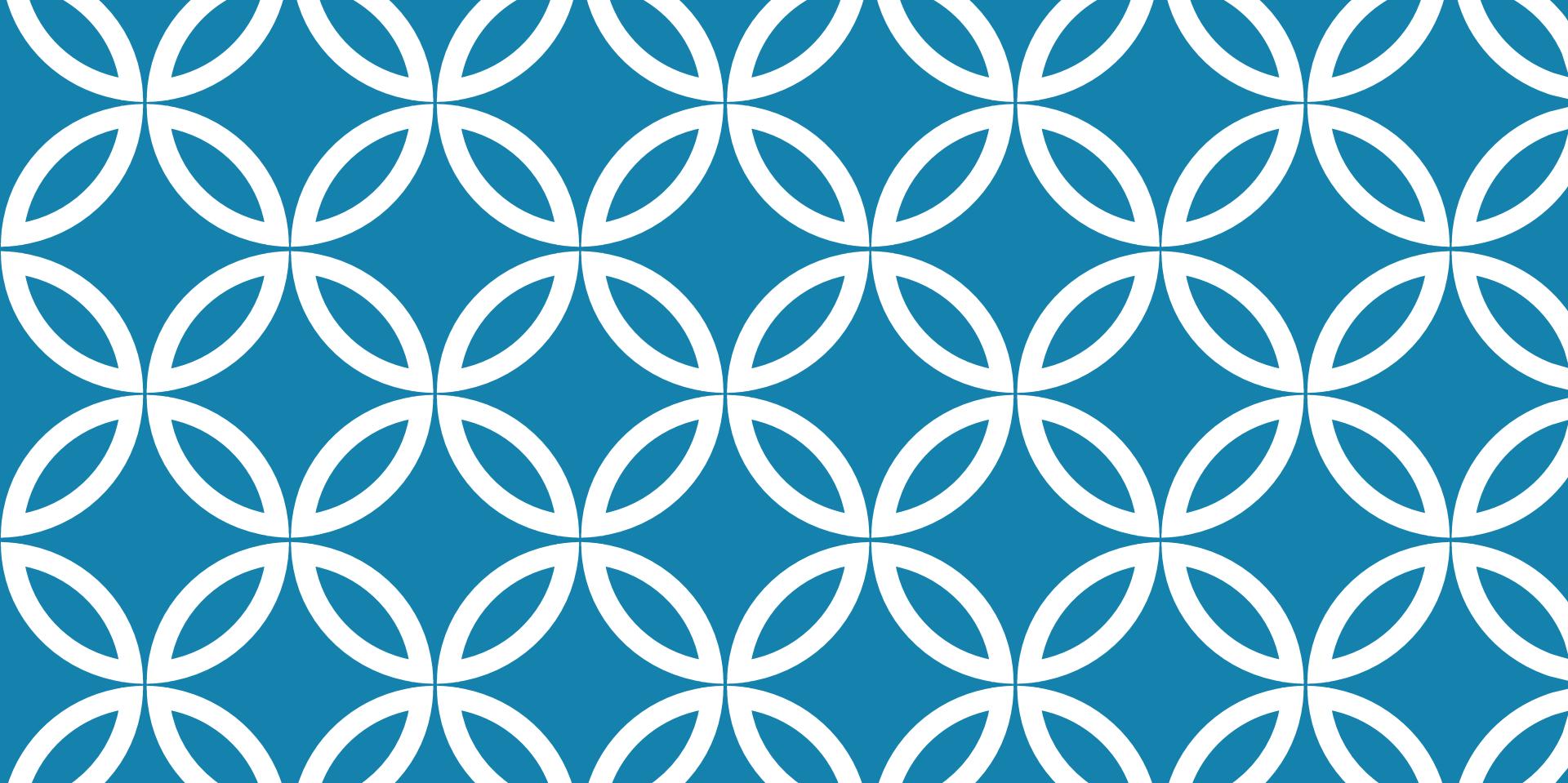
```
abstract class WidgetFactory {  
    public Window createWindow();  
    public Menu createMenu();  
    public Button createButton();  
}  
  
class MacWidgetFactory extends WidgetFactory {  
    public Window createWindow()  
    { return new MacWidow(); }  
    public Menu createMenu()  
    { return new MacMenu(); }  
    public Button createButton()  
    { return new MacButton(); }  
}
```

# CREATING PRODUCTS

... using product's factory methods

- subclass just provides the concrete products in the constructor
- spares the reimplementation of FM's in subclasses

```
abstract class WidgetFactory {  
    private Window windowFactory;  
    private Menu menuFactory;  
    private Button buttonFactory;  
  
    public Window createWindow()  
        { return windowFactory.createWindow() }  
    public Menu createMenu()  
        { return menuFactory.createWindow() }  
    public Button createButton()  
        { return buttonFactory.createWindow() }  
}  
  
class MacWidgetFactory extends WidgetFactory {  
    public MacWidgetFactory() {  
        windowFactory = new MacWindow();  
        menuFactory = new MacMenu();  
        buttonFactory = new MacButton();  
    }  
}
```



**SINGLETON**

# BASICS

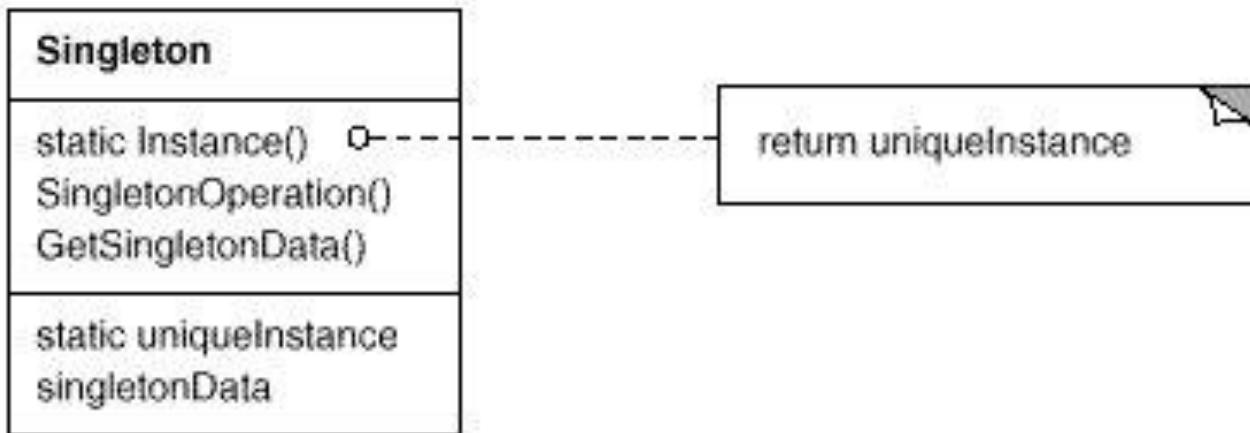
## Intent

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

## Applicability

- want exactly one instance of a class
- accessible to clients from one point
- want the instance to be extensible
- can also allow a countable number of instances
- improvement over global namespace

# STRUCTURE OF THE PATTERN



Put constructor in private/protected data section

# PARTICIPANTS AND COLLABORATIONS

## Singleton

- defines an **Instance** method that becomes the single "gate" by which clients can access its unique instance.
  - **Instance** is a class method (i.e. static)
- may be responsible for creating its own unique instance

*Clients access Singleton instances solely through the **Instance** method*

# CONSEQUENCES

- Controlled access to sole instance
- Permits refinement of operations and representation
- Permits a variable (but precise) number of instances
- Reduced global name space



# SOFTWARE DESIGN

Structural DP

# CONTENT

## Design Patterns

- Structural Patterns
  - Adapter
  - Composite
  - Decorator
  - Proxy
  - Bridge

# REFERENCES

- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.
- Univ. of Timisoara Course materials

# DESIGN PATTERN SPACE

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Singleton Prototype	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Command Chain of Responsibility Strategy Visitor Iterator Mediator Memento Observer State

# STRUCTURAL PATTERNS

Structural patterns are concerned with how classes and objects are composed to form larger structures.

Structural **class** patterns use **inheritance** to compose interfaces or implementations.

Rather than composing interfaces or implementations, structural **object** patterns describe ways to **compose** objects to realize new functionality.

# PAYING BY PAYPAL

```
class PayPal {  
    public PayPal() {  
        // Your Code here //  
    }  
    public void sendPayment(double amount) {  
        // Paying via Paypal //  
    }  
PayPal pay = new PayPal();  
pay.sendPayment(2629);
```

# CHANGES

`sendPayment -> payAmount`

**COMMIT TO AN INTERFACE, NOT AN IMPLEMENTATION!**

```
// Simple Interface for each Adapter we create

interface PaymentAdapter {
    public pay(double amount);
}

class PaypalAdapter implements PaymentAdapter {
    private PayPal paypal;
    public PaypalAdapter(PayPal p) {
        this.paypal = p;
    }
    public void pay(double amount) {
        paypal.sendPayment(amount);
    }
}

// Client Code
PaymentAdapter pp = new PaypalAdapter(new PayPal());
pp.pay(2629);
```

# CHANGING THE PAY SERVICE

```
class PayServiceAdapter implements PaymentAdapter
{
    private PayService pay;
    public PayServiceAdapter(PayService p) {
        this.pay = p;
    }
    public void pay(double amount) {
        pay.payAmount(amount);
    }
}
```

# ADAPTER PATTERN

## **Intent**

Convert the interface of a class into another interface clients expect.

Adapter enables classes to work together that couldn't otherwise because of incompatible interfaces.

## **Also known as**

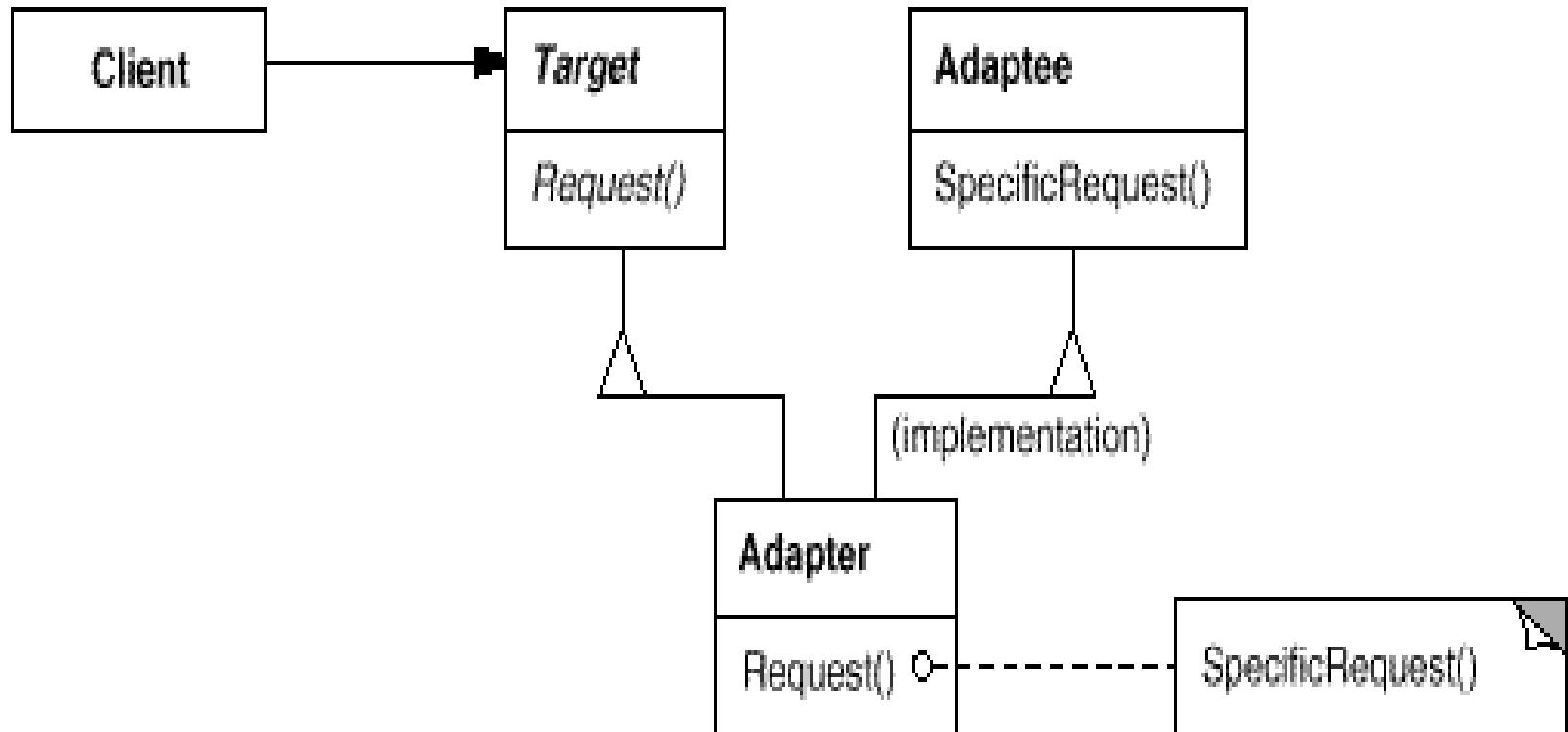
Wrapper

# ADAPTER PATTERN

## Applicability

- 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.

# ADAPTER PATTERN - INHERITANCE-BASED



```
class Adapter extends Adaptee implements Target
```

# ADAPTER PATTERN - PARTICIPANTS

## **Target (Shape)**

- defines the domain-specific interface that Client uses.

## **Client (DrawingEditor)**

- collaborates with objects conforming to the Target interface.

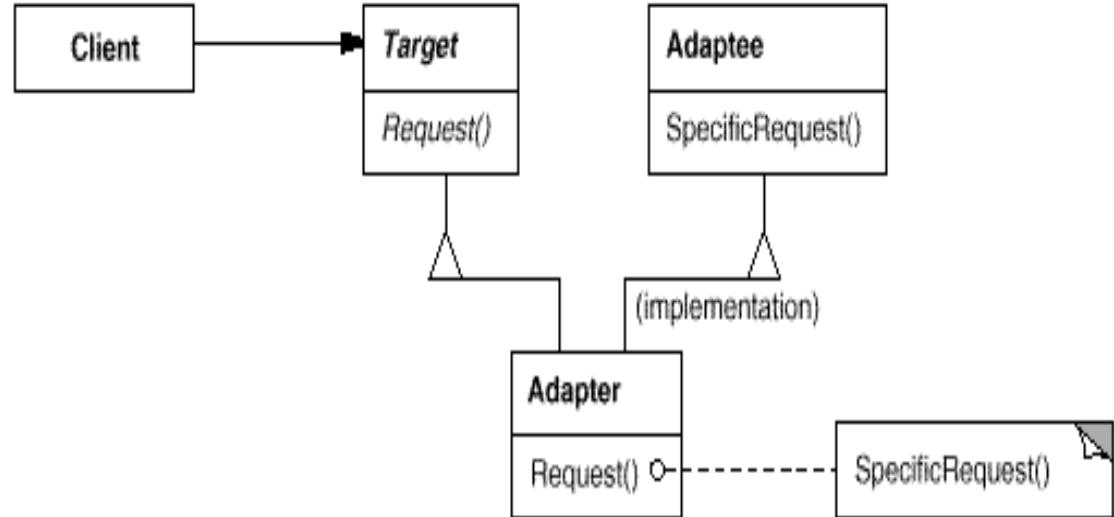
## **Adaptee (TextView)**

- defines an existing interface that needs adapting.

## **Adapter (TextShape)**

- adapts the interface of Adaptee to the Target interface.

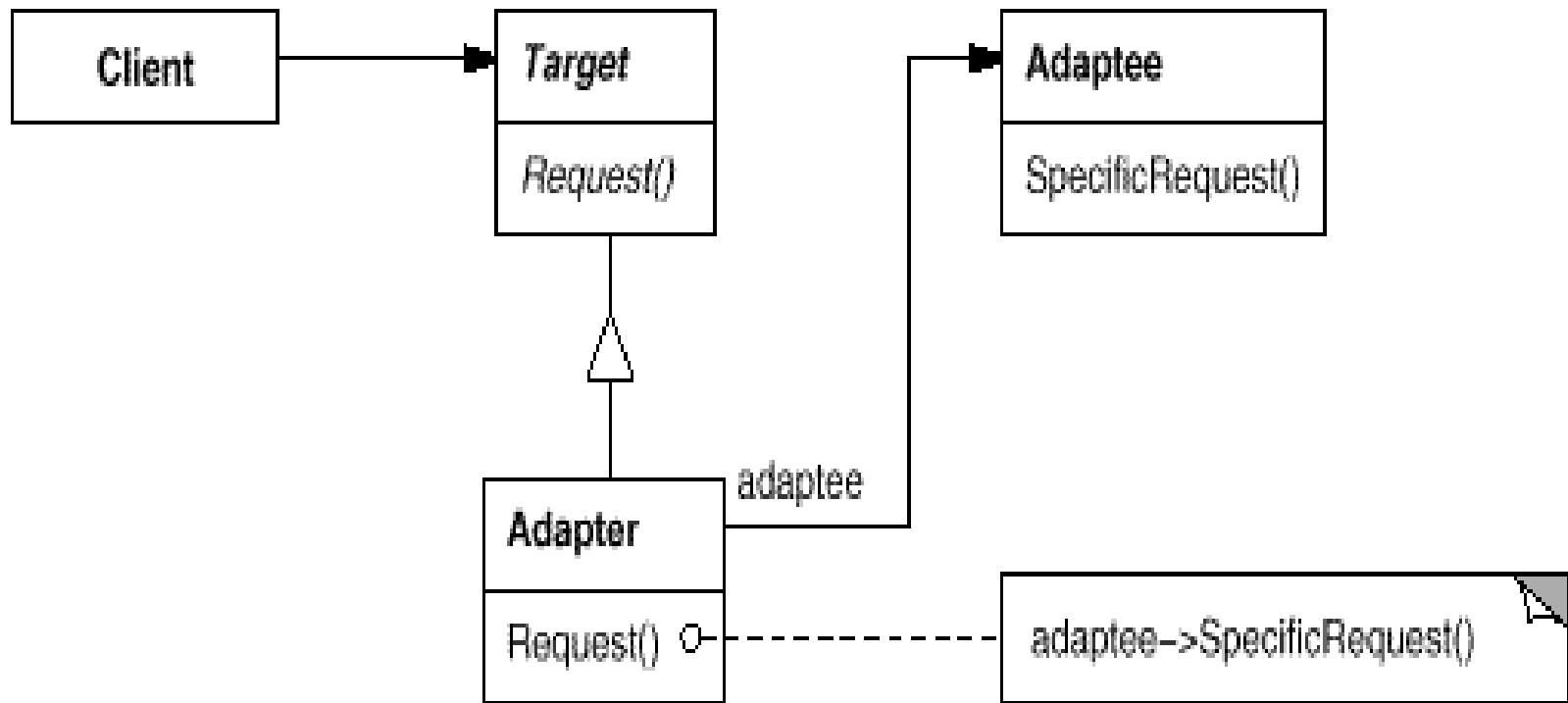
# CONSEQUENCES



## Class Adapter (Inheritance based)

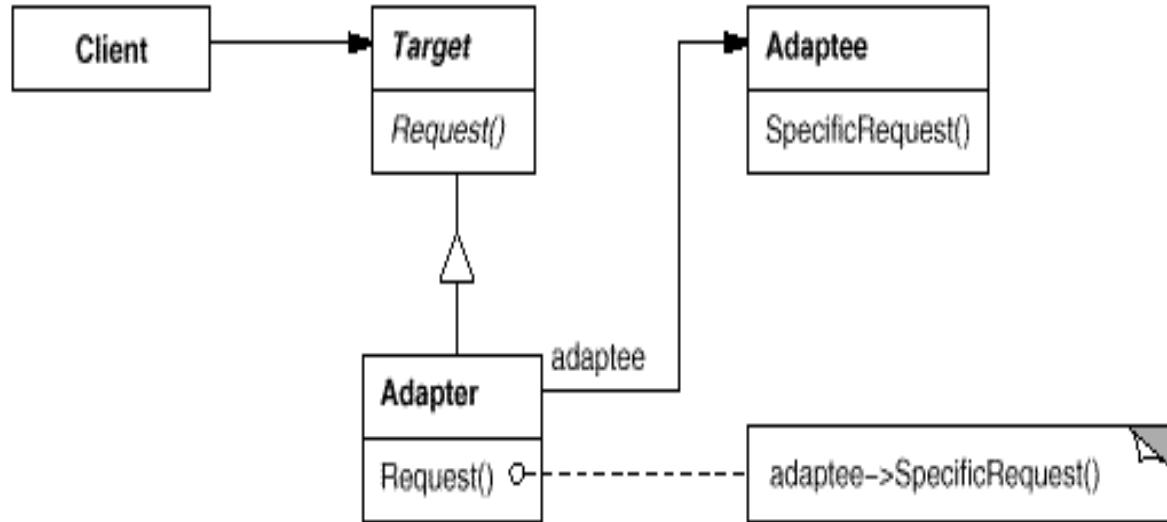
- adapts Adaptee to Target by committing to a concrete Adapter class => a class adapter won't work when we want to adapt a class and all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the Adaptee.

# ADAPTER PATTERN – COMPOSITION BASED



```
class Adapter implements Target {  
    Adaptee adaptee;  
}  
...
```

# CONSEQUENCES



## Object Adapter (Composition based)

- lets a single Adapter work with many Adaptees (that is, the Adaptee itself and all of its subclasses (if any)). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

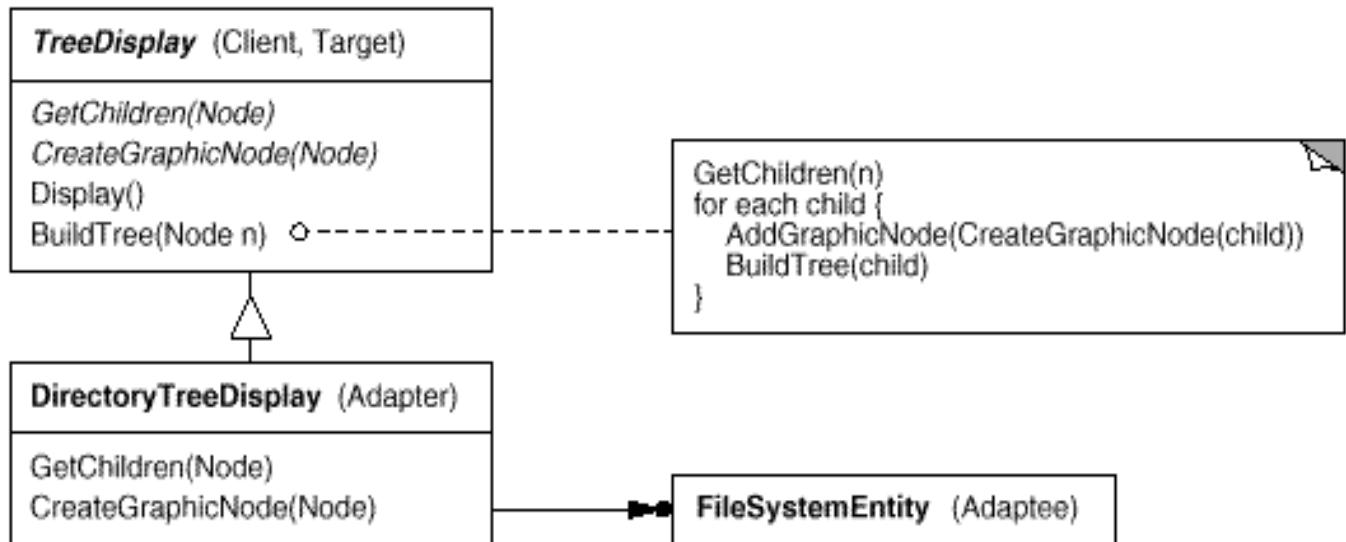
# OTHER ISSUES

*How much adapting does Adapter do?*

- from simple interface conversion, to supporting an entirely different set of operations

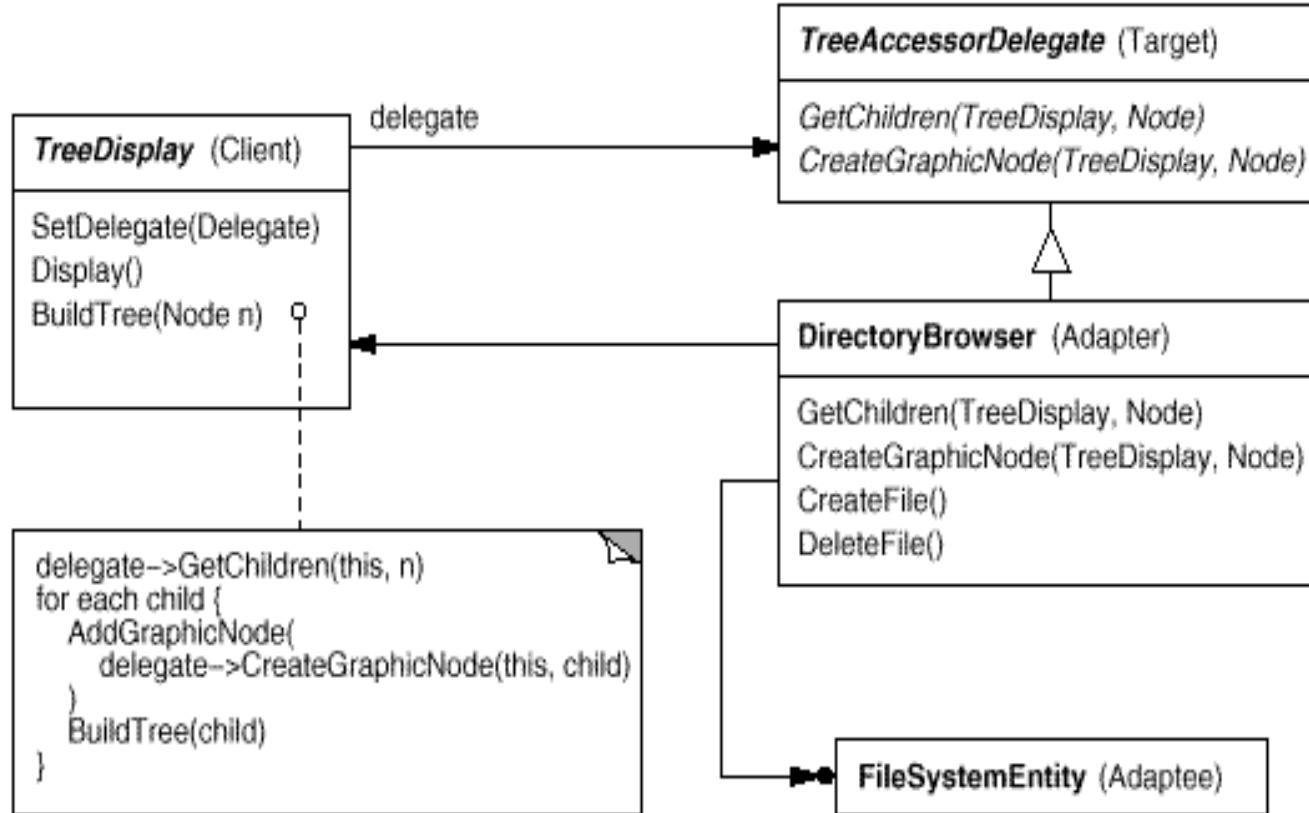
*Pluggable adapters.*

- Using abstract operations



# PLUGGABLE ADAPTERS CONTINUED

- Using delegate objects

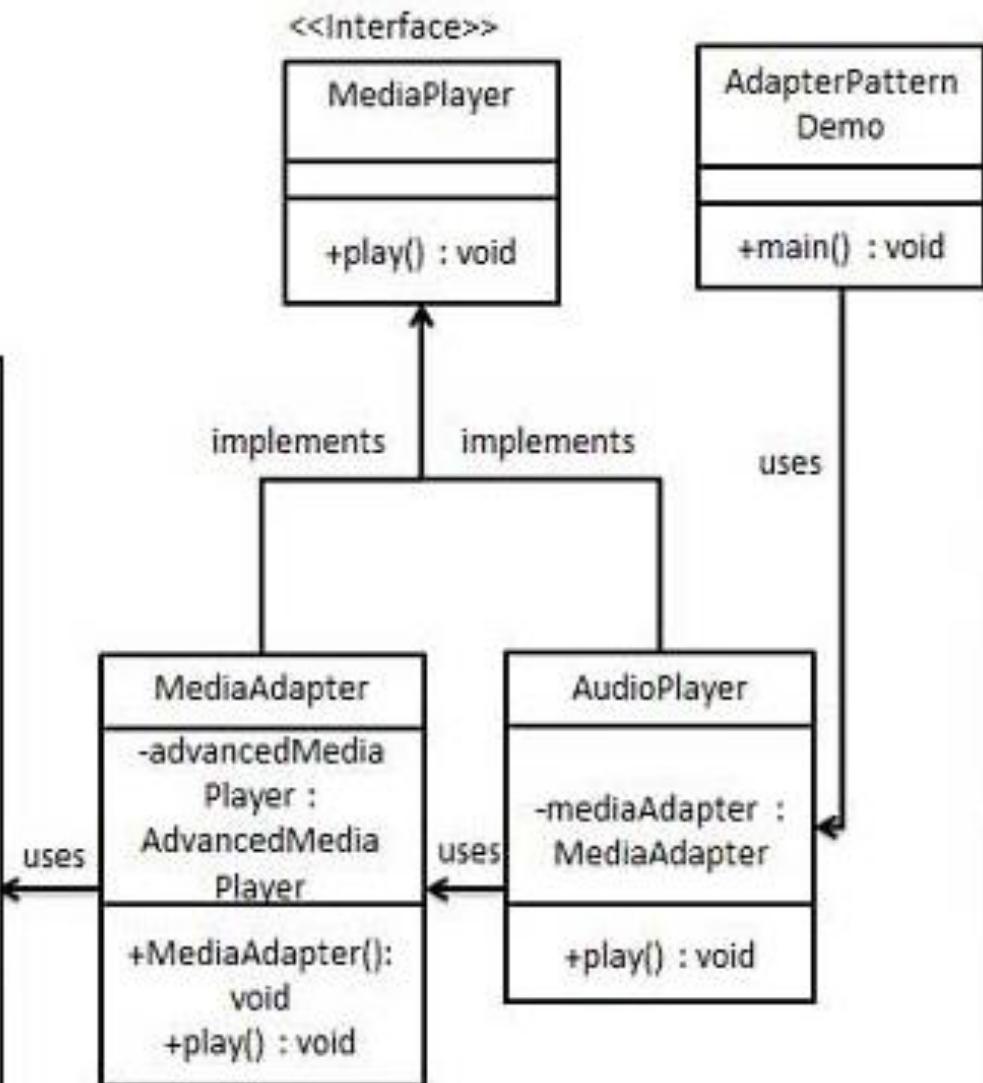
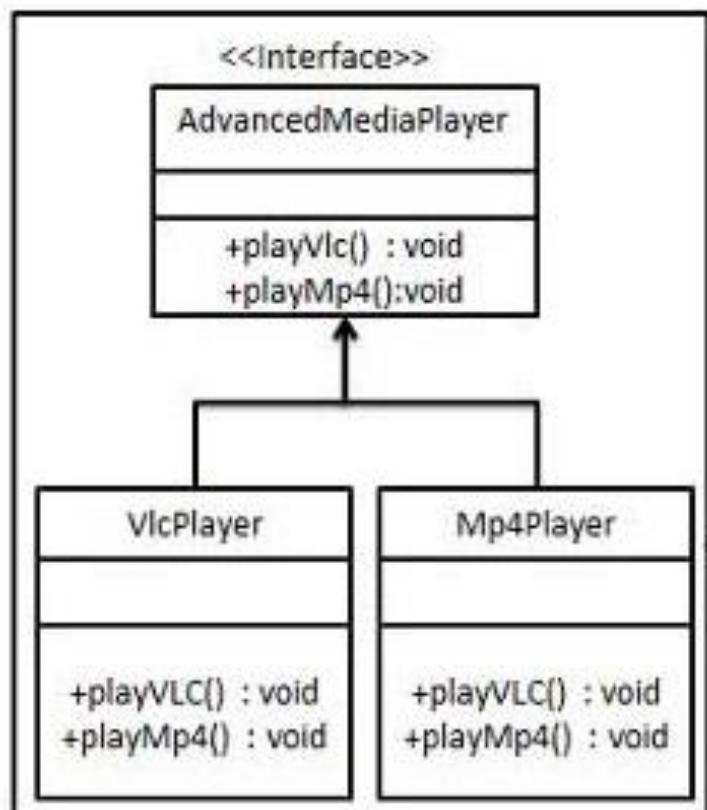


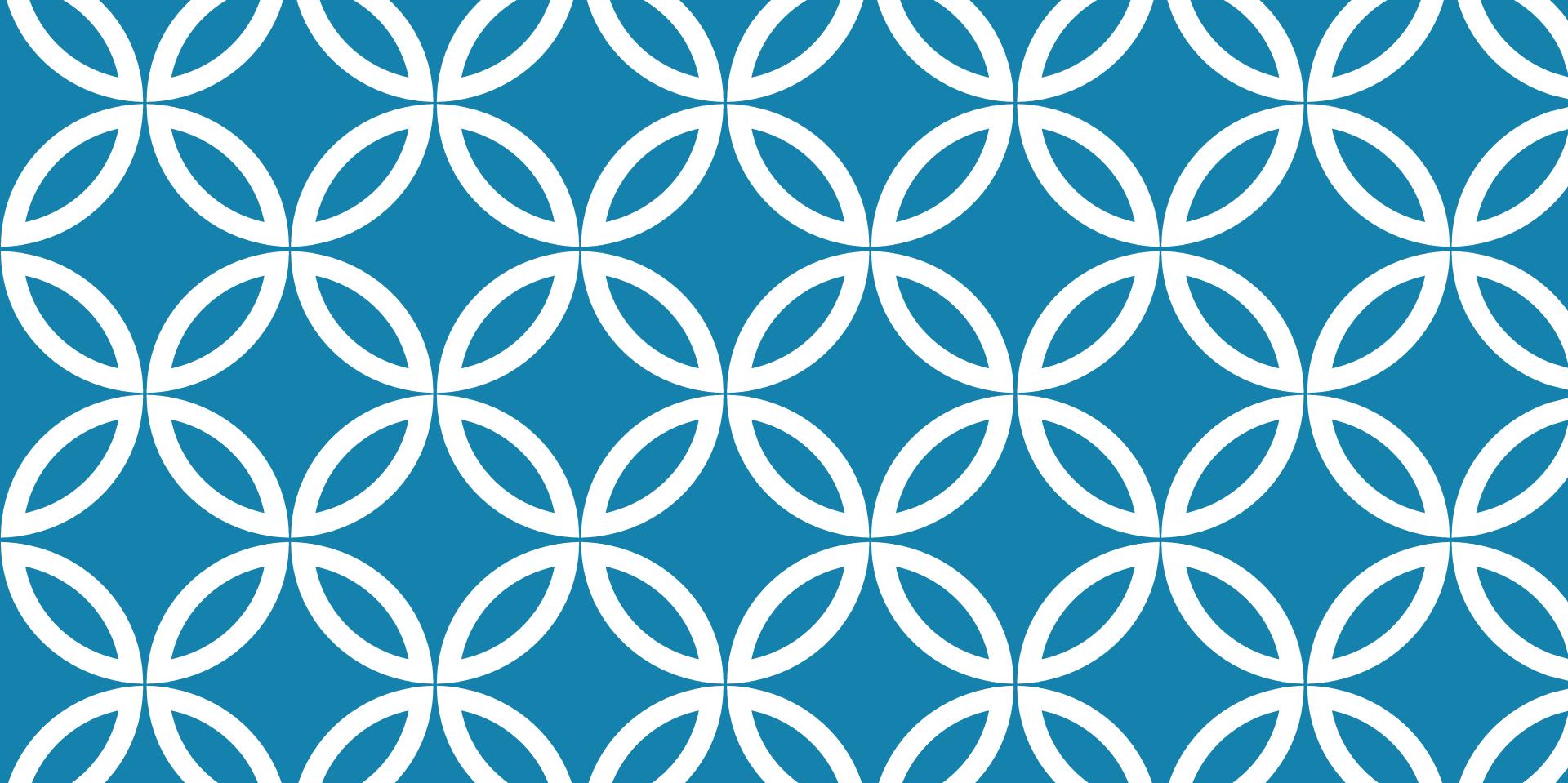
# ANOTHER (BAD!) EXAMPLE

Interface Segregation

Open Closed

How would you design it?

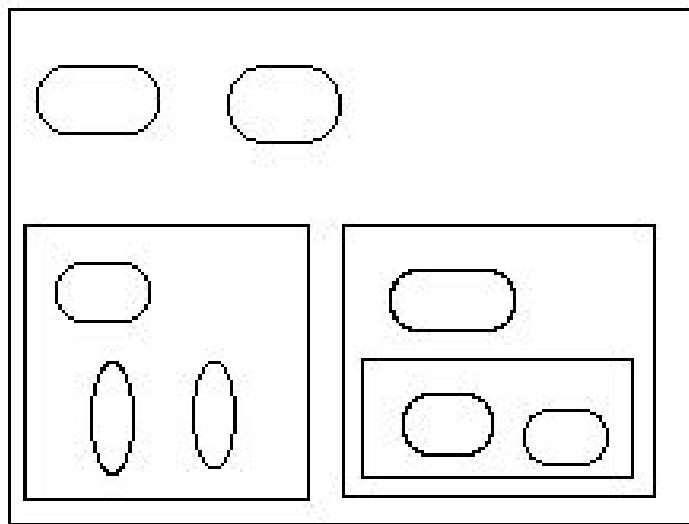




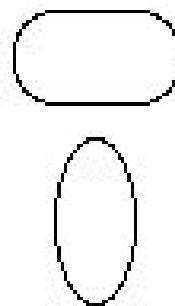
# COMPOSITE PATTERN

# MOTIVATION

## Application Window



Windows &  
WidgetContainers



Buttons  
Menus  
Text Areas  
etc

## GUI Windows and GUI elements

- How does the window hold and deal with the different items it has to manage?
- Widgets are different than WidgetContainers

# IMPLEMENTATION IDEAS

## Nightmare Implementation

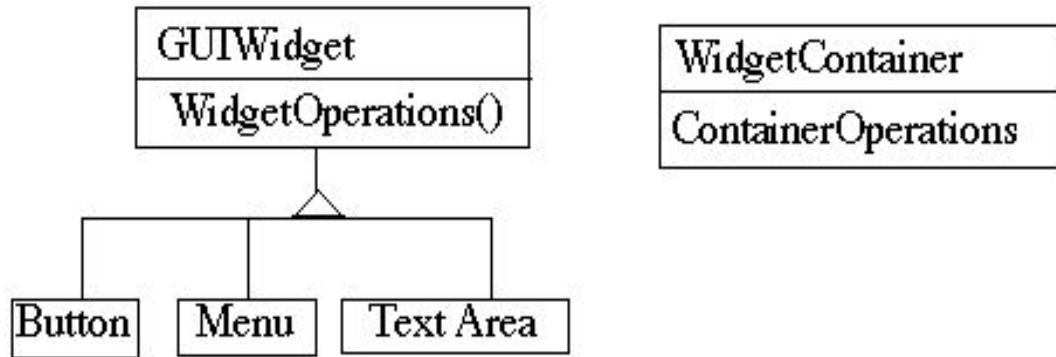
- for each operation deal with each category of objects individually
- no uniformity and no hiding of complexity
- a lot of code duplication

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if ( myButtons != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myButtons[k].refresh();
        if ( myMenus != null )
            for ( int k = 0; k < myMenus.length(); k++ )
                myMenus[k].display();
        if ( myTextAreas != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myTextAreas[k].refresh();
        if ( myContainers != null )
            for (int k = 0; k < myContainers.length();k++)
                myContainers[k].updateElements();
        // ...etc.
    }

    public void fooOperation()
    {
        if ( blah ) etc.
        // again and again... .
    }
}
```

# PROGRAM TO AN INTERFACE



- uniform dealing with widget operations
- but still containers are treated different

```
class Window {  
    GUIWidgets[] myWidgets;  
    WidgetContainer[] myContainers;  
  
    public void update() {  
        if(myWidgets != null)  
            for (int k = 0; k < myWidgets.length(); k++)  
                myWidgets[k].update();  
        if(myContainers != null)  
            for (int k = 0; k < myContainers.length(); k++)  
                myContainers[k].updateElements();  
        // ... etc.  
    }  
}
```

# BASIC ASPECTS OF COMPOSITE PATTERN

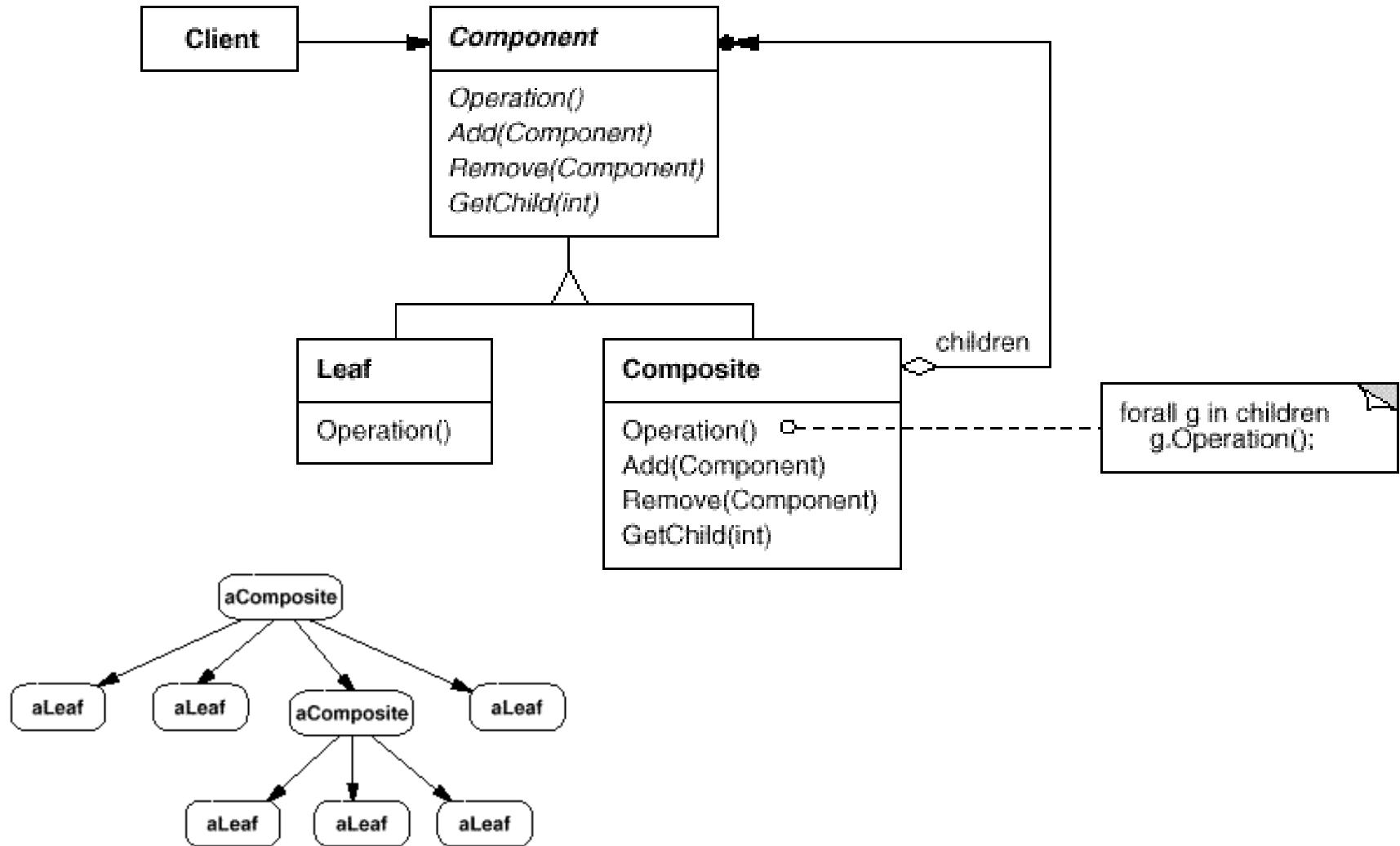
## Intent

- Treat individual objects and compositions of these object **uniformly**
- Compose objects into tree-structures to represent recursive aggregations

## Applicability

- represent part-whole hierarchies of objects
- be able to ignore the difference between compositions of objects and individual objects

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## Component

- declares interface for objects in the composition
- implements default behavior for components when possible

## Composite

- defines behavior for components having children
- stores child components
  - implement child-specific operations

## Leaf

- defines behavior for primitive objects in the composition

## Client

- manipulates objects in the composition through the Component interface

# CONSEQUENCES

Defines uniform class hierarchies

- recursive composition of objects

Make clients simple

- don't know whether dealing with a leaf or a composite
- simplifies code because it avoids to deal in a different manner with each class

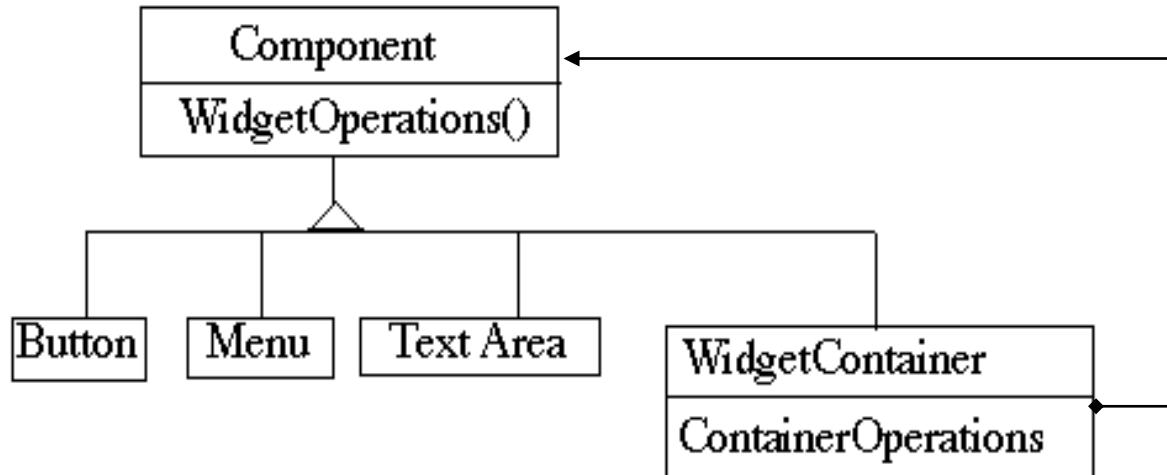
Easier to extend

- easy to add new Composite or Leaf classes

**Design excessively general**

- **type checks needed to restrict the types admitted in a particular composite structure**

# APPLYING COMPOSITE TO WIDGET PROBLEM



```
class WidgetContainer {
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }
}
```

# WHERE TO PLACE CONTAINER OPERATIONS ?

Adding, deleting, managing components in composite

- should they be placed in Component or in Composite?

## Pro-Transparency Approach

- Declaring them in the Component gives all subclasses the same interface
  - All subclasses can be treated alike.
- Safety costs!
  - clients may do stupid things like adding objects to leaves
  - `getComposite()` to improve safety.

## Pro-Safety Approach

- Declaring them in Composite is safer
  - Adding or removing widgets to non-Widget Containers is an error

# OTHER IMPLEMENTATION ISSUES

## Explicit references to Composite

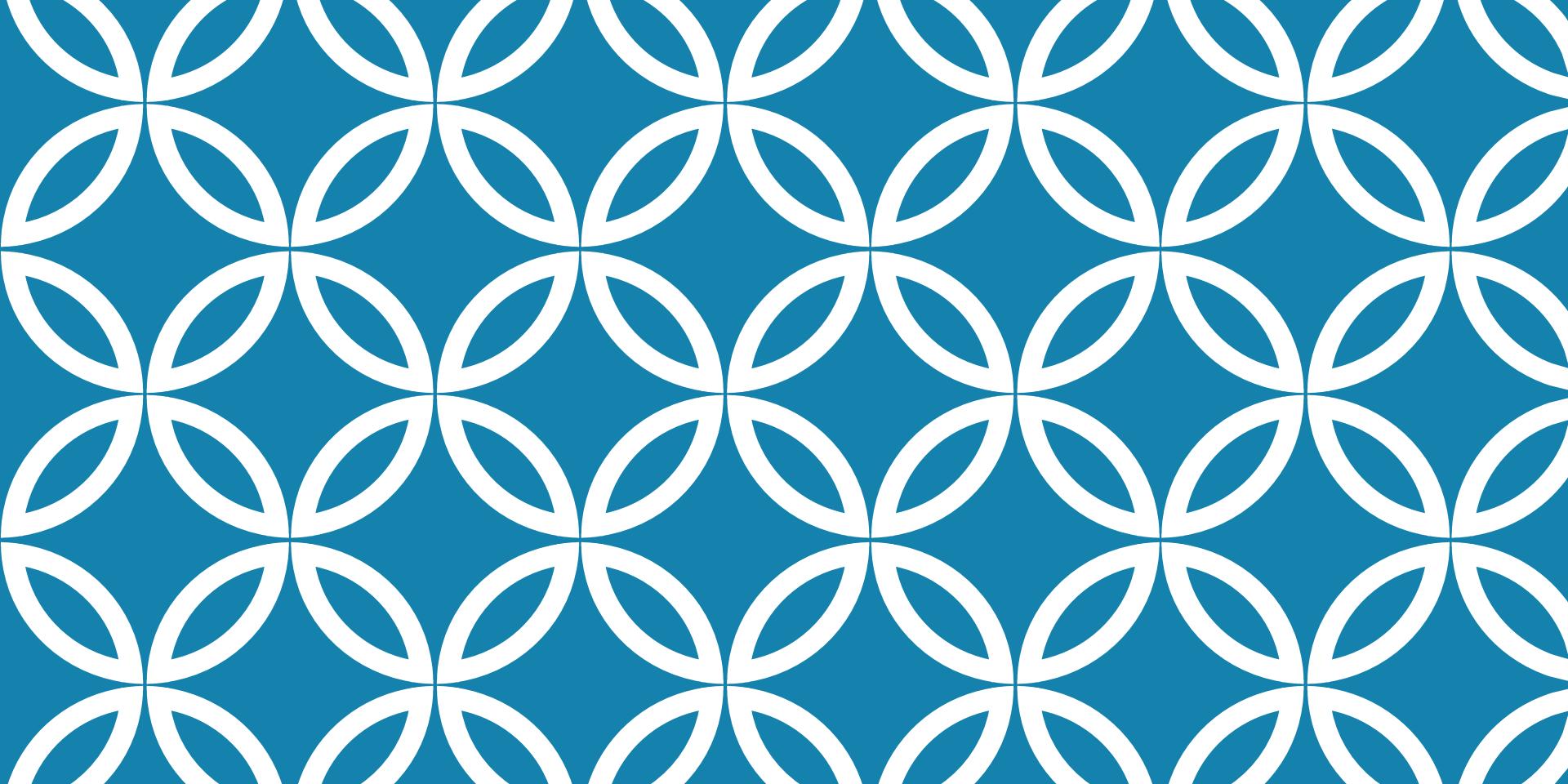
- simplifies traversal
- place it in Component
- the consistency issue
  - change reference to Composite **only** when add or remove component

## Component Ordering in Composites

- consider using Iterator

Who should **delete** components?

- Composite should delete its components



# DECORATOR PATTERN

Changing the skin of an object

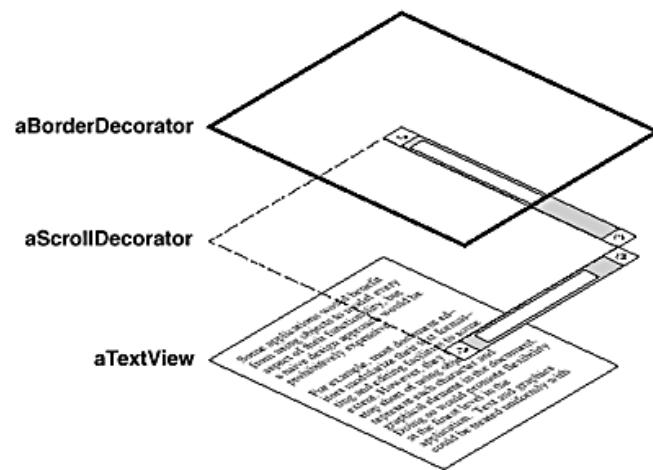
# MOTIVATION

A TextView has 2 features:

- borders:
  - 3 options: none, flat, 3D
- scroll-bars:
  - 4 options: none, side, bottom, both

**Inheritance** => How many Classes?

- $3 \times 4 = 12 !!!$
- e.g. TextView, TextViewWithNoBorder&SideScrollbar,  
TextViewWithNoBorder&BottomScrollbar,  
TextViewWithNoBorder&Bottom&SideScrollbar,  
TextViewWith3DBorder,  
TextViewWith3DBorder&SideScrollbar,  
TextViewWith3DBorder&BottomScrollbar, ... .....

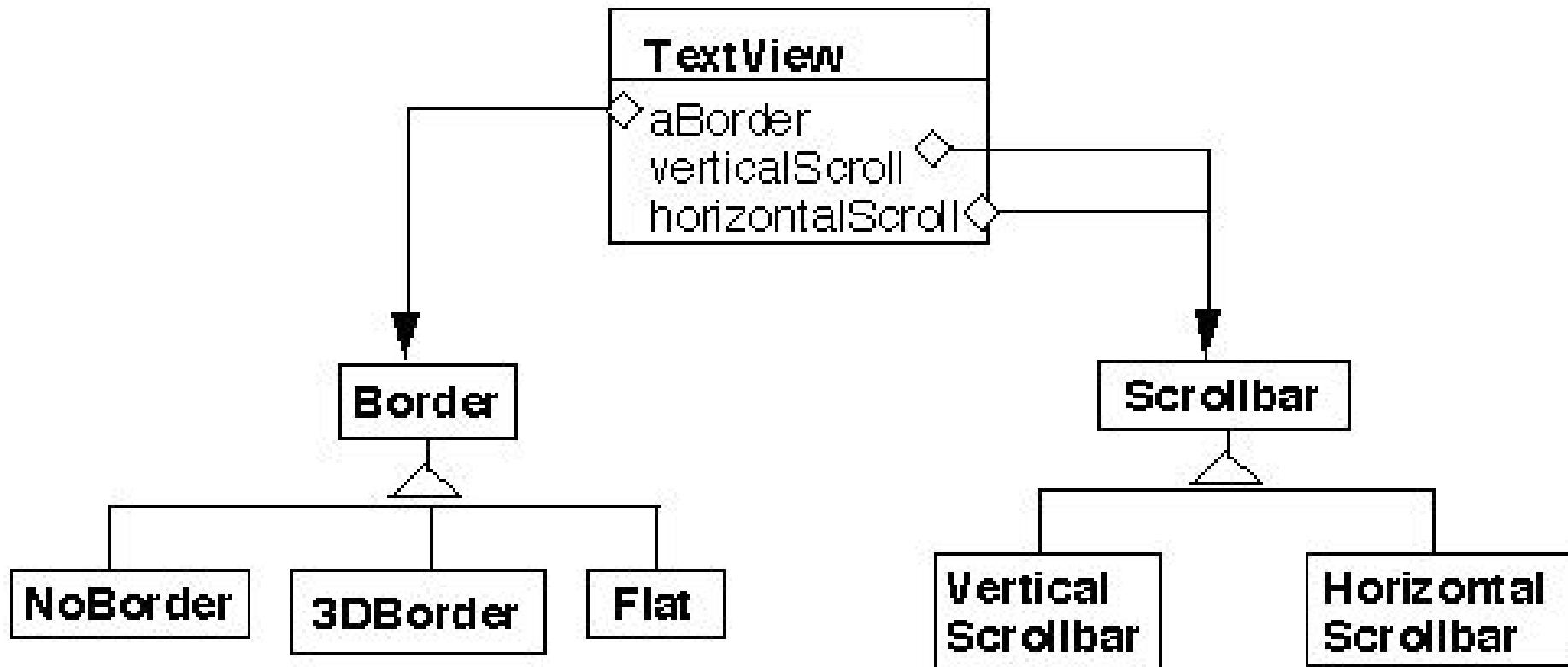


Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

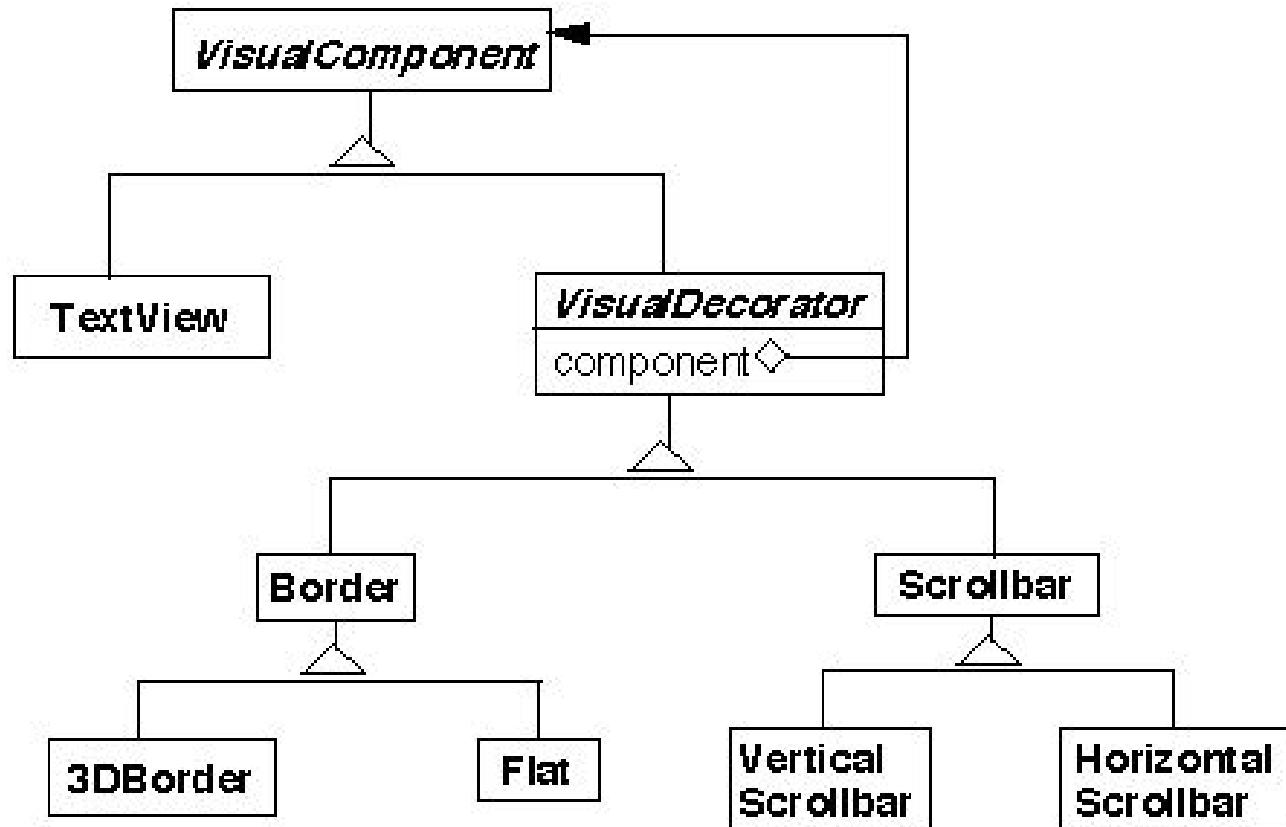
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with



# SOLUTION 1: USE OBJECT COMPOSITION



# SOLUTION 2: CHANGE THE SKIN, NOT THE GUTS!



TextView has **no** borders or scrollbars!

Add borders and scrollbars **on top of** a TextView

# BASIC ASPECTS

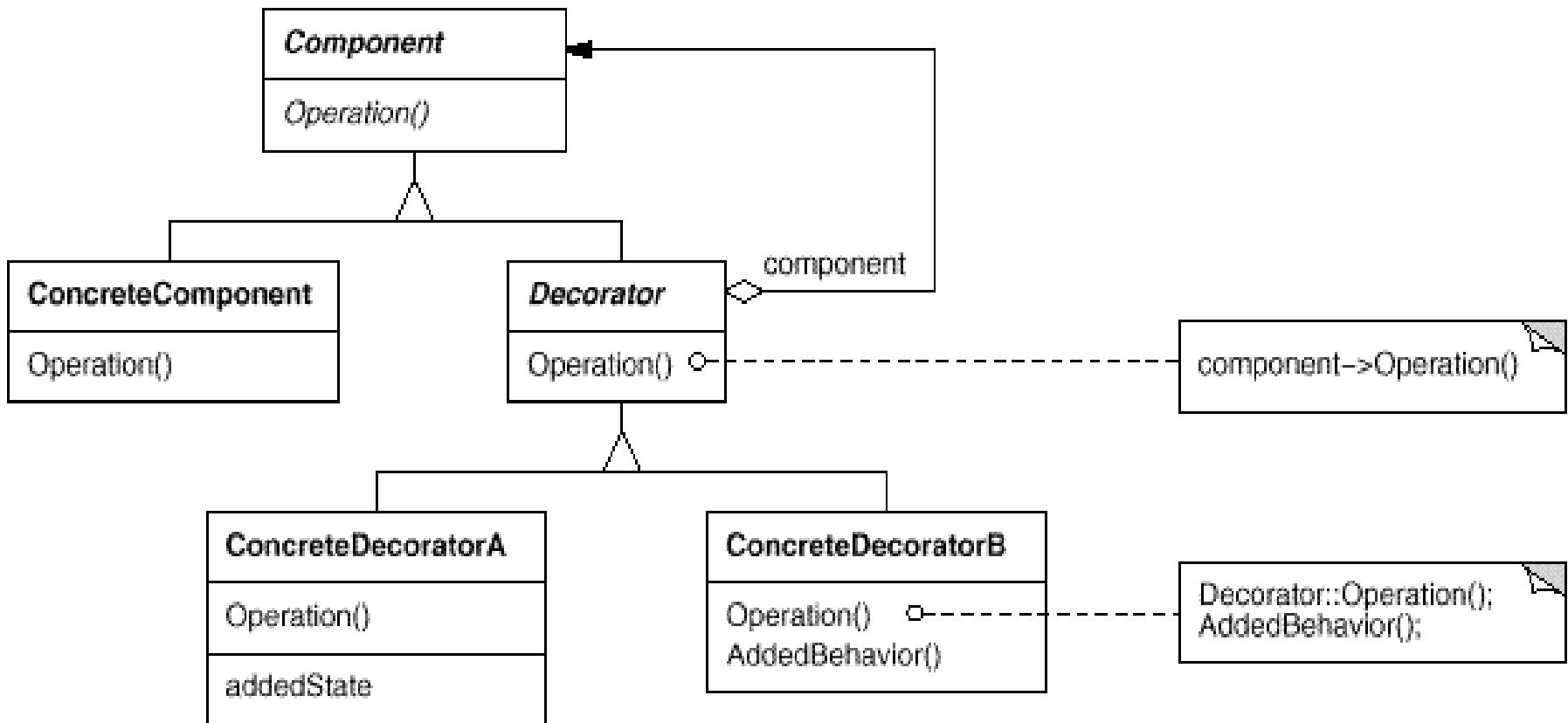
## Intent

- *Add responsibilities to a particular object rather than its class*
- Attach additional responsibilities to an object dynamically.
- Provide a flexible alternative to subclassing

## Applicability

- Add responsibilities to objects **transparently** and **dynamically**
  - i.e. without affecting other objects
- Extension by subclassing is impractical
  - may lead to too many subclasses

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## **Component**

- defines the interface for objects that can have responsibilities added dynamically

## **ConcreteComponent**

- the "bases" object to which additional responsibilities can be added

## **Decorator**

- defines an interface conformant to Component's interface
  - for transparency
- maintains a reference to a Component object

## **ConcreteDecorator**

- adds responsibilities to the component

# CONSEQUENCES

More **flexibility** than static inheritance

- allows to mix and match responsibilities
- allows to apply a property twice

Avoid feature-laden classes high-up in the hierarchy

- "pay-as-you-go" approach
- easy to define new types of decorations

Lots of little objects

- easy to customize, but hard to learn and debug

A decorator and its component aren't identical

- checking object identification can cause problems
  - e.g. if (aComponent instanceof TextView )

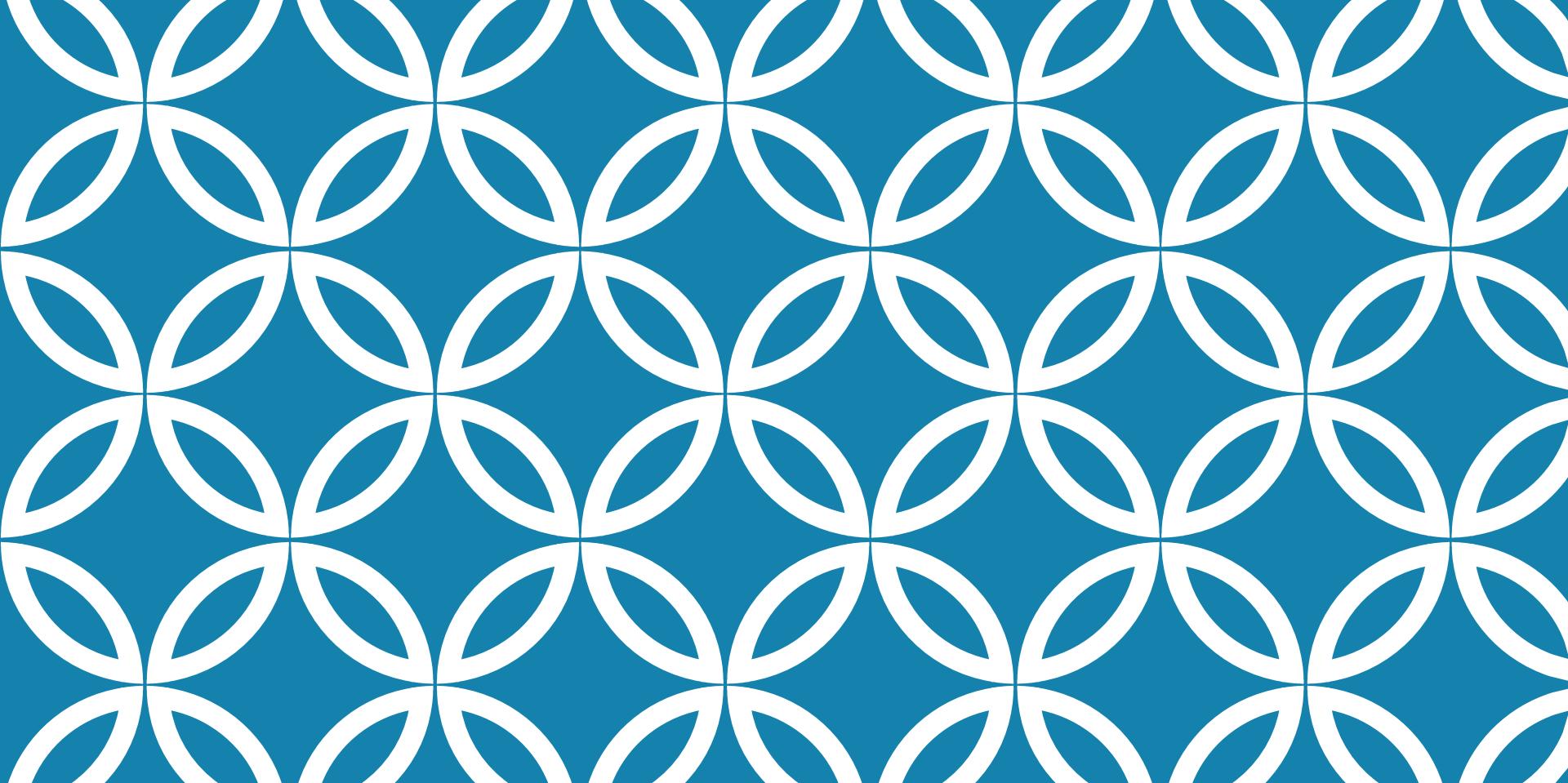
# IMPLEMENTATION ISSUES

Keep Decorators lightweight

- Don't put data members in Component
- Use it for shaping the interface

Omitting the abstract Decorator class

- If only one decoration is needed
- Subclasses may pay for what they don't need



**CHANGING THE GUTS**

# CHANGING THE "GUTS" OF AN OBJECT ...

## Control

- "shield" the implementation from direct access (**Proxy**)

## Decouple

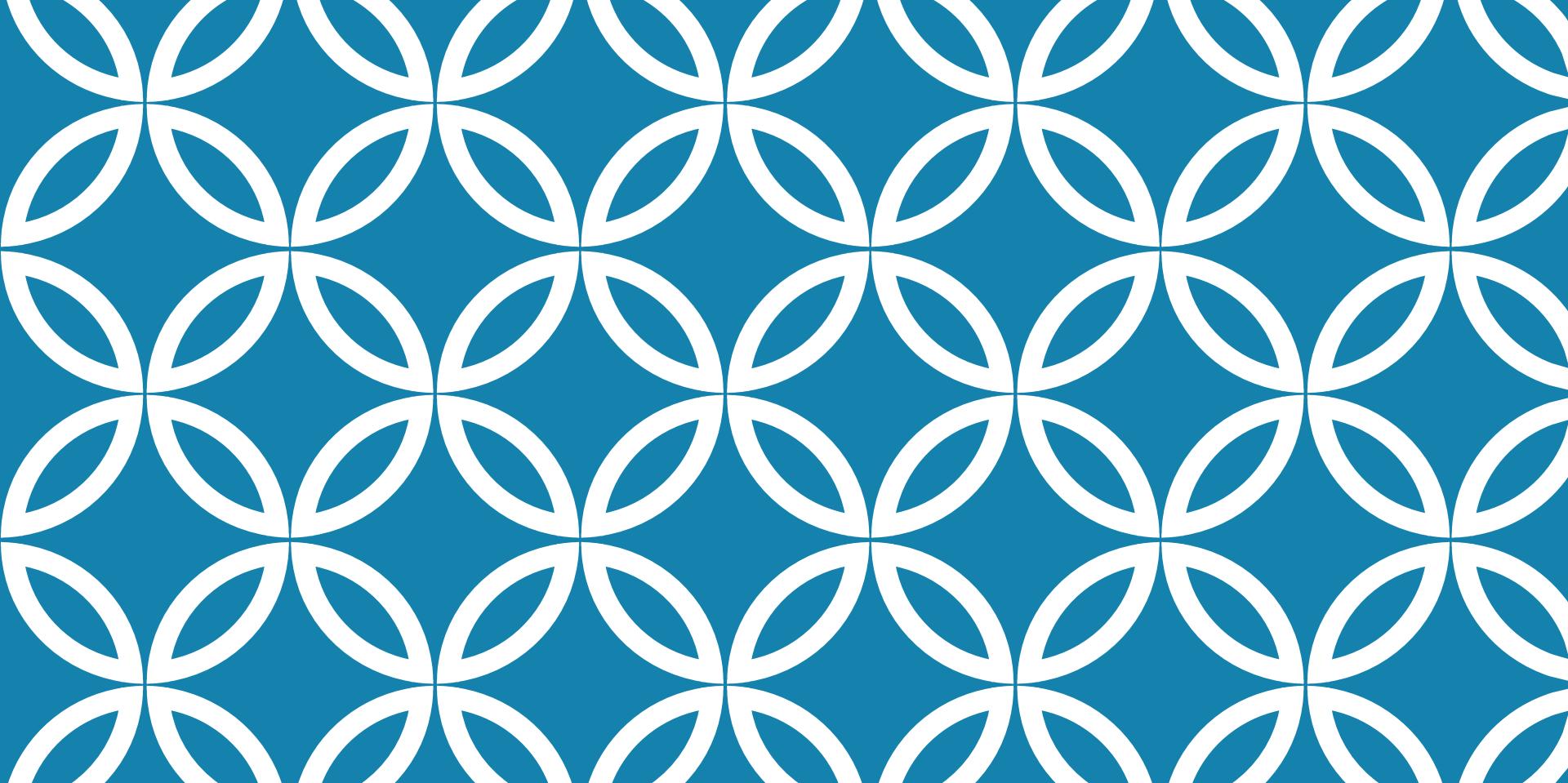
- let abstraction and implementation vary independently (**Bridge**)

## Optimize

- use an alternative algorithm to implement behavior (**Strategy**)

## Alter

- change behavior when object's state changes (**State**)



# PROXY PATTERN

---

# LOADING "HEAVY" OBJECTS

Document Editor that can embed multimedia objects

- MM objects are expensive to create ⇒ opening of document slow
- avoid creating expensive objects
  - they are not all necessary as they are not all visible at the same time

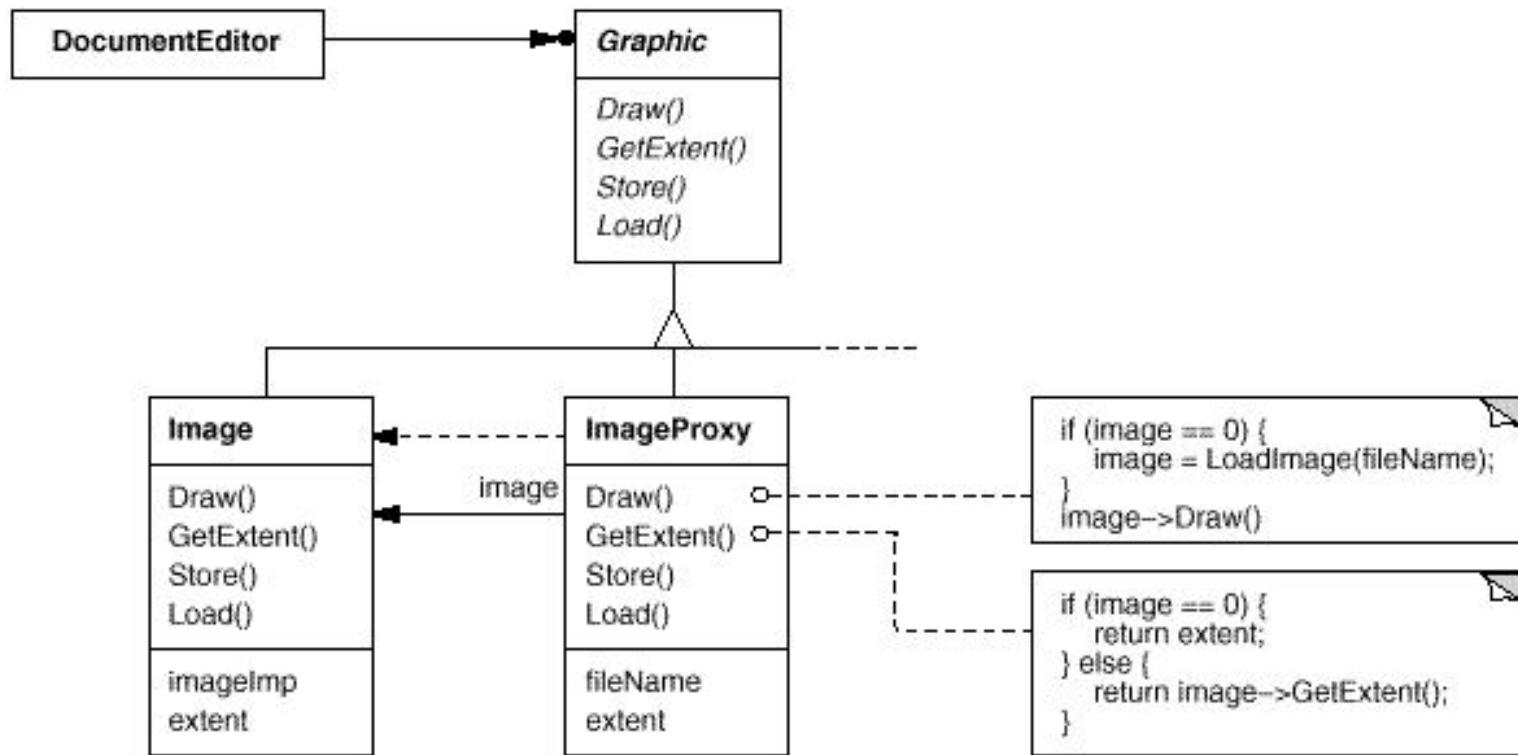
Creating each expensive object **on demand** !

- i.e. when image has to be displayed

What should we put instead?

- hide the fact that we are "lazy"!
- don't complicate the document editor!

# IDEA: USE A PLACEHOLDER!



- create only when needed for drawing
- keeps information about the dimensions (extent)

# BASIC ASPECTS

**Definition:** **proxy** (n. pl **prox-ies**) The agency for a person who acts as a substitute for another person, authority to act for another

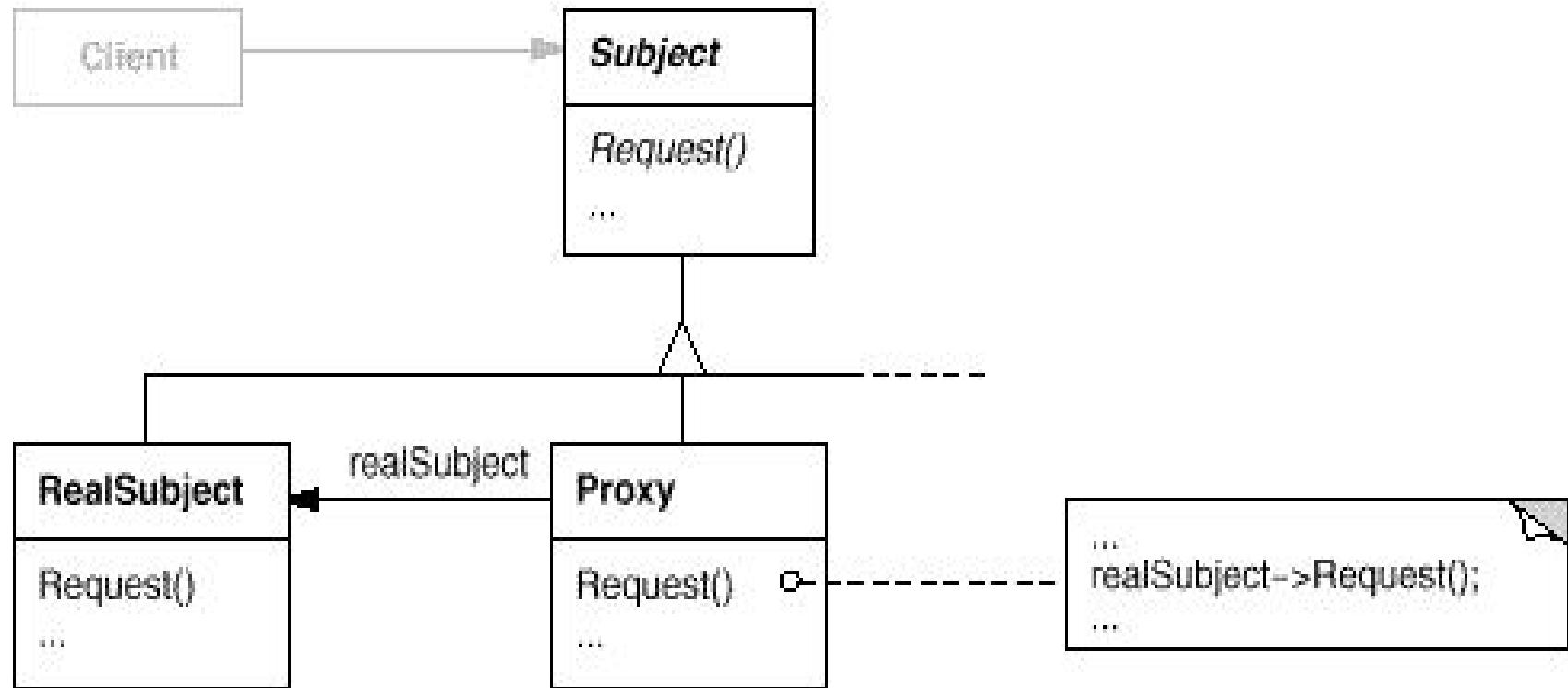
## Intent

- provide a surrogate or placeholder for another object to control access to it

**Applicability:** whenever there is a need for a more *flexible* or *sophisticated* reference to an object than a simple pointer

- remote proxy ... if real object is “far away”
- virtual proxy ... if real object is “expensive”
- protection proxy ... if real object is “vulnerable”
- enhancement proxies (*smart pointers*)
  - prevent accidental delete of objects (counts references)

# STRUCTURE



# PARTICIPANTS

## Proxy

- maintains a reference that lets the proxy access the real subject.
- provides an interface identical to Subject's
  - so that proxy can be substituted for the real subject
  - controls access to the real subject
  - may be responsible for creating or deleting it

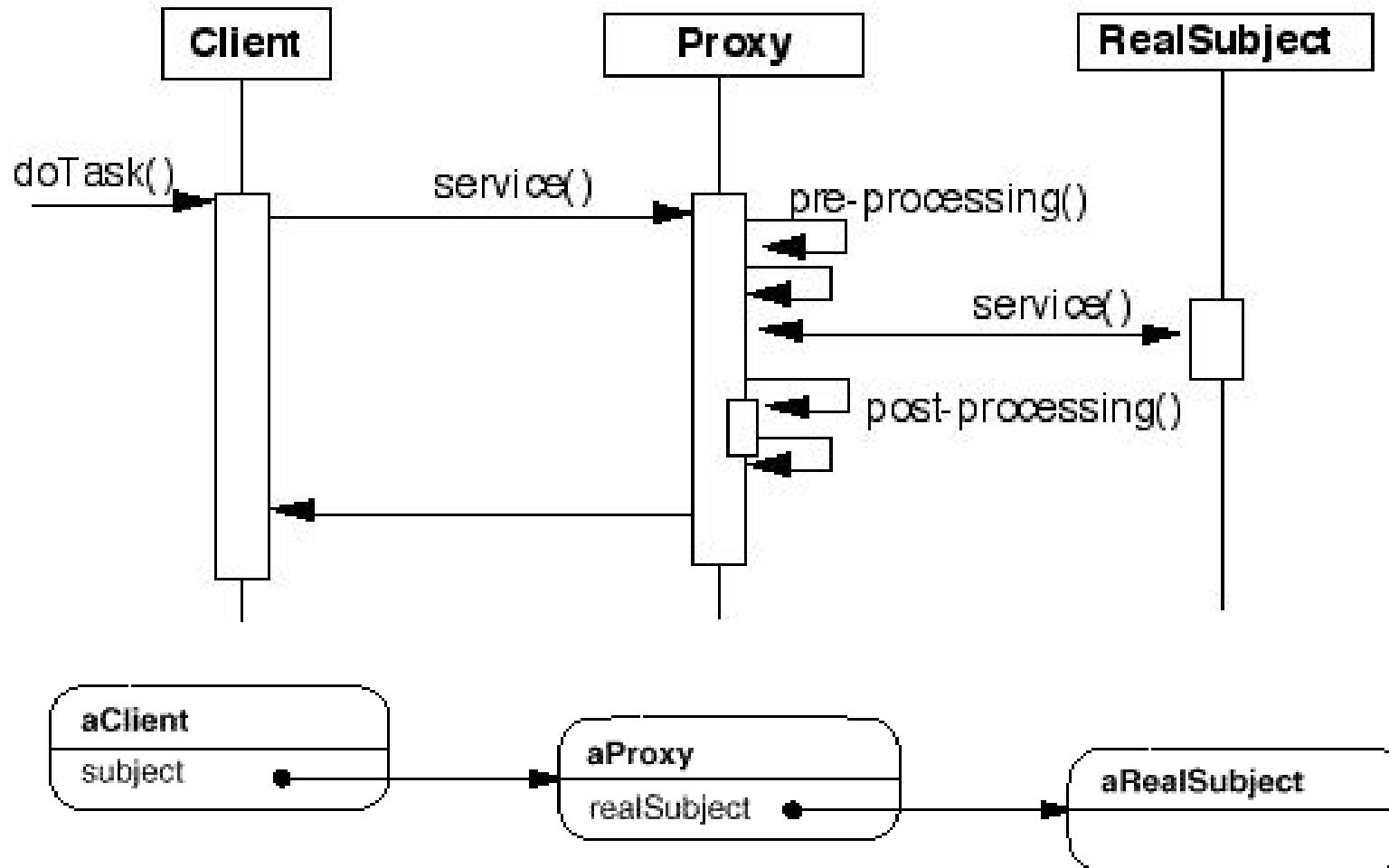
## Subject

- defines the common interface for RealSubject and Proxy

## RealSubject

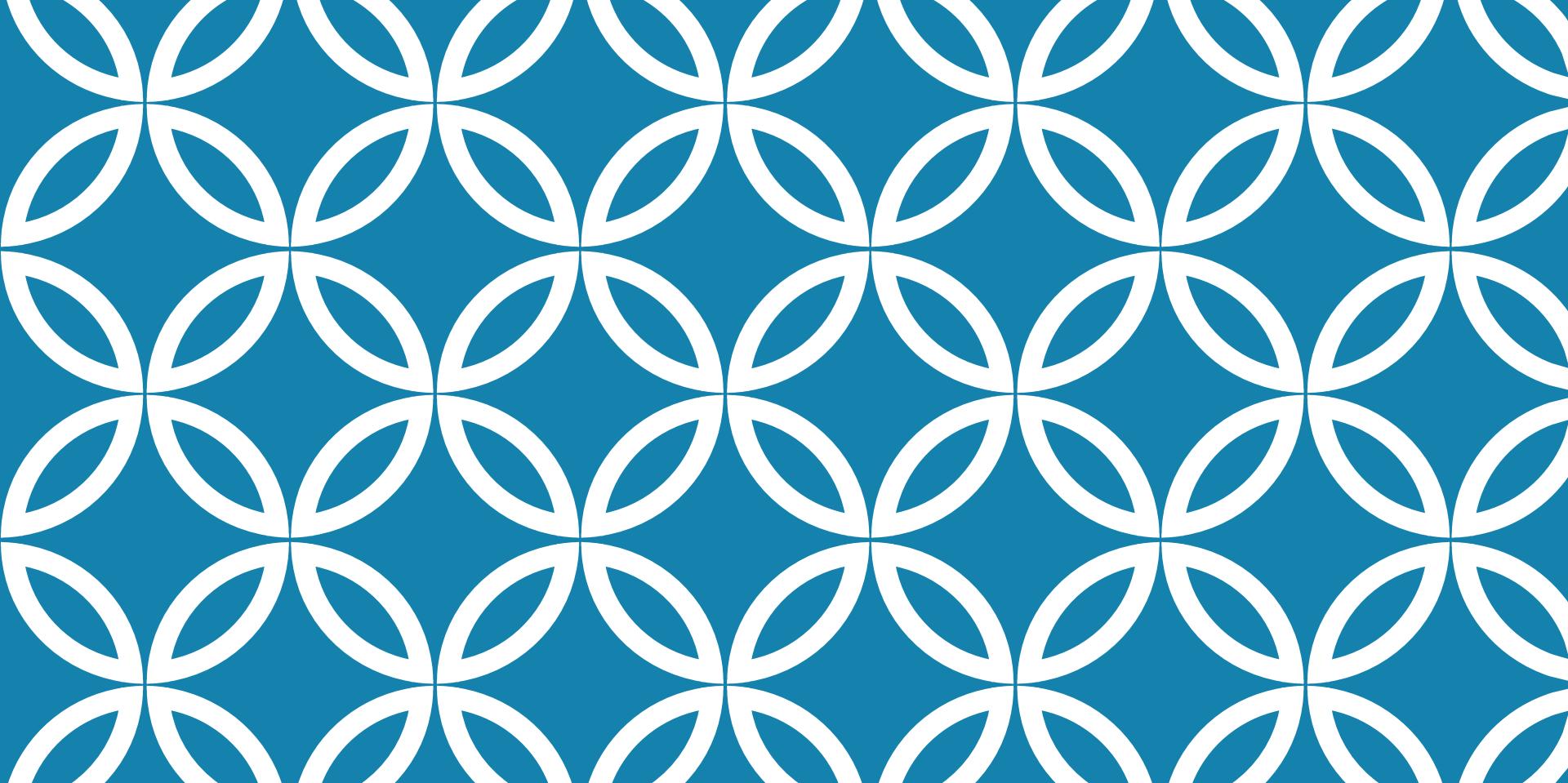
- defines the real object that the proxy holds place for

# COLLABORATIONS



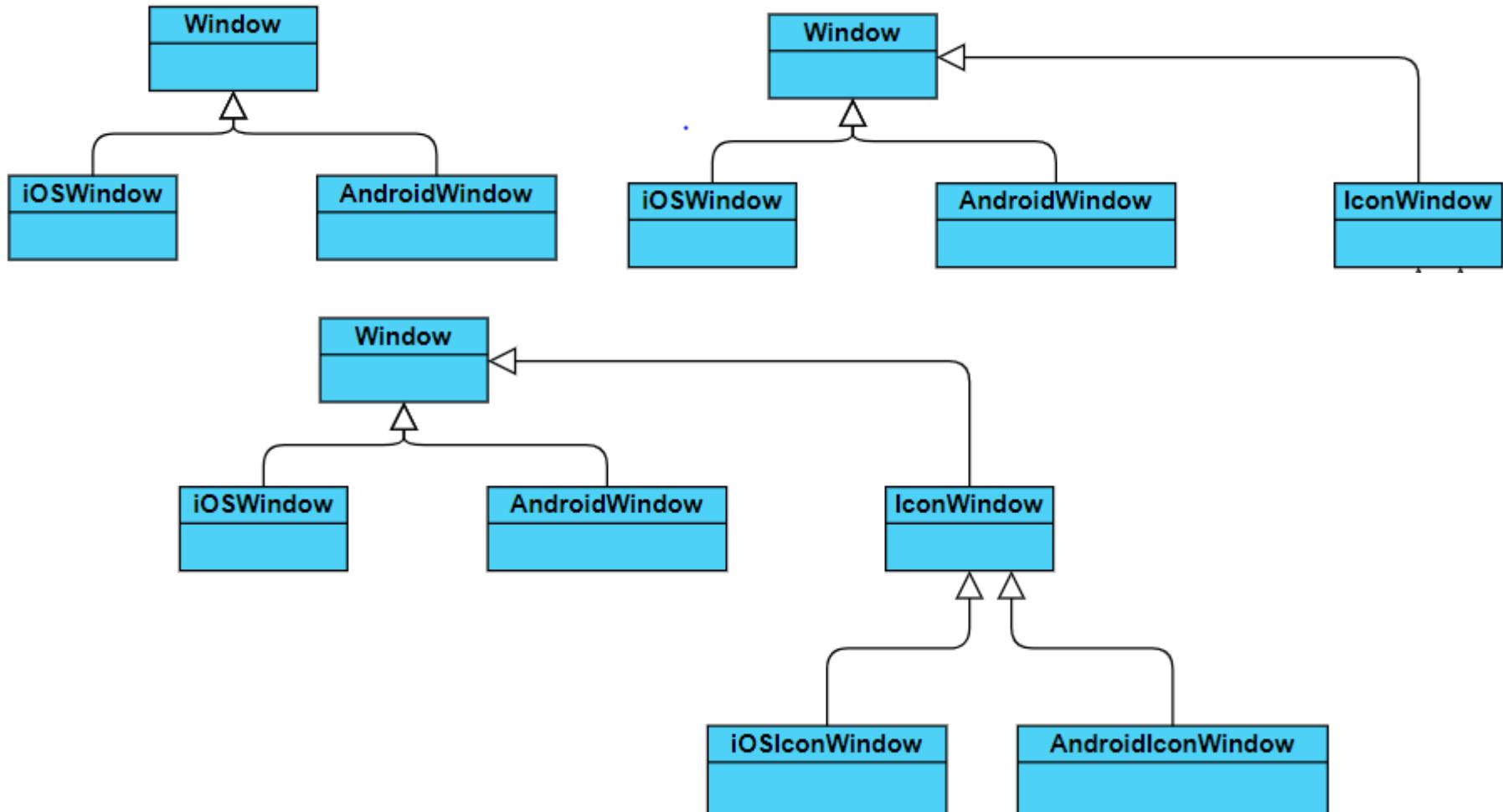
# CONSEQUENCES

- introduces a level of indirection
- used differently depending on the kind of proxy:
  - hide different address space (remote p.)
  - creation on demand (virtual p.)
  - allow additional housekeeping activities  
(protection, smart pointers)



# BRIDGE PATTERN

# INHERITANCE THAT LEADS TO EXPLOSION!



# BASIC ASPECTS OF BRIDGE PATTERN

## Intent

- decouple an abstraction from its implementation
- allow implementation to vary independently from its abstraction
- abstraction defines and implements the interface
  - all operations in abstraction call methods from its implementation obj.

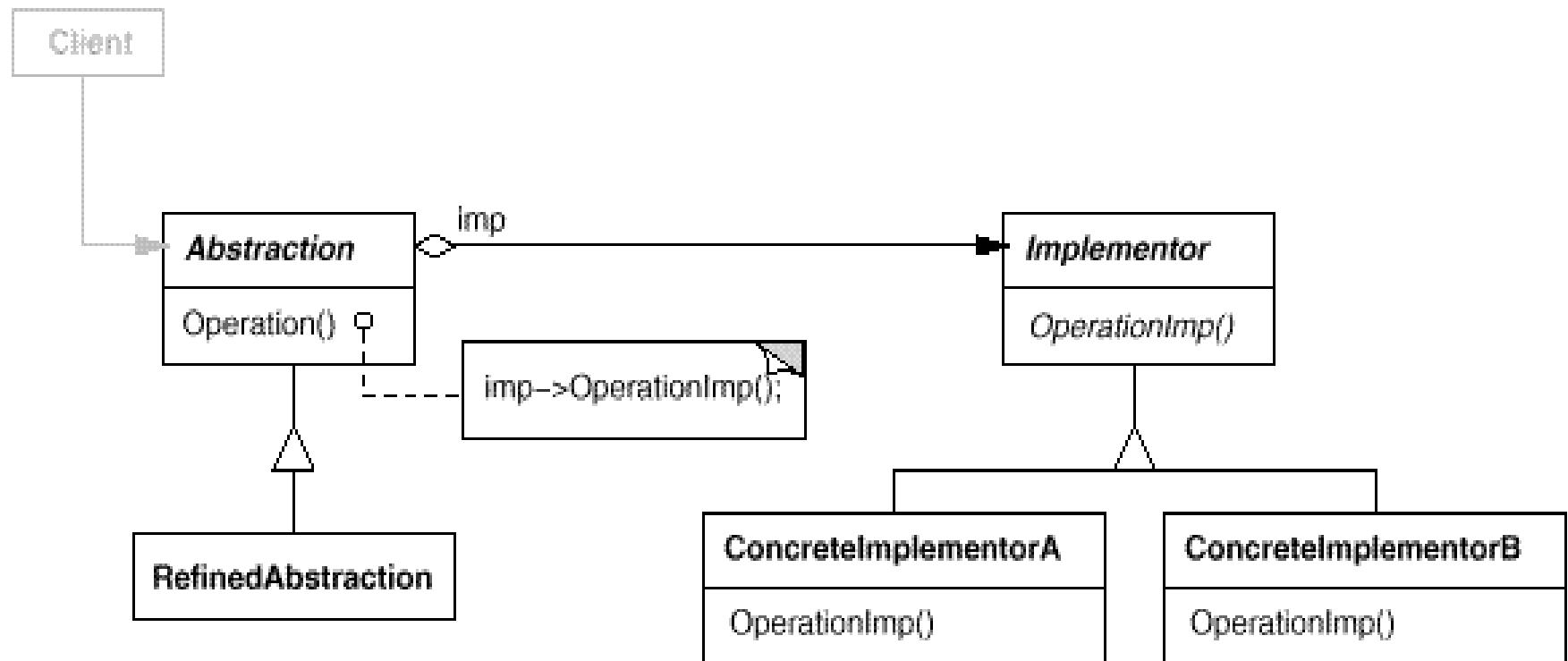
In the Bridge pattern ...

- ... an abstraction can use different implementations
- ... an implementation can be used in different abstractions

# APPLICABILITY

- Avoid permanent binding btw. an abstraction and its implementation
- Abstractions and their implementations should be *independently extensible* by subclassing
- Hide the implementation of an abstraction completely from clients
  - their code should not have to be recompiled when the implementation changes
- Share an implementation among multiple objects
  - and this fact should be hidden from the client

# STRUCTURE



# PARTICIPANTS

## Abstraction

- defines the abstraction's interface
- maintains a reference to an object of type Implementor

## Implementor

- defines the interface for implementation classes
- does not necessarily correspond to the Abstraction's interface
- Implementor contains primitive operations,
- Abstraction defines the higher-level operations based on these primitives

## RefinedAbstraction

- extends the interface defines by Abstraction

## ConcreteImplementer

- implements the Implementor interface, defining a concrete impl.

# CONSEQUENCES

Decoupling interface and implementation

- Implementation is **configurable** and **changeable** at run-time
- reduce compile-time dependencies
  - implementation changes do not require Abstraction to recompile

Improved extensibility

- extend by subclassing independently Abstractions and Implementations

Hiding implementation details from clients

- shield clients from implementations details
  - e.g. sharing implementor objects together with reference counting

# IMPLEMENTATION

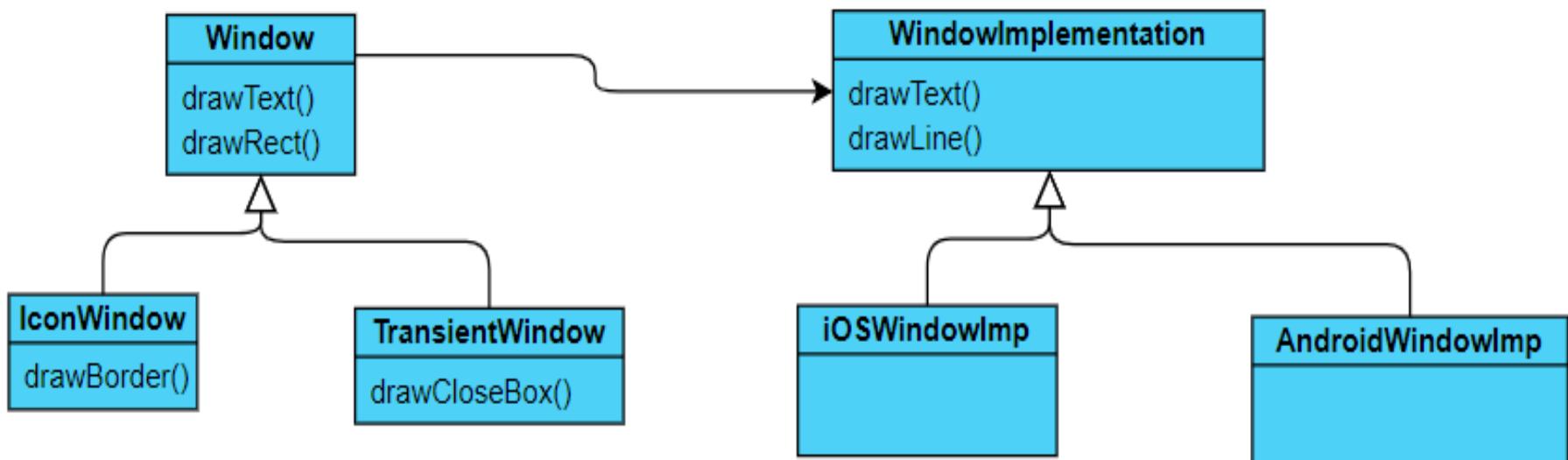
## Only one Implementor

- not necessary to create an abstract implementor class
- degenerate, but useful due to decoupling

## Which Implementor should I use ?

- Variant 1: let Abstraction know all concrete implem. and choose
- Variant 2: choose initially default implem. and change later
- Variant 3: use an Abstract Factory
  - no coupling btw. Abstraction and concrete implem. classes

# WINDOWS EXAMPLE REVISITED



# ADAPTER VS. BRIDGE

## Common features

- promote flexibility by providing a level of indirection to another object.
- involve forwarding requests to this object from an interface other than its own.

## Differences

- **Adapter resolves incompatibilities** between two existing interfaces. No focus on how those interfaces are implemented, nor how they might evolve independently
- **Bridge links an abstraction and its implementations.** It provides a stable interface to clients as it lets you vary the classes that implement it.
- **The Adapter pattern makes things work after they're designed; Bridge makes them work before they are.**

# NEXT TIME

Behavioral DP



# SOFTWARE DESIGN

Behavioural DP

# CONTENT

## Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioural Patterns
  - Observer
  - Strategy
  - State
  - Command
  - Chain of Responsibility

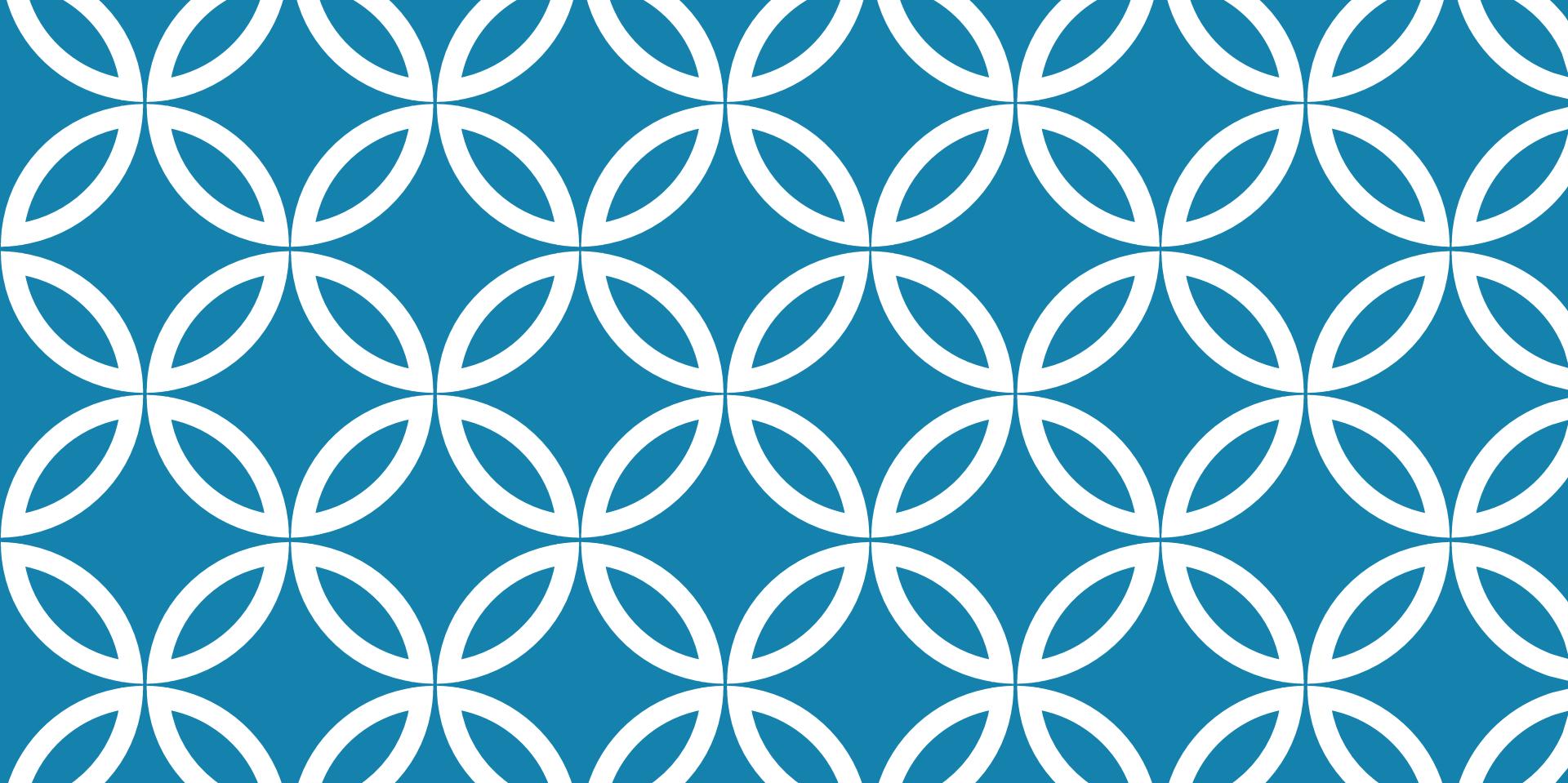
# REFERENCES

Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.

Univ. of Timisoara Course materials

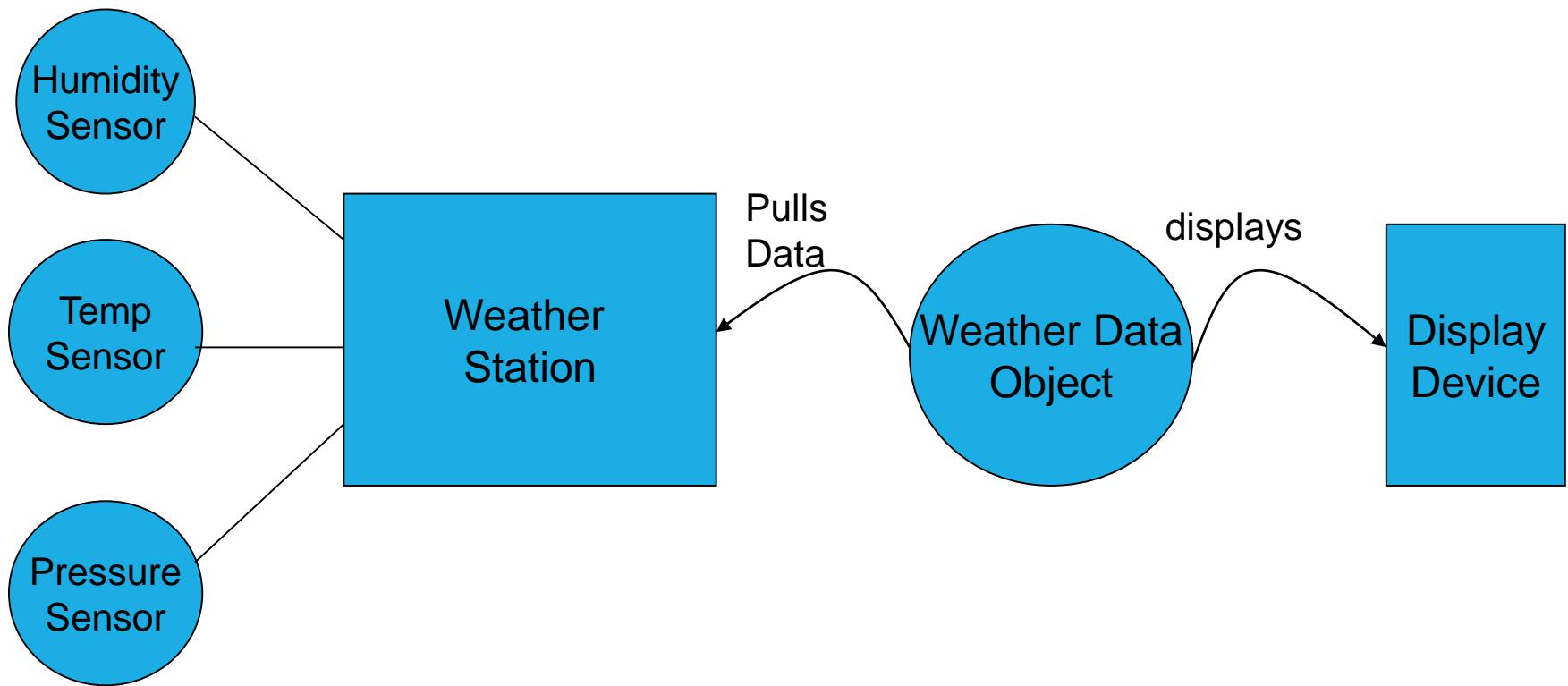
# BEHAVIOURAL PATTERNS

- are concerned with **algorithms** and the **assignment of responsibilities** between objects.
- describe **patterns of communication** between classes/objects.
- **class** patterns use inheritance to distribute behavior between classes.
- **object** patterns use object composition rather than inheritance.

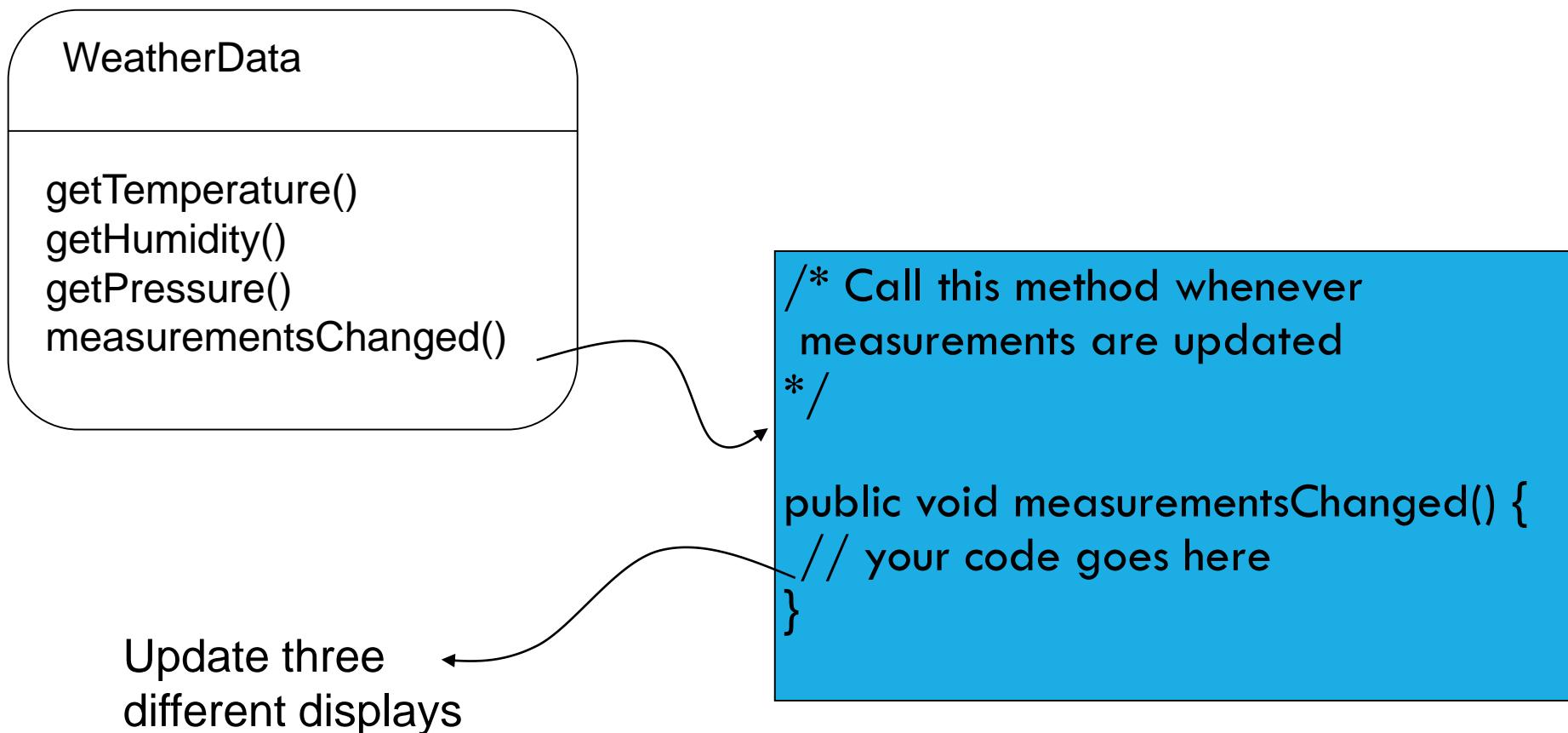


# OBSERVER PATTERN

# WEATHER MONITORING APPLICATION



# WHAT NEEDS TO BE DONE?



# PROBLEM SPECIFICATION

- WeatherData class has three getter methods
- measurementsChanged() method called whenever there is a change
- Three display methods needs to be supported:
  - current conditions,
  - weather statistics and
  - simple forecast
- System should be expandable

# FIRST CUT AT IMPLEMENTATION

```
public class WeatherData {  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
        currentConditionsDisplay.update (temp, humidity,  
pressure);  
        statisticsDisplay.update (temp, humidity, pressure);  
        forecastDisplay.update (temp, humidity, pressure);  
    }  
    // other methods  
}
```

# FIRST CUT AT IMPLEMENTATION

```
public class WeatherData {  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
        currentConditionsDisplay.update (temp, humidity,  
                                         pressure);  
        statisticsDisplay.update (temp, humidity, pressure);  
        forecastDisplay.update (temp, humidity, pressure);  
    }  
    // other methods  
}
```

Area of change which can be managed better by encapsulation

By coding to concrete implementations there is no way to add additional display elements without making code change <sup>10</sup>

# BASIS FOR OBSERVER PATTERN

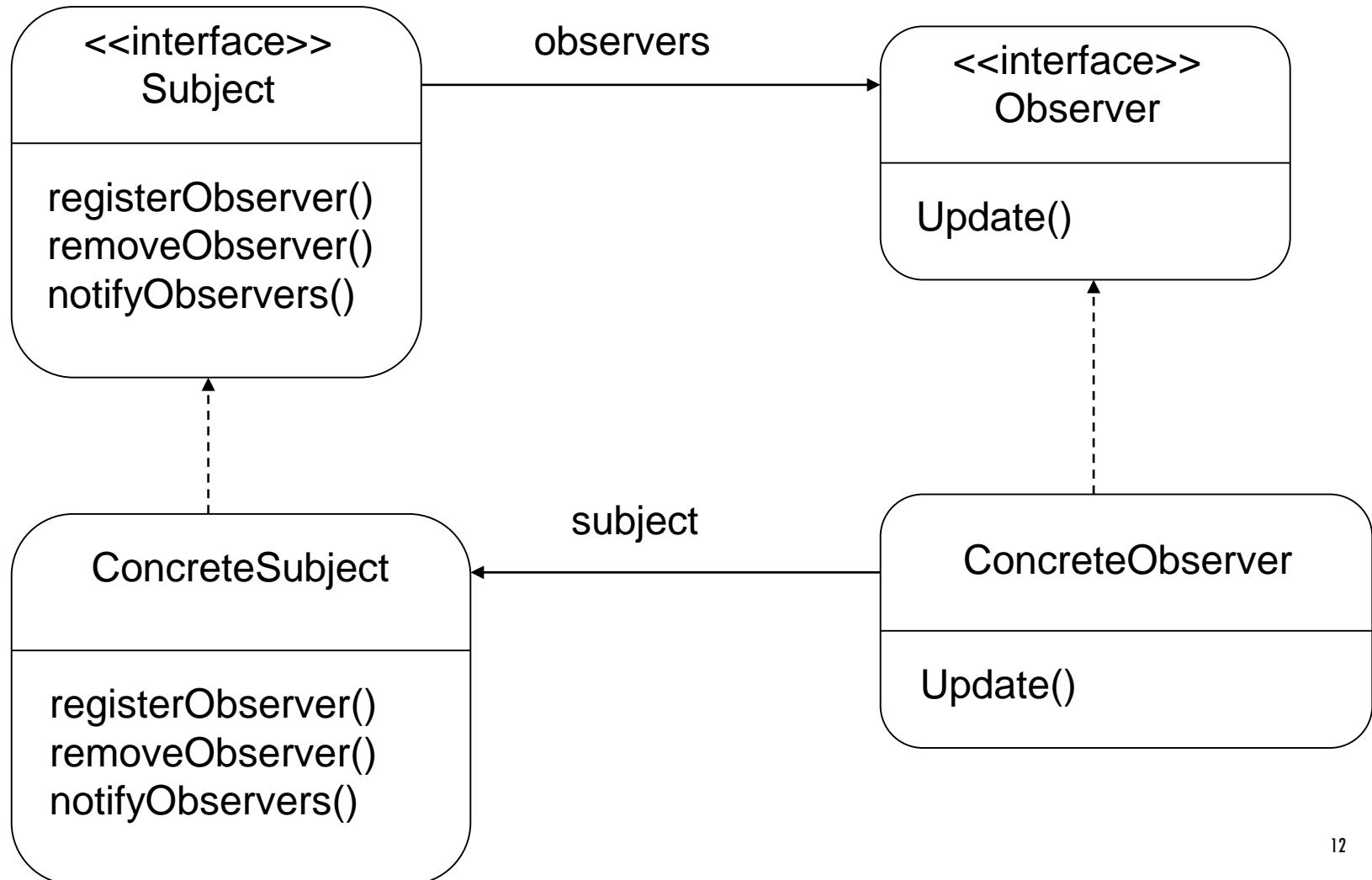
Fashioned after the publish/subscribe model

Works off similar to any subscription model

- Buying newspaper
- Magazines
- List servers

**The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

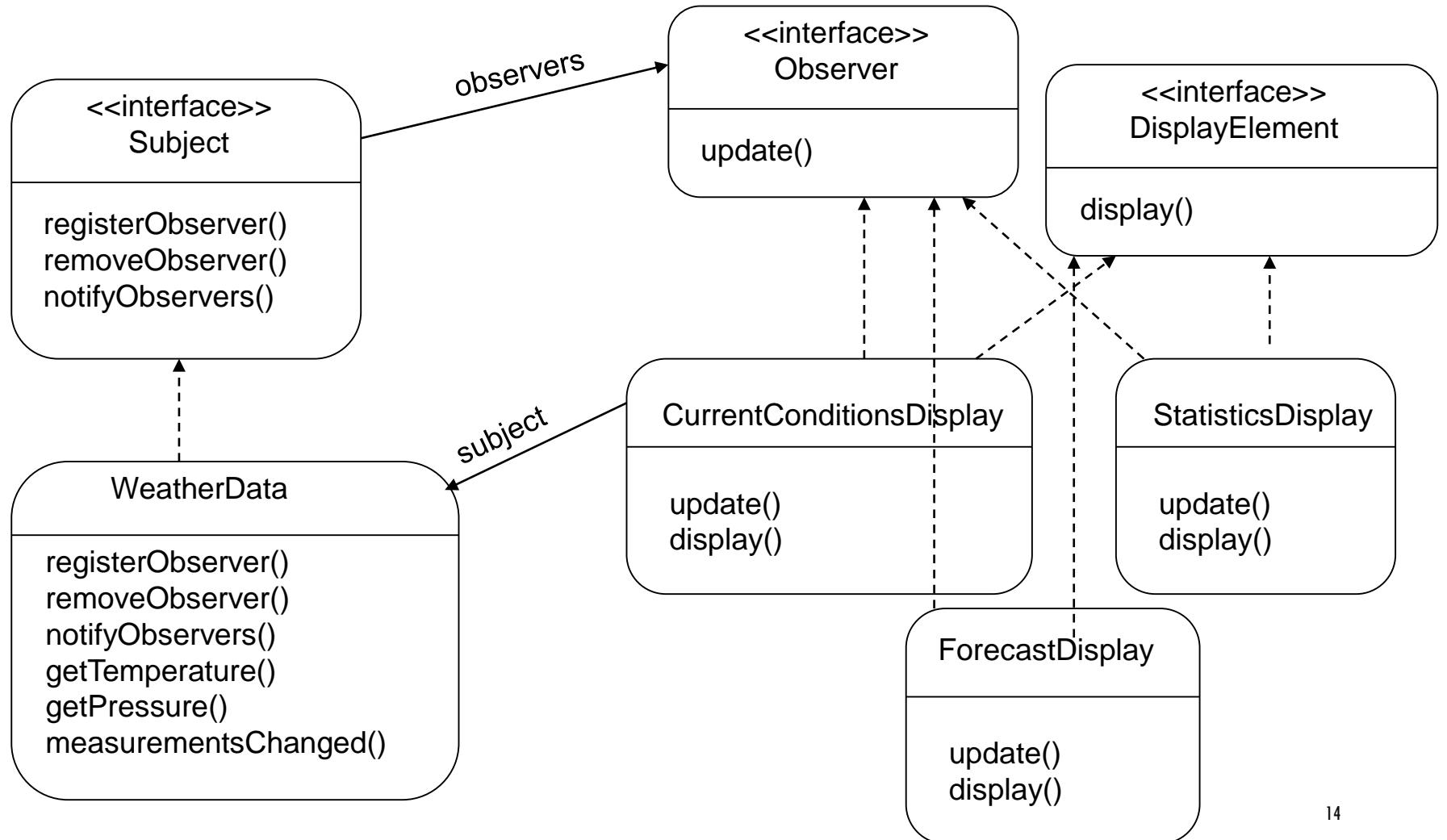
# OBSERVER PATTERN – CLASS DIAGRAM



# POWER OF LOOSE COUPLING

- The only thing that the subject knows about an observer is that it implements an interface
- Observers can be added at any time and subject need not be modified to add observers
- Subjects and observers can be reused or modified without impacting the other [as long as they honor the interface commitments]

# WEATHER DATA REVISITED



# WEATHER DATA INTERFACES

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}  
  
public interface Observer {  
    public void update(float temp, float humidity, float  
pressure);  
}  
  
public interface DisplayElement {  
    public void display();  
}
```

# IMPLEMENTING SUBJECT INTERFACE

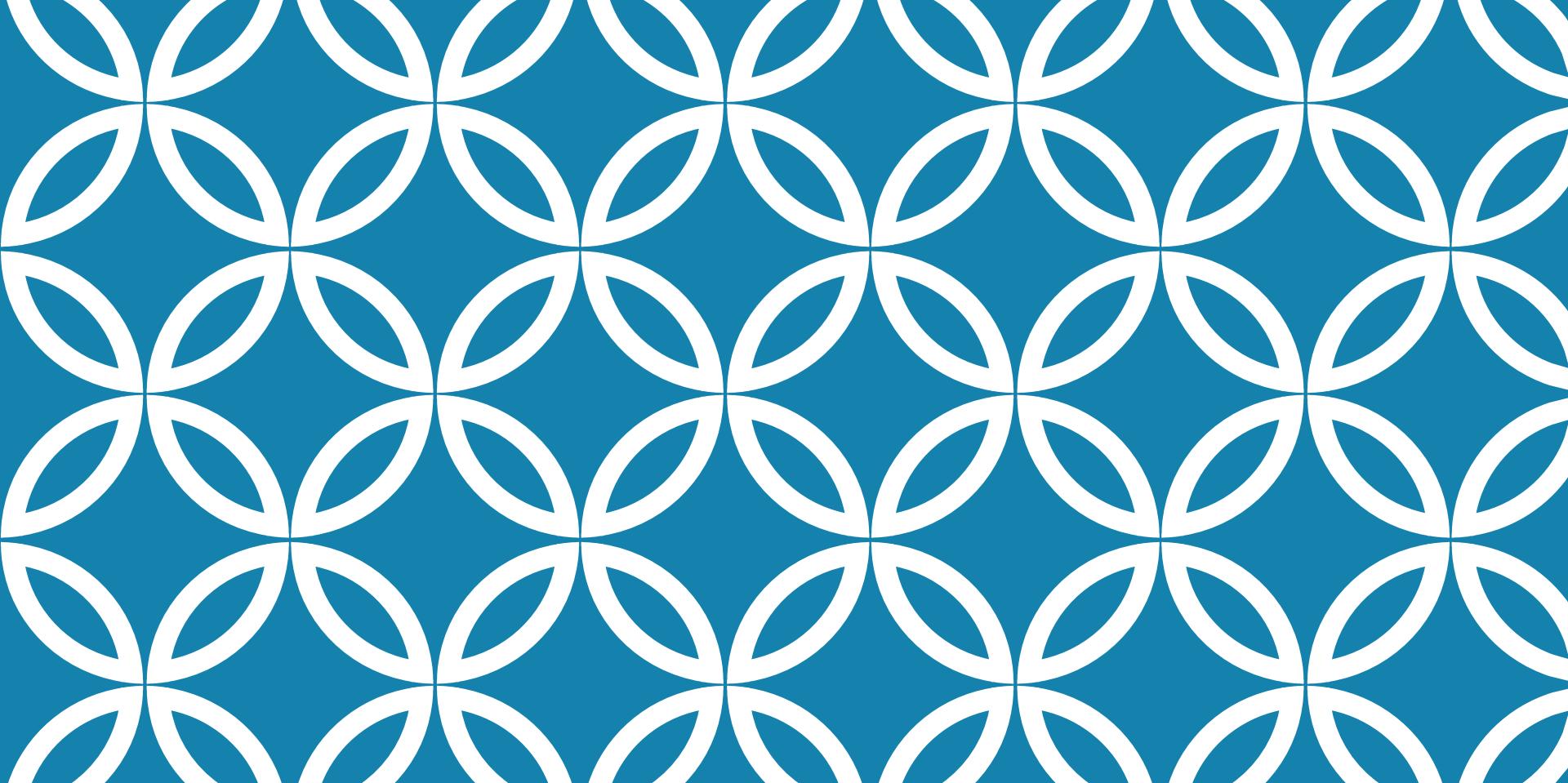
```
public class WeatherData implements Subject {  
    private ArrayList observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList();  
    }  
    ...}
```

# REGISTER AND UNREGISTER

```
public void registerObserver(Observer o) {  
    observers.add(o);  
}  
  
public void removeObserver(Observer o) {  
    int i = observers.indexOf(o);  
    if (i >= 0) {  
        observers.remove(i);  
    }  
}
```

# NOTIFY METHODS

```
public void notifyObservers() {  
    for (int i=0; i<observers.size(); i++)  
    {  
        Observer observer = (Observer)observers.get(i);  
        observer.update(temperature, humidity, pressure);  
    }  
}  
  
public void measurementsChanged() {  
    notifyObservers()  
}
```



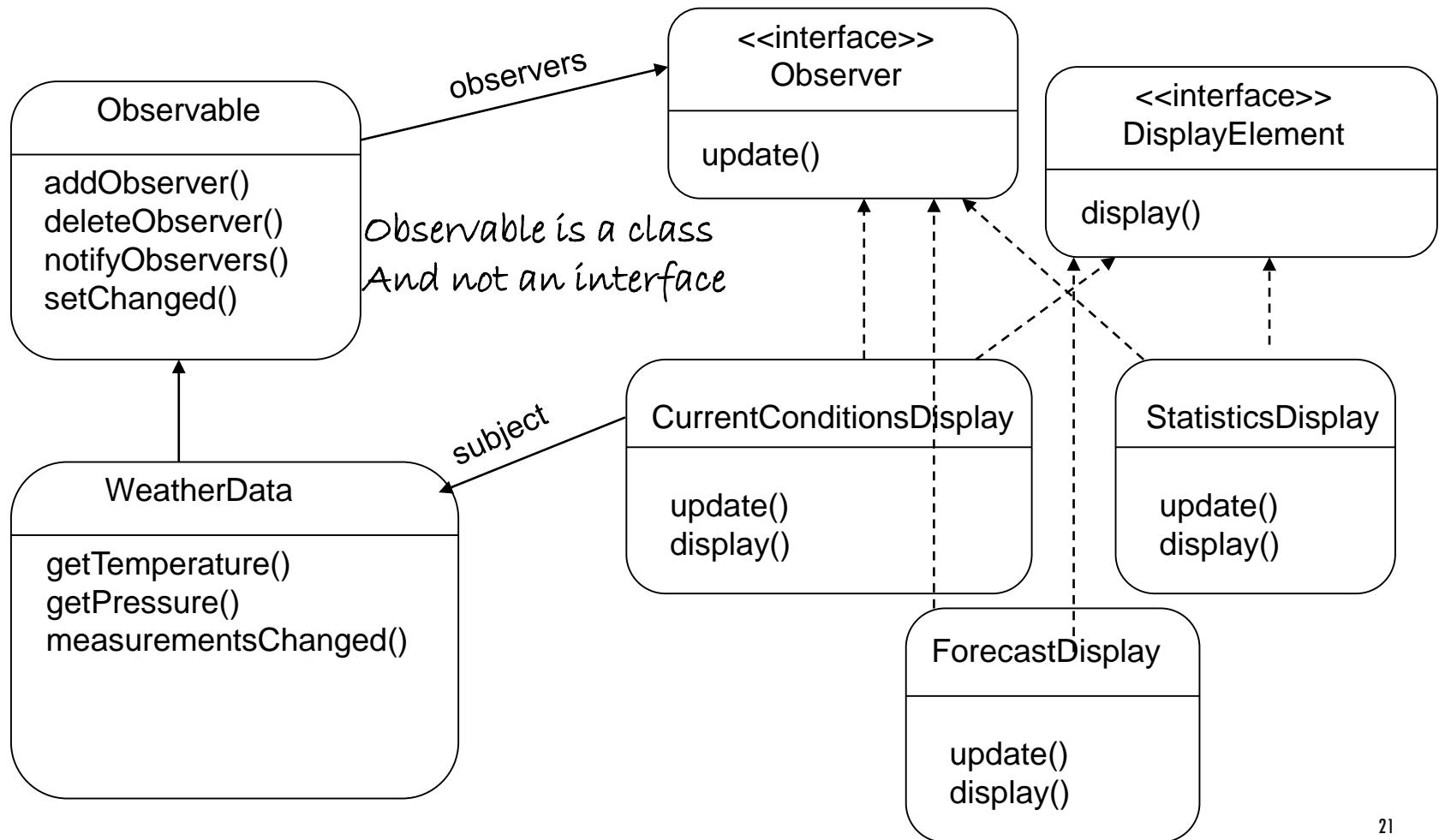
# OBSERVER PATTERN

More analysis

# PUSH OR PULL

- The notification approach used so far **pushes** all the state to all the observers
- One can also just send a notification that some thing has changed and let the observers **pull** the state information
- Java observer pattern support has built in support for both push and pull in notification
  - **java.util.Observable**
  - **java.util.Observer**

# JAVA OBSERVER PATTERN – WEATHER DATA



# PROBLEMS WITH JAVA IMPLEMENTATION

Observable is a class

- You have to subclass it
- You cannot add observable behavior to an existing class that already extends another superclass
- **You have to program to an implementation – not interface**

Observable protects crucial methods

- Methods such as setChanged() are protected and not accessible unless one subclasses Observable.
- **You cannot favor composition over inheritance.**

You may have to write your own observer interface if Java utilities don't work for your application

# CHANGING THE "GUTS" OF AN OBJECT ...

## Control

- "shield" the implementation from direct access (**Proxy**)

## Decouple

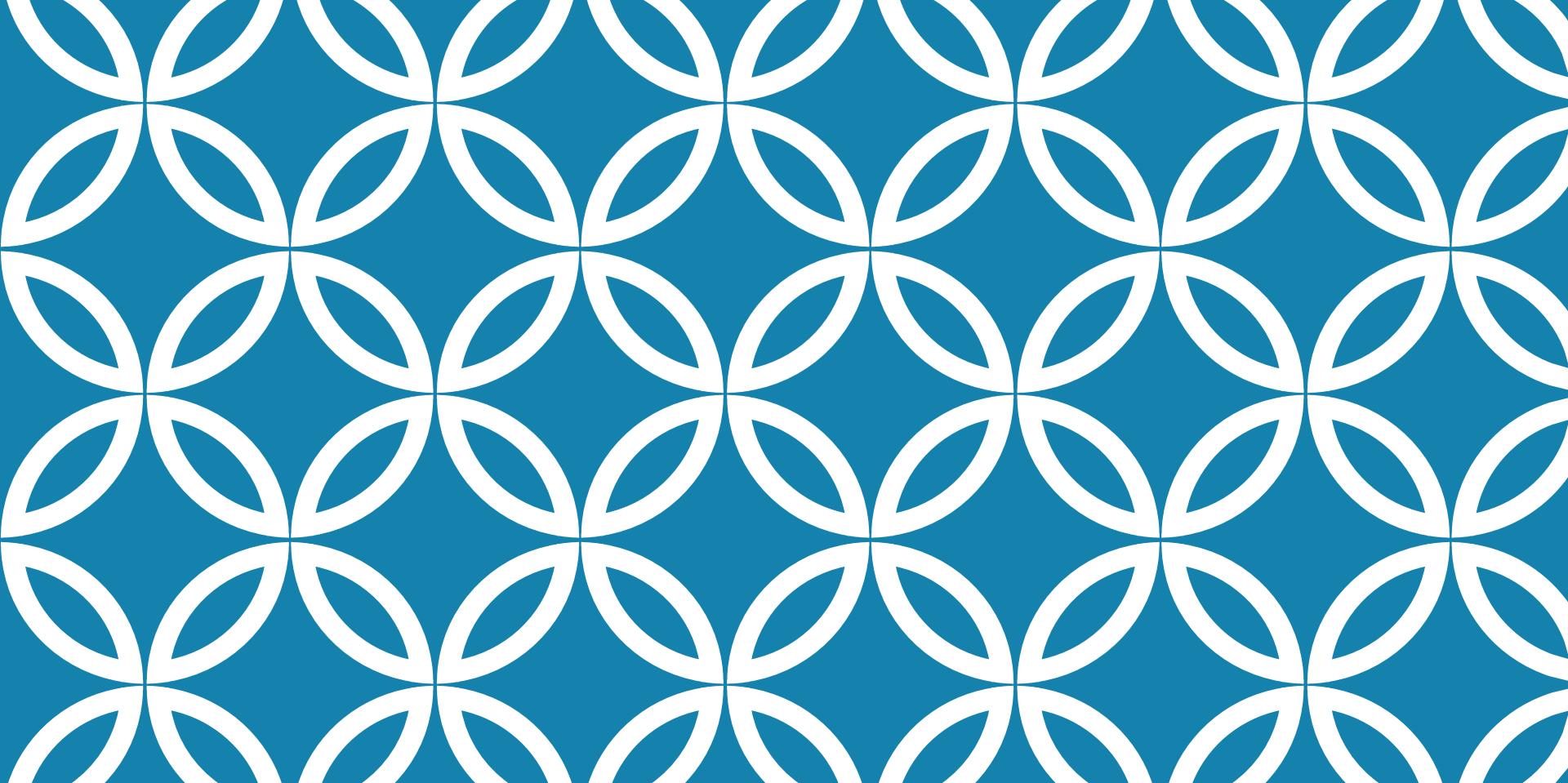
- let abstraction and implementation vary independently (**Bridge**)

## Optimize

- use an alternative algorithm to implement behavior (**Strategy**)

## Alter

- change behavior when object's state changes (**State**)



# STRATEGY PATTERN

# JAVA LAYOUT MANAGERS

GUI container classes in Java

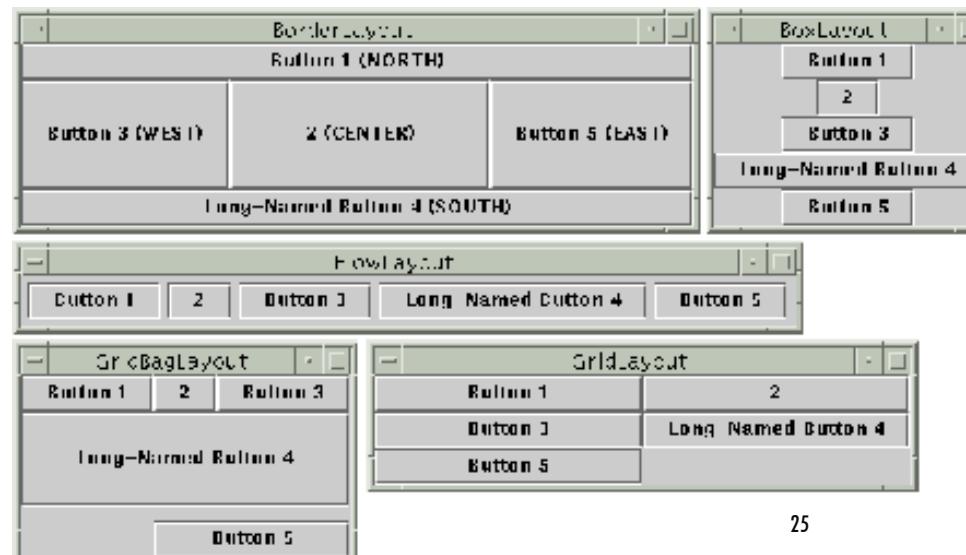
- frames, dialogs, applets (top-level)
- panels (intermediate)

Each container class has a layout manager

- determine the size and position of components
- 20 types of layouts
- ~40 container-types
- imagine to combine them freely by inheritance

Consider also sorting...

- open-ended number of sorting criteria



# BASIC ASPECTS

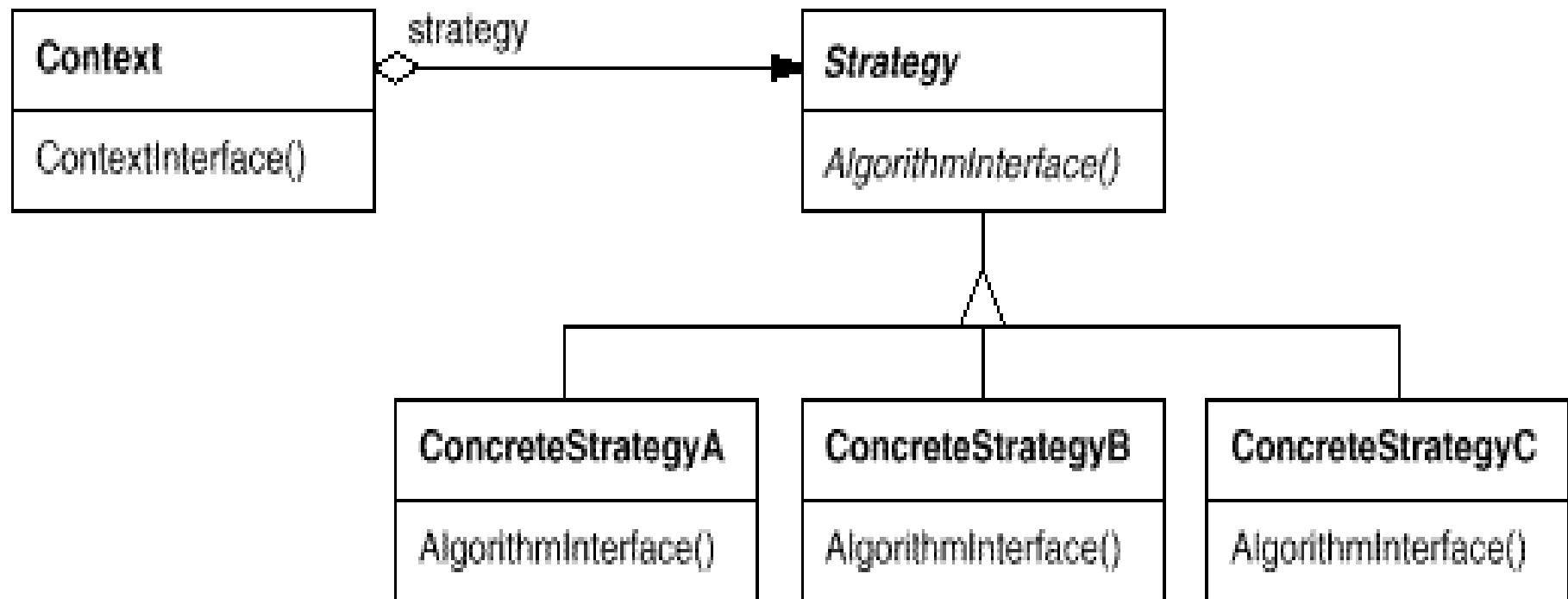
## Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Let the algorithm vary independently from clients that use it

## Applicability

- You need different variants of an algorithm
- An algorithm uses data that clients shouldn't know about
  - avoid exposing complex, algorithm-specific data structures
- Many related classes differ only in their behavior
  - configure a class with a particular behavior

# STRUCTURE



# PARTICIPANTS

## Strategy

- declares an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a `ConcreteStrategy`

## ConcreteStrategy

- implements the algorithm using the `Strategy` interface

## Context

- configured with a `ConcreteStrategy` object
- may define an interface that lets `Strategy` objects to access its data

# CONSEQUENCES

## Families of related algorithms

- usually provide different implementations of the same behavior
- choice decided by time vs. space trade-offs

## Alternative to subclassing

- see examples with layout managers
- We still subclass the strategies...

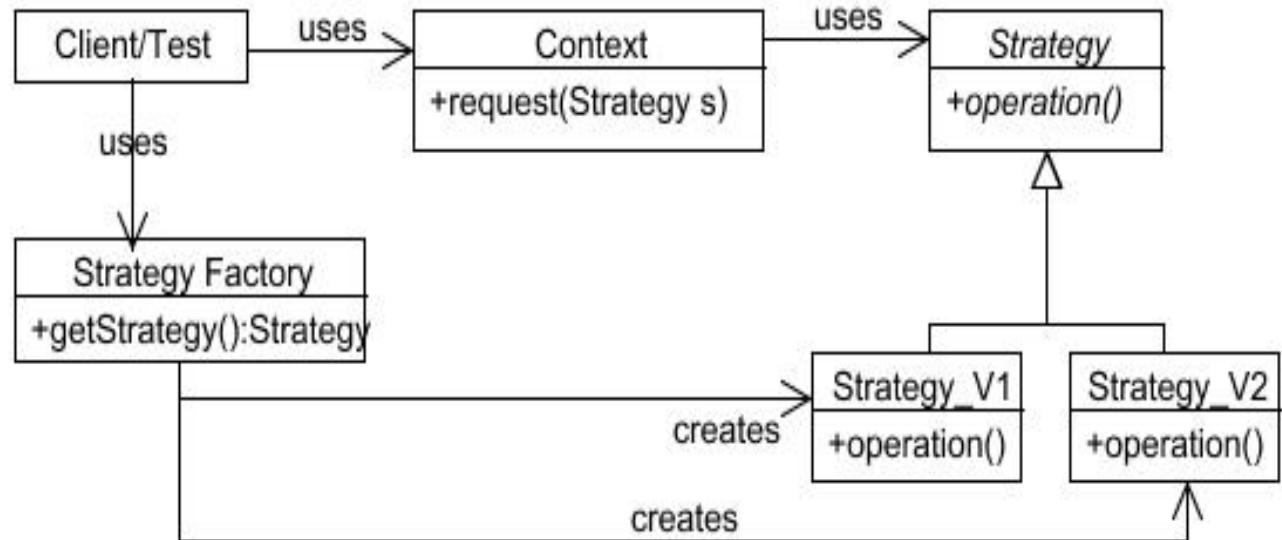
## Eliminates conditional statements

- many conditional statements → "invitation" to apply Strategy!

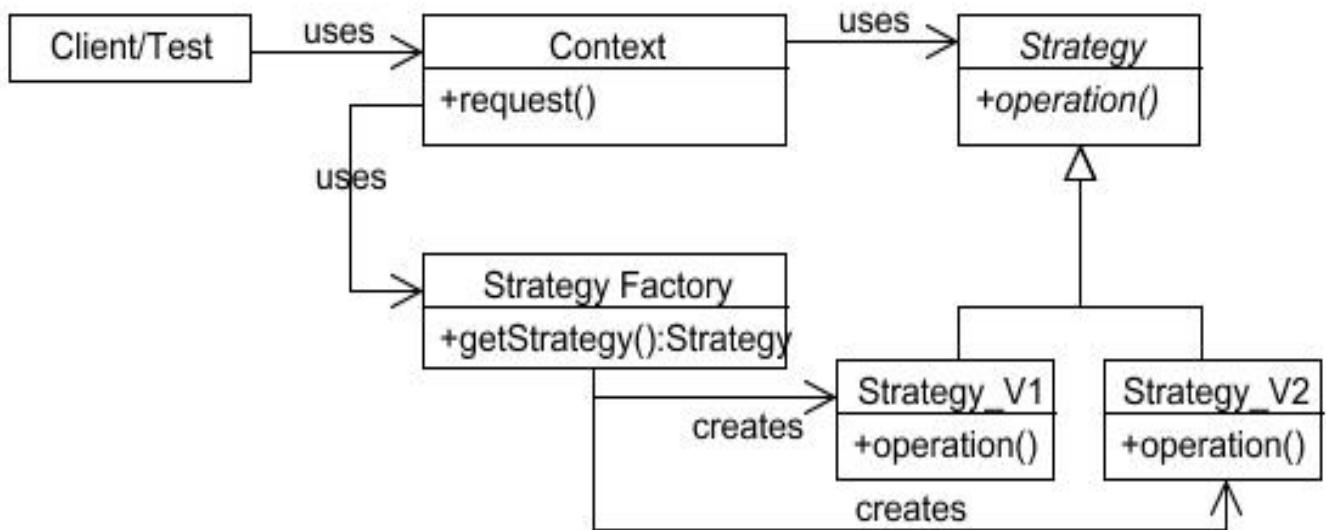
# ISSUES

Who chooses  
the strategy?

Client



Context



# IMPLEMENTATION

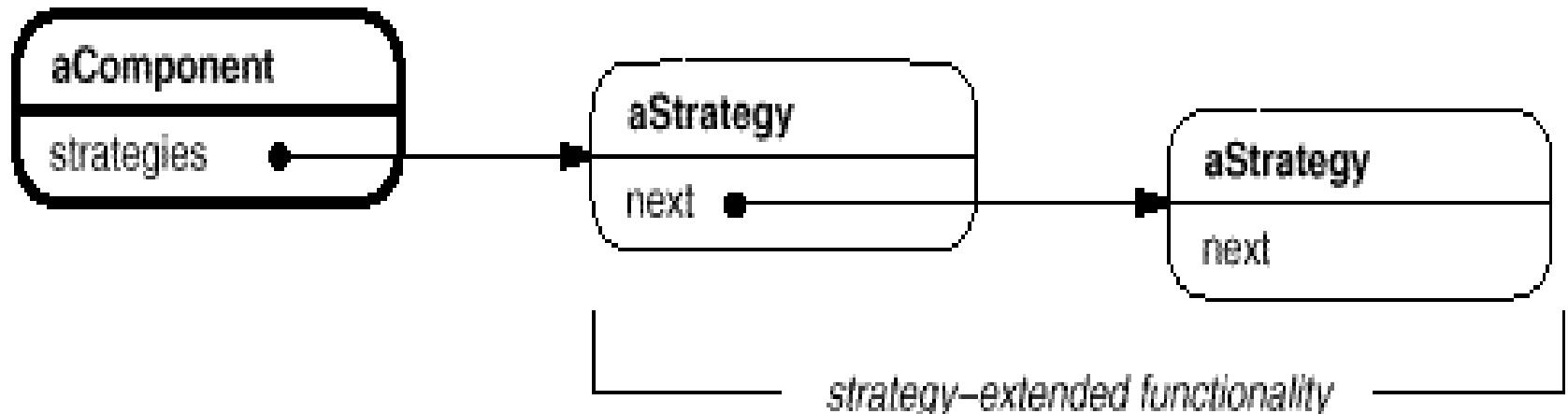
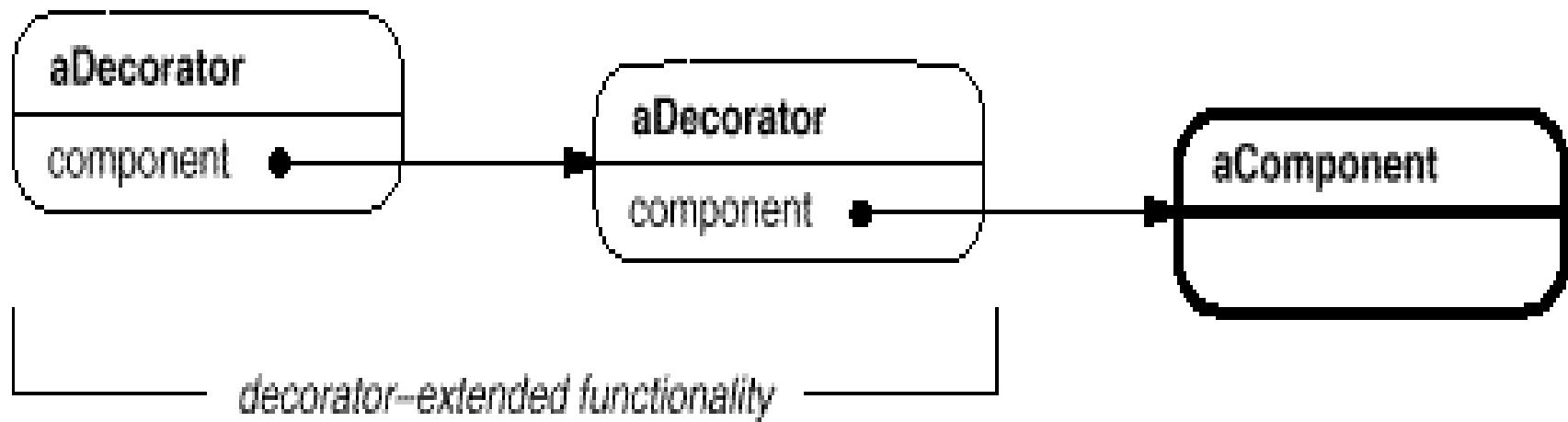
How does data flow between Context and Strategies?

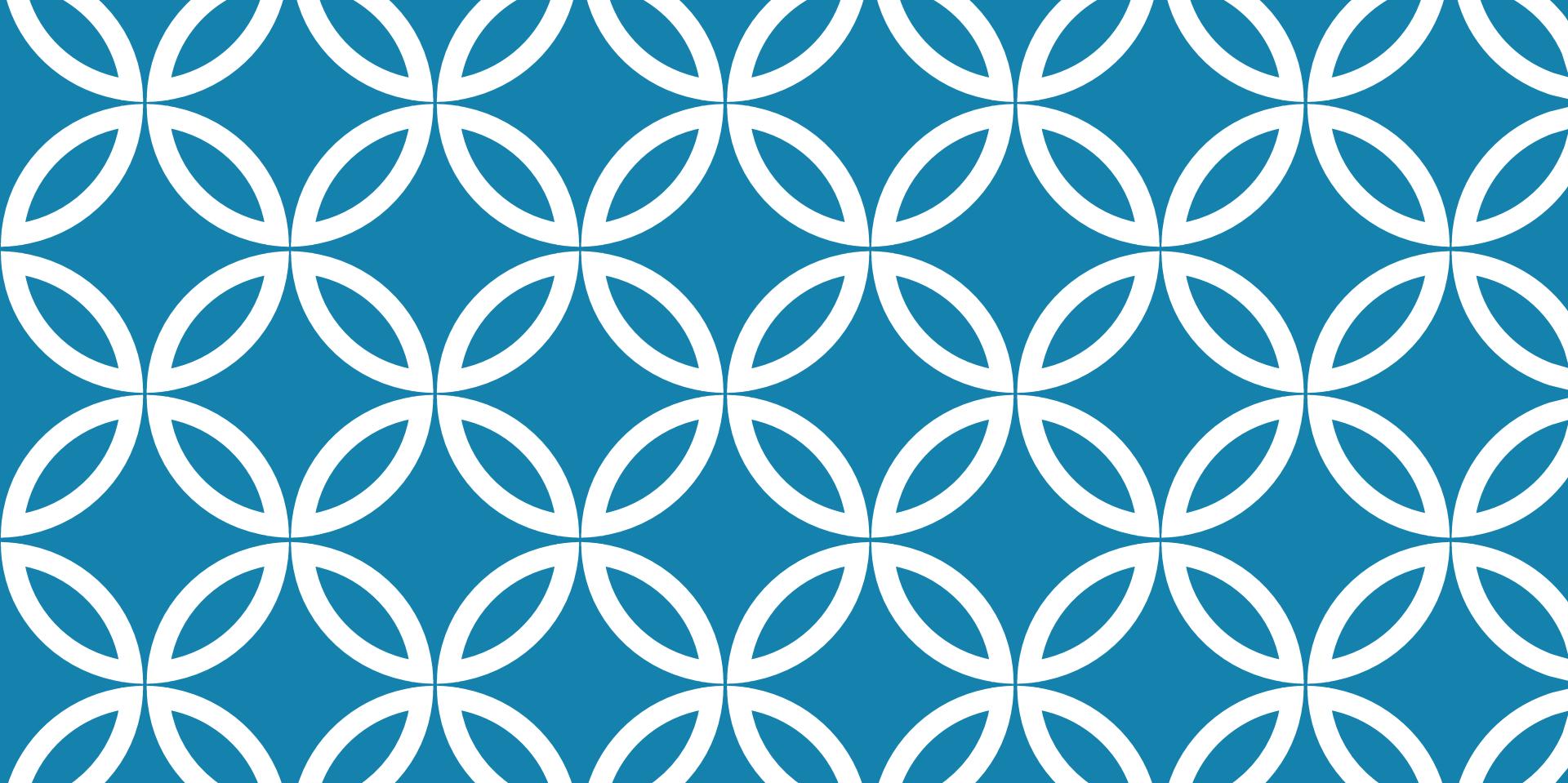
- **Approach 1:** take data to the strategy
  - decoupled, but might be inefficient
- **Approach 2:** pass Context itself and let strategies take data
  - Context must provide a more comprehensive access to its data => more coupled
- In Java, the strategy hierarchy might be **inner classes**

Making Strategy object optional

- provide Context with default behavior
  - if default used no need to create Strategy object
- don't have to deal with Strategy unless you don't like the default behavior

# DECORATOR VS. STRATEGY





# STATE PATTERN

---

# EXAMPLE: SPOP

SPOP = Simple Post Office Protocol

- used to download emails from server

SPOP supports the following commands:

- USER <username>
- PASS <password>
- LIST
- RETR <message number>
- QUIT

USER & PASS commands

- USER with a username must come first
- PASS with a password or QUIT must come after USER
- If the username and password are valid, the user can use other commands

# SPOP (CONTD.)

## LIST command

- Arguments: a message-number (optional)
- Returns: size of message in octets
  - if message number, returns the size of that message
  - otherwise return size of all mail messages in the mail-box

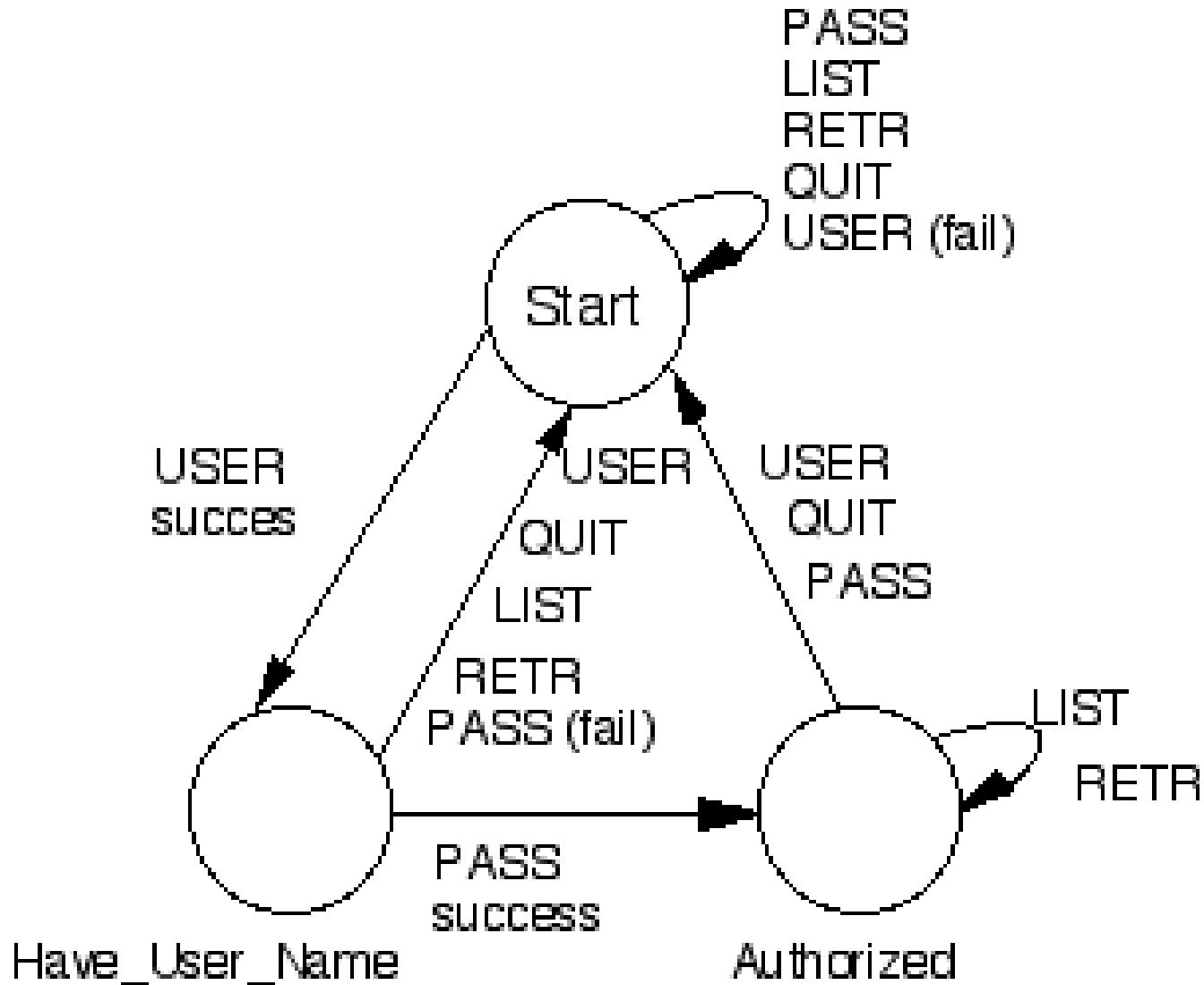
## RETR command

- Arguments: a message number
- Returns: the mail message indicated by that number

## QUIT command

- Arguments: none
- updates mailbox to reflect transactions taken during the transaction state, the logs user out
- if session ends by any method except the QUIT command, the updates are not done

# SPOP STATES



# THE "DEAR, OLD" SWITCHES IN ACTION

Think about adding a new state  
to the protocol...

Why?

- object's behavior depends  
on its state

```
class Spop {
    static final int HAVE_USER_NAME = 2;
    static final int START = 3;
    static final int AUTHORIZED = 4;

    private int state = START;

    String userName;
    String password;

    public void user( String userName ) {
        switch (state) {
            case START:
                this.userName = userName;
                state = HAVE_USER_NAME;
                break;

            case HAVE_USER_NAME:
            case AUTHORIZED:
                endLastSessionWithoutUpdate();
                goToStartState()
        }
    }

    public void pass( String password ) {
        switch (state) {
            case START:
                giveWarningOfIllegalCommand();
            break;
            case HAVE_USER_NAME:
                this.password = password;
                if (validateUser())

```

# BASIC ASPECTS OF STATE PATTERN

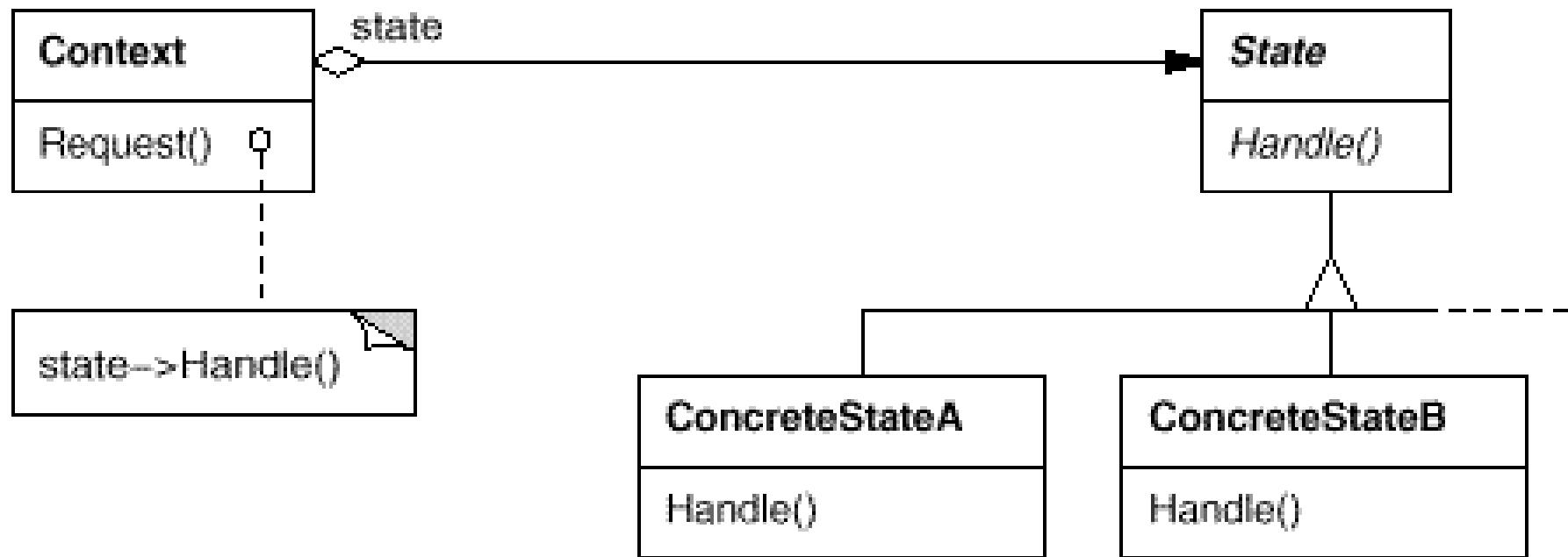
## Intent

- allow an object to alter its behavior when its internal state changes
  - object will appear to change its class

## Applicability

- object's behavior depends on its state
- it must change behavior at run-time depending on that state
- operations with multipart conditional statements depending on the object's state
  - state represented by one or more enumerated constants
  - several operations with the same (or similar) conditional structure

# STRUCTURE



# PARTICIPANTS

## **Context**

- defines the interface of interest for clients
- maintains an instance of **ConcreteState** subclass

## **State**

- defines an interface for encapsulating the behavior associated with a particular state of the Context

## **ConcreteState**

- each subclass implements a behavior associated with a state of the Context

# COLLABORATIONS

Context delegates state-specific requests to the State objects

- the Context may pass itself to the State object
  - if the State needs to access it in order to accomplish the request

State transitions are managed either by Context or by State

- see discussion on the coming slides

Clients interact exclusively with Context

- but they might configure contexts with states
  - e.g initial state

# CONSEQUENCES

**Localizes** state-specific behavior and **partitions** behavior for different states

- Put all behavior associated with a state in a state-object
- Easy to add new states and transitions
  - context becomes O-C
- Behavior spread among several State subclasses
  - number of classes increases, less compact than a single class
  - good if many states...

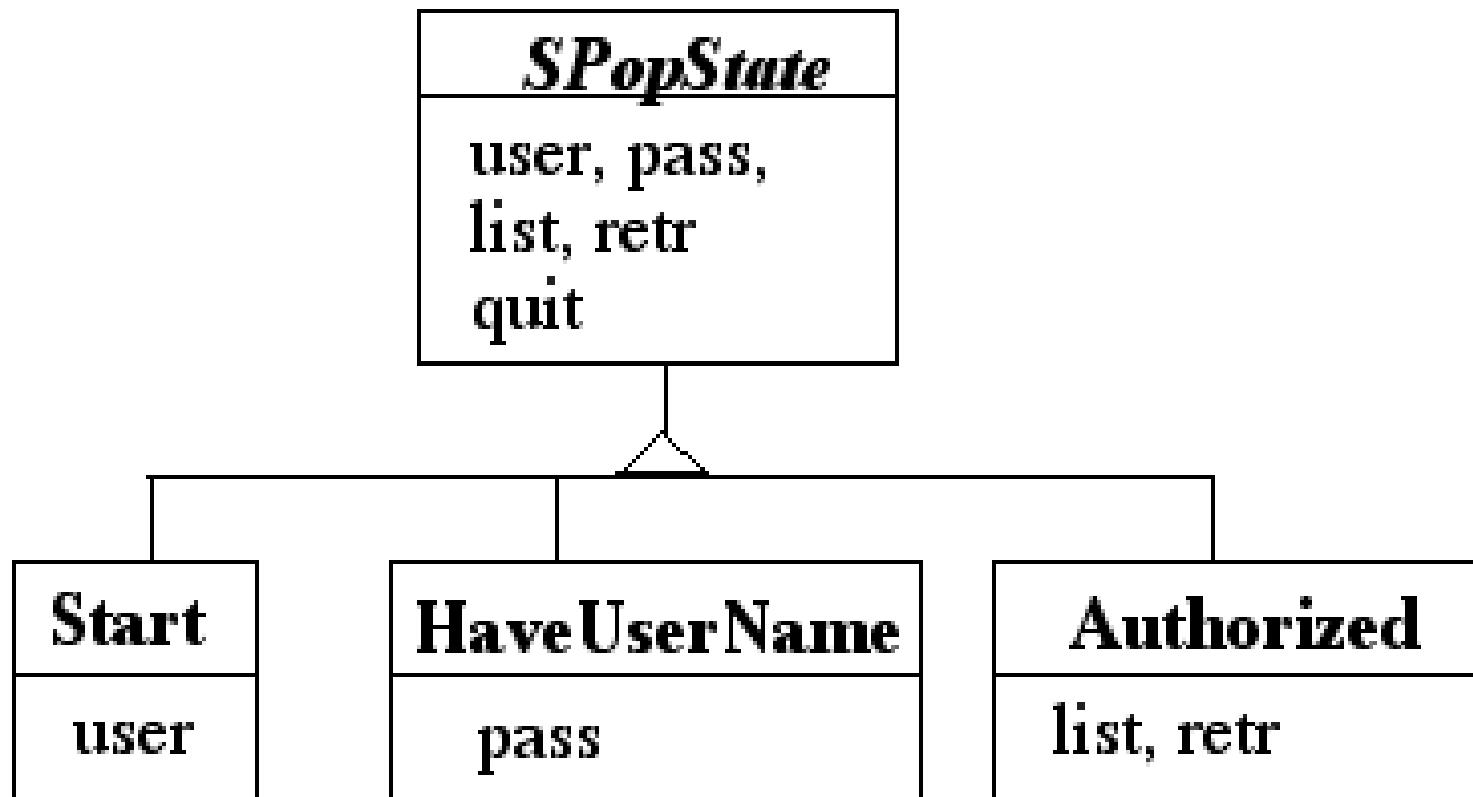
Makes state transitions explicit

- Not only a change of an internal value
- **States receive a full-object status!**
- Protects Context from inconsistent internal states

# APPLYING STATE TO SPOP

```
class SPop {  
    private SPopState state = new Start();  
    public void user( String userName ) {  
        state = state.user( userName );  
    }  
    public void pass( String password ) {  
        state = state.pass( password );  
    }  
    public void list( int messageNumber ) {  
        state = state.list( messageNumber );  
    }  
    // . . .  
}
```

# SPOP STATES



# HOW MUCH STATE IN THE STATE?

Let's identify the roles...

- **SPop** is the Context
- **SPopState** is the abstract State
- **Start**, **HaveUserName** are ConcreteStates

All the state and *real* behavior is in SPopState and subclasses

- this is an extreme example

In general Context has data and methods

- besides State & State methods
- this data will not change states

Only some aspects of the Context will alter its behavior

# WHO DEFINES THE STATE TRANSITION?

The Context if ...

- ...states will be **reused** in different state machines with different transitions
- ... the criteria for changing states are fixed

```
class Spop {  
    private SPopState state = new Start();  
  
    public void user( String userName ) {  
        state.user( userName );  
        state = new HaveUserName( userName );  
    }  
  
    public void pass( String password ) {  
        if ( state.pass( password ) )  
            state = new Authorized( );  
        else  
            state = new Start();  
    }  
}
```

# OR...THE STATES

More flexible to let State subclasses specify the next state

```
class SPop {  
    private SPopState state = new Start();  
  
    public void user( String userName ) {  
        state = state.user( userName );  
    }  
  
    public void pass( String password ) {  
        state = state.pass( password );  
    }  
  
    public void list( int messageNumber ) {  
        state = state.list( messageNumber );  
    }  
  
    // . . .  
}  
  
class Start extends SPopState {  
    public SPopState user( String userName ) {  
        return new HaveUserName( userName );  
    }  
}  
  
class HaveUserName extends SPopState {  
    String userName;  
  
    public HaveUserName( String userName ) {  
        this.userName = userName;  
    }  
}
```

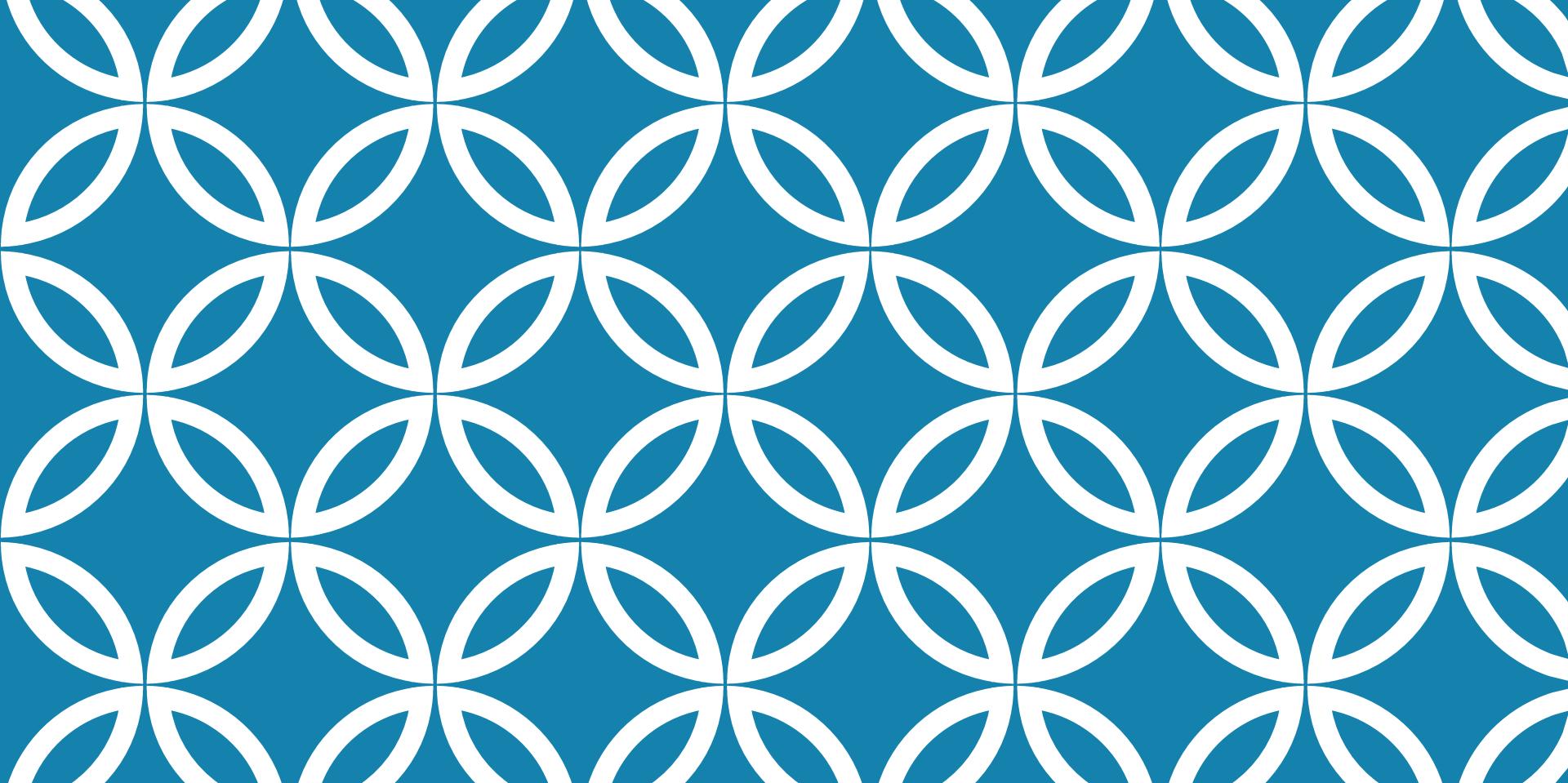
# STATE VERSUS STRATEGY

## Rate of Change

- Strategy
  - Context object usually contains one of several possible **ConcreteStrategy** objects
- State
  - Context object often changes its **ConcreteState** object over its lifetime

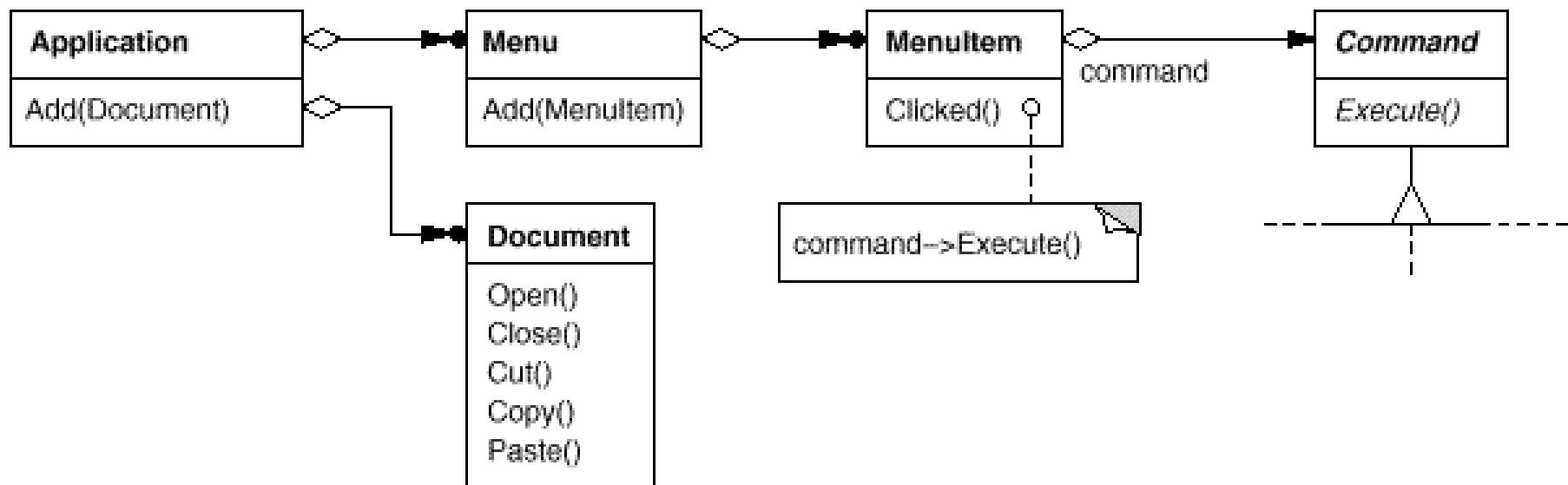
## Visibility of Change

- Strategy
  - All **ConcreteStrategy** do the same thing, but differently
  - Clients do not see any difference in behavior in the Context
- State
  - **ConcreteState** acts differently
  - Clients see different behavior in the Context



# COMMAND PATTERN

# MENU ITEMS USE COMMANDS



# BASIC ASPECTS

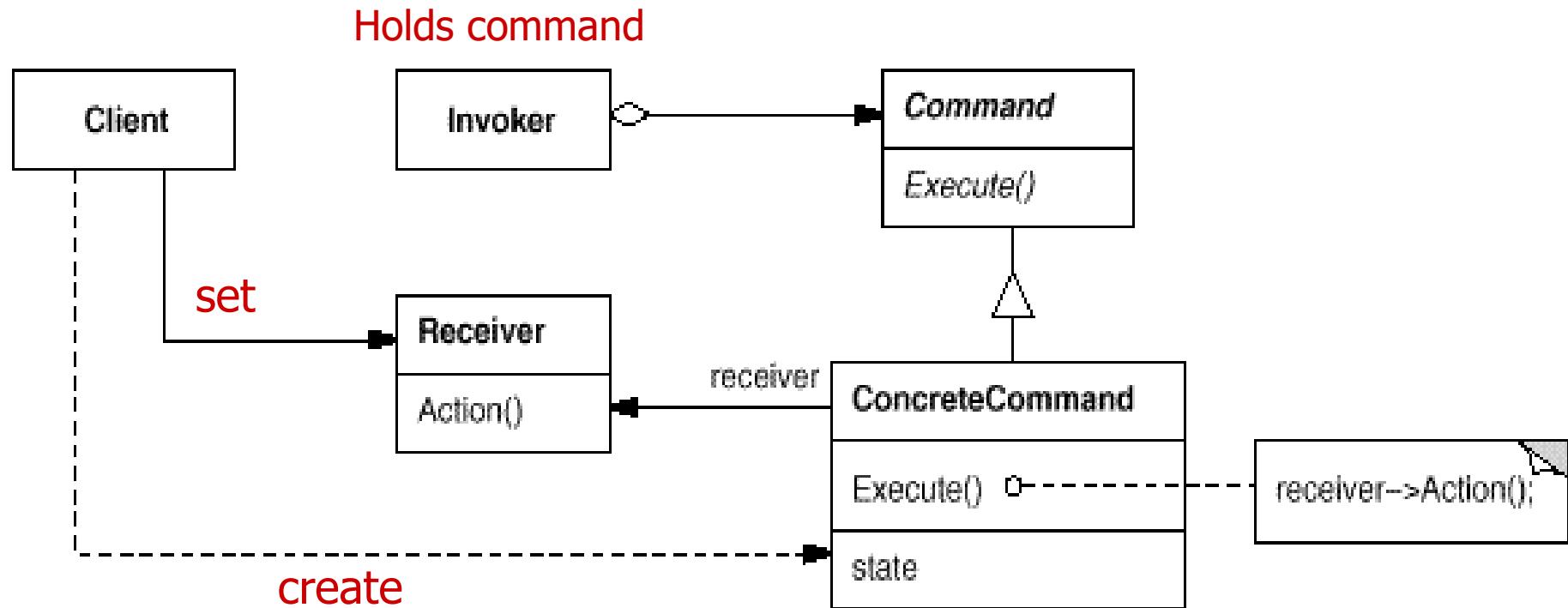
## Intent

- Encapsulate requests as objects, letting you to:
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations

## Applicability

- Parameterize objects
- Specify, queue, and execute requests at different times
- Support undo
  - recover from crashes → needs undo operations in interface
- Support for logging changes
  - recover from crashes → needs load/store operations in interface
- Model transactions

# STRUCTURE



Transforms: **concreteReceiver.action()** in **command.execute()**

# PARTICIPANTS

## **Command**

- declares the interface for executing the operation

## **ConcreteCommand**

- binds a request with a concrete action

## **Invoker**

- asks the command to carry out the request

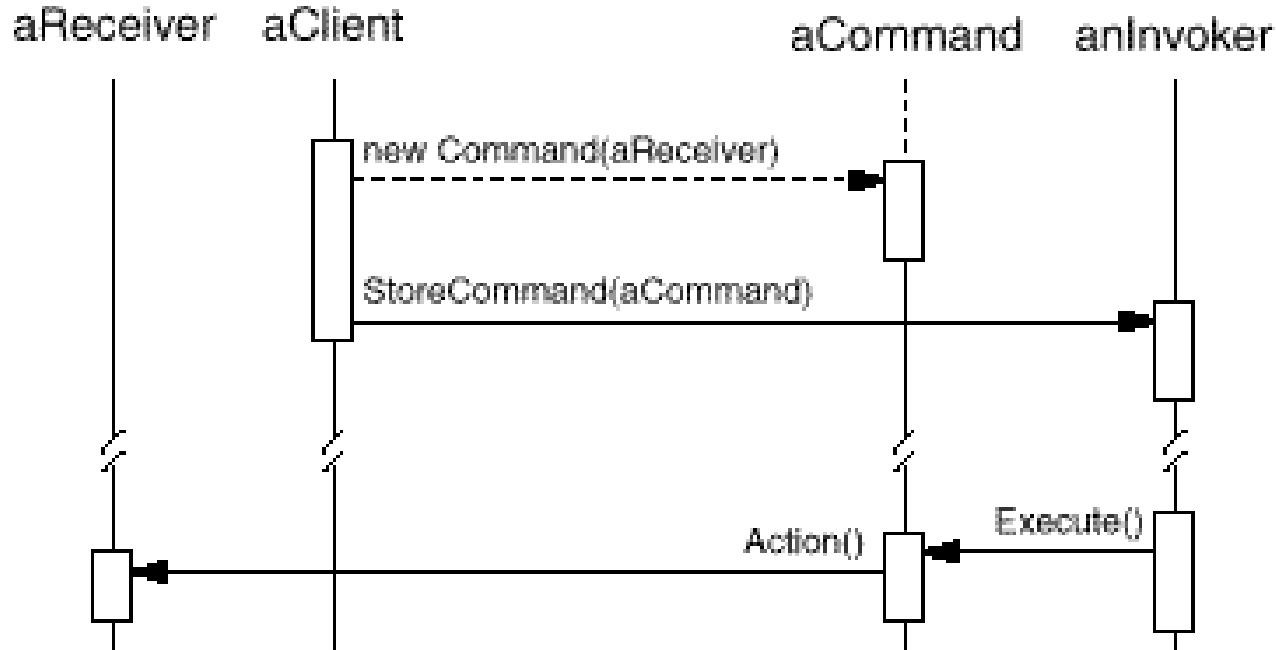
## **Receiver**

- knows how to perform the operations associated with carrying out a request.

## **Client**

- creates a ConcreteCommand and sets its receiver

# COLLABORATIONS



Client → ConcreteCommand

- creates and specifies receiver

Invoker → ConcreteCommand

ConcreteCommand → Receiver

# CONSEQUENCES

Decouples Invoker from Receiver

Commands are **first-class objects**

- can be manipulated and **extended**

Composite Commands

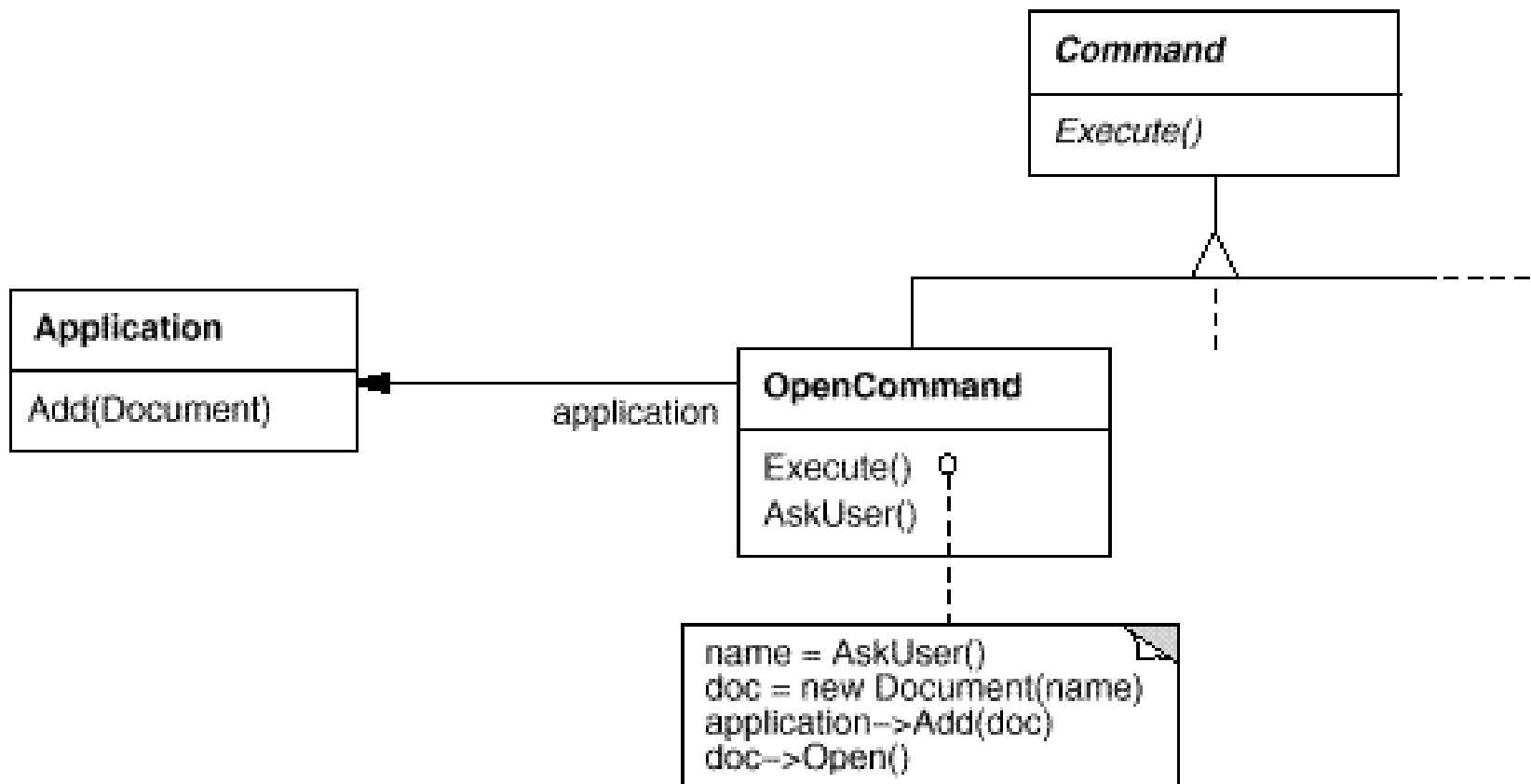
- see also **Composite pattern**

Easy to add new commands

- Invoker does not change
- it is Open-Closed

Potential for an excessive number of command classes

# EXAMPLE: OPEN DOCUMENT



# INTELLIGENCE OF COMMAND OBJECTS

"Dumb"

- delegate everything to Receiver
- used just to decouple Sender from Receiver

"Genius"

- does everything itself without delegating at all
- useful if no receiver exists
- let ConcreteCommand be independent of further classes

"Smart"

- find receiver dynamically

# UNDOABLE COMMANDS

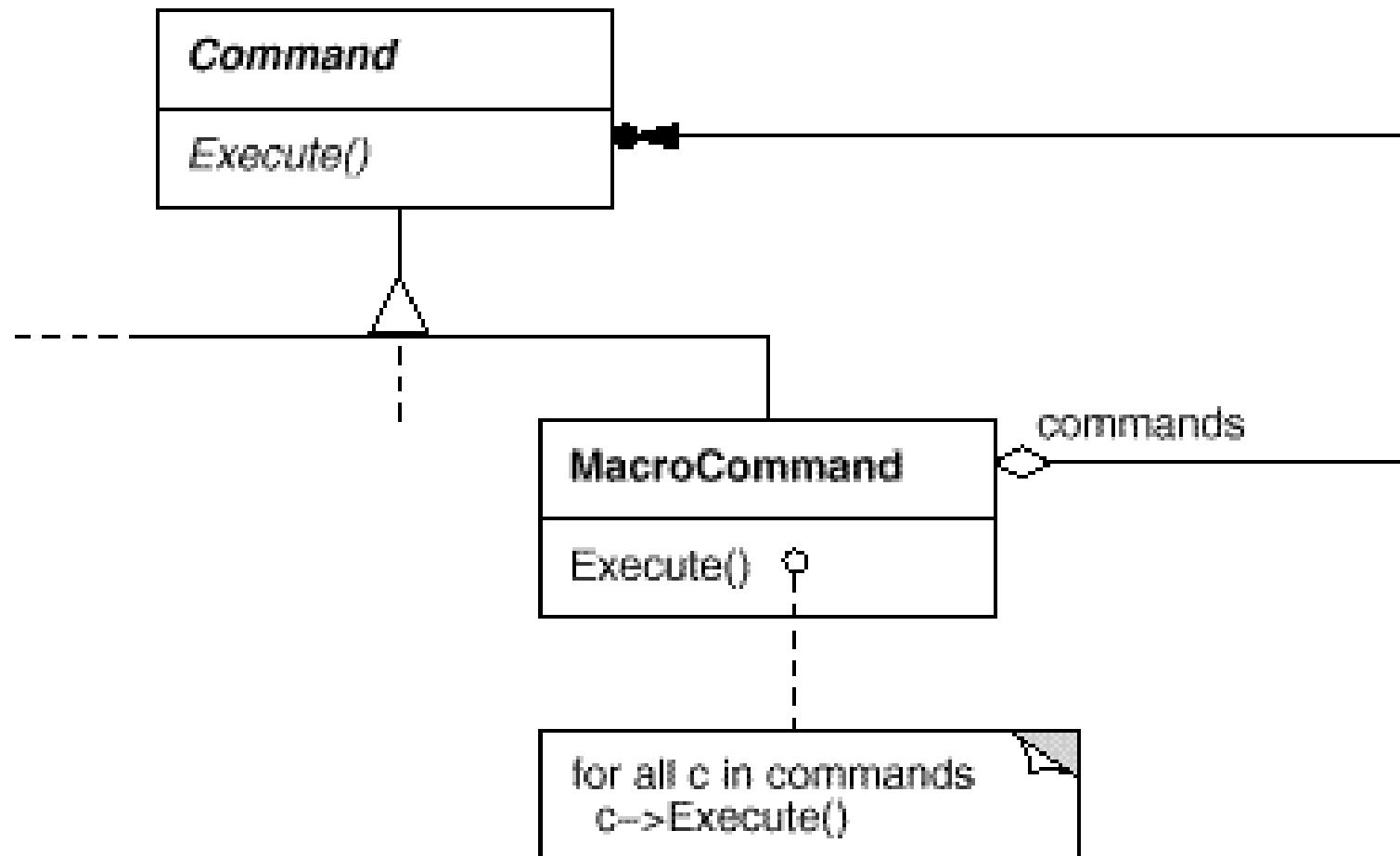
Need to store additional state to reverse execution

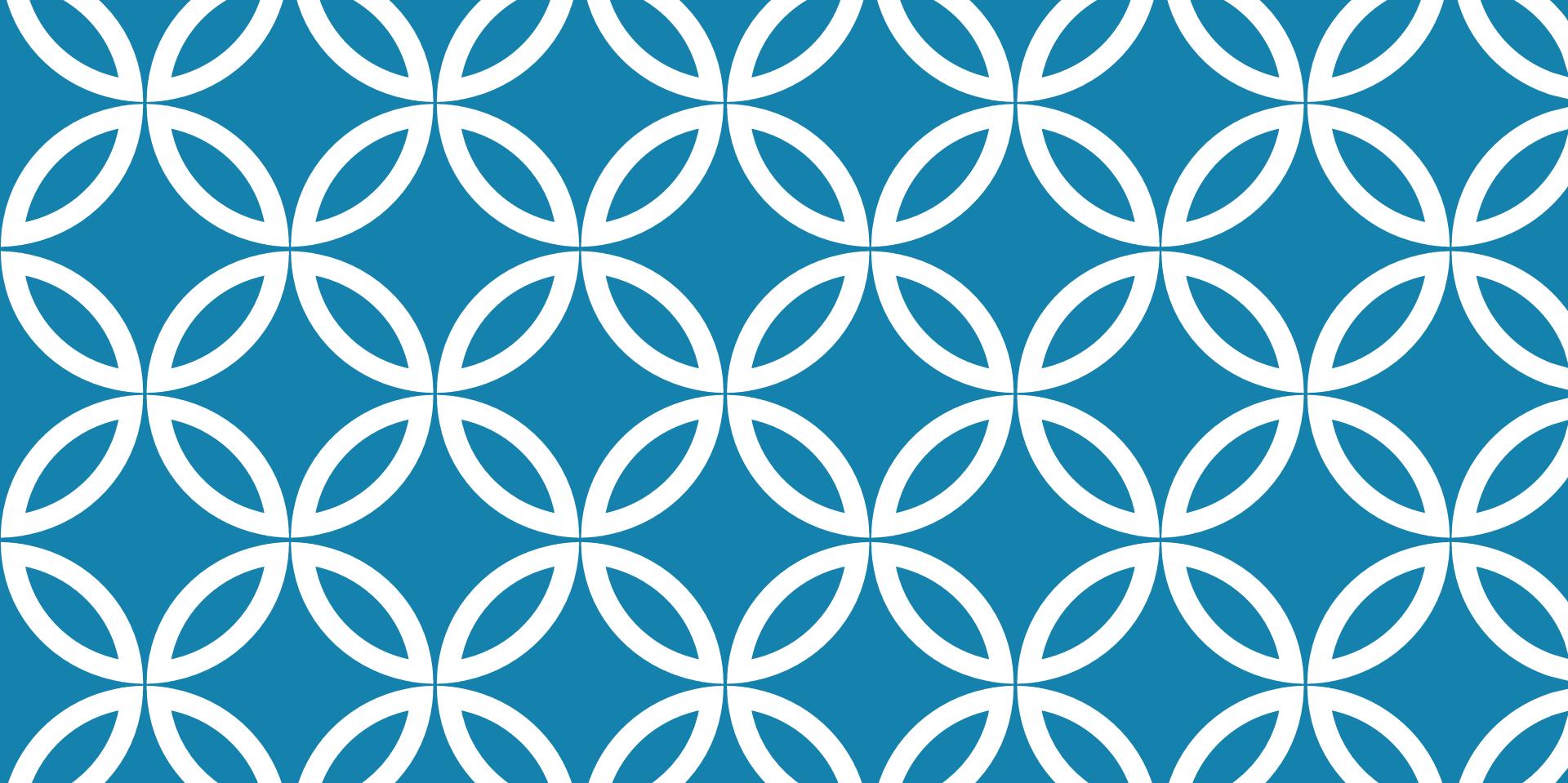
- receiver object
- parameters of the operation performed on receiver
- original values in receiver that may change due to request
  - receiver must provide operations that makes possible for command object to return it to its prior state

History list

- Sequence of commands that have been executed
  - used as LIFO with reverse-execution  $\Rightarrow$  undo
  - used as FIFO with execution  $\Rightarrow$  redo
- Commands may need to be copied
  - when state of commands change by execution

# COMPOSED COMMANDS





# CHAIN OF RESPONSIBILITY PATTERN

---

# BASIC ASPECTS

## Intent

- Decouple sender of request from its receiver
  - by giving more than one object a chance to handle the request
  - Put receivers in a chain and pass the request along the chain
    - until an object handles it

## Motivation

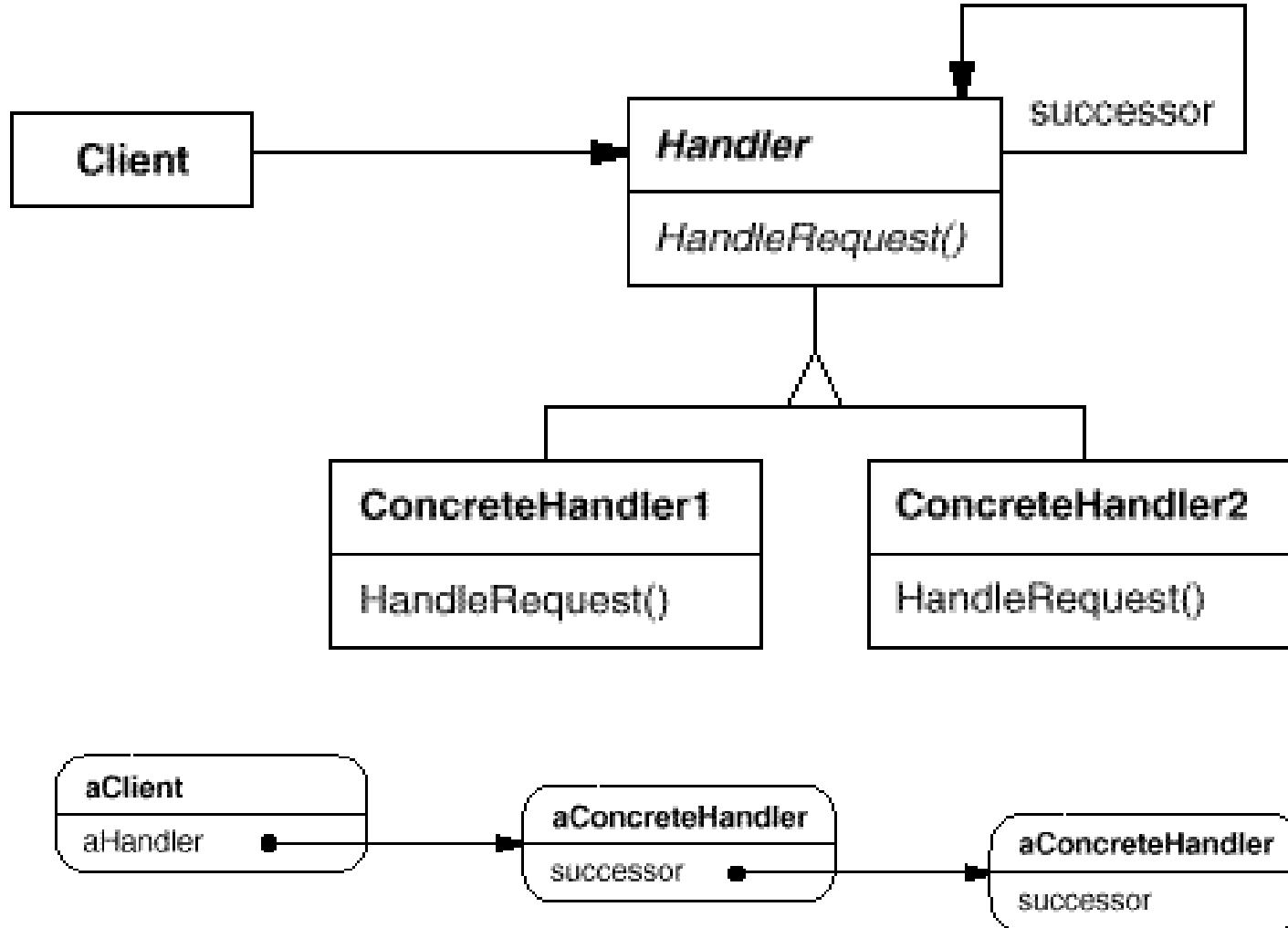
- context-sensitive help
  - a help request is handled by one of several UI objects
  - Which one?
    - depends on the context
  - The object that initiates the request does not know the object that will eventually provide the help

# WHEN TO USE?

## Applicability

- more than one object may handle a request
  - and handler isn't known a priori
- set of objects that can handle the request should be dynamically specifiable
- send a request to several objects without specifying the receiver

# STRUCTURE



# PARTICIPANTS & COLLABORATIONS

## **Handler**

- defines the interface for handling requests
- may implement the successor link

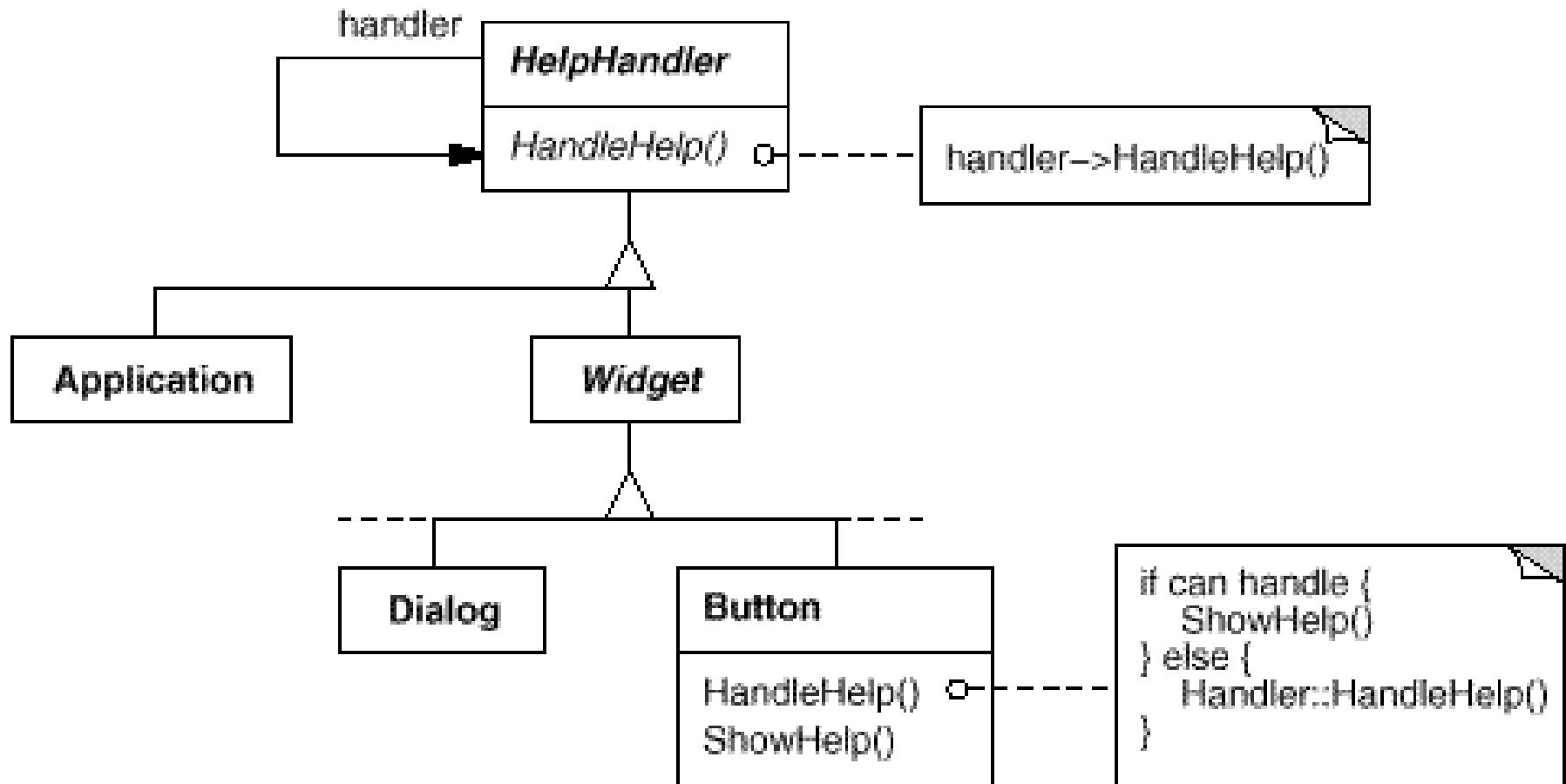
## **ConcreteHandler**

- either handles the request it is responsible for ...
- ... or it forwards the request to its successor

## **Client**

- initiates the request to a **ConcreteHandler** object in the chain

# THE CONTEXT-HELP SYSTEM



# CONSEQUENCES

## Reduced Coupling

- frees the client (sender) from knowing who will handle its request
- sender and receiver don't know each other
- instead of sender knowing all potential receivers, just keep a single reference to next handler in chain.
  - simplify object interconnections

## Flexibility in assigning responsibilities to objects

- responsibilities can be added or changed
- chain can be modified at run-time

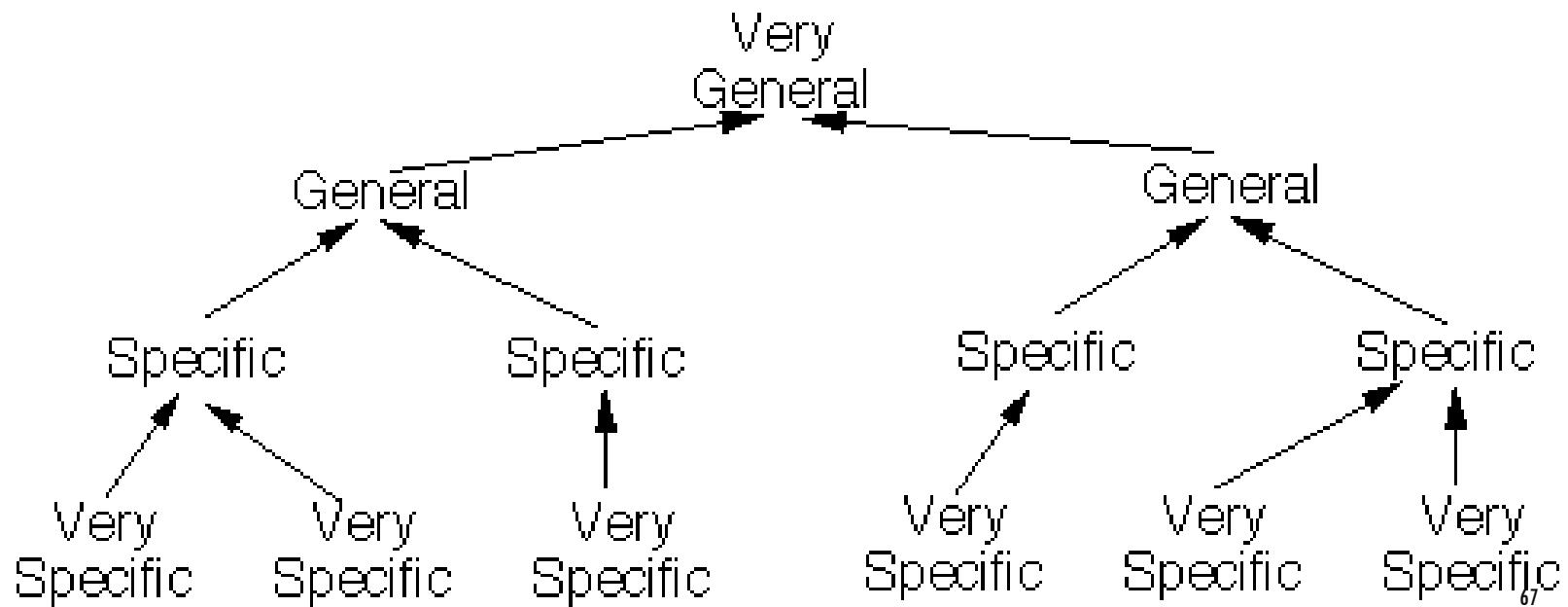
## Requests can go unhandled

- chain may be configured improperly!!

# HOW TO DESIGN CHAINS OF COMMANDS?

Like the military

- a request is made
- it goes up the chain of command until someone has the authority to answer the request



# IMPLEMENTING THE SUCCESSOR CHAIN

Define new link

- Give each handler a link to its successor

Use existing links

- concrete handlers may already have pointers to their successors
- references in a part-whole hierarchy
  - can define a part's successor
- spares work and space ...
- ... but it must reflect the chain of responsibilities that is needed

# REPRESENTING MULTIPLE REQUESTS USING ONE CHAIN

Each request is hard-coded

- convenient and safe
- not flexible (limited to the fixed set of requests defined by handler)

```
abstract class HardCodedHandler {  
    private HardCodedHandler successor;  
  
    public HardCodedHandler( HardCodedHandler aSuccessor)  
        { successor = aSuccessor; }  
  
    public void handleOpen()  
        { successor.handleOpen(); }  
  
    public void handleClose()  
        { successor.handleClose(); }  
  
    public void handleNew( String fileName)  
        { successor.handleNew( fileName ); }  
}
```

## Unique handler with parameters

- more flexible
- but it requires conditional statements for dispatching request
- less type-safe to pass parameters

```
abstract class SingleHandler {  
    private SingleHandler successor;  
  
    public SingleHandler( SingleHandler aSuccessor) {  
        successor = aSuccessor;  
    }  
  
    public void handle( String request) {  
        successor.handle( request );  
    }  
}  
  
class ConcreteOpenHandler extends SingleHandler {  
    public void handle( String request) {  
        switch ( request ) {  
            case "Open" : // do the right thing;  
            case "Close" : // more right things;  
            case "New" : // even more right things;  
            default: successor.handle( request );  
        }  
    }  
}
```

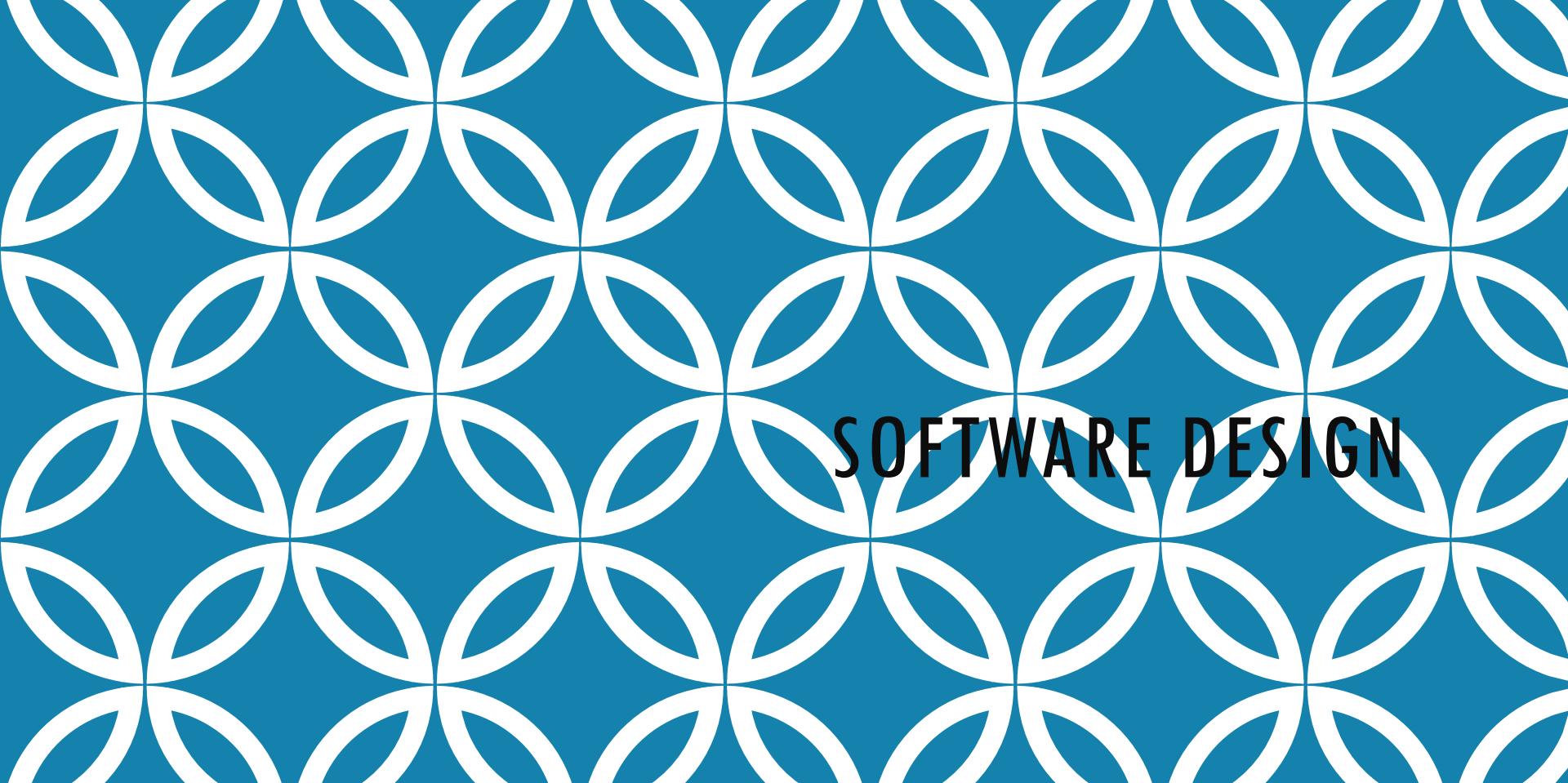
# DECORATOR VS. CHAIN OF RESPONSIBILITY

Chain of Responsibility	Decorator
Comparable to “event-oriented” architecture	Comparable to layered architecture (layers of an onion)
The “filter” objects are of equal rank	A “core” object is assumed, all “layer” objects are optional
User views the chain as a “launch and leave” pipeline	User views the decorated object as an enhanced object
A request is routinely forwarded until a single filter object handles it. many (or all) filter objects could contrib. to each request's handling.	A layer object always performs pre or post processing as the request is delegated.
All the handlers are peers (like nodes in a linked list) – “end of list” condition handling is required.	All the layer objects ultimately delegate to a single core object - “end of list” condition handling is not required. <sup>71</sup>

# NEXT TIME

Quality attributes

- Representation
- Strategies



# SOFTWARE DESIGN

Quality attributes

# CONTENT

Requirements Engineering

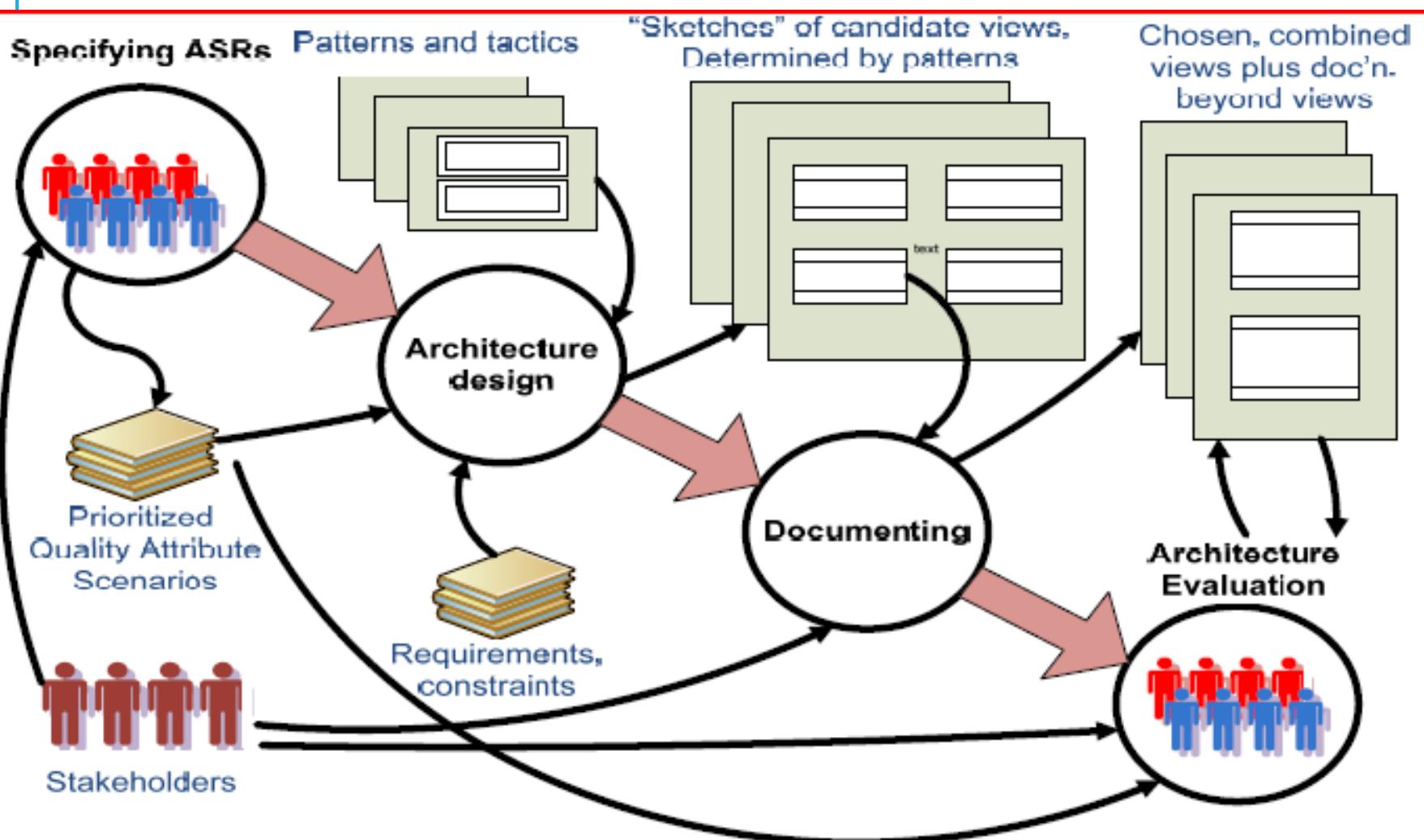
Achieving Quality Attributes

Attribute-Driven Design (ADD)

# REFERENCES

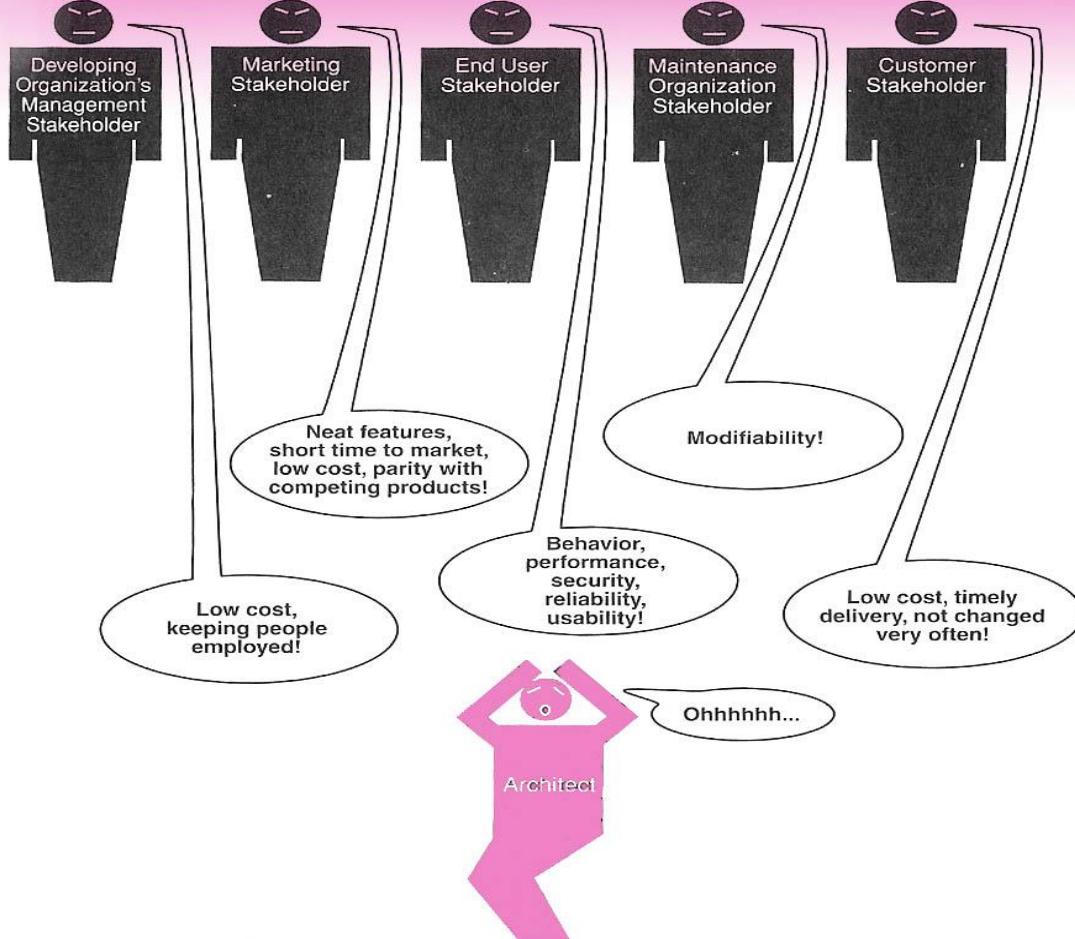
- Len Bass, Paul Clements, Rick Kazman, **Software Architecture in Practice, Second Edition**, Addison Wesley, 2003, ISBN: 0-321-15495-9
- Felix Bachmann, Len Bass, Mark Klein, **Deriving Architectural Tactics: A Step Toward Methodical Architectural Design**,
- TECHNICAL REPORT CMU/SEI-2003-TR-004
- IBM Rational
- Microsoft MSF
- <http://www.cs.uu.nl/wiki/bin/view/Swa/CourseLiterature>

# SOFTWARE ARCHITECTURE PROCESS



Adapted from Hofmeister et al., 2006.

# STAKEHOLDERS



- Developing Organization Management
- Marketing
- End User
- Maintenance Organization
- Customer

FIGURE 1.2 Influence of stakeholders on the architect

# REQUIREMENTS ELICITATION

Requirements elicitation = the activities involved in discovering the requirements of the system

Techniques:

- **Asking:** interview, (structured) brainstorm, questionnaire
- **Task analysis**
- **Scenario-based analysis:** ‘think aloud’, use case analysis
- **Ethnography:** active observation
- **Form and document analysis**
- Start from **existing system**
- **Prototyping**
- **Own insight**

General advice: use more than one technique!

# REQUIREMENTS SPECIFICATIONS

Requirements specification = rigorous modeling of requirements, to provide formal definitions for various aspects of the system

Requirements specification document should be

- as **precise** as possible: starting point for architecture/design
- as **readable** as possible: understandable for the user

Other preferred properties:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance
- Verifiable
- Modifiable
- Traceable

# SPECIFICATION TECHNIQUES

Specification techniques:

- Entity-Relationship (E-R) modeling
- Use cases (UML)
- Epics and User stories

# REQUIREMENTS VALIDATION

- Requirements validation = checking the requirements document for consistency, completeness and accuracy
- Validation of requirements needs user interaction!
- Techniques:
  - Reviews (reading, checklists, discussion)
  - Prototyping
  - Animation

# USE-CASES

- Technique for specifying **functional** requirements (What should the system do?)
- A **use case** captures a **contract** between the stakeholders of a system about its behavior
- The use case describes the system's behavior **under various conditions** as the system responds to a request from one of the stakeholders, called the primary actor
- Use cases are fundamentally a text form describing use scenarios

# BASIC USE-CASE FORMAT

- Use case: <use case goal>
- Level: <one of: summary level, user-goal level, subfunction>
- Primary actor: <a role name for the actor who initiates the use case>
- Description: <the steps of the **main success scenario** from trigger to goal delivery and any cleanup after>
- Extension: <alternate scenarios of success or failure>

# USE-CASES: BEST PRACTICES

- A use case is a prose essay
- Should be easy to read
- Sentence form for the scenario: active voice, present tense, describing an actor successfully achieving a goal
- Include sub-use cases where appropriate
- Keep the GUI out
- Use UML use case diagrams to visualize relations between actors and use cases or among use cases. Use text to specify use cases themselves!
- It is hard, and important, to keep track of the various use cases

# FUNCTIONAL REQUIREMENTS

## Pitfalls:

- Undefined or inconsistent system boundary (scope!)
- System point of view instead of actor centered
- Spiderweb actor-to-use case relations
- Long, too many, confusing use case specifications, incomprehensible to customer

## Beware of:

- ‘Shopping cart’ mentality
- The ‘all requirements are equal’ fallacy
- Stakeholders who won’t read use case descriptions because they find them too technical or too complicated

# ARCHITECTURAL DRIVERS

- Functionality is the ability of the system to do the work for which it was intended
- Functionality may be achieved through the use of any number of possible structures
  - => functionality is largely independent of structure
- Architecture constrains the mapping of functionality on various structures if **quality attributes** are important

# ARCHITECTURAL DRIVERS [2]

- Get the functionality right and then accommodate nonfunctional requirements – NOT POSSIBLE!!!
- Non-functional requirements must be taken into account EARLY ON!!!
- There are two broad categories of non-functional requirements
  - Design-time
  - Run-time

# SOFTWARE ARCHITECTURE AND QUALITY ATTRIBUTES

- Quality isn't something that can be added to a software intensive system after development finishes
- Quality concerns need to be addressed during all phases of the software development.
- BUSINESS GOALS determine the qualities attributes, which are over and above of system's functionality!!!

# QUALITY REQUIREMENTS OBJECTIVES

- Input for architecture definition
- Driving architecture evaluation
- Communicating architectural parts and requirements
- Finding missing requirements
- Guiding the testing process

# QUALITY REQUIREMENTS: BEST PRACTICES

Not all quality attributes are equally important:  
prioritize!

Make the quality requirements measurable

- Not: ‘The system should perform well’ but ‘The response time in interactive use is less than 200 ms’

Link the quality requirements to use cases

- Example: ‘The ATM validates the PIN within 1 second’

# CHANGE SCENARIOS

Some quality requirements do not concern functionality but other aspects of the system. These cannot be linked to any use case

- Maintainability and Portability, e.g. Changeability and Adaptability

Link these quality requirements to specific change scenarios

- **Not** ‘The system should be very adaptable’ **but** ‘The software can be installed on all Windows and Unix platforms without change to the source code’
- **Not** ‘The system should be changeable’ **but** ‘Functionality can be added to the ATM within one month that makes it possible for users to transfer money from savings to checking account’

# CONSTRAINTS

- Functional and quality requirements specify the goal, constraints limit the (architecture) solution space
- Stakeholders should therefore not only specify requirements, but also constraints
- Possible constraint categories:
  - Technical, e.g. platform, reuse of existing systems and components, use of standards
  - Financial, e.g. budget
  - Organizational, e.g. processes, availability of customer
  - Time, e.g. deadline

# ACHIEVING QUALITY

Once determined, the quality requirements provide guidance for **architectural decisions**

An architectural decision that influences the qualities of the product is sometimes called a **tactic**

Mutually connected tactics are bundled together into **architectural patterns**: schemas for the structural organization of entire systems

# TYPES OF REQUIREMENTS (FURPS MODEL)

## Functionality

- feature sets
- capabilities
- security

## Usability

- human factors
- aesthetics
- consistency in the user interface
- online and context-sensitive help
- wizards and agents
- user documentation
- training materials

## Reliability

- frequency and severity of failure
- recoverability
- predictability
- accuracy
- mean time between failure

# TYPES OF REQUIREMENTS [2]

## Performance

- speed
- efficiency
- availability
- accuracy
- response time
- recovery time
- resource usage

## Supportability

- testability
- extensibility
- adaptability
- maintainability
- compatibility
- configurability
- installability
- localizability (internationalization)

# QUALITY ATTRIBUTES

Business quality:

- Time to market – “Time”
- Cost and benefit – “Economy”
- Projected lifetime – “Form”
- Target market – “Function”
- Rollout schedule – “Time”
- Integration with legacy – “Time”

# QUALITY ATTRIBUTES

Architectural quality

- Conceptual integrity
- Correctness and completeness
- Buildability

# QUALITY ATTRIBUTES

System quality

- Availability
- Performance
- Security
- Modifiability
- Testability
- Usability

# SYSTEM QUALITY ATTRIBUTES

Are defined in terms of scenarios

- **source of stimulus** [the entity (human or another system) that generated the stimulus or event.] **who?**
- **stimulus** [a condition that determines a reaction of the system.] **what?**
- **environment** [the current condition of the system when the stimulus arrives.] **when?**
- **artifact** [is a component that reacts to the stimulus. It may be the whole system or some pieces of it.] **where?**
- **response** [the activity determined by the arrival of the stimulus.] **which?**
- **response measure** [the quantifiable indication of the response.] **how?**

# GENERAL SCENARIO

76 Part Two Creating an Architecture

4—Understanding Quality Attributes

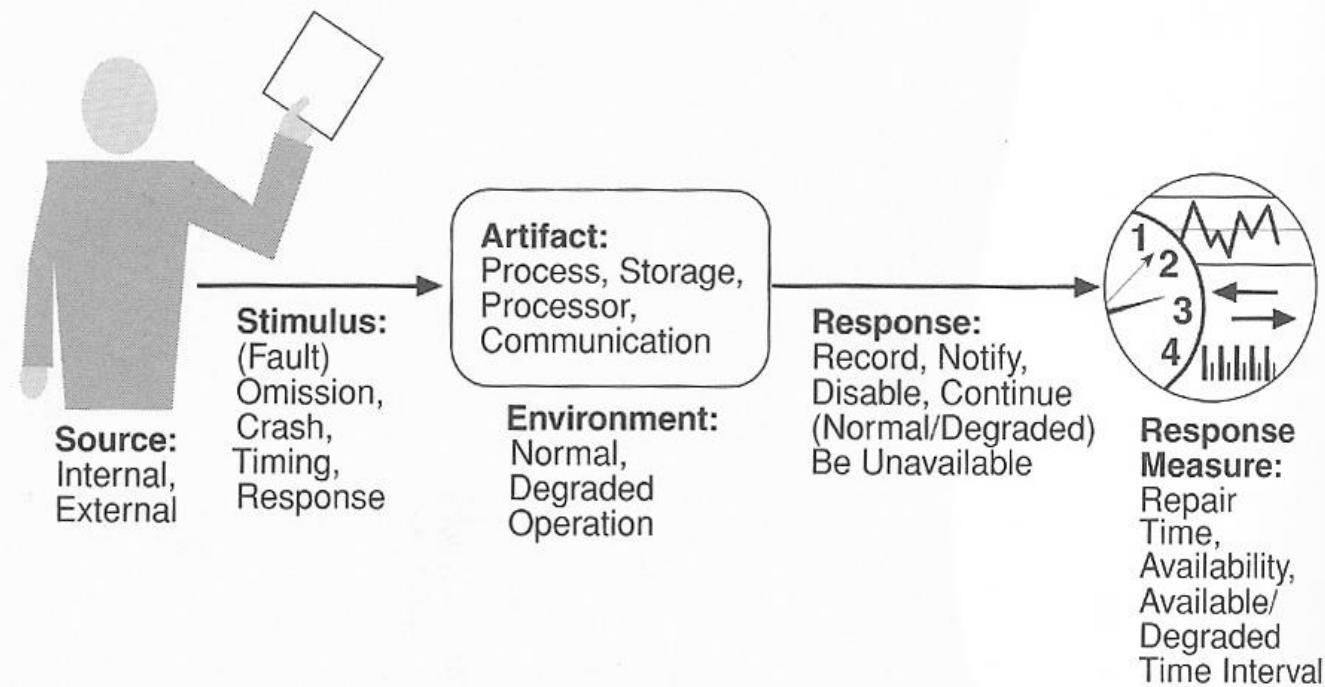
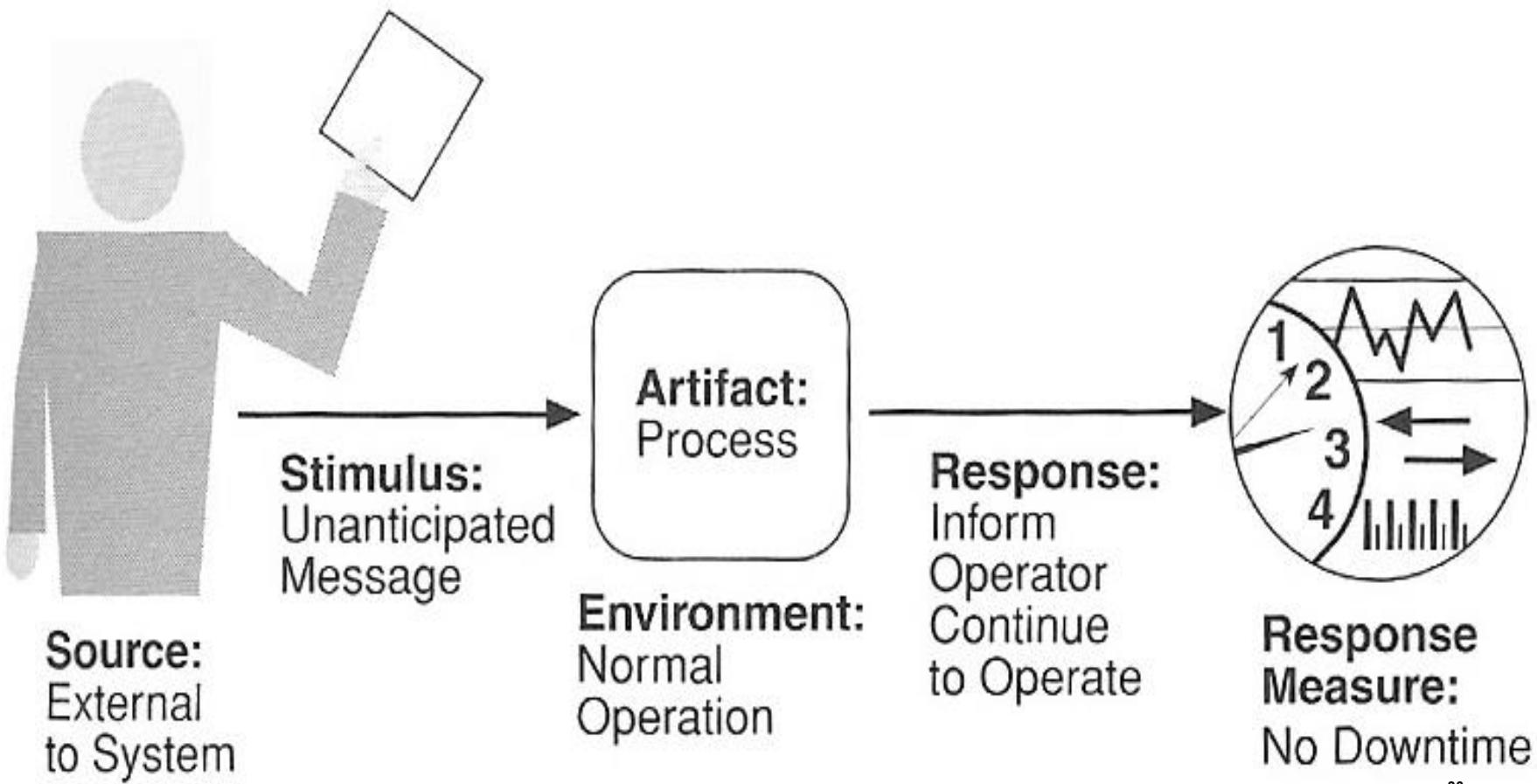


FIGURE 4.2 Availability general scenarios

# CONCRETE SCENARIO



# AVAILABILITY

Typically defined as the probability of a system to be operational when needed in terms of

**mean time to failure / (mean time to failure + mean time to repair)**

# AVAILABILITY SCENARIO

**Source of stimulus:** internal or external

**Stimulus:**

- omission: a component fails to respond to an input.
- crash: a component repeatedly suffers omission faults.
- timing: a component responds, but the response is early or late.
- response: a component responds with an incorrect value.

**Artifact:** the resource that is required to be available (i.e. processor, communication channel, process, or storage device).

# AVAILABILITY SCENARIO

**Environment:** defines the state of the system when the fault or failure occurred: normal, degraded

**Response:** logging, notification, switching to backup, restart, shutdown

**Response measure:**

- the availability percentage,
- a time for repair,
- certain times during which the system must be available,
- the duration for which the system must be available.

# POS – QUALITY ATTRIBUTE SCENARIO 1

**Scenario(s):** The barcode scanner fails; failure is detected, signalled to user at terminal; continue in degraded mode

**Stimulus Source :** Internal to system

**Stimulus:** Fails

**Environment:** Normal operation

**Artefact (If Known):** Barcode scanner

**Response:** Failure detected, shown to user, continue to operate

**Response Measure:** No downtime, React in 2 seconds

# POS – QUALITY ATTRIBUTE SCENARIO 2

**Scenario(s):** The inventory system fails and the failure is detected. The system continues to operate and queue inventory requests internally; issue requests when inventory system is running again

**Stimulus Source :** Internal to system

**Stimulus:** Fails

**Environment:** Normal operation

**Artefact (If Known):** Inventory system

**Response:** Failure detected, operates in degraded mode, queues requests, detects when inventory system is up again

**Response Measure:** Degraded mode is entered for maximum one hour

# TACTICS TO ACHIEVE AVAILABILITY

- for fault detection
- for fault recovery
- for fault prevention

# TACTICS FOR FAULT DETECTION

- Ping/echo
  - Signal is issued, response is waited for
  - Estimates round-trip time and rate of package loss
- Heartbeat
  - Periodic signal is broadcasted
- Exception handling

# TACTICS FOR FAULT RECOVERY

- Voting
  - run the same algorithm on different processors.
  - "majority rules" approach uncovers any deviant behavior in the processors.
- Active redundancy
  - set up redundant components and keep them synchronized
- Passive redundancy
  - have only one component respond to events, but have that component inform redundant components about the changes

# TACTICS FOR FAULT RECOVERY [2]

- Spare
  - a standby spare computing platform is prepared to replace many different failed components. Backup + logs needed.
- Shadow operation
  - a previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behaviour of the working component before it is restored to service.

# TACTICS FOR FAULT PREVENTION

- Removal from service
  - removal of a component of the system from operation in order to update it and avoid potential failures.
    - Automatic
    - Manual
- Transactions
  - set of operations where either all or none are executed successfully.
- Process monitor
  - if a fault is detected in a process, an automated monitoring process can delete the failed process and create a new instance of it, initializing it to some appropriate state as in the spare tactic

# PERFORMANCE

**Performance** refers to the time it takes the system to respond to an event. The event can be fired by:

- a user,
- another system,
- the system itself.

# PERFORMANCE SCENARIO

**Source of stimulus:** The stimuli arrive either from external (possibly multiple) or internal sources.

**Stimulus:** The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic, or sporadic.

- **Periodic** means that the events arrive in regular intervals of time
- **Stochastic** means that the arrival of events is based on some probabilistic distribution
- **Sporadic** means that the events arrive rather randomly.

**Artifact.** The artifact is always the system's service, which has to respond to the event.

**Environment.** The system can be in various operational modes, such as normal, emergency, or overload. The response varies depending on the current state of the system.

# PERFORMANCE SCENARIO

**Response.** The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). The response of the system can be characterized by:

- **latency** (the time between the arrival of the stimulus and the system's response to it),
- **deadlines** in processing (a specific action should take place before another),
- **throughput** of the system (e.g., the number of transactions the system can process in a second),
- **jitter** of the response (the variation in latency),
- **number of events not processed** because the system was too busy to respond,
- **lost data** because the system was too busy.

**Response measure.** Response measures include the time it takes to process the arriving events (latency, or deadlines by which the events must be processed), variations in this time (jitter), the number of events that can be processed within a particular time interval (throughput), and the characterization of the events that cannot be processed (miss rate, data loss).

# SCENARIO PROFILE FOR PERFORMANCE

Quality Factor	Scenario description
Initialization	The system Must perform all initialization activities within 10 minutes.
Latency	The system shall Run simulations with no instantaneous lags greater than five seconds, no average lags greater than three seconds.
Capacity	The system shall be able to provide run-time simulation with debug enabled.
Latency	A sensor shall finish data collection within 30 seconds of simulation termination.
Throughput	The system shall finish data collection request from three network sensors within 10 seconds.

# POS CASE STUDY

**Scenario(s):** The POS system scans a new item, item is looked up, total price updated within two seconds

**Stimulus Source :** End user

**Stimulus:** Scan item, fixed time between events for limited time period

**Environment:** Development time

**Artefact (If Known):** POS system

**Response:** Item is looked up, total price updated

**Response Measure:** Within two seconds

# THROUGHPUT

Measure of the amount of work an application must perform in unit time

- Transactions per second (TPS)
- Messages per second (MPS)

Is required throughput:

- Average?
- Peak?

Many system have low average but high peak throughput requirements

# RESPONSE TIME

- Measure of the latency an application exhibits in processing a request
- Usually measured in (milli)seconds
- Often an important metric for users
- Is required response time:
  - Guaranteed?
  - Average?

E.g. 95% of responses in sub-4 seconds, and all within 10 seconds

# DEADLINES

“something must be completed before some specified time”

- Payroll system must complete by 2am so that electronic transfers can be sent to bank
- Weekly accounting run must complete by 6am Monday so that figures are available to management

Deadlines often associated with batch jobs in IT systems.

# ATTENTION!

What is a

- Transaction?
- Message?
- Request?

All are application specific measures.

Ex. System must achieve 100 mps throughput

- BAD!!

System must achieve 100 mps peak throughput for  
*PaymentReceived* messages

- GOOD!!!

# FACTORS AFFECTING PERFORMANCE

- Resource Consumption
- Blocked time
  - Contention for resources
  - Availability of resources
  - Dependency on other computation

# TACTICS TO ACHIEVE PERFORMANCE

- resource demand,
- resource management
- resource arbitration.

# RESOURCE DEMAND

**Increase computational efficiency.**

- Improving the algorithms,
- Trading one resource for another.

**Reduce computational overhead.**

- Use local class vs. RMI
- Use of intermediaries => the classic flexibility/performance tradeoff.

# RESOURCE DEMAND - REDUCE THE NUMBER OF PROCESSED EVENTS

## **Manage event rate.**

- Reduce the sampling frequency at which environmental variables are monitored

## **Control frequency of sampling.**

- If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.

# RESOURCE DEMAND – CONTROL THE USE OF RESOURCES

## **Bound execution times.**

- Place a limit on how much execution time is used to respond to an event.

## **Bound queue sizes.**

- This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.

# RESOURCE MANAGEMENT

## **Introduce concurrency.**

- process different streams of events on different threads
- create additional threads to process different sets of activities.
- appropriately allocate the threads to resources (load balancing)

## **Maintain multiple copies of either data or computations.**

- clients in a client-server pattern are replicas of the computation.
- caching is a tactic in which data is replicated,
- keeping the copies consistent and synchronized becomes a responsibility that the system must assume.

## **Increase available resources.**

- faster processors, additional processors, additional memory, faster networks.

# RESOURCE ARBITRATION - SCHEDULING

## First In, First Out (FIFO).

- queues that treat all requests equally (all have the same priority). Requests are ordered by time of arrival.

## Fixed priority.

- assign to each resource requester a priority, and treat the requests issued by high-priority requesters first.
- Strategies:
  - semantic importance.** Priority is assigned based on some domain characteristic.
  - deadline monotonic.** This is a static strategy that assigns higher priority to requesters with shorter deadlines.
  - rate monotonic.** This is a static strategy for periodic requesters; higher priority is assigned to requesters with shorter periods.

# RESOURCE SCHEDULING

## **Dynamic priority.**

- ordering according to a criterion and then, at every assignment possibility, assign the resource to the next request in that order.
- earliest deadline: the priority is assigned to the pending request with the earliest deadline.

## **Static scheduling.**

- determine the sequence of assignment offline.

# SECURITY

Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users.

An attempt to breach security is called an attack

# SECURITY FEATURES

**Nonrepudiation** - a transaction (access to or modification of data or services) cannot be denied by any of the parties to it.

**Confidentiality** - data or services are protected from unauthorized access.

**Integrity** - data or services are being delivered as intended.

**Assurance** - the parties to a transaction are who they purport to be.

**Availability** - the system will be available for legitimate use.

**Auditing** - the system tracks activities within it at levels sufficient to reconstruct them.

# SECURITY SCENARIO

**Source** - Individual or system

- that is correctly identified, identified incorrectly, of unknown identity
- who is internal/external, authorized/not authorized
- with access to limited resources, vast resources

**Stimulus** - Tries to

- display data, change/delete data, access system services, reduce availability to system services

**Artifact** - System services; data within system

**Environment** -

- online or offline,
- connected or disconnected,
- firewalled or open

# SECURITY SCENARIO CONTINUED

## **Response**

- Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services

## **Response Measure**

- Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

# SECURITY SCENARIOS

Series No.	Security Requirements	Security properties
SR1	A system shall accept online payments for the services, which means the transactions between the system and financial institutes must be protected.	Private communication/information protection- Defense in depth
SR2	A system provides secured storage to customers' credit details and other information.	Data protection
SR3	A system shall be able to identify different users and verify their access privileges according to their account types.	User identification Access verification
SR4	A system shall be able to detect and prevent Denial Of Service (DOS) attacks. The system shall be able to run reliably most of the time.	Reducing exposure to attack / Error prevention & handling
SR5	A system is an evolving system that shall be easily modifiable to introduce changes in the security policy and other security checks.	Encapsulation of Security policy/ Initializaiton process

# DEALING WITH SECURITY RISKS

Vulnerability Category	Potential Problem Due to Bad Design
Input / Data Validation	Insertion of malicious strings in user interface elements or public APIs. These attacks include command execution, cross - site scripting (XSS), SQL injection, and buffer overflow. Results can range from information disclosure to elevation of privilege and arbitrary code execution.
Authentication	Identity spoofing, password cracking, elevation of privileges, and unauthorized access.
Authorization	Access to confidential or restricted data, data tampering, and execution of unauthorized operations.
Configuration Management	Unauthorized access to administration interfaces, ability to update configuration data, and unauthorized access to user accounts and account profiles.
Sensitive Data	Confidential information disclosure and data tampering.
Cryptography	Access to confidential data or account credentials, or both.
Exception Management	Denial of service and disclosure of sensitive system-level details.
Auditing and Logging	Failure to spot the signs of intrusion, inability to prove a user's actions, and difficulties in problem diagnosis.

# PRINCIPLES FOR SECURITY STRATEGIES

Category	Guidelines
Input / Data Validation	<p>Do not trust input; consider centralized input validation. Do not rely on client-side validation. Be careful with canonicalization issues. Constrain, reject, and sanitize input. Validate for type, length, format, and range.</p>
Authentication	<p>Use strong passwords. Support password expiration periods and account disablement. Do not store credentials (use one-way hashes with salt). Encrypt communication channels to protect authentication tokens.</p>
Authorization	<p>Use least privileged accounts. Consider authorization granularity. Enforce separation of privileges. Restrict user access to system-level resources.</p>
Configuration Management	<p>Use least privileged process and service accounts. Do not store credentials in clear text. Use strong authentication and authorization on administration interfaces. Do not use the Local Security Authority (LSA). Secure the communication channel for remote administration.</p>

# PRINCIPLES FOR SECURITY STRATEGIES

Sensitive Data	Avoid storing secrets. Encrypt sensitive data over the wire. Secure the communication channel. Provide strong access controls for sensitive data stores.
Cryptography	Do not develop your own. Use proven and tested platform features. Keep unencrypted data close to the algorithm. Use the right algorithm and key size. Avoid key management (use DPAPI). Cycle your keys periodically. Store keys in a restricted location.
Exception Management	Use structured exception handling. Do not reveal sensitive application implementation details. Do not log private data such as passwords. Consider a centralized exception management framework.
Auditing and Logging	Identify malicious behavior. Know what good traffic looks like. Audit and log activity through all of the application tiers. Secure access to log files. Back up and regularly analyze log files.

# SECURITY TACTICS

- Resisting attacks
- Detecting attacks
- Recovering from attacks

# RESISTING ATTACKS

## **Authenticate users.**

- Ensure that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.

## **Authorize users.**

- ensure that an authenticated user has the rights to access and modify either data or services. This is usually managed by providing some access control patterns within a system. Access control can be by user or by user class. Classes of users can be defined by user groups, by user roles, or by lists of individuals.

## **Maintain data confidentiality.**

- encryption to data and to communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for a Web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

# RESISTING ATTACKS

## Maintain integrity.

- Data should be delivered as intended. It can have redundant information encoded in it, such as checksums or hash results, which can be encrypted either along with or independently from the original data.

## Limit exposure.

- The architect can design the allocation of services to hosts so that limited services are available on each host.

## Limit access.

- Firewalls restrict access based on message source or destination port. It is not always possible to limit access to known sources. One configuration used in this case is the so-called demilitarized zone (DMZ).

# SECURITY TACTICS

## Detecting attacks

- Intrusion detection system

## Recovering from attacks

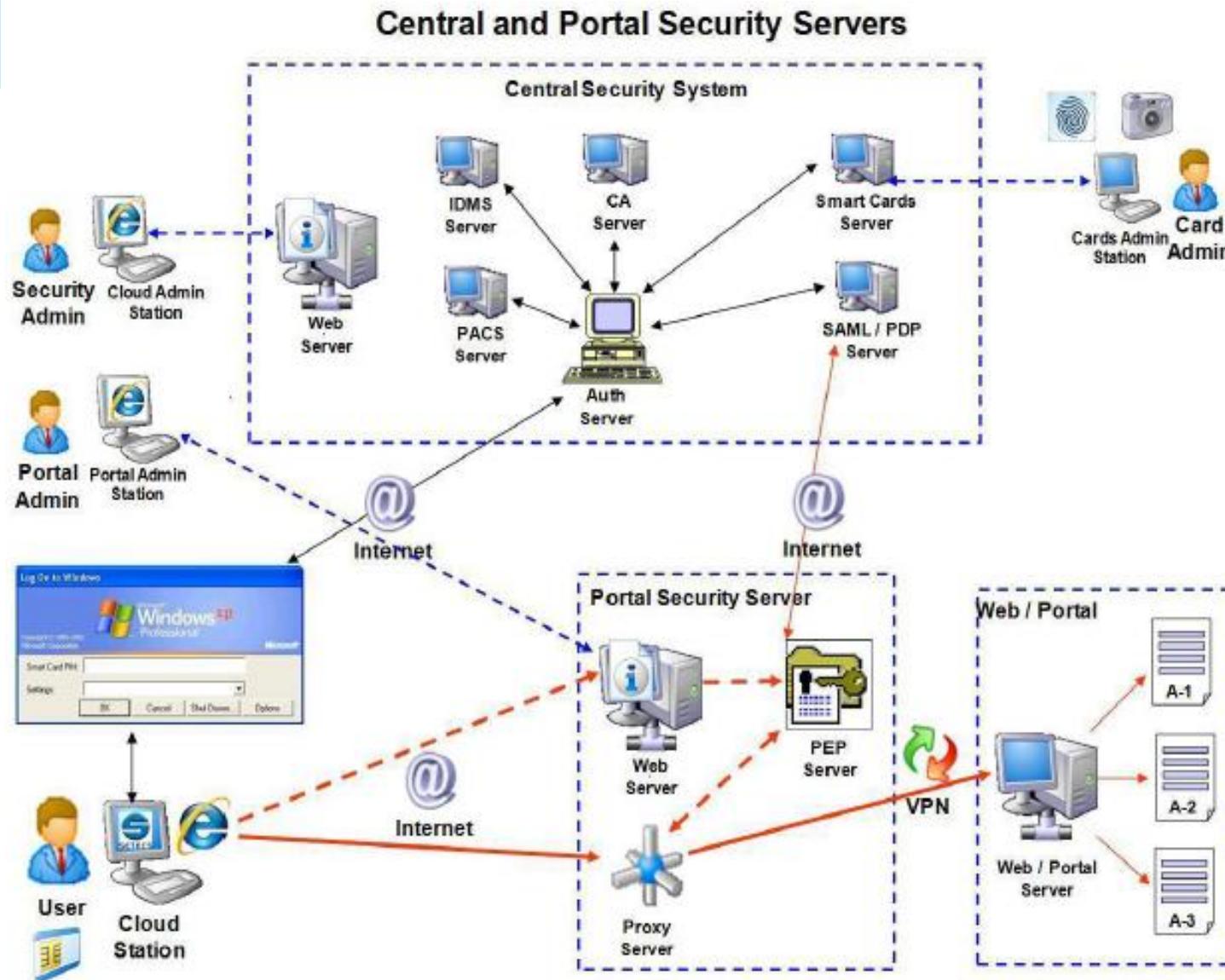
- restoring state (availability)
- attacker identification (nonrepudiation)

# CLOUD-BASED SECURITY VULNERABILITIES

Identified by CSA, NIST and ENISA

- Data Privacy and Reliability
- Data Integrity
- Authentication and Authorization
  - authentication frameworks like OpenID, SAML, Shibboleth

# CLOUD SECURITY INFRASTRUCTURE EXAMPLE



# OTHER QUALITY ATTRIBUTES

## Modifiability

- **Source:** developer, administrator, user
- **Stimulus:** add/delete/modify function or quality
- **Artifact:** UI, platform, environment
- **Environment:** design, compile, build, run
- **Response:** make change and test it
- **Measure:** effort, time, cost

# TESTABILITY

**Source:** developer, tester, user

**Stimulus:** milestone completed

**Artifact:** design, code component, system

**Environment:** design, development, compile, deployment, run

**Response:** can be controlled and observed

**Measure:** coverage, probability, time

# EXAMPLE TESTABILITY SCENARIO

**Source:** Unit tester

**Stimulus:** Performs unit test

**Artifact:** Component of the system

**Environment:** At the completion of the component

**Response:** Component has interface for controlling behavior, and output of the component is observable

**Response Measure:** Path coverage of 85% is achieved within 3 hours

# USABILITY

**Source:** end user

**Stimulus:** wish to learn/use/minimize errors/adapt/feel comfortable

**Artifact:** system

**Environment:** configuration or runtime

**Response:** provide ability or anticipate

**Measure:** task time, number of errors, user satisfaction, efficiency

# LESSONS LEARNED

Requirements engineering is important and not trivial. Involves:

- Elicitation
- Specification
- Validation

Architecture is driven by requirements:

- Functional
- Non-functional (quality attributes)

Quality attributes

- Defined as scenarios
- Achieved using appropriate tactics

# DESIGN TRADE-OFFS

## **QAs are rarely orthogonal**

- They interact, affect each other
- highly secure system may be difficult to integrate
- highly available application may trade-off lower performance for greater availability
- high performance application may be tied to a given platform, and hence not be easily portable

Architects must create solutions that makes **sensible design compromises**

- Not possible to fully satisfy all competing requirements
- Must satisfy all stakeholder needs
- This is the difficult bit!

# NEXT TIME

- Review
- Your questions