



SOFTWARE DESIGN

Spring 2023

MANAGEMENT DETAILS

- Instructor: Mihaela Dînşoreanu
- Contact:
 - room D01, Baritiu 26-28
 - E-mail: mihaela.dinsoreanu@campus.utcluj.ro
 - Teams, Moodle (Mon-Fri, 8.00a.m. – 6.00p.m.)

COURSE SETUP

- **Synchronous Communication** (Lectures, Lab/Project)
 - **Onsite** according to the Schedule
- **Asynchronous communication** (Course materials, Quizzes)
 - **Moodle** platform
moodle.cs.utcluj.ro
course enrollment key: **course_PS/SD2023**
 - **Teams** platform

GRADING (SEE FISA DISCIPLINEI)

- Project 20%
- Lab 20%
- Course Evaluation 60%
 - Quizzes during Lectures 10%
 - Final Exam 50%

YOUR TEACHING ASSISTANTS (TA)

30231 – Iulia Cornea

30232 – Grigore Vlad

30233 – Anca Iordan

30234 – Georgiana
Cornea

30235 – Anca Iordan

30236 – Adriana Tornea

30237 – Paul Stanescu

30238 – Lucian Braescu

30431 – Cristina Mihai

30432 – Andrei Isfan

30433 – Bogdan Bindea

30434 – Anca Iordan

CSC – Anca Iordan

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 ≥ 5 for both the lab and project to attend the final exam
- **Attend the lab sessions only when your group is scheduled**

RESEARCH (EARLY START 😊)

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, mihaela.dinsoreanu@cs.utcluj.ro, camelia.lemnaru@cs.utcluj.ro} **by 17th 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 post any ideas/suggestions/inputs in the discussions area in Teams.
- TODO by next week
 - Enroll in Moodle
 - Review the lecture (recommended to read from references)
 - Read any announcements I might post
 - Attend the lab/project meeting

REFERENCES [1]

Software Architectures

- Juval Lowy, *Righting software*, O'Reilly, 2020
- Robert Martin, *Clean Architecture*, Prentice Hall, 2018
- 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*, 3rd 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 Sornmerlad, 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

COURSE CONTENT [TENTATIVE]

1. Introduction. OOP Concepts. SOLID
2. Class design principles (GRASP). Package Design principles
3. Architectural Patterns
- 4-5 Organizing the Business Logic
6. Data access
7. Live coding session/Midterm exam
8. Concurrency. Presentation
- 9-11. Design Patterns
12. Quality Attributes
13. Review, Q&A

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:

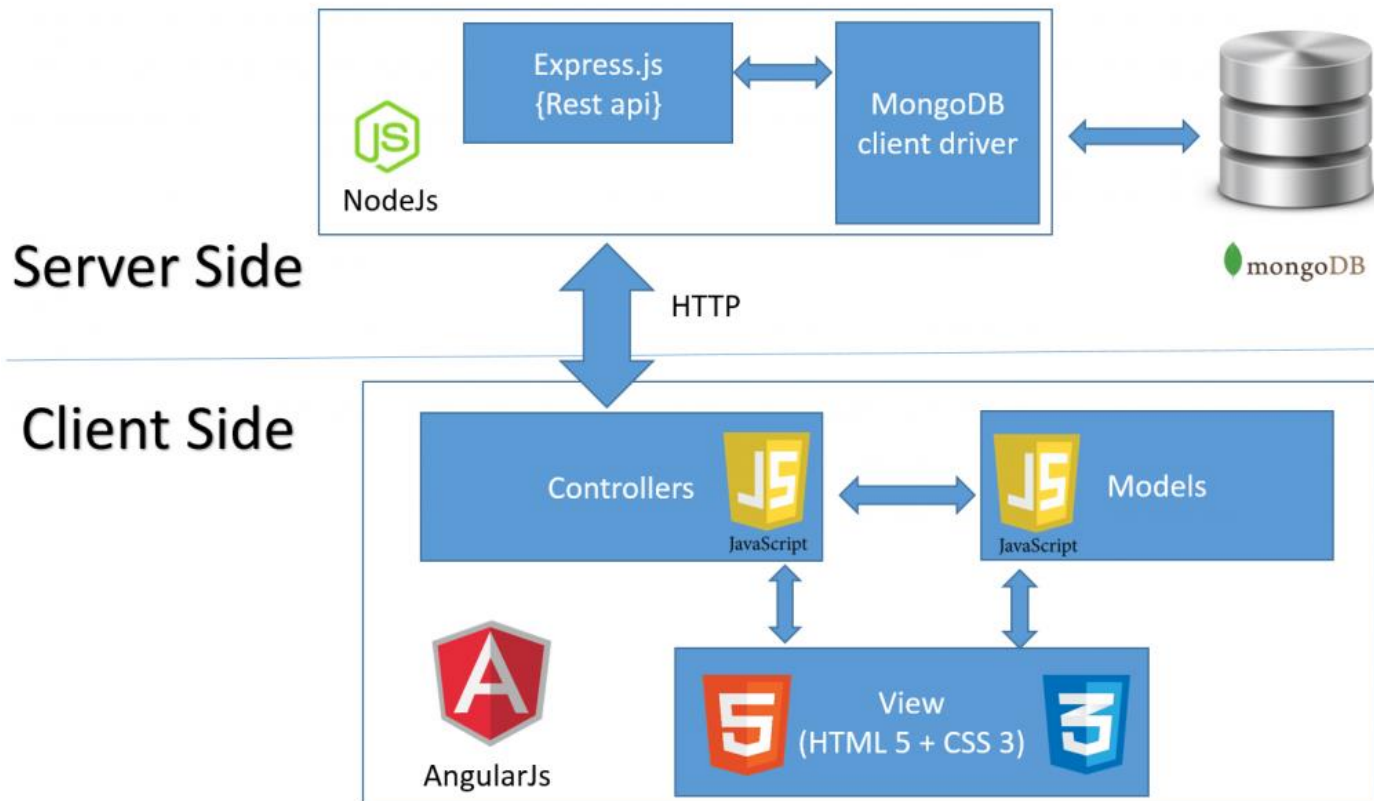
- The **use cases** and **operation** of the system (“screaming” architecture).
- The **maintenance** of the system.
- The **development** of the system.
- The **deployment** of the system.

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

HOW DO WE GET TO DEVELOP ARCHITECTURES?

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, 2018.

[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  
}
```

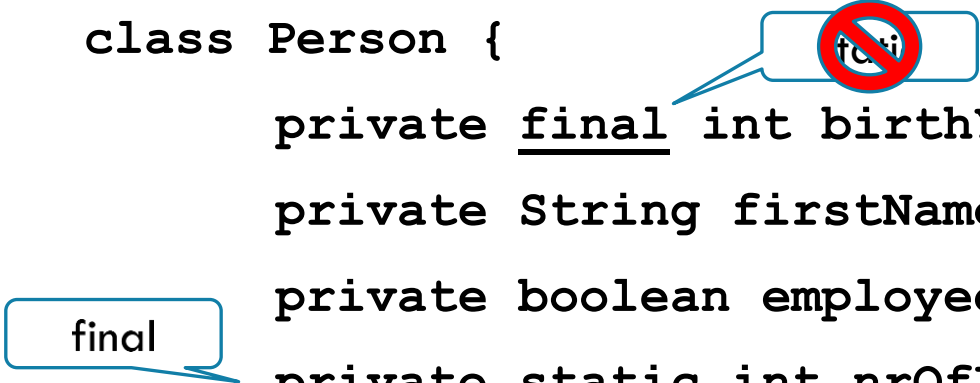
What should we make static?

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;  
  
    //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 !

ATTENTION! What does that mean for

- 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;
        birthYear = bY;
        employed = e;
    }
    public void setFirstName(String n)
    {
        firstName = n;
    }
    public void setLastName(String n)
    {
        lastName = n;
    }
    public int calculateAge()
    {
        Date dt = new Date();
        int year = dt.getYear();
        return (1900+year - birthYear);
    }
    public String toString(){
        return "The person "+firstName + ' ' +lastName+ "is "+calculateAge() + " years old";
    }

    public static void changeName(Person p)
    {
        System.out.println("Before change "+p);
        p.setFirstName("Pandele");
        System.out.println("Inside change method "+p);
    }
    public static void main(String args[])
    {
        Person me = new Person("Mihaela", "Dinsoreanu", 1970, true);
        changeName(me);
        System.out.println("Outside change method "+me);
    }
}

```

run:

Before change The person Mihaela Dinsoreanuis 52 years old

Inside change method The person Pandele Dinsoreanuis 52 years old

Outside change method The person Pandele Dinsoreanuis 52 years old

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;}  
}
```

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 ....."
```

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

EXAMPLE

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

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

CLASS DESIGN PRINCIPLES

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion



SOLID

Software Development is not a Jenga game

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**.

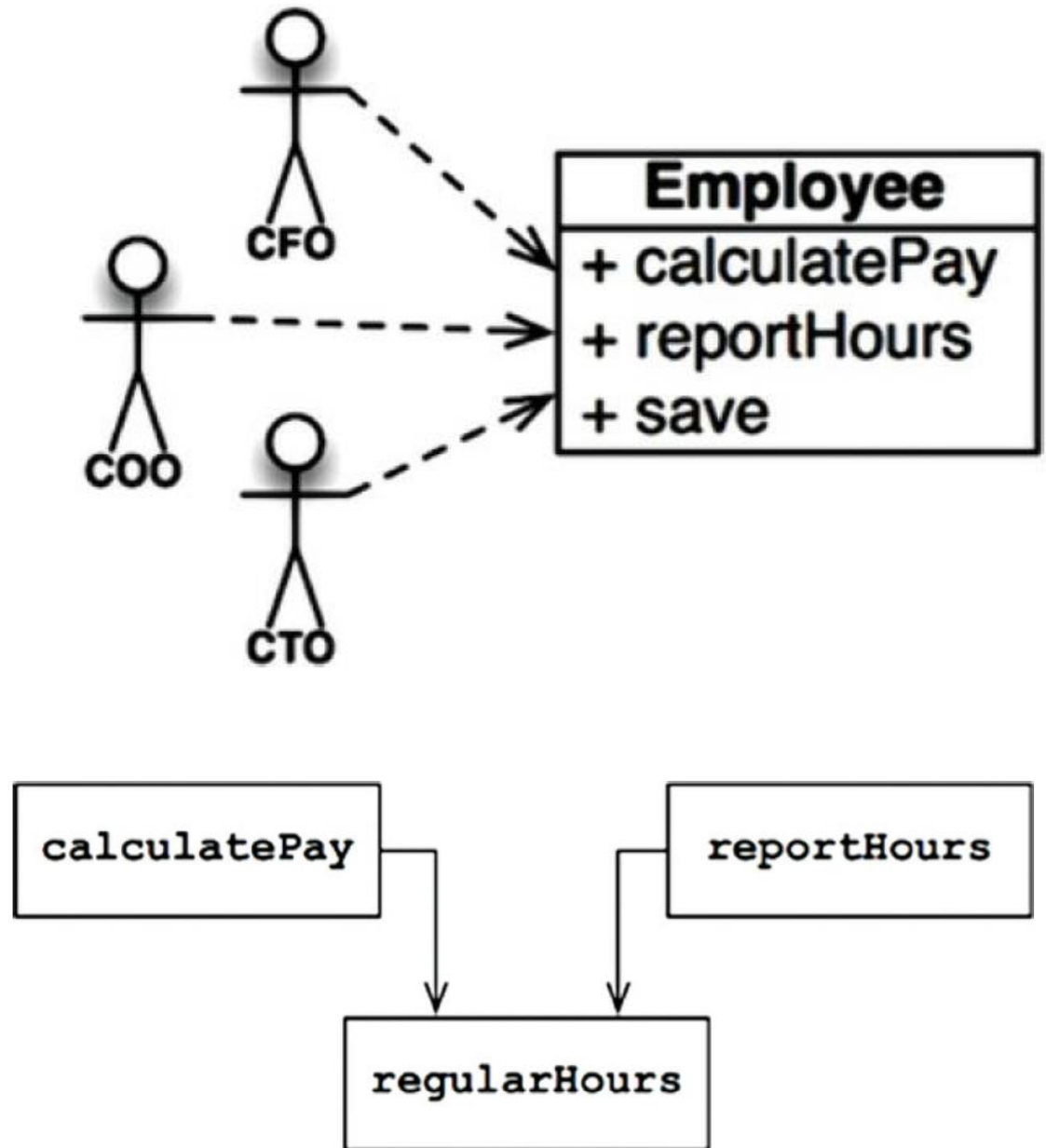
A module should be **responsible to one**, and only one, **actor**.



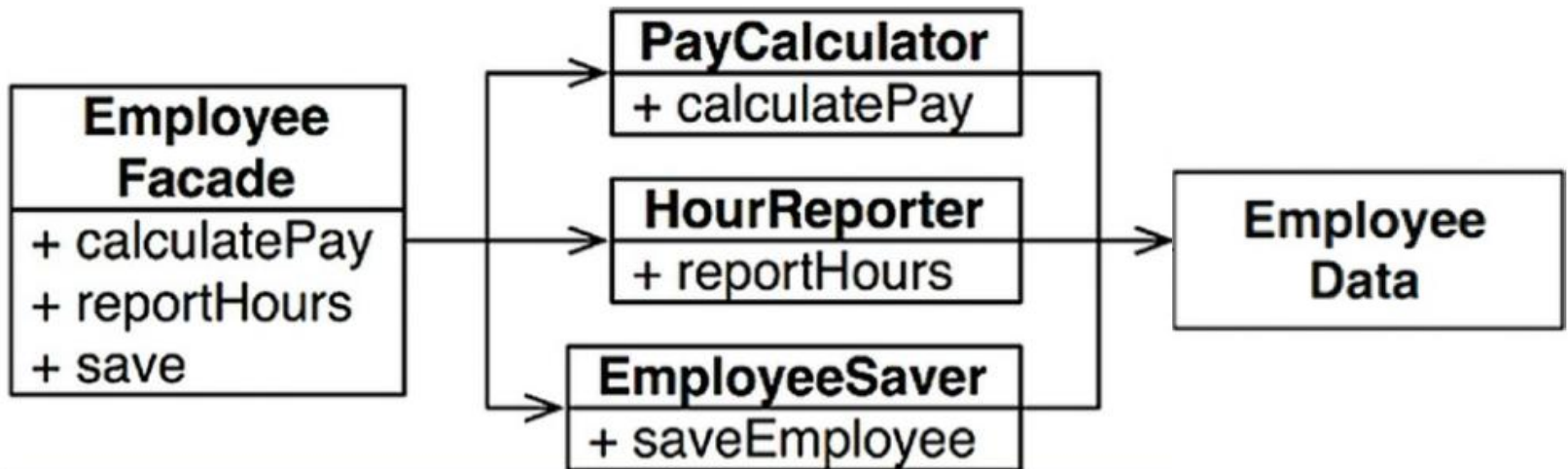
SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

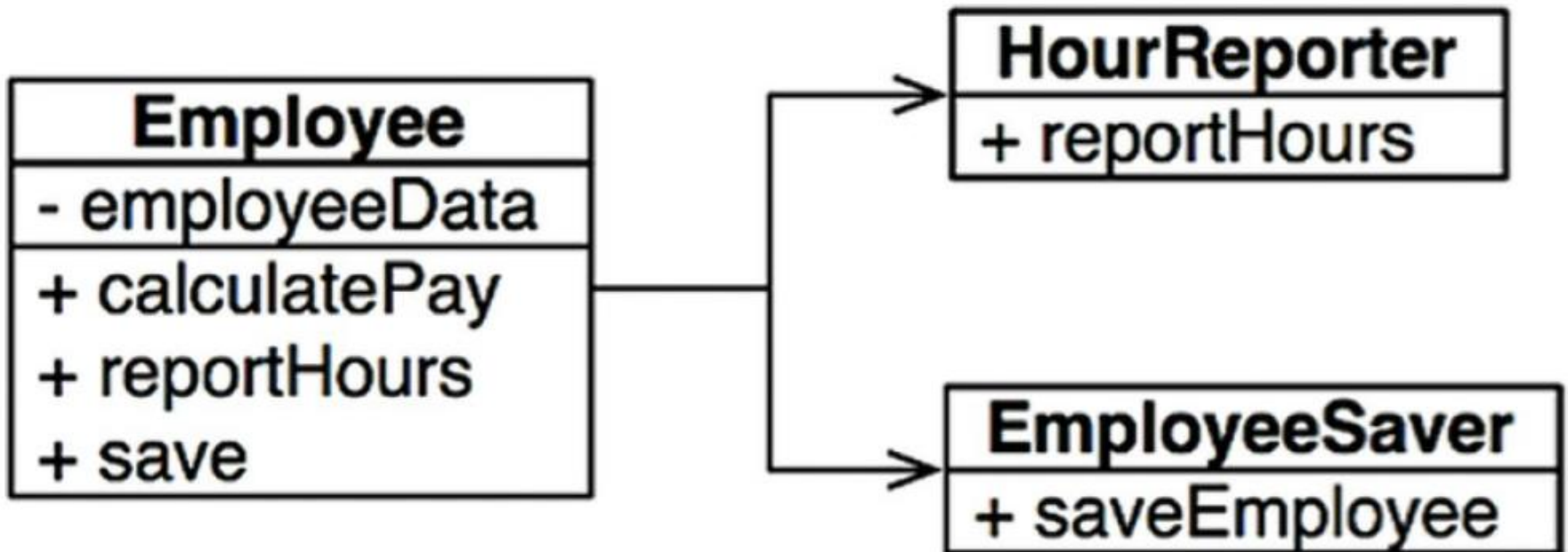
EXAMPLE



SOLUTION(S)



OR...



OPEN-CLOSED PRINCIPLE (OCP)

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

EXTENSION??

- by inheritance?
- by composition?



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

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

Define an abstraction

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

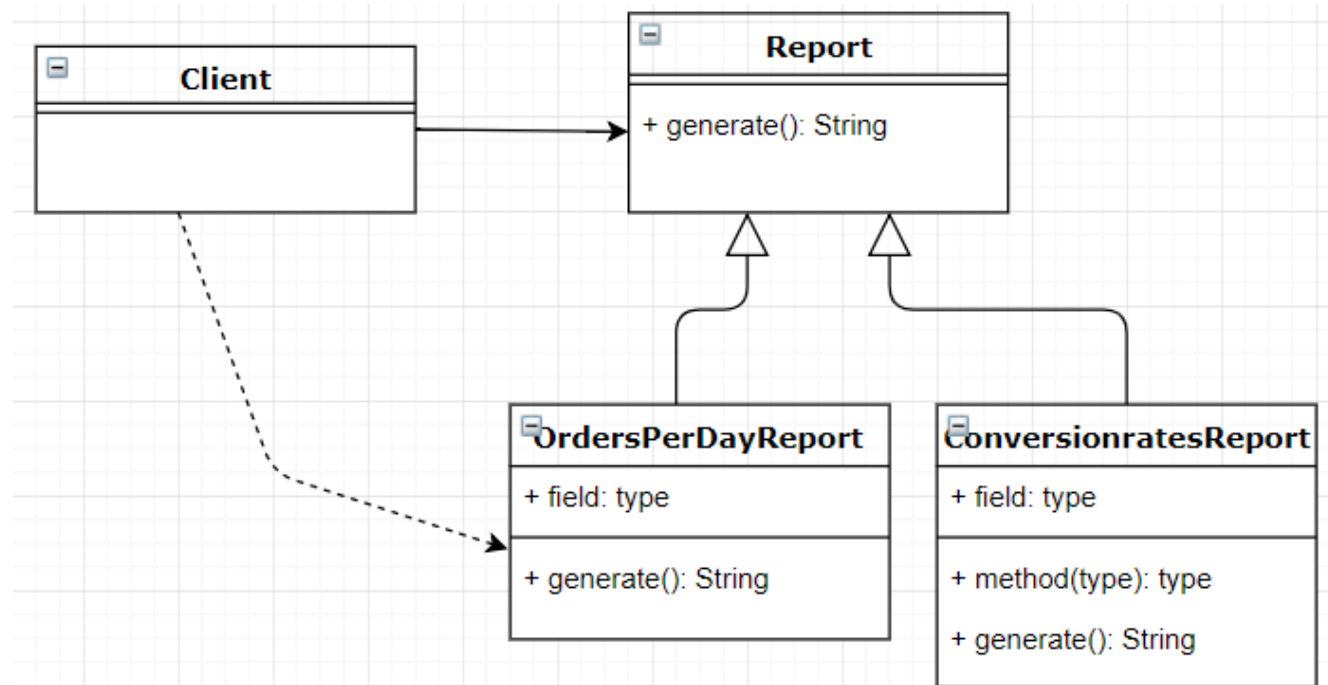
Implement the abstraction

```
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

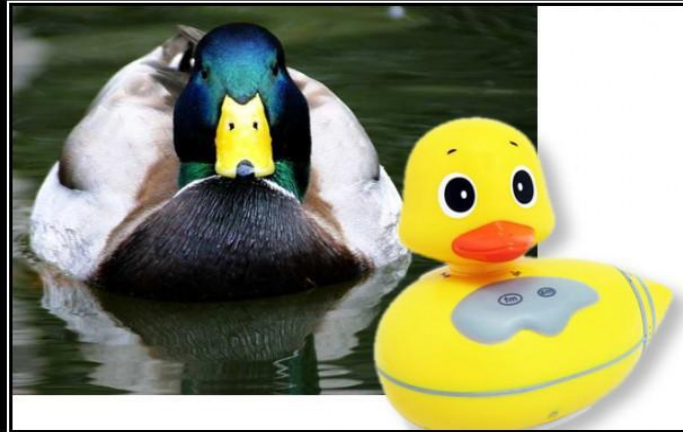
- place volatile policy decisions in a separate location (e.g. a configuration file or a separate object)
- minimizes future change locations

LSKOV 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 .”

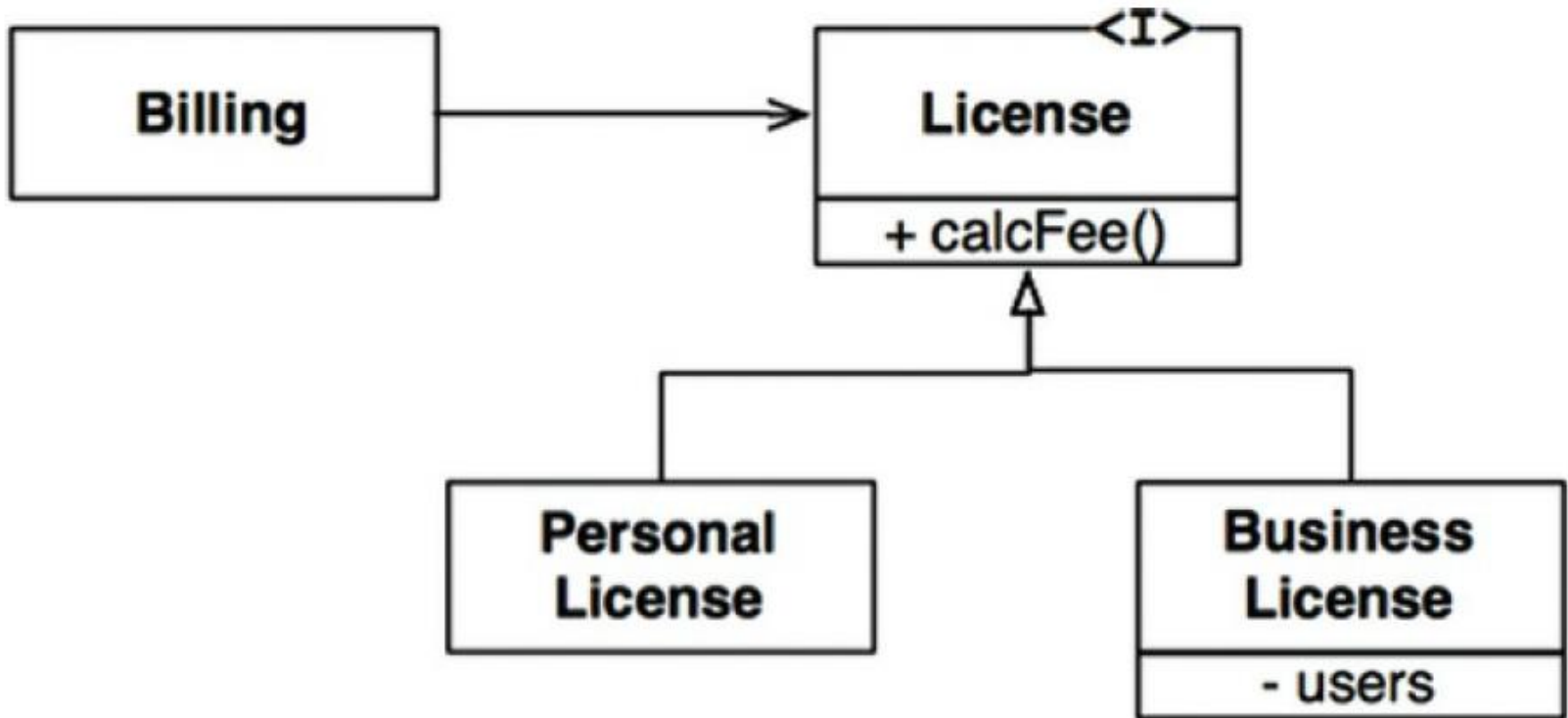
[Barbara Liskov, 1988]



LSKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

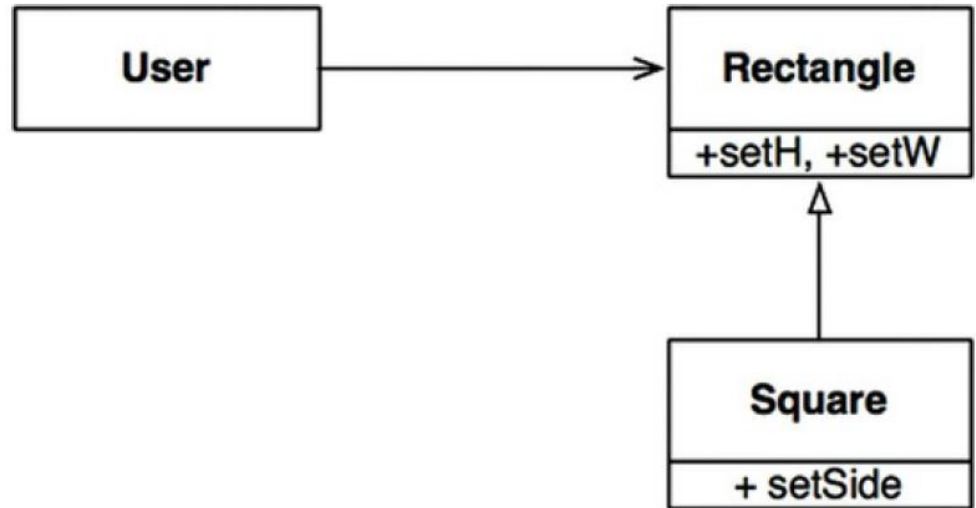
EXAMPLE



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 subclass method are no stronger than the base class method.**
- 2. Postconditions Q of the subclass 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)

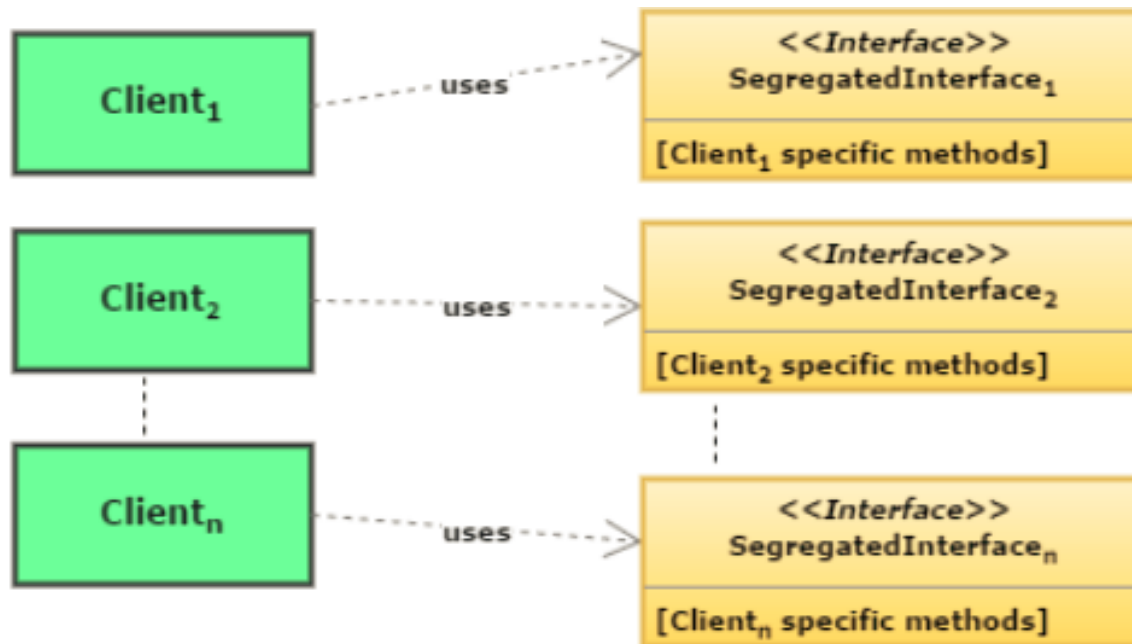
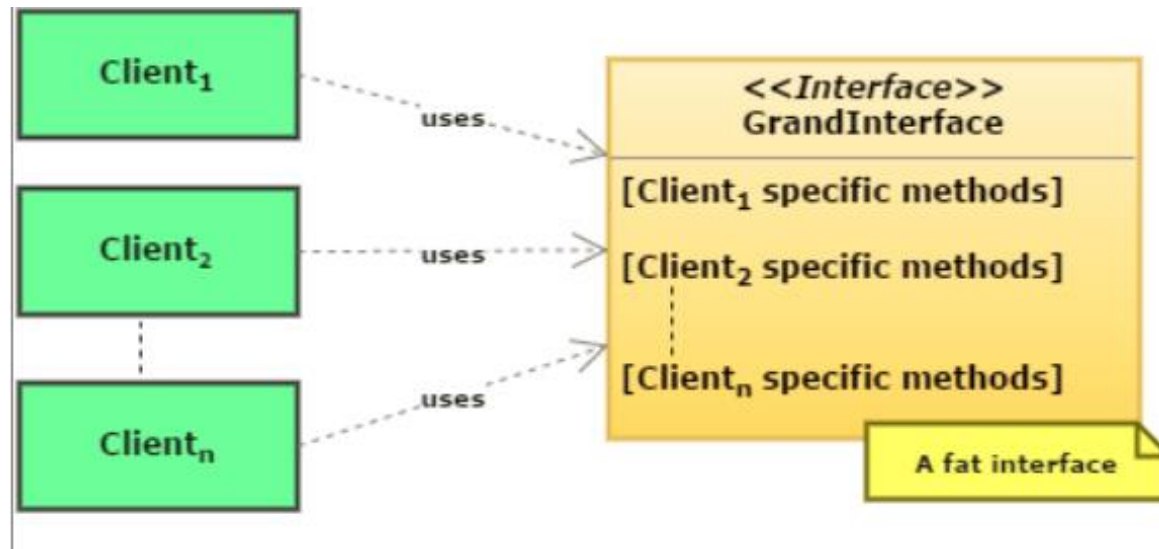
Clients should not be forced to depend upon interfaces that they don't use.



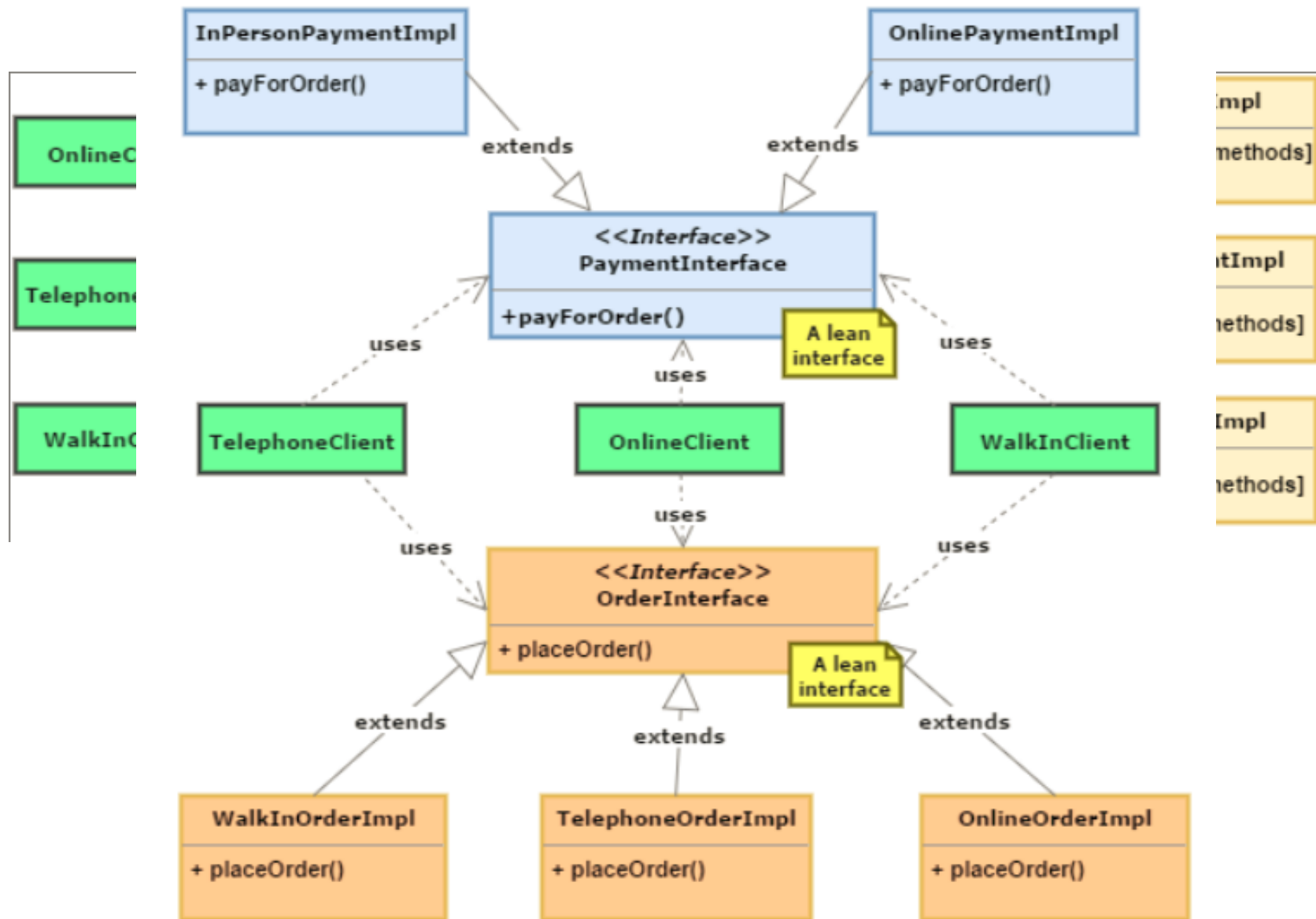
INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

ISP

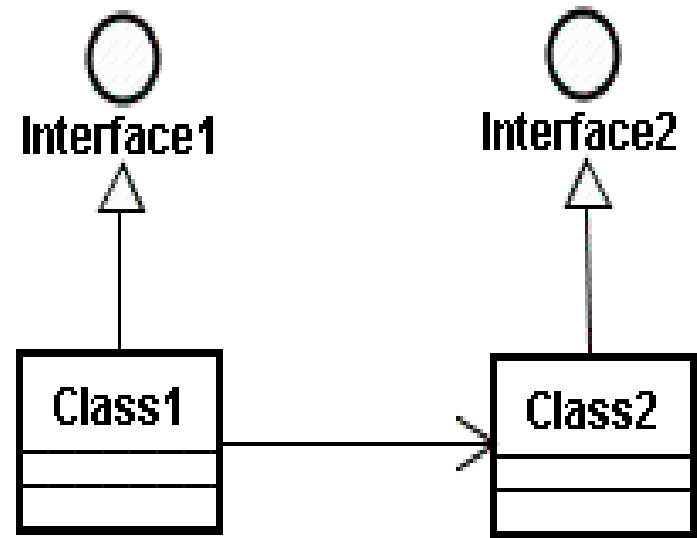
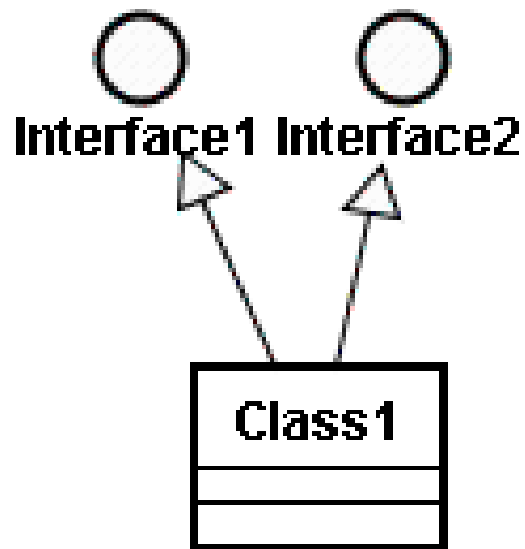


ISP EXAMPLE



ISP EXAMPLE

Separation thru Multiple Inheritance vs. separation thru Composition



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 PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

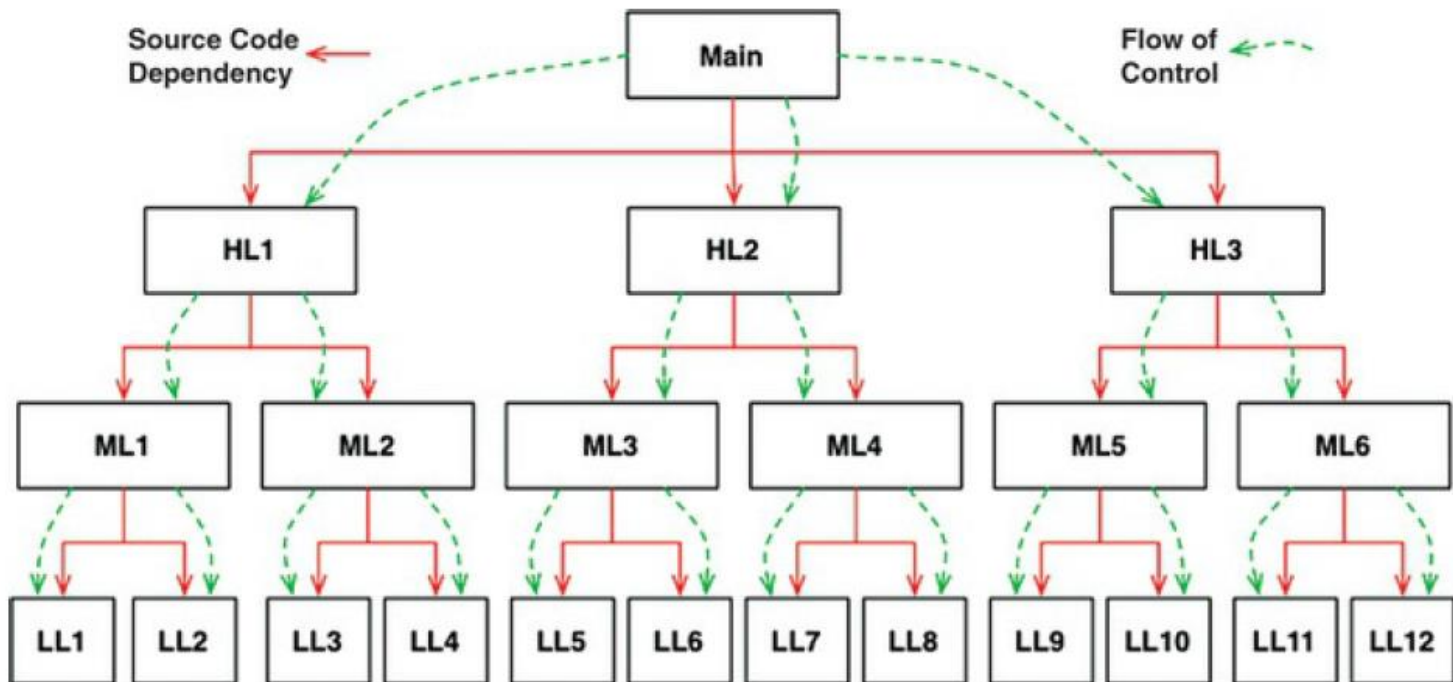
DEPENDENCY INVERSION MOTIVATION

Traditional calling tree Main

#include (C++)

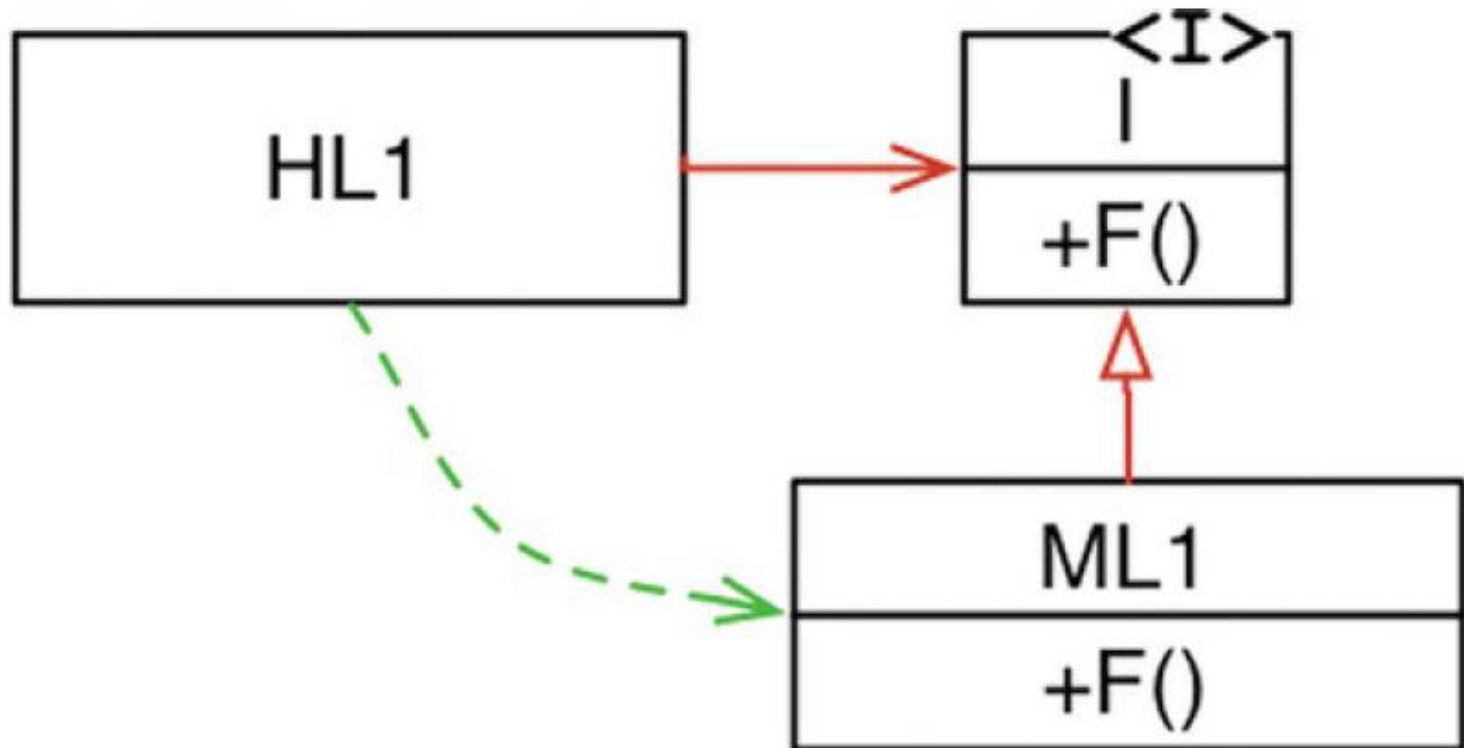
import (Java)

using (C#)

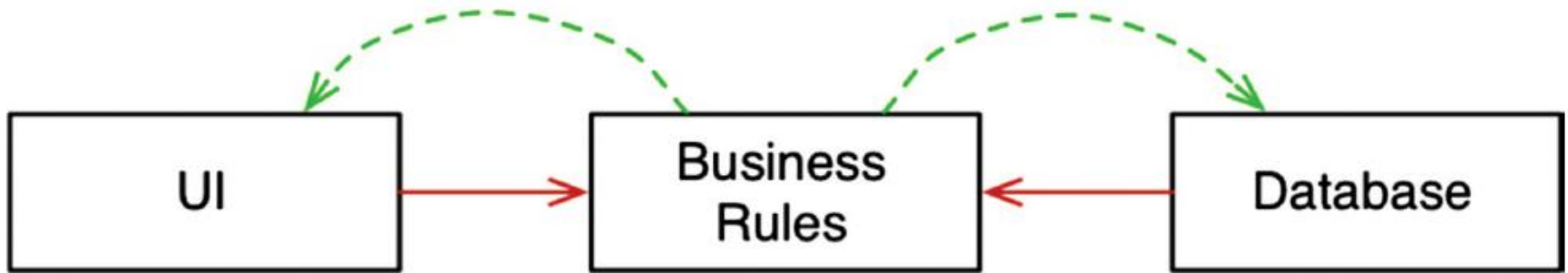


DEPENDENCY INVERSION

The power of polymorphism!

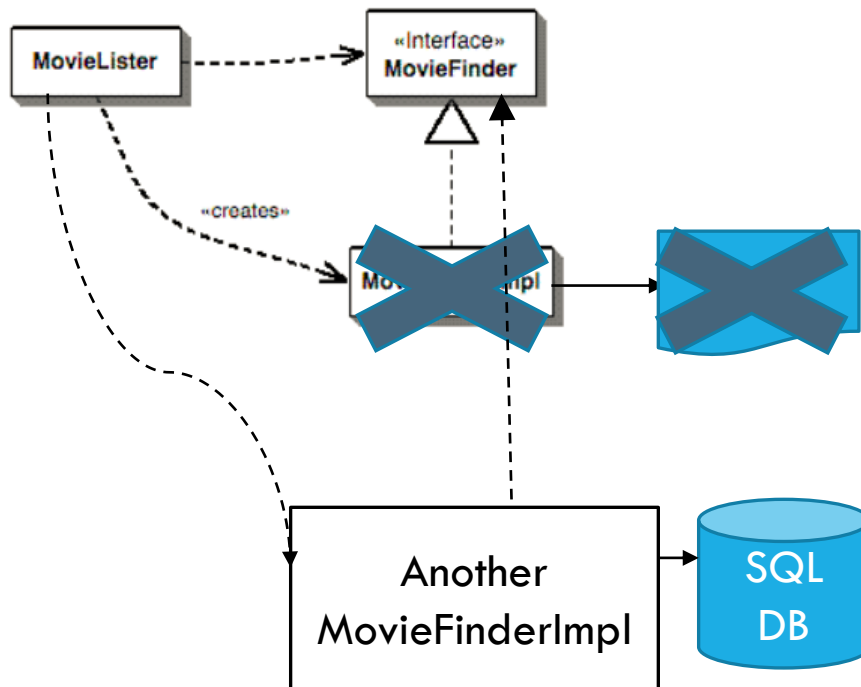


DEPENDENCY INVERSION IN LAYERS



- 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 MovieList {
    private MovieFinder finder;

    public MovieList() {
        this.finder = new
        MovieFinderImpl("movies.txt");
    }...}
```

TYPES OF DEPENDENCY INJECTION

Constructor

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

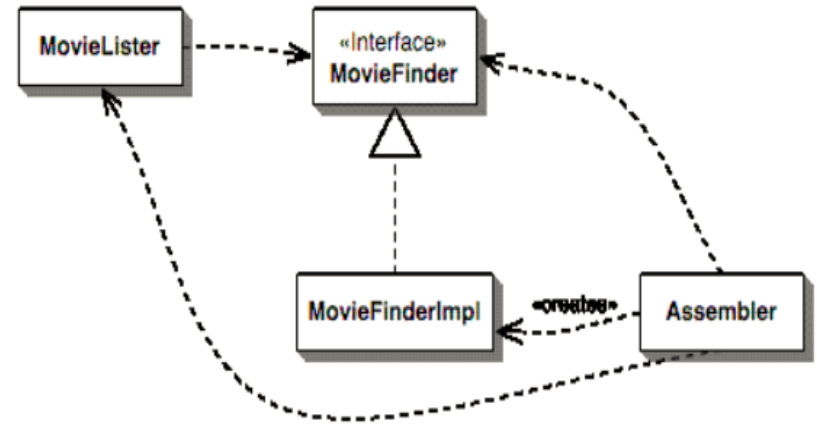
class TextMovieFinder implements MovieFinder
    public TextMovieFinder(String filename) {
        this.filename = filename; }

//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(MovieLi
ster.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 BUT NOT EQUALLY GOOD!

- Design
- Technology
- Code
- Deployment

Basic Design Principles have to be considered!