

Software Design and Architecture (OO Design & Programming)

Course Introduction
Term 3, 2025

COMP2511, CSE, UNSW



Our Team

Course Convenor and Lecturer [Week 07 to 10]:

Dr Jesse Laeuchli (j.laeuchli@unsw.edu.au)

Lecturer [Week 01 to 05]:

Dr Ashesh Mahidadia (a.mahidadia@unsw.edu.au)

Course Admin Team:

Alvin Cherk, Daniel Khuu, Michael Mospan, Grace Kan

Tutors:

30+ passionate tutors!

Course Account Email: **cs2511@cse.unsw.edu.au**

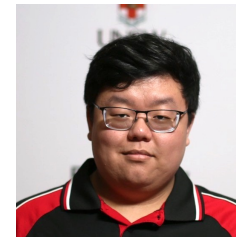
*(Unless you specifically require to contact a member of the admin team, please use the **above email** for any queries related to the course.)*



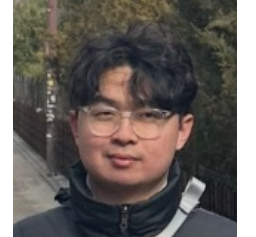
Jesse



Ashesh

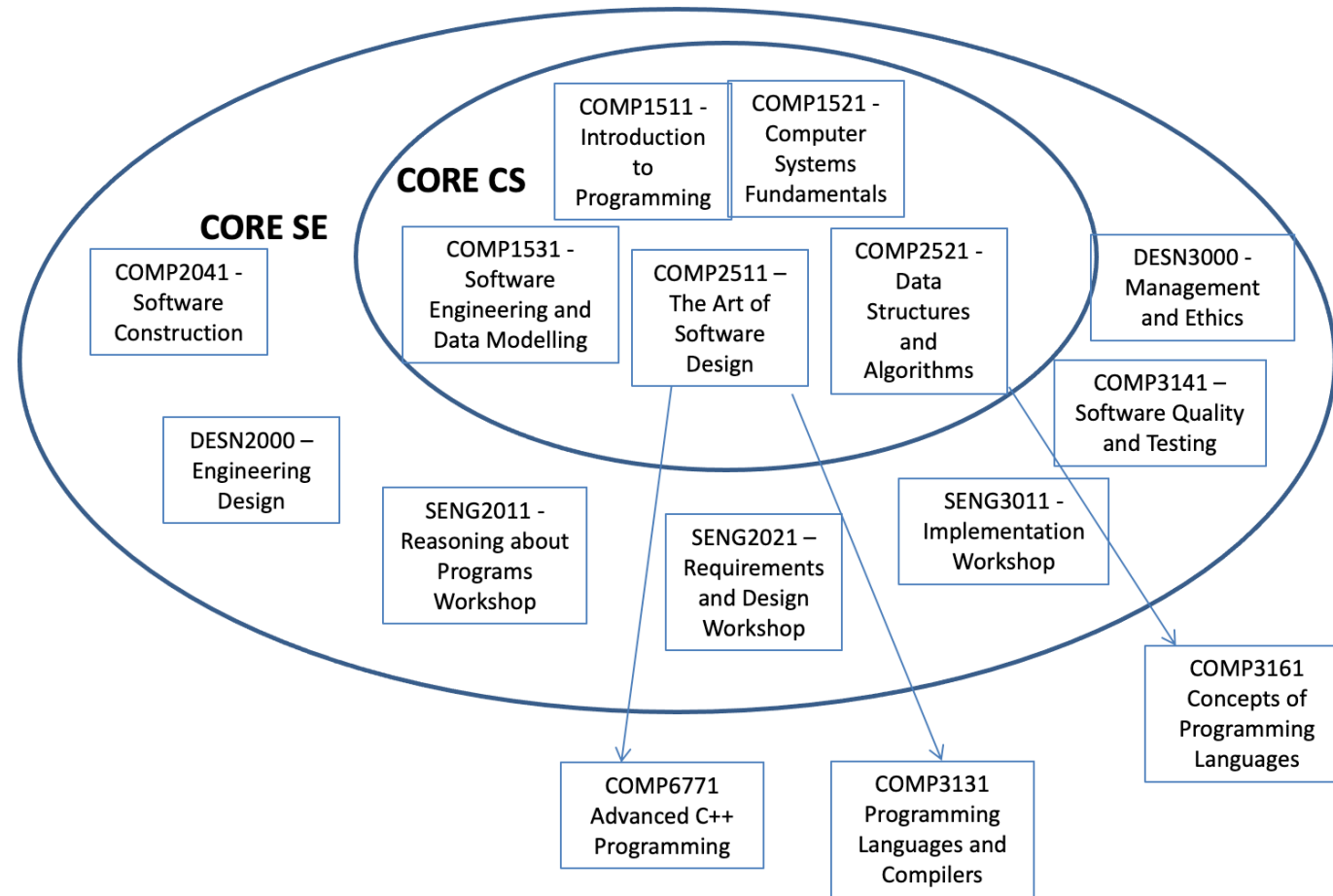


Alvin



Daniel

Course Context



The Story So Far: Course Context

COMP1511: Solving problems with computers, the wonder and joy of programming

COMP1521: Getting right down into the silicon

COMP1531: Solving problems in a team; programming in the large

COMP2521: Solving problems at scale using data structures and algorithms

COMP2511???

COMP2511

- ❖ We can write code, but how do we write good code?
- ❖ Designing elegant and beautiful software.
- ❖ Shades of Grey - things aren't clear cut; writing good software is an art.
- ❖ Grow from a programmer into a software engineer by following a systematic design and development strategy.

COMP 2511 Major Themes

- ❖ Analyse characteristics of **elegantly written software**, and learn how to create and maintain well-designed codebases
- ❖ Apply widely used **Software Design** and **Architectural Patterns** to create **extensible, flexible, maintainable** and **reusable software systems**
- ❖ Apply the principles of **Object-Oriented Design** to solve problems.

COMP 2511 Major Themes

- ❖ Create **medium-scale systems** from scratch, and work on existing systems as part of the Software Development Life Cycle.
- ❖ For **specific software development scenarios**, evaluate different design and architectural paradigms and methodologies based on their origins and suitability.
- ❖ Create software solutions using **an enterprise programming language** within an integrated development environment (IDE).

Credit teaching material

- ❖ No textbook, the lecture slides cover the required topics.
- ❖ However, you are strongly encouraged to read additional material and the reference books.
- ❖ In the lecture notes, some content and ideas are drawn from:
 - *Head First Design Patterns* , by Elisabeth Freeman and Kathy Sierra, The State University of New Jersey
 - *Head First Software Architecture*, by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc.
 - *Fundamentals of Software Architecture*, 2nd Edition, by Mark Richards, Neal Ford
 - *Refactoring: Improving the design of existing code*, by Martin Fowler
 - Material from many popular websites.

How do we obtain our educational objectives?

- ❖ **Lectures:** 4-hour lectures (9 weeks)
- ❖ **Tutorials:**
 - ❖ A 1-hour tutorial session per week, which is scheduled before the lab.
 - ❖ Online Tutorials/Labs will be run via **MS Teams** .
 - ❖ Tutorials are understanding-driven - interactive examples to illustrate concepts discussed in lectures
 - ❖ Solutions and recording to tutorials posted at the end of each week

How do we obtain our educational objectives?

❖ Labs:

- ❖ 2 hours each week, straight after tutorial
- ❖ Like most CSE core courses
- ❖ Lab retros posted after due date on course website
- ❖ Online Run via MS Teams

Assessments

Coursework (15%)

- ❖ Your coursework mark is made up of marks associated with the lab exercises.
- ❖ There are **eight** labs, each worth **ten** marks.
- ❖ We will cap total coursework marks at 70 (which will translate to 15%), leaving **one lab as a buffer**.
- ❖ If you attend all seven labs, we will add all seven lab marks and cap the total coursework marks to 60.
- ❖ The specific marking criteria for each lab will be outlined in the respective specifications.
- ❖ A general guide for the criteria that your tutor/lab assistant will use to assess you is available on the class webpage.
- ❖ You (students) must get your lab **manually marked** each week
- ❖ Lab09 (in Week 9) is an attendance and participation lab

Assignment I (15%)

- ❖ The marking criteria for the assignment will be outlined in the specification which will be released Tuesday of Week 2.
- ❖ Due Wednesday 3pm Week 5.
- ❖ Completed **individually**.

Assignment II (20 %)

- ❖ The marking criteria for the project will be outlined in the specification which will be released Thursday Week 5.
- ❖ Due **Wednesday 3pm week 10**
- ❖ Completed **individually**.

Final Exam (50%)

- ❖ In 25T3 the COMP2511 exam will be **held in person in the CSE Labs, and invigilated**.
- ❖ **All students** are required to take **the final exam in person**, even if they have enrolled in online classes. In 25T3, there will be no online exams.
- ❖ **Hurdle** : In order to pass the course, it is required for the student to achieve a **minimum of 40%** (20 out of 50) marks **in the final examination**.
- ❖ Students are eligible for a Supplementary Exam if and only if:
 - Students cannot attend the final exam due to illness or misadventure. Students must formally apply for a special consideration, and it must be approved by the respective authority.

Assumed Knowledge

- ❖ Confident programmers
 - Familiar with C and Python/JS programming concepts
- ❖ Able to work in a team
 - Git
 - Working with others
- ❖ Understand basic testing principles
- ❖ Understand basic software engineering design principles (DRY, KISS)

Assumed Knowledge

- ❖ What we don't assume:
 - Knowledge of Java
 - Understanding of Object-Oriented Programming
- ❖ **This is not a Java course**

Course philosophy

- ❖ A step up from first year courses
- ❖ Challenging but achievable
- ❖ Develop skills in time management, teamwork as well as critical thinking
- ❖ Highly rewarding

Support

- ❖ Supporting you is our job :)
- ❖ Help Sessions
 - Lots of them with fantastic tutors
 - Feedback on work, help with problems, clarifying ideas
 - You are expected to have done your own research and debugging before arriving

Support

- ❖ Course [Forum](#)
 - Ask questions and everyone can see the answers!
 - Make private posts for sharing code
 - Response time
- ❖ Course Account - cs2511@cse.unsw.edu.au
 - Sensitive/personal information
- ❖ During the project - [your tutor](#)
- ❖ Go to [help sessions](#) for help on concepts
- ❖ [Post on the forum](#) if you need more immediate lab feedback
- ❖ There are [no late extensions](#) on labs unless in extenuating circumstances, email cs2511@cse.unsw.edu.au

Support - UNSW

- ❖ **Special Consideration** -
<https://student.unsw.edu.au/special-consideration>
- ❖ **Equitable Learning Services** -
<https://student.unsw.edu.au/els>

Mental Health & Wellbeing

- ❖ UNSW Psychology & Wellness - <https://student.unsw.edu.au/mhc>
- ❖ UNSW Student Advisors - <https://student.unsw.edu.au/advisors>
- ❖ Reach out to us at cs2511@cse.unsw.edu.au
- ❖ Check in with each other
- ❖ Talk to someone

Technology Stack

- ❖ Java Version – JDK 17
- ❖ VSCode
- ❖ Gradle 8.8
- ❖ Gitlab (+ CI pipelines)

Feedback

- ❖ We love feedback :)
- ❖ Changes made to the course this term based on constructive student feedback
- ❖ We always want to continuously improve
 - This term, we are incorporating software architecture topics to enhance the course's relevance to real-world applications.
- ❖ Feedback form
- ❖ Course account

Respect

- ❖ Yourself, each other, course staff

It's time to lift off for 25T3 !!!!



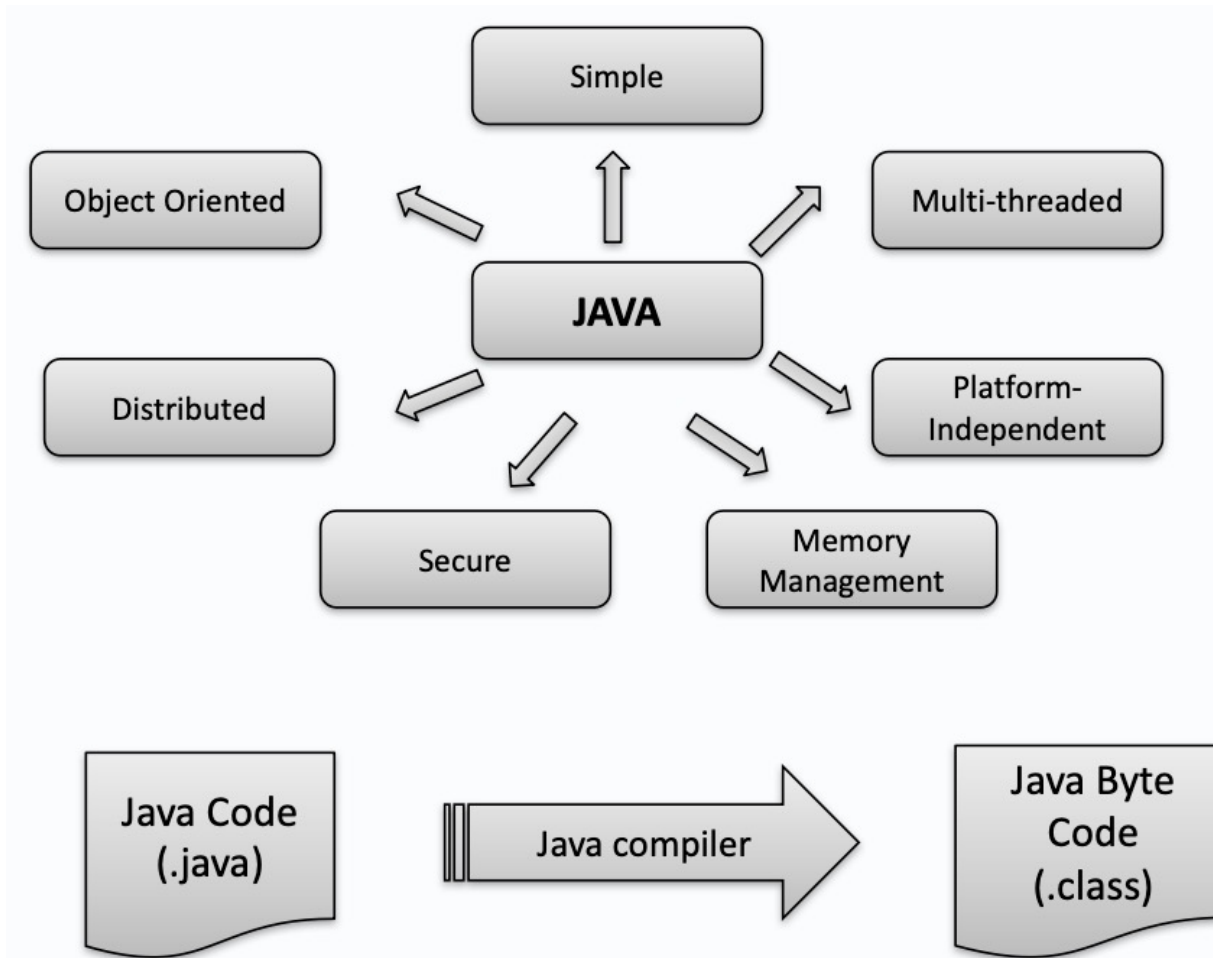
OOP in Java

COMP2511, CSE, UNSW



UNSW
SYDNEY

The Java Platform



OOP in Java

- ❖ Object Oriented Programming (OOP) v Inheritance in OOP
- ❖ Introduction to Classes and Objects v Subclasses and Inheritance
- ❖ Abstract Classes
- ❖ Single Inheritance versus Multiple Inheritance
- ❖ Interfaces
- ❖ Method Forwarding (Has-a relationship) v Method Overriding (Polymorphism)
- ❖ Method Overloading
- ❖ Constructors

Object Oriented Programming (OOP)

In procedural programming languages (like 'C'), programming tends to be **action-oriented**, whereas in Java - programming is **object-oriented**.

In **procedural** programming,

- groups of actions that perform some task are formed into functions and functions are grouped to form programs.

In **OOP**,

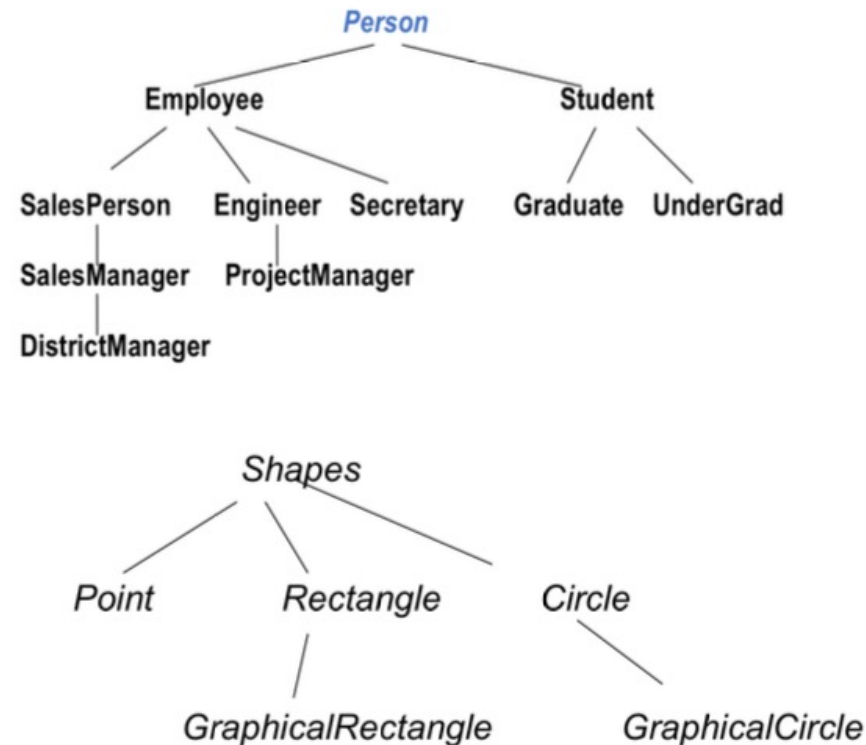
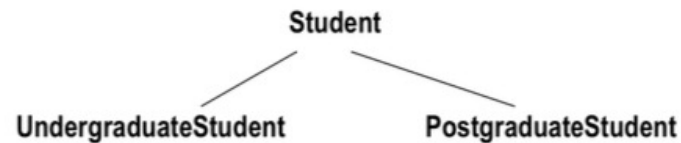
- programmers concentrate on creating their own user-defined types called **classes**.
- each **class** contains **data** as well as the set of **methods** (procedures) that manipulate the data.
- an instance of a user-defined type (i.e. a **class**) is called an **object**.
- OOP **encapsulates** data (attributes) and methods (behaviours) into **objects**, the data and methods of an object are intimately tied together.
- Objects have the property of *information hiding*.

Inheritance in Object Oriented Programming (OOP)

- ❖ **Inheritance** is a form of software reusability in which new classes are created from the existing classes by absorbing their attributes and behaviours.
- ❖ Instead of defining completely (separate) new class, the programmer can designate that the new class is to **inherit** attributes and behaviours of the existing class (called **superclass**). The new class is referred to as **subclass**.
- ❖ Programmer can **add more attributes and behaviours** to the *subclass*, hence, normally **subclasses** have **more features** than their *super classes*.

Inheritance in Object Oriented Programming (OOP)

Inheritance relationships form **tree-like hierarchical** structures. For example,



“Is-a” - Inheritance relationship

- ❖ In an “**is-a**” relationship, an object of a subclass may also be treated as an object of the superclass.
- ❖ For example, *UndergraduateStudent* can be treated as *Student* too.
- ❖ You should use *inheritance* to model “is-a” relationship.

Very Important:

- ❖ Don’t use inheritance unless **all or most** inherited attributes and methods **make sense**.
- ❖ For example, mathematically a *circle* is-a (an) *oval*, however you should **not** inherit a class *circle* from a class *oval*. A class *oval* can have one method to set *width* and another to set *height*.

“Has-a” - Association relationship

- ❖ In a “has-a” relationship, a **class object has an object of another class** to store its state or do its work, i.e. it “has-a” reference to that other object.
- ❖ For example, a Rectangle Is-NOT-a Line.
However, we may use a Line to draw a Rectangle.
- ❖ The “has-a” relationship is quite different from an “is-a” relationship.
- ❖ “Has-a” relationships are examples of creating new classes by *composition* of existing classes (as oppose to *extending* classes).

Very Important:

- ❖ Getting “Is-a” versus “Has-a” relationships correct is both *subtle* and potentially *critical*. You should *consider* all *possible* future *usages* of the classes before finalising the hierarchy.
- ❖ It is possible that *obvious solutions may not work* for some applications.

Designing a Class

- Think carefully about the functionality (methods) a class should offer.
- Always **try to keep data private** (local).
- Consider **different ways** an object may be **created**.
- Creating an object may require different actions such as initializations.
- Always initialize data.
- If the object is no longer in use, free up all the associated resources.
- **Break up** classes with **too many responsibilities**.
- In OO, classes are often closely related. “**Factor out**” common attributes and behaviours and place these in a class. Then use suitable relationships between classes (for example, “is-a” or “has-a”).

Introduction to Classes and Objects

- ❖ A class is a collection of **data** and **methods** (procedures) that operate on that data.
- ❖ For example,
a **circle** can be described by the **x, y position** of its centre and by its **radius**.
- ❖ We can define some useful methods (procedures) for circles,
compute **circumference**, compute area, check whether points are inside the circle,
etc.
- ❖ By defining the **Circle class** (as below), we can create a **new data type**.

The Class Circle

For simplicity, the methods for *getter* and *setters* are not shown in the code.

```
public class Circle {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
}
```

Objects are Instances of a class

In Java, objects are created by instantiating a class.

For example,

```
Circle c ;  
c = new Circle ( ) ;
```

OR

```
Circle c = new Circle ( ) ;
```

Accessing Object Data

We can access data fields of an object.

For example,

```
Circle c = new Circle ( ) ;  
  
// Initialize our circle to have centre (2, 5)  
// and radius 1.  
// Assuming, x, y and r are not private  
  
c.x = 2;  
c.y = 5;  
c.r = 1;
```

Using Object Methods

To access the methods of an object, we can use the same syntax as accessing the data of an object:

```
Circle c = new Circle ( ) ;  
double a;
```

```
c.r = 2;      // assuming r is not private
```

```
a = c.area( );
```

```
//Note that its not :  a = area(c) ;
```

Subclasses and Inheritance:

First Approach

We want to implement *GraphicalCircle*.

This can be achieved in at least 3 different ways.

First Approach:

- ❖ In this approach we are creating the **new separate class** for *GraphicalCircle* and **re-writing** the code already available in the class *Circle*.
- ❖ For example, we re-write the methods *area* and *circumference*.
- ❖ Hence, this approach is NOT elegant, in fact its the **worst** possible solution.
Note again, its the **worst** possible solution!

```
// The class of graphical circles

public class GraphicalCircle {
    int x, y;
    int r;
    Color outline, fill;

    public double circumference( ) {
        return 2 * 3.14159 * r ;
    }
    public double area ( ) {
        return 3.14159 * r * r ;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

Subclasses and Inheritance:

Second Approach

- ❖ We want to implement *GraphicalCircle* so that it can make use of the code in the class *Circle*.
- ❖ This approach uses “**has-a**” relationship.
- ❖ That means, a *GraphicalCircle* has a (mathematical) *Circle*.
- ❖ It uses methods from the class *Circle* (*area* and *circumference*) to define some of the new methods.
- ❖ This technique is also known as **method forwarding**.

```
public class GraphicalCircle2 {  
    // here's the math circle  
    Circle c;  
    // The new graphics variables go here  
    Color outline, fill;  
  
    // Very simple constructor  
    public GraphicalCircle2() {  
        c = new Circle();  
        this.outline = Color.black;  
        this.fill = Color.white;  
    }  
  
    // Another simple constructor  
    public GraphicalCircle2(int x, int y, int r,  
                             Color o, Color f) {  
        c = new Circle(x, y, r);  
        this.outline = o;  
        this.fill = f;  
    }  
  
    // draw method , using object 'c'  
    public void draw(Graphics g) {  
        g.setColor(outline);  
        g.drawOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
        g.setColor(fill);  
        g.fillOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);  
    }  
}
```


Subclasses and Inheritance:

Third Approach – Extending a Class

- ❖ We can say that *GraphicalCircle* **is-a** *Circle*.
- ❖ Hence, we can define *GraphicalCircle* as an **extension**, or *subclass* of *Circle*.
- ❖ The subclass *GraphicalCircle* **inherits** all the variables and methods of its superclass *Circle*.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color outline, fill;
    public GraphicalCircle(){
        super();
        this.outline = Color.black;
        this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                           int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }
}
```

Subclasses and Inheritance: Example

We can assign an instance of *GraphicCircle* to a *Circle* variable. For example,

```
GraphicCircle gc = new GraphicCircle();  
...  
double area = gc.area();  
...  
Circle c = gc;  
// we cannot call draw method for "c".
```

Important:

- ❖ Considering the variable "c" is of type *Circle*,
- ❖ we can only access attributes and methods available in the class *Circle*.
- ❖ we **cannot** call *draw* method for "c".

Super classes, Objects, and the Class Hierarchy

- ❖ Every class has a superclass.
- ❖ If we don't define the superclass, by default, the superclass is the class **Object**.

Object Class :

- ❖ Its the only class that does not have a superclass.
- ❖ The methods defined by **Object** can be called by any Java object (instance).
- ❖ Often we need to **override** the following methods:
 - **toString()**
 - read the API at [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString\(\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString())
 - **equals()**
 - read the API at [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object))
 - **hashCode()**

Abstract Classes

Using **abstract** classes,

- ❖ we can declare classes that define **only** part of an implementation,
- ❖ leaving extended classes to provide specific implementation of some or all the methods.

The benefit of an **abstract** class

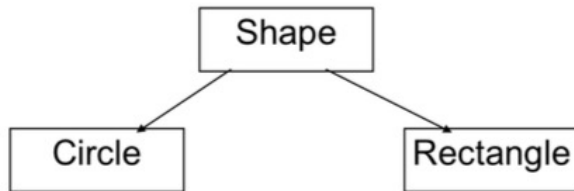
- ❖ is that methods may be declared such that the programmer knows **the interface definition** of an object,
- ❖ however, methods can be **implemented differently** in different subclasses of the abstract class.

Abstract Classes

Some rules about abstract classes:

- ❖ An abstract class is a class that is **declared abstract**.
- ❖ If a class **includes abstract methods**, then the class itself must be declared abstract.
- ❖ An abstract class **cannot be instantiated**.
- ❖ A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass and provides **an implementation** for **all** of them.
- ❖ If a subclass of an abstract class **does not implement** all the abstract methods it inherits, that subclass is itself abstract.

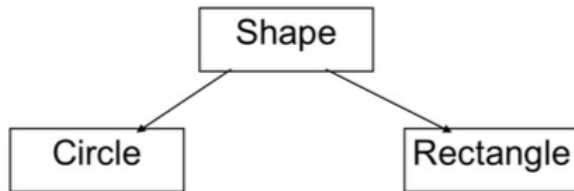
Abstract Class: Example



```
public abstract class Shape {  
  
    public abstract double area();  
    public abstract double circumference();  
  
}
```

```
public class Circle extends Shape {  
  
    protected static final double pi = 3.14159;  
    protected int x, y;  
    protected int r;  
  
    // Very simple constructor  
    public Circle(){  
        this.x = 1;  
        this.y = 1;  
        this.r = 1;  
    }  
    // Another simple constructor  
    public Circle(int x, int y, int r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    /**  
     * Below, methods that return the circumference  
     * area of the circle  
     */  
    public double circumference( ) {  
        return 2 * pi * r ;  
    }  
    public double area ( ) {  
        return pi * r * r ;  
    }  
  
}
```

Abstract Class: Example



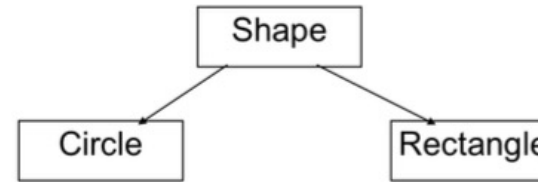
```
public abstract class Shape {  
    public abstract double area();  
    public abstract double circumference();  
}
```

```
public class Rectangle extends Shape {  
    protected double width, height;  
  
    public Rectangle() {  
        width = 1.0;  
        height = 1.0;  
    }  
  
    public Rectangle(double w, double h) {  
        this.width = w;  
        this.height = h;  
    }  
  
    public double area(){  
        return width*height;  
    }  
  
    public double circumference() {  
        return 2*(width + height);  
    }  
}
```


Abstract Class: Example

Some points to note:

- ❖ As **Shape** is an abstract class, we cannot instantiate it.
- ❖ Instantiations of **Circle** and **Rectangle** can be assigned to variables of **Shape**.
No cast is necessary
- ❖ In other words, subclasses of **Shape** can be assigned to elements of an array of **Shape**.
No cast is necessary.
- ❖ We can invoke **area()** and **circumference()** methods for **Shape** objects.



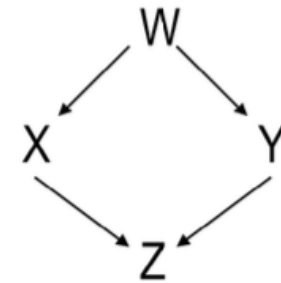
We can now write code like this:

```
// create an array to hold shapes
Shape[] shapes = new Shape[4];
shapes[0] = new Circle(4, 6, 2);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
shapes[3] = new GraphicalCircle(1, 1, 6,
                                Color.green, Color.yellow);

double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    // compute the area of the shapes
    total_area += shapes[i].area();
}
```


Single Inheritance versus Multiple Inheritance

- In Java, a new class can extend exactly one superclass - a model known as *single inheritance*.
- Some object-oriented languages employ *multiple inheritance*, where a new class can have two or more *super classes*.
- In multiple inheritance, **problems** arise when a superclass's behaviour is *inherited in two/multiple ways*.
- Single inheritance precludes some useful and correct designs.
- In Java, **interface** in the class hierarchy can be used to add multiple inheritance, more discussions on this later.

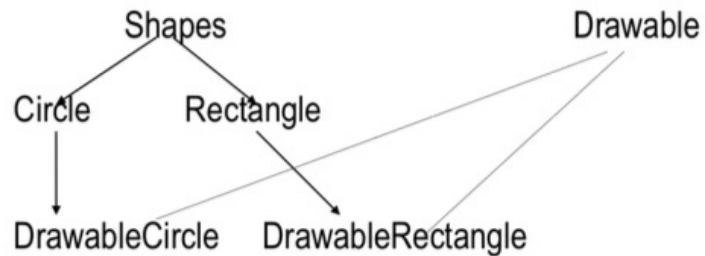


Diamond inheritance problem

Interfaces in Java

- ❖ Interfaces are like abstract classes, but with few **important differences**.
- ❖ All the methods defined within an interface are **implicitly abstract**. (We don't need to use abstract keyword, however, to improve clarity one can use abstract keyword).
- ❖ **Variables** declared in an interface must be **static and final**, that means, they must be **constants**.
- ❖ Just like a class **extends** its superclass, it also can optionally **implements** an interface.
- ❖ In order to implement an interface, a class must first declare the interface in an **implements** clause, and then it must provide an implementation for all of the abstract methods of the interface.
- ❖ A class can “**implements**” **more** than one **interfaces**.
- ❖ More discussions on “**interfaces**” later in the course.

Interfaces in Java: Example



```
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(Graphics g);
}

public class DrawableRectangle
    extends Rectangle
    implements Drawable {

    private Color c;
    private double x, y;

    .....

    // Here are implementations of the
    // methods in Drawable
    // we also inherit all public methods
    // of Rectangle

    public void setColor(Color c) { this.c = c; }
    public void setPosition(double x, double y) {
        this.x = x; this.y = y; }
    public void draw(Graphics g) {
        g.drawRect(x,y,w,h,c); }
}
```

Using Interfaces: Example

- ❖ When a class **implements** an interface, instance of that class can also be **assigned to** variables of the **interface type**.

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];

DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3,
4.5);

// The shapes can be assigned to both arrays
shapes[0] = dc; drawables[0] = dc;
shapes[1] = ds; drawables[1] = ds;
shapes[2] = dr; drawables[2] = dr;

// We can invoke abstract method
// in Drawable and Shapes

double total_area = 0;
for(int i=0; i< shapes.length; i++) {

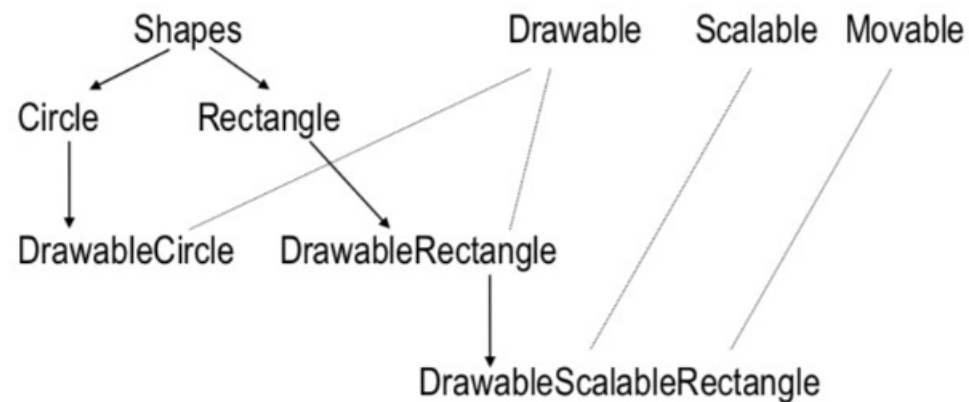
    total_area += shapes[i].area();

    drawables[i].setPosition(i*10.0, i*10.0);

    // assume that graphic area 'g' is
    // defined somewhere
    drawables[i].draw(g);
}
```

Implementing Multiple Interfaces

A class can **implements** more than one interfaces. For example,



```
public class DrawableScalableRectangle
    extends DrawableRectangle
    implements Movable, Scalable {

    // methods go here ....

}
```

Extending Interfaces

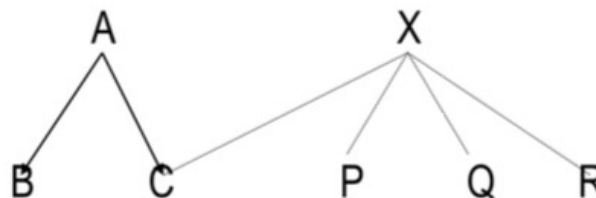
- ❖ Interfaces can have **sub-interfaces**, just like classes can have subclasses.
- ❖ A sub-interface **inherits all** the abstract methods and constants of its super-interface, and may define new abstract methods and constants.
- ❖ Interfaces **can extend** more than one interface at a time. For example,

```
public interface Transformable
    extends Scalable, Rotable, Reflectable {}

public interface DrawingObject
    extends Drawable, Transformable{}

public class Shape implements DrawingObject {
    ... }
```

Method Forwarding



- ❖ Suppose class C extends class A, and also implements interface X.
- ❖ As all the methods defined in interface X are abstract, class C needs to implement all these methods.
- ❖ However, there are three implementations of X (in P,Q,R).
- ❖ In class C, we may want to use one of these implementations, that means, we may want to use some or all methods implemented in P, Q or R.
- ❖ Say, we want to use methods implemented in P. We can do this by creating an object of type class P in class C, and through this object access all the methods implemented in P.
- ❖ Note that, in class C, we do need to provide required stubs for all the methods in the interface X. In the body of the methods we may simply call methods of class P via the object of class P.
- ❖ This approach is also known as **method forwarding**.

Methods Overriding (Polymorphism)

- ❖ When a class defines a method using the **same** name, return type, and by the number, type, and position of its arguments as a method in its *superclass*, the method in the class **overrides** the method in the *superclass*.
- ❖ If a method is invoked for an object of the class, it's the **new definition** of the method that is called, and **not** the superclass's **old definition**.

Polymorphism

- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called *polymorphism*.

Methods Overriding: Example

In the example below,

- ❖ if **p** is an instance of class **B**,
p.f() refers to **f()** in class **B**.
- ❖ However, if **p** is an instance of class **A**,
p.f() refers to **f()** in class **A**.

The example also shows how to refer to the **overridden** method using **super** keyword.

```
class A {  
    int i = 1;  
    int f() { return i;}  
}  
  
class B extends A {  
    int i;                                // shadows i from A  
    int f() {                             // overrides f() from A  
        i = super.i + 1;                 // retrieves i from A  
        return super.f() + i;           // invokes f() from A  
    }  
}
```

Methods Overriding: Example

Suppose class C is a subclass of class B, and class B is a subclass of class A.

Class A and class C both define method `f()`.

From class C, we can refer to the overridden method by,

`super.f()`

This is because class B inherits method `f()` from class A.

However,

- ❖ if **all the three** classes define `f()`, then calling `super.f()` in class C invokes class B's definition of the method.
- ❖ **Importantly**, in this case, there is **no way** to invoke `A.f()` from within class C.
- ❖ Note that `super.super.f()` is **NOT legal** Java syntax.

Method Overloading

Defining methods with the **same name** and **different** argument or return types is called *method overloading*.

In Java,

- ❖ a method is distinguished by its **method signature** - its name, return type, and by the number, type, and position of its arguments

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

Data Hiding and Encapsulation

We can **hide** the **data** within the class and make it available only through the methods.

This can help in maintaining the consistency of the data for an object, that means the state of an object.

Visibility Modifiers

Java provides five access modifiers (for variables/methods/classes),

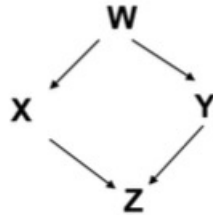
- ❖ **public** - visible to the world
- ❖ **private** - visible to the class only
- ❖ **protected** - visible to the package and all subclasses
- ❖ **No modifier (default)** - visible to the package

Constructors

- ❖ Good practice to **define** the required constructors for **all** classes.
- ❖ If a constructor is **not defined** in a class,
 - **no-argument** constructor is **implicitly inserted**.
 - this no-argument constructor invokes the **superclass's no-argument** constructor.
 - if the parent class (superclass) doesn't have a visible constructor with no-argument, it results in a compilation **error**.
- ❖ If the **first statement** in a constructor is **not** a call to **super()** or **this()**, a call to **super ()** is **implicitly** inserted.
- ❖ If a constructor is **defined** with **one or more arguments**, **no-argument** constructor is **not inserted** in that class.
- ❖ A class can have **multiple** constructors, with **different signatures**.
- ❖ The word **"this"** can be used to call another constructor in the same class.

Diamond Inheritance Problem: A Possible Solution

Using **multiple inheritance** (in C++):

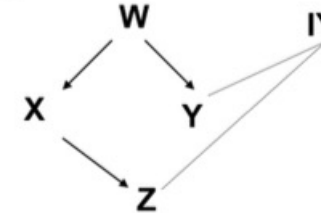


we achieve the following:

- In class Z, we can use methods and variables defined in X, W **and** Y.
- Objects of classes Z and Y can be assigned to variables of **type Y**.
- and more ...

Using **single inheritance** in Java:

```
class W {}  
interface IY {}  
class X extends W {}  
class Y extends W implements IY {}  
class Z extends X implements IY {}
```



we achieve the following:

- In class Z, we can use methods and variables defined in X and W. In class Z, if we want to use methods implemented in class Y, we can use **method forwarding** technique. That means, in class Z, we can create an object of type class Y, and via this object we can access (in class Z) all the methods defined in class Y.
- Objects of classes Z and Y can be assigned to variables of **type IY** (instead of Y).
- and more

Some References to Java Tutorials

- ❖ <https://docs.oracle.com/javase/tutorial/>
- ❖ <https://www.w3schools.com/java/default.asp>
- ❖ <https://www.tutorialspoint.com/java/index.htm>

Domain Modelling

COMP2511, CSE, UNSW



UNSW
SYDNEY

Domain Models

- Domain Models are used to **visually represent** important domain **concepts** and **relationships** between them.
- Domain Models help **clarify** and **communicate** important domain specific concepts and are used during the requirements gathering and **designing phase**.
- **Domain modeling** is the activity of expressing related domain concepts into a domain model.
- Domain models are also often referred to as *conceptual models* or *domain object models*.
- We will be use Unified Modeling Language (**UML**) **class diagrams** to represent domain models.
- There are many different modelling frameworks, like: UML, Entity-Relationship, Mind maps, Context maps, Concept diagrams. etc.

Requirements Analysis vs Domain modelling

- Requirements analysis determines *external behaviour*
“*What are the features of the system-to-be and who requires these features (actors)*”
- Domain modelling determines (internal behavior) –
“*how elements of system-to-be interact to produce the external behaviour*”
- Requirements analysis and domain modelling are **mutually dependent** - domain modelling supports clarification of requirements, whereas requirements help building up the model.

What is a domain?

- *Domain* – A sphere of knowledge particular to the problem being solved
- *Domain expert* – A person expert in the domain
- For example, in the domain of cake decorating, cake decorators are the domain experts

Problem

A motivating example:

- Tourists have schedules that involve at least one and possibly several cities
- Hotels have a variety of rooms of different grades: standard and premium
- Tours are booked at either a standard or premium rate, indicating the grade of hotel room
- In each city of their tour, a tourist is booked into a hotel room of the chosen grade
- Each room booking made by a tourist has an arrival date and a departure date
- Hotels are identified by a name (e.g. Melbourne Hyatt) and rooms by a number
- Tourists may book, cancel or update schedules in their tour

Ubiquitous language

- **Things in our design must represent real things in the domain expert's mental model.**
- For example, if the domain expert calls something an "order" then in our domain model (and ultimately our implementation) we should have something called an Order.
- Similarly, our domain model should not contain an OrderHelper, OrderManager, etc.
- Technical details do not form part of the domain model as they are not part of the design.

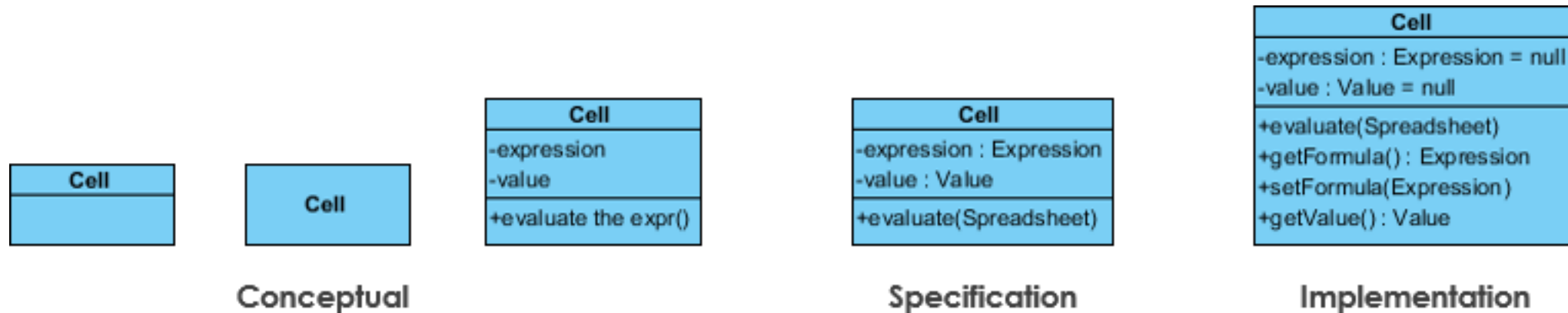
Noun/verb analysis

- Finding the ubiquitous language of the domain by finding the **nouns** and **verbs** in the requirements
- The **nouns** are possible entities in the domain model and the **verbs** possible behaviours

Problem

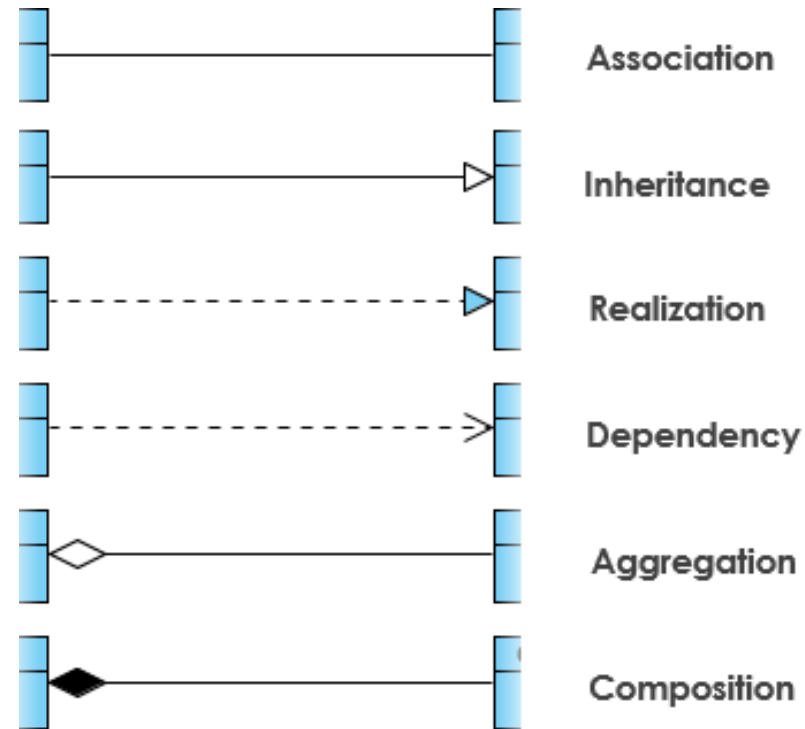
- The **nouns** and **verbs**:
 - **Tourists** have **schedules** that involve at least one and possibly several **cities**
 - **Hotels** have a variety of **rooms** of different **grades**: standard and premium
 - **Tours** are **booked** at either a standard or premium rate, indicating the **grade** of hotel **room**
 - In each **city** of their **tour**, a **tourist** is **booked** into a hotel **room** of the chosen **grade**
 - Each room **booking** made by a tourist has an arrival **date** and a departure **date**
 - **Hotels** are identified by a **name** (e.g. Melbourne Hyatt) and **rooms** by a **number**
 - **Tourists** may **book**, **cancel** or **update** **schedules** in their **tour**

UML Class diagrams: Perspectives



* Above diagram is from: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

UML Class diagrams: Relationships



* Above diagram is from: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

UML Class diagrams

Dependency ----->

- The loosest form of relationship. A class in some way *depends* on another.

Association _____

- A class "uses" another class in some way. When undirected, it is not yet clear in what direction dependency occurs.

Directed
Association ----->

- Refines association by indicating which class has knowledge of the other

UML Class diagrams

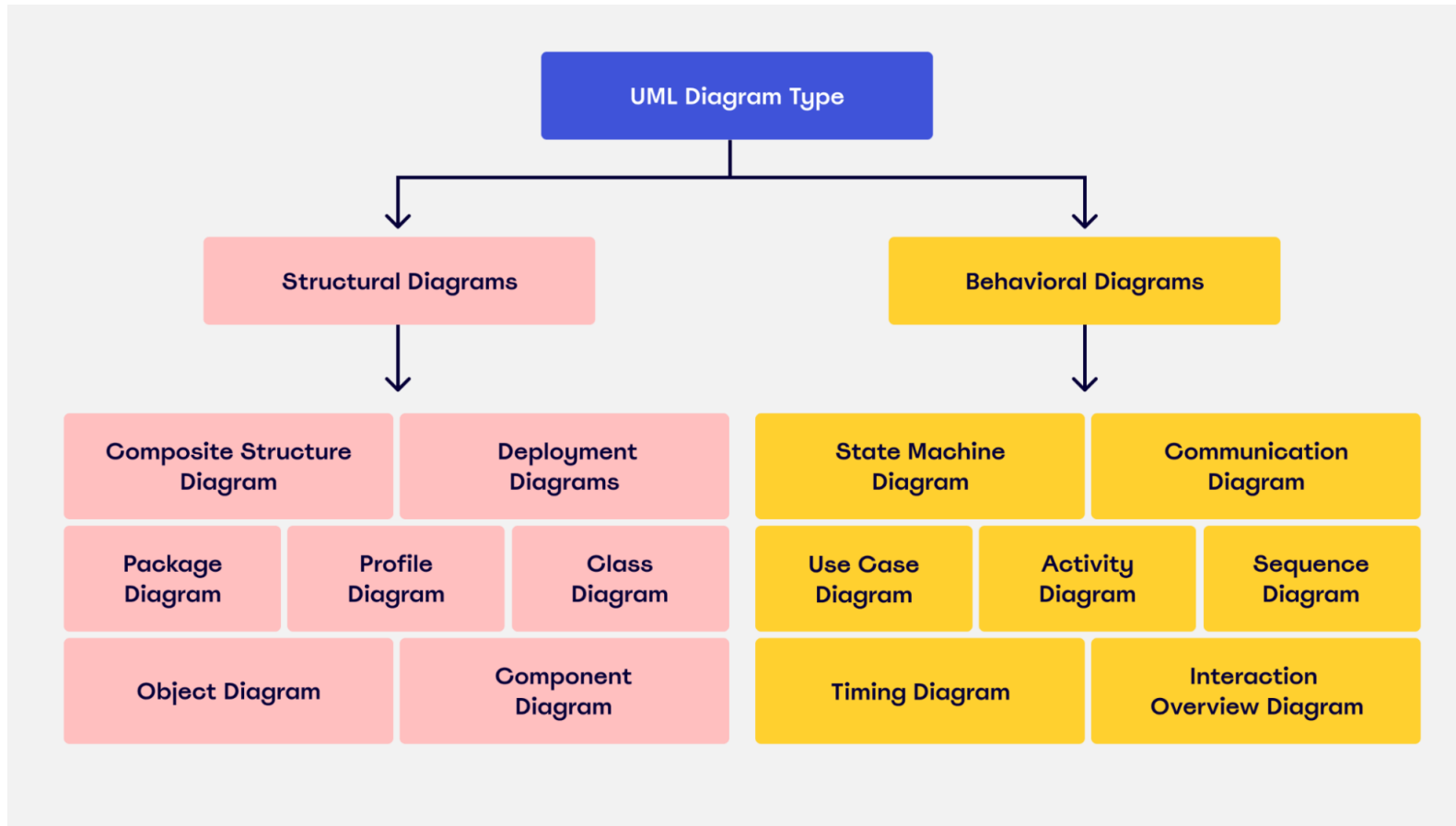
Aggregation 

- A class contains another class (e.g. a course contains students). Note that the diamond is at the end with the containing class.

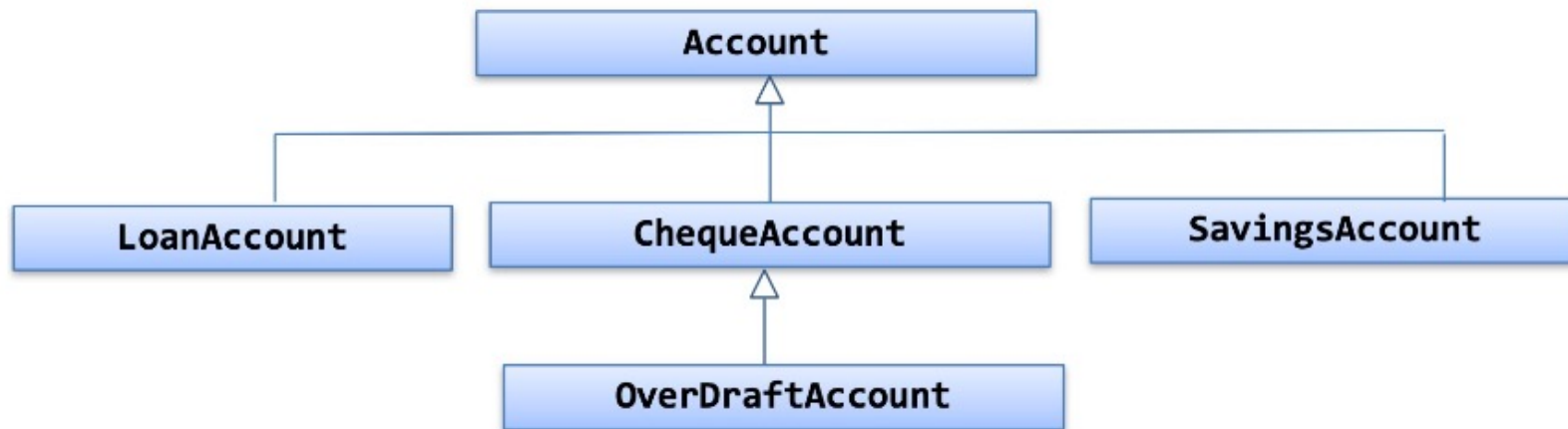
Composition 

- Like aggregation, but the contained class is integral to the containing class. The contained class cannot exist outside of the container (e.g. the leg of a chair)

UML Diagram Types



Examples



Examples

class (class diagram)

Account

-name: String
-balance: float

+getBalance(): float
+getName() : String
+withDraw(float)
+deposit(float)

object instances (object diagram)

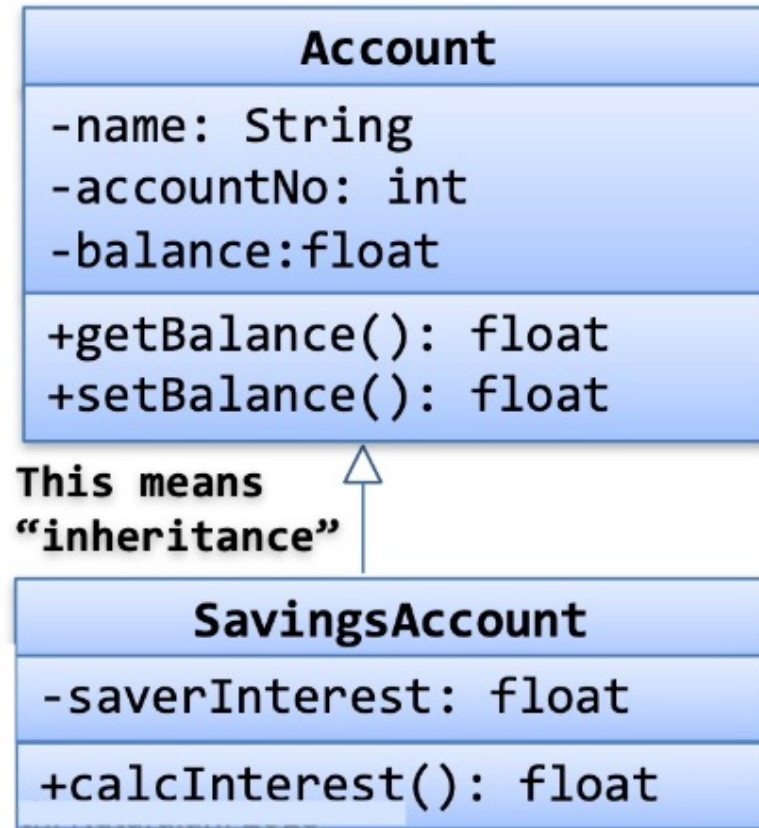
a1:Account

name = "John Smith"
balance = 40000

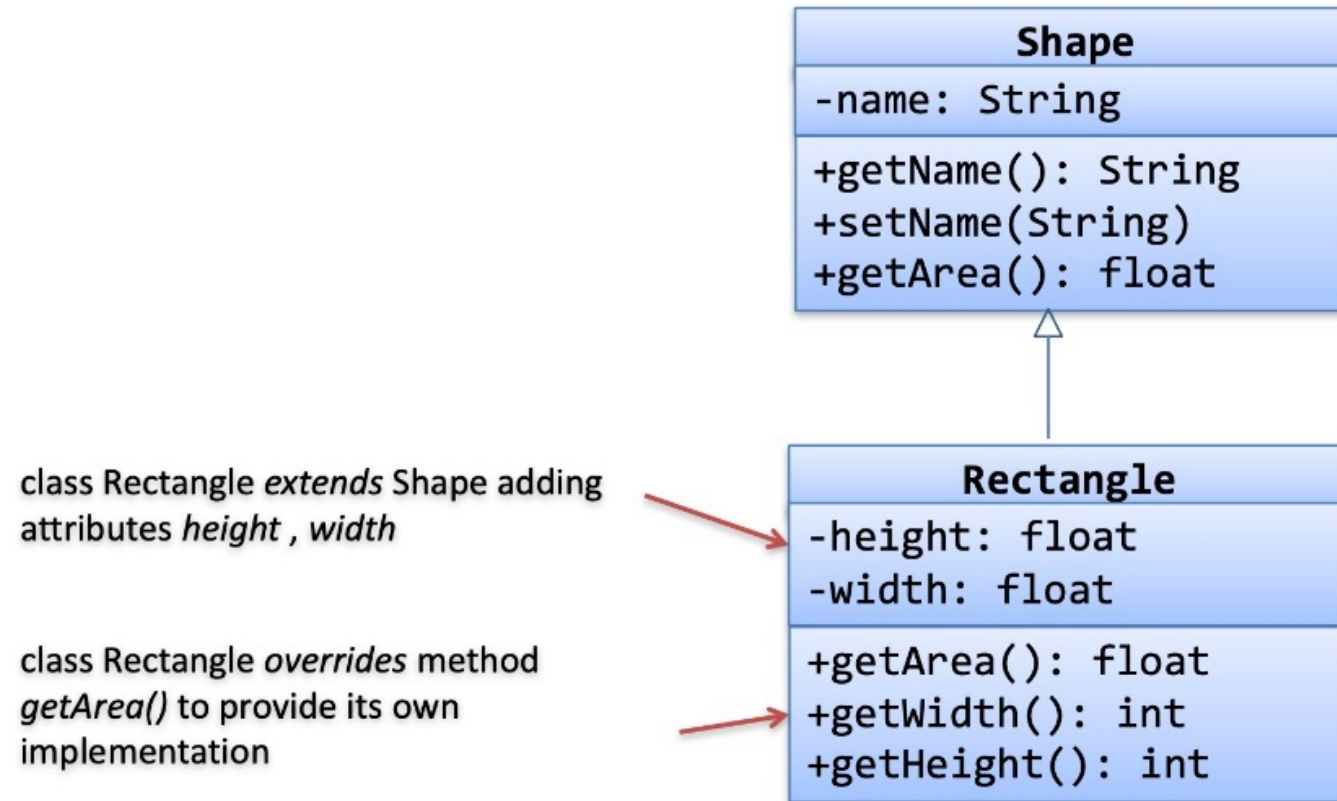
a2:Account

name = "Joe Bloggs"
balance = 50000

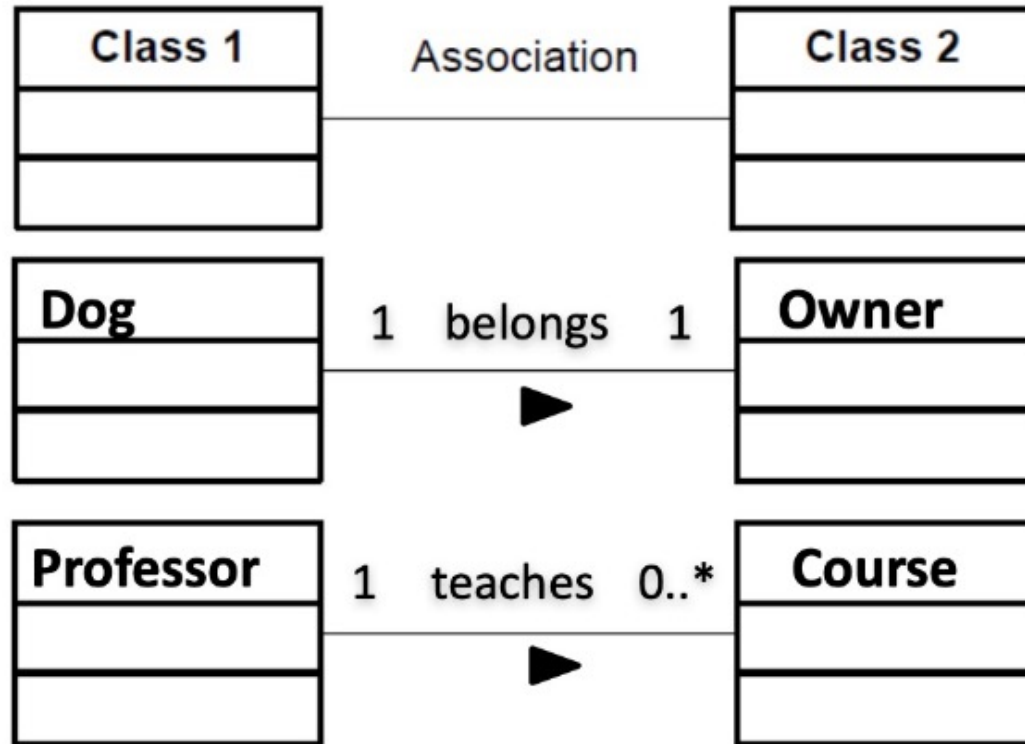
Representing classes in UML



Representing classes in UML



Representing Association in UML

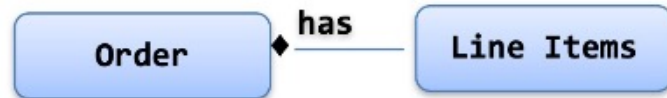


Representing Association in UML

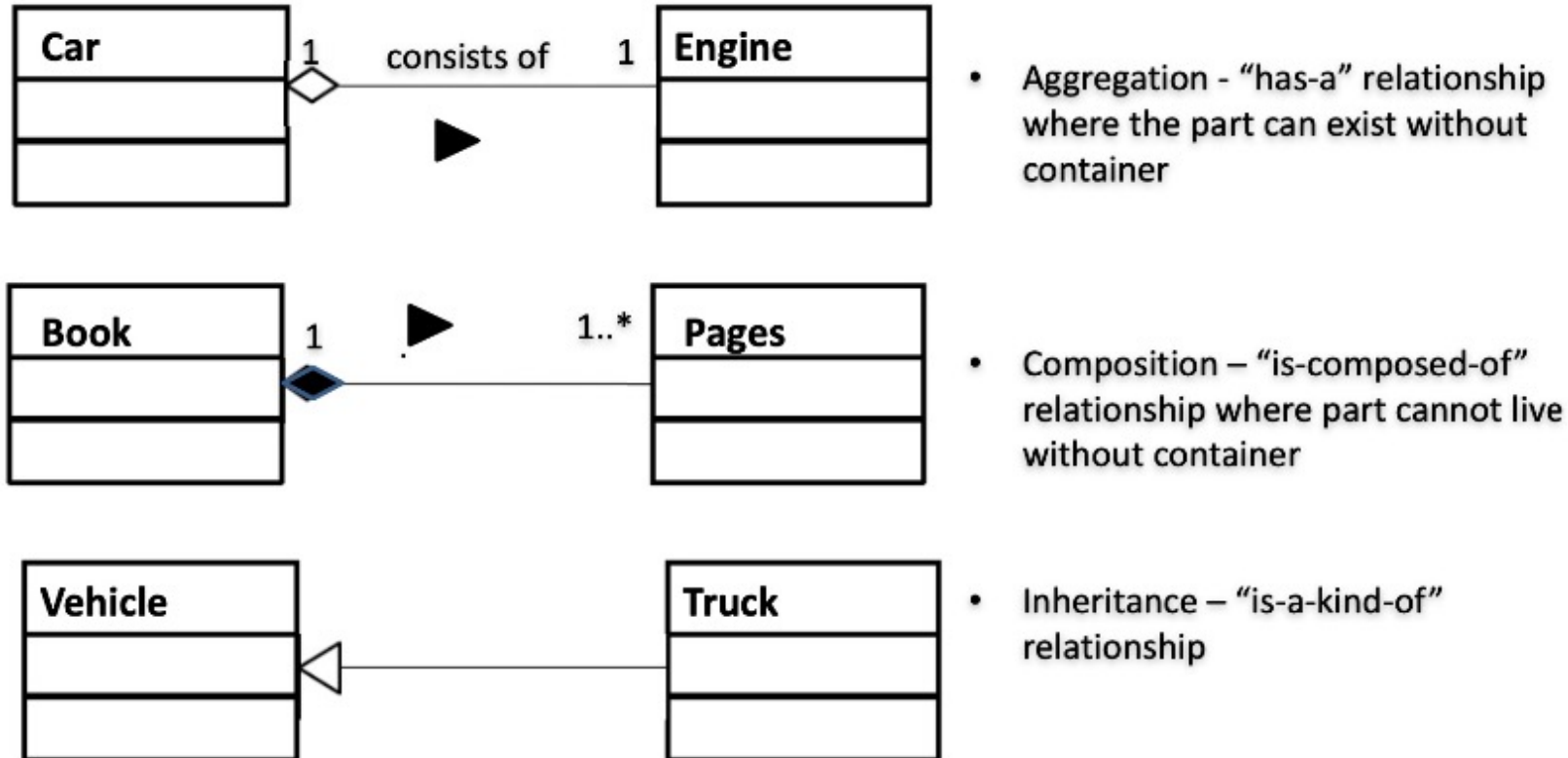
- Associations can model a ***“has-a”*** relationship where one class “contains” another class
- Associations can further be refined as:
Aggregation relationship (hollow diamond symbol ◇): The contained item is an element of a collection but it can also exist on its own, e.g., a lecturer in a university or a student at a university



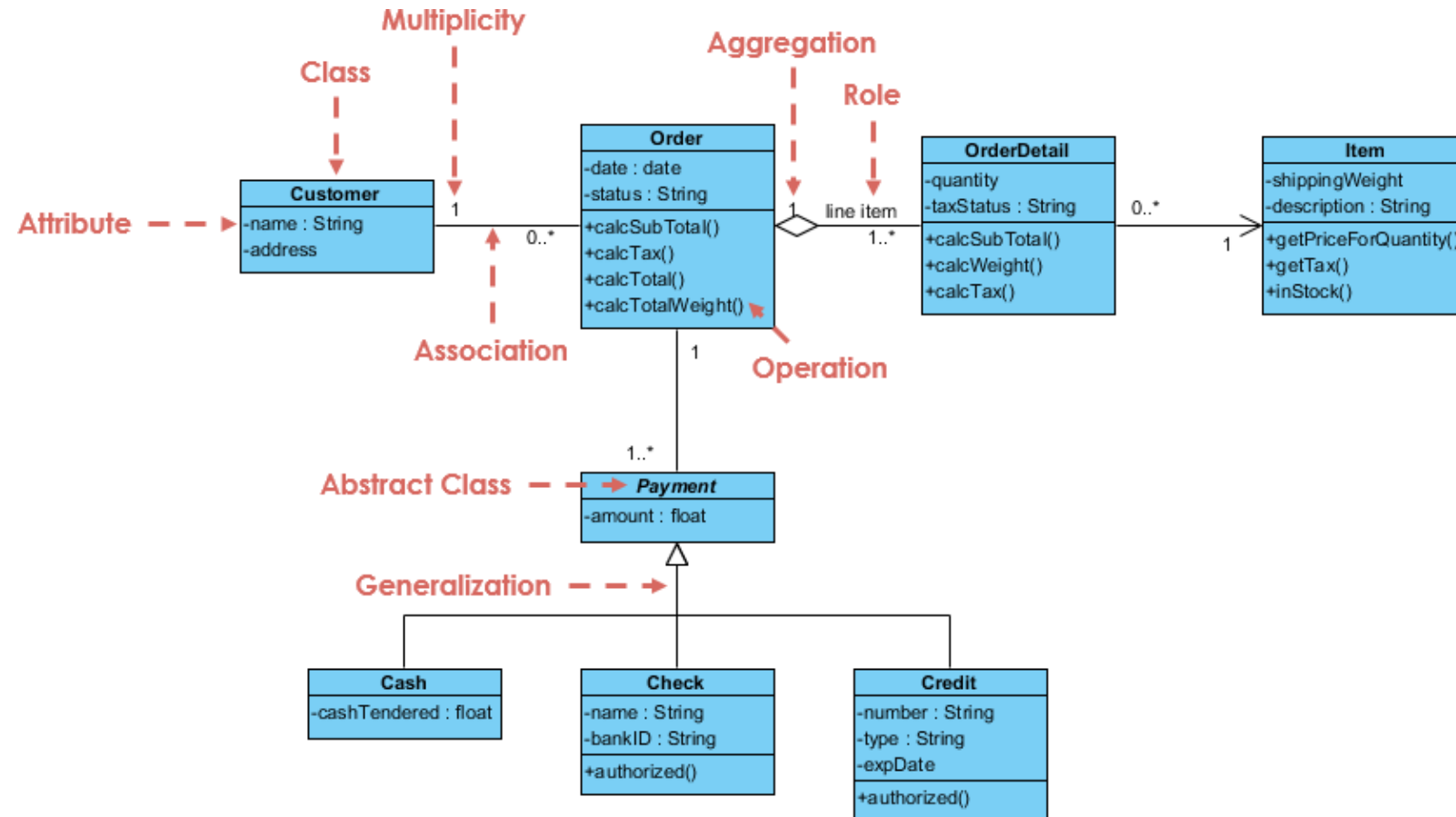
Composition relationship (filled diamond symbol ◆ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car



Representing Association in UML

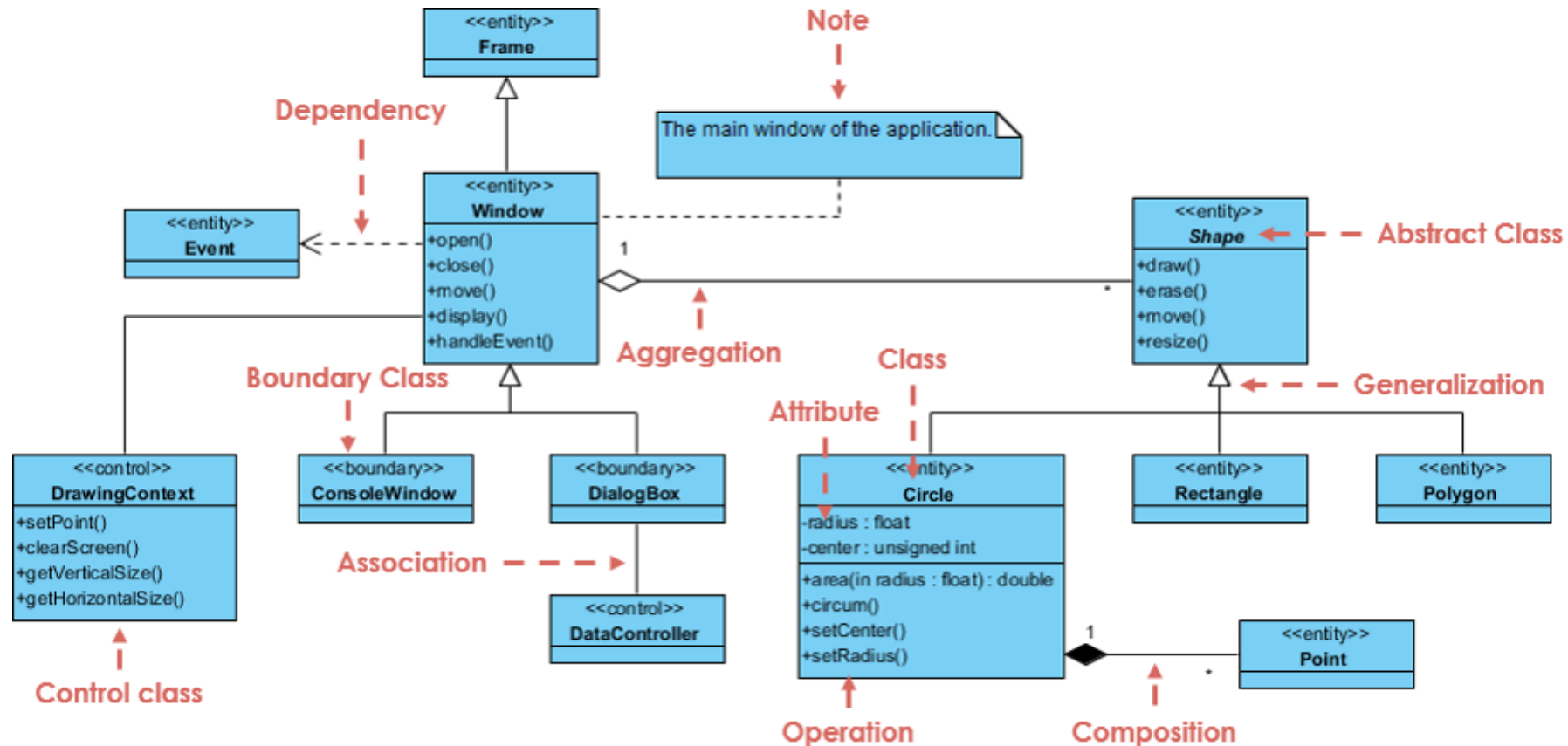


Class Diagram Example: Order System



* Above diagram is from: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

Class Diagram Example: GUI



* Above diagram is from: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

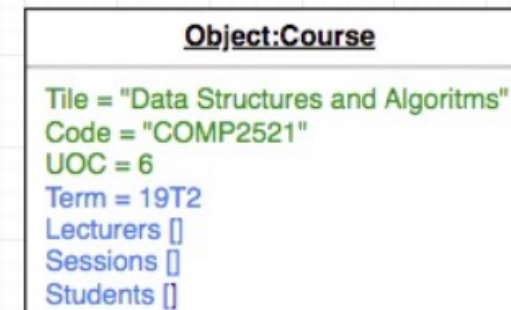
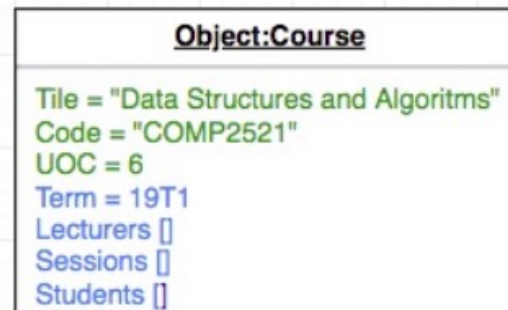
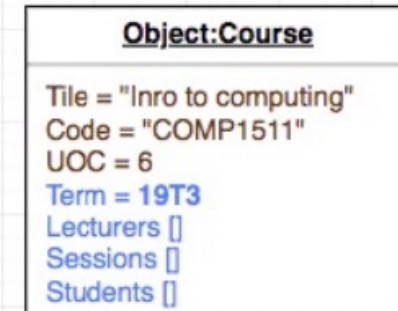
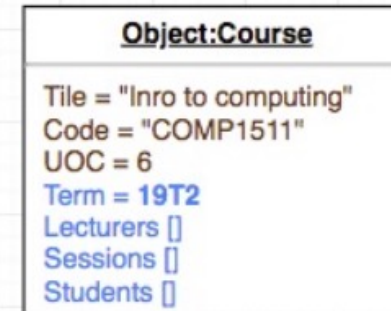
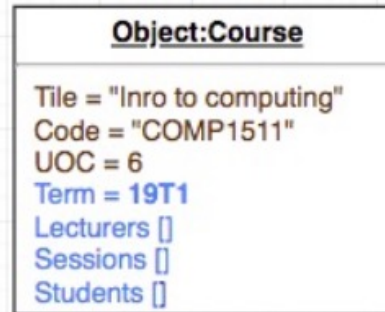
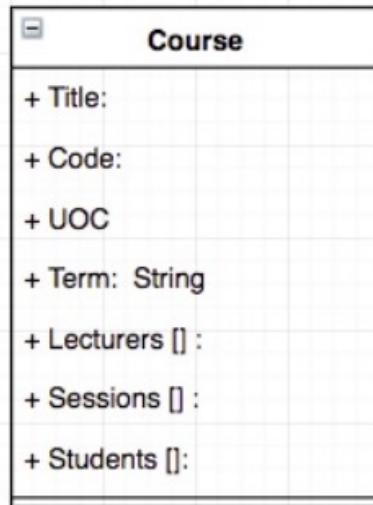
Attributes vs. Classes

- ❖ The most common confusion – *should it be an attribute or a class?*
 - when creating a domain model, often we need to decide whether to represent *something* as an *attribute* or a conceptual *class*.
- ❖ If a concept is **not** representable by a *number* or a *string*, most likely it is a *class*.
- ❖ For example:
 - a *lab mark* **can** be represented by a *number*, so we should represent it as an *attribute*
 - a *student* **cannot** be represented by a *number* or a *string*, so we should represent it as a *class*

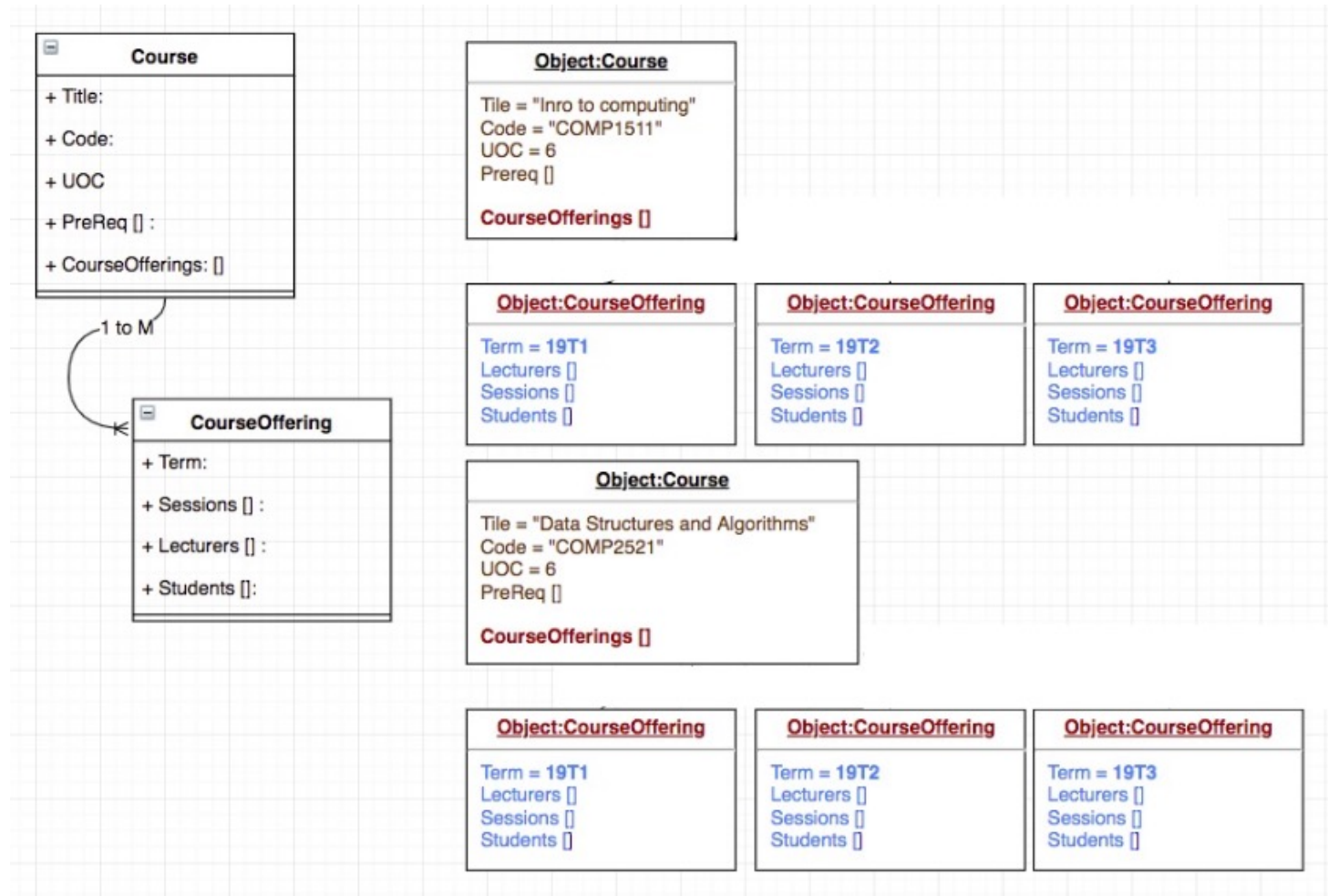
Attributes vs. Classes



What is wrong with the following?



A Possible solution



References

- ❖ A very detailed description of UML
 - <https://www.uml-diagrams.org/>
- ❖ Books that go into detail on Domain Driven Design
 - *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans.
 - *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#* by Scott Wlaschin.

Design By Contract

COMP2511, CSE, UNSW



UNSW
SYDNEY

Defensive Programming Vs Design by Contract

Defensive programming:

Tries to **address unforeseen** circumstances, in order to ensure the continuing functionality of the software element. For example, it makes the software behave in a predictable manner despite unexpected inputs or user actions.

- often used where **high availability**, safety or security is needed.
- results in **redundant checks** (both client and supplier may perform checks), more **complex software** for maintenance.
- difficult to locate errors, considering there is **no clear demarcation** of responsibilities.
- may safeguard against errors that will never be encountered, thus incurring run-time and maintenance costs.

Design by Contract:

At the design time, **responsibilities** are **clearly assigned** to different software elements, clearly documented and enforced during the development using unit testing and/or language support.

- clear demarcation of responsibilities helps **prevent redundant checks**, resulting in **simpler** code and **easier** maintenance.
- crashes if the required conditions are not satisfied! May **not** be **suitable** for **high availability** applications.

Design by Contract (DbC)

- ❖ Bertrand Meyer coined the term for his design of the Eiffel programming language (in 1986). Design by Contract (DbC) has its roots in work on formal specification, formal verification and Hoare logic.
- ❖ In business, when two parties (supplier and client) *interact* with each other, often they write and sign **contracts** to clarify the **obligations** and **expectations**. For example,

	Obligations	Benefits
Client	<i>(Must ensure precondition)</i> Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	<i>(May benefit from post-condition)</i> Reach Chicago.
Supplier	<i>(Must ensure post-condition)</i> Bring customer to Chicago.	<i>(May assume pre-condition)</i> No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

The example is from <https://www.eiffel.com/values/design-by-contract/introduction/>

Design by Contract (DbC)

Every software element should define a **specification** (or a **contract**) that governs its interaction with the rest of the software components.

A **contract** should address the following three questions:

- ❖ **Pre-condition** - what does the contract expect?

If the precondition is true, it can avoid handling cases outside of the precondition.

For example, expected argument value (`mark ≥ 0`) and (`marks ≤ 100`).

- ❖ **Post-condition** - what does the contract guarantee?

Return value(s) is guaranteed, provided the precondition is true.

For example: correct return value representing a grade.

- ❖ **Invariant** - what does the contract maintain?

Some values must satisfy constraints, before and after the execution (say of the method).

For example: a value of `mark` remains between zero and 100.

Design by Contract (DbC)

A **contract** (precondition, post-condition and invariant) should be,

- ❖ **declarative** and must **not** include implementation details.
- ❖ as far as possible: **precise**, **formal** and **verifiable**.

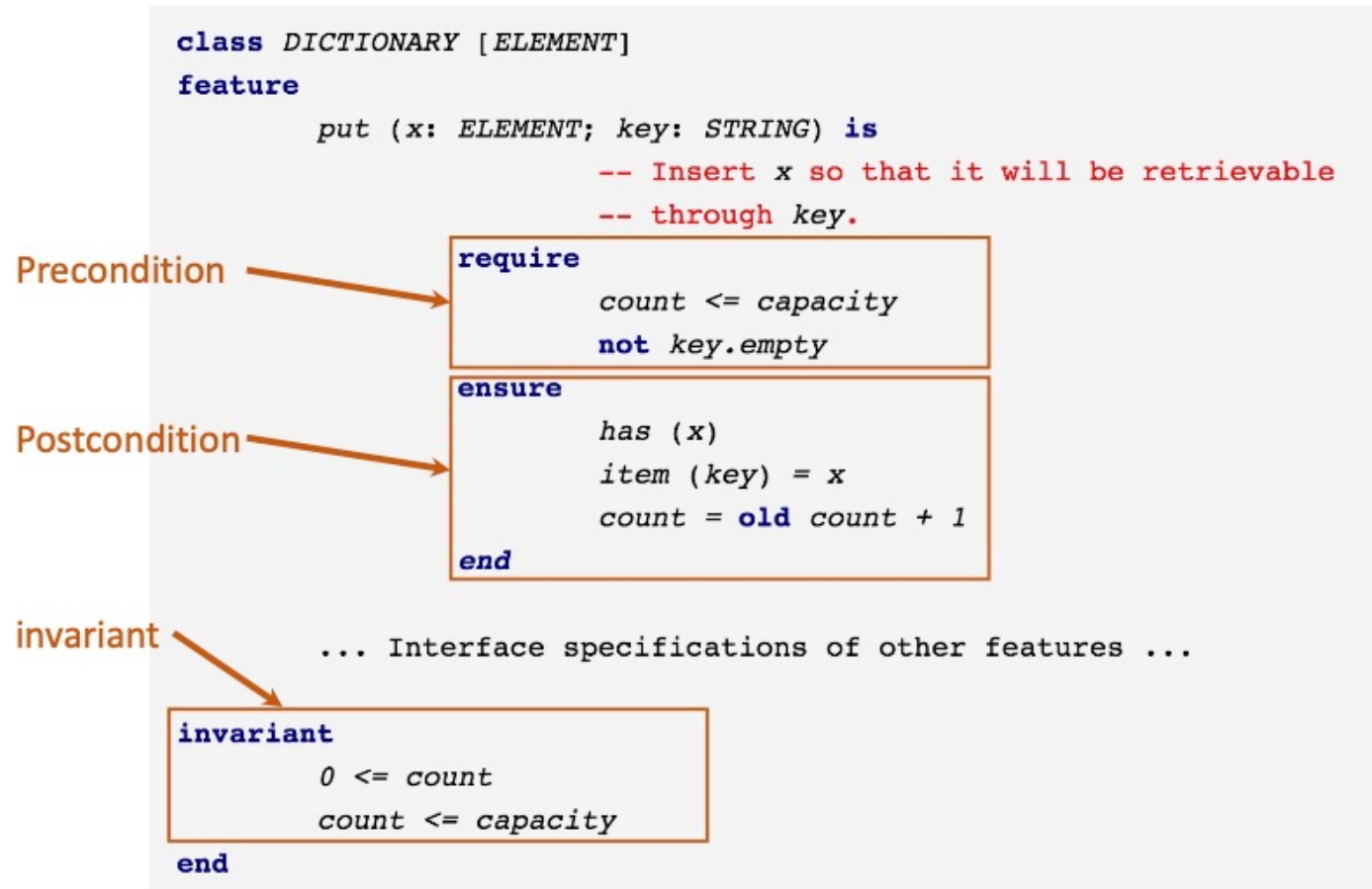
Benefits of Design by Contract (DbC)

- ❖ Do **not need to do error checking** for conditions that not satisfy the preconditions!
- ❖ **Prevents** redundant validation tasks.
- ❖ Given the preconditions are satisfied, clients can **expect** the specified post-conditions.
- ❖ Responsibilities are **clearly assigned**, this helps in locating errors and resulting in easier code maintenance.
- ❖ Helps in **cleaner** and **faster** development.

Design by Contract (DbC) : Implementation Issues

- ❖ Some programming languages (like Eiffel) offer **native support** for DbC.
- ❖ **Java** does **not have native support** for DbC, there are various libraries to support DbC.
- ❖ In the absence of a native language support, **unit testing** is used to test the contracts (preconditions, post-conditions and invariants).
- ❖ Often preconditions, post-conditions and invariants are **included** in the **documentation**.
- ❖ As indicated earlier, **contracts** should be,
 - **declarative** and must not include implementation details.
 - as far as possible: **precise**, **formal** and **verifiable**.

Design by Contract : Example using Eiffel



Design by Contract: Examples in Java

```
/**
 * @param value to calculate square root
 * @returns sqrt - square root of the value
 * @pre  value >= 0
 * @post value = sqrt * sqrt
 */
public double squareRoot ( double value );
```

```
/**
 * @invariant age >= 0
 */
public class Student {
```

```
/**
 * @param amount to be deposited into the account
 * @pre  amount > 0
 * @post balance = old balance + amount
 */
public void deposit( double amount);
```

Pre-Conditions

- ❖ A **pre-condition** is a condition or predicate that must always be true just **prior** to the execution of some section of code
- ❖ If a precondition is violated, the effect of the section of code becomes **undefined** and thus may or may not carry out its intended work.
- ❖ Security **problems** can **arise** due to **incorrect** pre-conditions.
- ❖ Often, preconditions are **included** in the **documentation** of the affected section of code.
- ❖ Preconditions are sometimes **tested** using **guards** or **assertions** within the code itself, and some languages have **specific** syntactic **constructions** for testing .
- ❖ **In Design by Contract**, a software element can **assume** that **preconditions are satisfied**, resulting in removal of redundant error checking code.
- ❖ See the next slide for the examples.

Pre-Conditions: Examples

```
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDbC(double mark) {
    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

Incorrect behaviour if *mark*
is outside the expected range

```
/**
 * Get Student at i'th position
 * @pre i < number_of_students
 * @param i - student's position
 * @return student at i'th position
 */
public Student getStudentDbC(int i) {
    return students.get(i);
}
```

Throws runtime exception
if (*i* >= *number_of_students*)

Design by Contract

No additional error checking for pre-conditions

```
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDefensive(double mark) {
    if( (mark < 0) || (mark > 100) ){
        System.out.println("Error");
    }

    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

Defensive Programming:

Additional error checking for pre-conditions

Pre-Conditions in Inheritance

- ❖ An implementation or redefinition (method overriding) of an inherited method **must comply** with the **inherited contract** for the method.
- ❖ Preconditions **may be weakened** (relaxed) in a subclass, but it must comply with the inherited contract.
- ❖ An implementation or redefinition **may lessen** the obligation of the client, but not increase it.
- ❖ For example,

```
/**  
 * @pre (theta >=0) and (theta <= 90)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```

valid

Weaker Pre-condition

```
/**  
 * @pre (theta >=0) and (theta <= 180)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```

X - not valid

Stronger Pre-condition

```
/**  
 * @pre (theta >=0) and (theta <=45)  
 * @param theta - angle to calculate trajectory  
 * @return trajectory at angle theta  
 */  
public double calculateTrajectory(double theta) {
```


Post-Conditions

- ❖ A **post-condition** is a condition or predicate that must always be true just **after** the execution of some section of code
- ❖ The **post-condition** for any routine is a declaration of the properties which are guaranteed upon completion of the routine's execution^[1].
- ❖ Often, preconditions are **included** in the **documentation** of the affected section of code.
- ❖ Post-conditions are sometimes **tested** using **guards** or **assertions** within the code itself, and some languages have **specific** syntactic **constructions** for testing .
- ❖ **In Design by Contract**, the properties declared by **the post-condition(s) are assured**, provided the software element is called in a state in which its pre-condition(s) were true.

[1] Meyer, Bertrand, Object-Oriented Software Construction, second edition, Prentice Hall, 1997.

```
/**  
 * @param value to calculate square root  
 * @returns sqrt - square root of the value  
 * @pre value >= 0  
 * @post value = sqrt * sqrt  
 */  
public double squareRoot ( double value );
```

Post-Conditions in Inheritance

- ❖ An implementation or redefinition (method overriding) of an inherited method **must comply** with the **inherited contract** for the method.
- ❖ Post-conditions **may be strengthened** (more restricted) in a subclass, but it must comply with the inherited contract.
- ❖ An implementation or redefinition (overridden method) **may increase** the benefits it provides to the client, but **not decrease** it.
- ❖ For example,
 - ❖ the original contract requires returning a **set**.
 - ❖ the redefinition (overridden method) returns **sorted set**, offering *more* benefit to a client.

Class Invariant

- ❖ The class invariant **constrains** the **state** (i.e. values of certain variables) stored in the object.
- ❖ Class invariants are **established** during construction and constantly **maintained** between calls to public methods. Methods of the class must make sure that the class invariants are satisfied / preserved.
- ❖ **Within a method**: code **within** a method **may break** invariants as long as the invariants are **restored** before a public method ends.
- ❖ Class invariants help programmers to rely on a valid state, avoiding risk of inaccurate / invalid data. Also helps in locating errors during testing.

Class invariants in Inheritance

- ❖ Class invariants are **inherited**, that means,
*"the **invariants** of **all the parents** of a class apply to the class itself."* !
- ❖ A subclass can access implementation data of the parents, however, **must always satisfy** the **invariants** of **all the parents** – preventing invalid states!

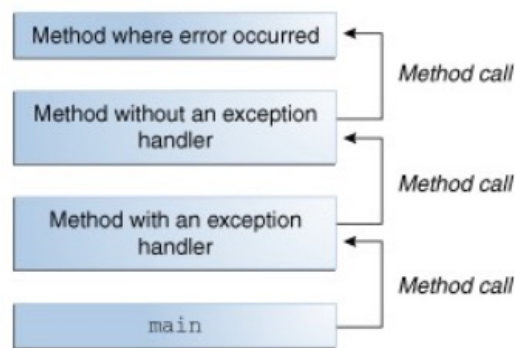
END

Exceptions in Java

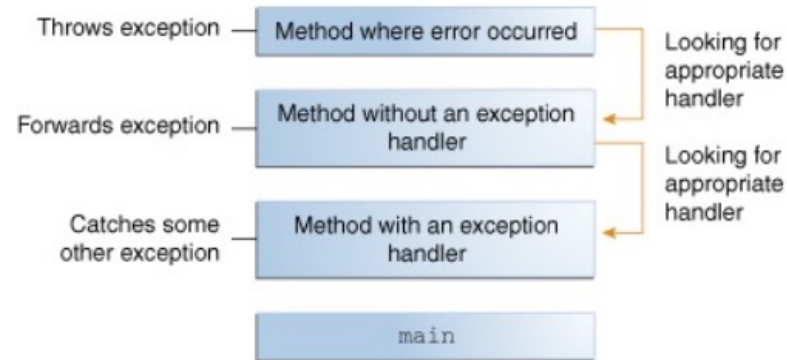


Exceptions in Java

- ❖ An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- ❖ When error occurs, an *exception* object is created and given to the runtime system, this is called **throwing** an exception.
- ❖ The runtime system searches the call stack for a method that contains a block of code that can **handle** the exception.
- ❖ The exception handler chosen is said to **catch** the exception.



The call stack.



Searching the call stack for the exception handler.

Exceptions in Java

The **Three Kinds** of Exceptions

- ❖ Checked exception (IOException, SQLException, etc.)
- ❖ Error (VirtualMachineError, OutOfMemoryError, etc.)
- ❖ *Runtime exception* (ArrayIndexOutOfBoundsException, ArithmeticException, etc.)

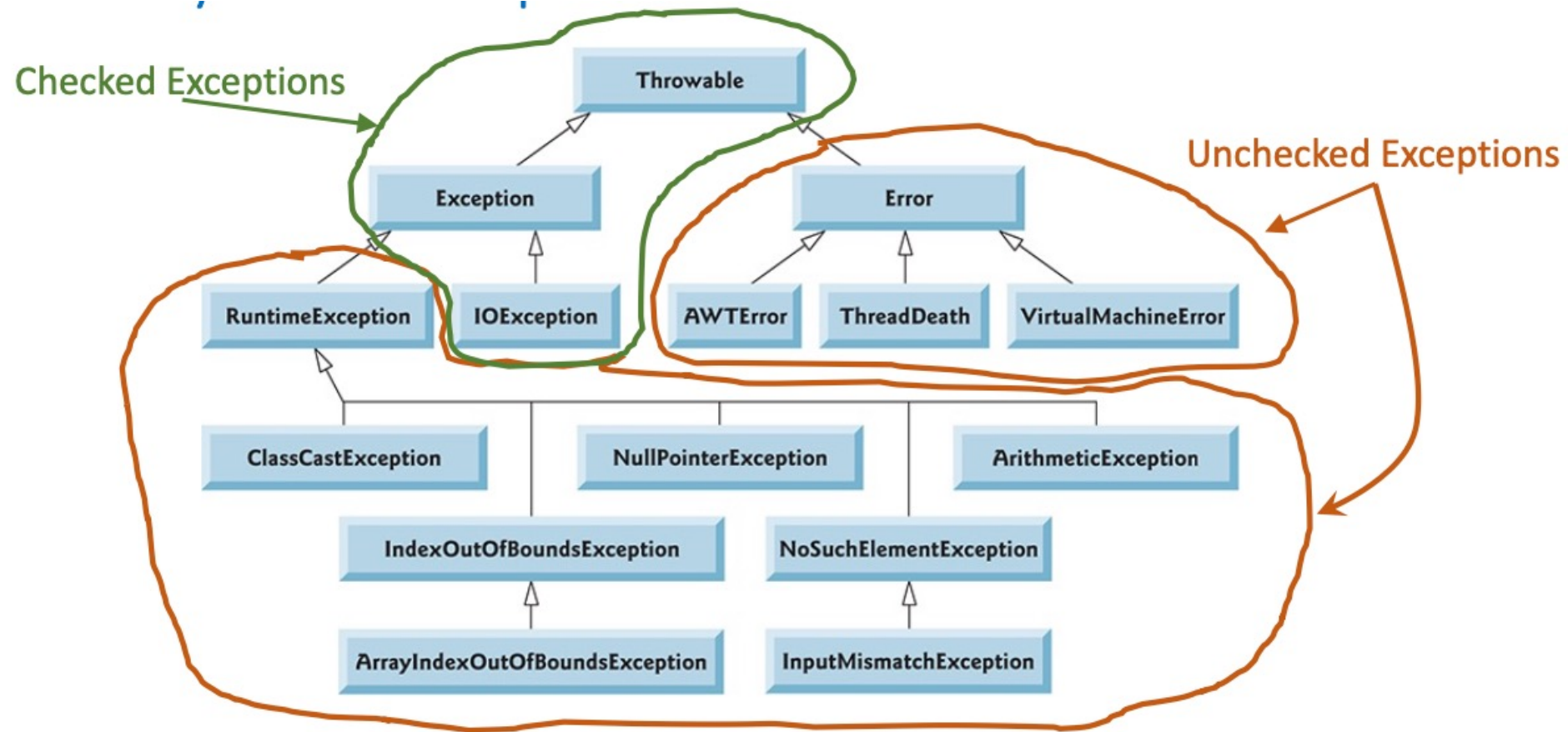
Checked vs. Unchecked Exceptions

- ❖ An exception's type determines whether it's checked or unchecked.
- ❖ All classes that are subclasses of *RuntimeException* (typically caused by defects in your program's code) or *Error* (typically 'system' issues) are **unchecked** exceptions.
- ❖ All classes that inherit from class *Exception* but not directly or indirectly from class *RuntimeException* are considered to be **checked** exceptions.

Exceptions in Java

- ❖ **Good introduction on Exceptions** at
<https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>
- ❖ **Unchecked Exceptions — The Controversy**
<https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>

Hierarchy of Java Exceptions



From the book "Java How to Program, Early Objects", 11th Edition, by Paul J. Deitel; Harvey Deitel

Example

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

try

catch

finally

User Defined Exceptions in Java

- ❖ We can also create **user defined** exceptions.
- ❖ All exceptions must be a child of **Throwable**.
- ❖ A **checked** exception need to extend the **Exception** class,
but **not** directly or indirectly from class **RuntimeException**.
- ❖ An **unchecked** exception (like a runtime exception) need to extend the **RuntimeException** class.

User Defined / Custom Checked Exception

- Normally we define a *checked* exception, by extending the `Exception` class.

```
class MyException extends Exception {  
    public MyException(String message){  
        super( message );  
    }  
}
```

User Defined / Custom Exceptions: A Simple Example

```
try {
    out = new PrintWriter(new FileWriter("myData.txt"));
    for(int i=0; i<SIZE; i++){
        int idx = i + 5;
        if(idx >= SIZE){
            throw new MyException("idx is out of index range!");
        }
        out.println(list.get(idx));
    }
}
catch(IOException e){
    System.out.println(" In writeln ....");
}
catch(MyException e){
    System.out.println(e.getMessage());
}
catch(Exception e){
    System.out.println(" In writeln, Exception ....");
}
}
```

Exceptions in Inheritance

- ❖ If a subclass method **overrides** a superclass method,
a subclass's **throws** clause can contain a subset of
a superclass's **throws** clause.
It must **not** throw more exceptions!
- ❖ Exceptions are **part of** an **API** documentation and **contract**.

Demo: Exceptions in Java

Demo

Assertions in Java

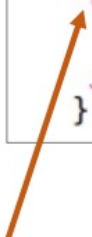
- An **assertion** is a statement in the Java that enables you to test your assumptions about your program. Assertions are **useful** for checking:
 - Preconditions, Post-conditions, and Class Invariants (DbC!)
 - Internal Invariants and Control-Flow Invariants
- You should **not** use assertions:
 - for argument checking in **public methods**.
 - to do any work that your application requires for correct operation.
- Evaluating assertions should **not** result in side effects.
- The following document shows how to use **assertions in Java** :
<https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>

Important: for backward compatibility, by **default**, Java **disables** assertion validation feature. It needs to be explicitly **enabled** using the following command line argument:

- **-enableassertions** command line argument, or
- **-ea** command line argument

Assert : Example

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param interval refresh interval in milliseconds.
 */
private void setRefreshInterval(int interval) {
    // Confirm adherence to precondition in nonpublic method
    assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;
    .... // Set the refresh interval
}
```



Exceptions: Summary Points

- ❖ Consider your exception-handling and error-recovery strategy in the **design process**.
- ❖ Sometimes you can **prevent an exception** by validating data first.
- ❖ If an exception can be handled meaningfully in a method, the method should **catch** the exception **rather than declare** it.
- ❖ If a subclass method overrides a superclass method, a subclass's **throws** clause can contain a subset of a superclass's **throws** clause. It must not throw more exceptions!
- ❖ Programmers should **handle checked** exceptions.
- ❖ If **unchecked** exceptions are **expected**, you must handle them **gracefully**.
- ❖ Only the **first** matching **catch** is executed, so select your catching class(es) carefully.
- ❖ Exceptions are part of an API documentation and contract.
- ❖ Assertions can be used to check preconditions, post-conditions and invariants.

Generics and Collections in Java

Generics in Java

Generics enable **types** (classes and interfaces) to be **parameters** when defining:

- classes,
- interfaces and
- methods.

Benefits

- ❖ Removes **casting** and offers stronger type checks at compile time.
- ❖ Allows implementations of **generic algorithms**, that work on collections of **different types**, can be customized, and are type safe.
- ❖ Adds stability to your code by making more of your bugs detectable at compile time.

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();  
listG.add("hello");  
String sg = listG.get(0);    // no cast
```

With Generics

Generic Types

❖ A generic type is a generic **class** or **interface** that is **parameterized** over types.

❖ A generic class is defined with the following format:

class name< **T1, T2, ..., Tn** > { /* ... */ }

❖ The most commonly used type parameter names are:

- ❖ E - Element (used extensively by the Java Collections Framework)
- ❖ K - Key
- ❖ N - Number
- ❖ T - Type
- ❖ V - Value
- ❖ S,U,V etc. - 2nd, 3rd, 4th types

❖ For example,

Box<**Integer**> integerBox = new **Box**<**Integer**>();

OR

Box<**Integer**> integerBox = new **Box**<>();

```
public class Box {  
    private Object object;  
  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
/**  
 * Generic version of the Box class.  
 * @param <T> the type of the value being boxed  
 */  
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Multiple Type Parameters

- ❖ A generic class can have **multiple type** parameters.
- ❖ For example, the generic **OrderedPair** class, which implements the generic **Pair** interface
- ❖ Usage examples,

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}  
  
public class OrderedPair<K, V> implements Pair<K, V> {  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);  
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");  
... ..  
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);  
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");  
... ..  
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Generic Methods

Generic methods are methods that **introduce** their **own type** parameters.

```
public class Util {  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
            p1.getValue().equals(p2.getValue());  
    }  
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.

Generally, this can be left out and the compiler will **infer** the **type** that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.compare(p1, p2);
```

Bounded Type Parameters

- ❖ There may be times when you want to **restrict the types** that can be used as type arguments in a parameterized type.
- ❖ For example, a method that operates on numbers might only want to accept instances of **Number** or its subclasses.

```
public <U extends Number> void inspect(U u){  
    System.out.println("U: " + u.getClass().getName());  
}
```

```
public class NaturalNumber<T extends Integer> {
```


Multiple Bounds

- ❖ A type parameter can have multiple bounds:

`< T extends B1 & B2 & B3 >`

- ❖ A type variable with multiple bounds is a subtype of **all** the **types** listed in the bound.
- ❖ Note that **B1, B2, B3**, etc. in the above refer to **interfaces** or a **class**. There can be at most one class (single inheritance), and the rest (or all) will be **interfaces**.
- ❖ If **one** of the bounds is a **class**, it must be specified **first**.

Generic Methods and Bounded Type Parameters

```
public static <T> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
    for (T e : anArray)  
        if (e > elem) // compiler error X - invalid  
            ++count;  
    return count;  
}
```

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

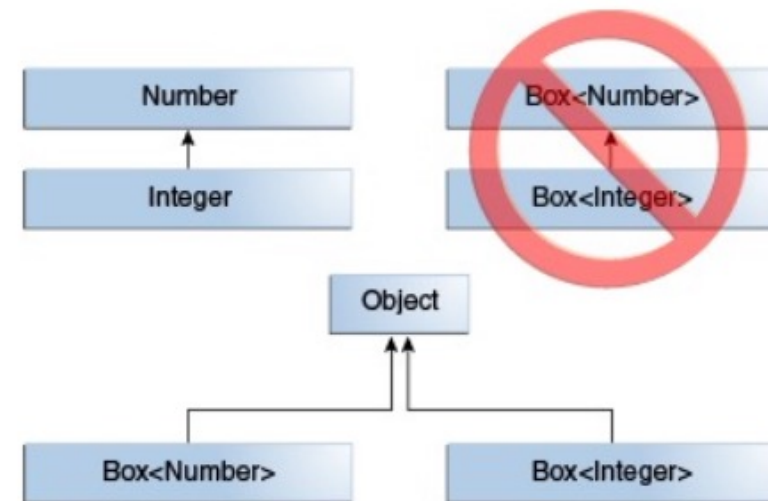
```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem)  
{  
    int count = 0;  
    for (T e : anArray)  
        if (e.compareTo(elem) > 0) Valid  
            ++count;  
    return count;  
}
```


Generics, Inheritance, and Subtypes

- ❖ Consider the following method:

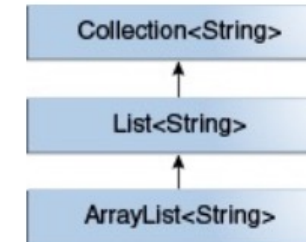
```
public void boxTest( Box<Number> n ) { /* ... */ }
```

- ❖ What type of argument does it accept?
- ❖ Are you allowed to pass in **Box<Integer>** or **Box<Double>** ?
- ❖ The answer is "no", because **Box<Integer>** and **Box<Double>** are **not** subtypes of **Box<Number>**.
- ❖ This is a **common misunderstanding** when it comes to programming with generics.



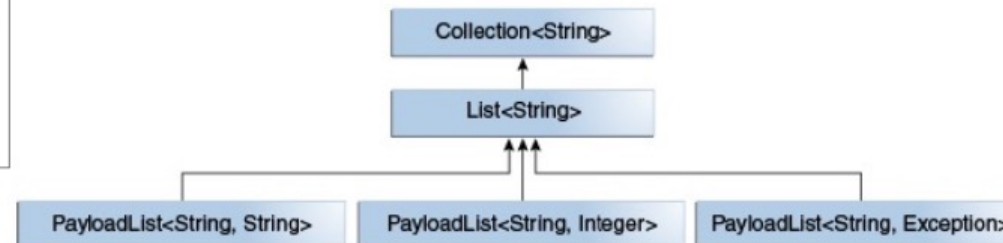
Generic Classes and Subtyping

- ❖ You **can subtype** a generic class or interface by extending or implementing it.
- ❖ The relationship between the type parameters of one **class** or **interface** and the type parameters of another are determined by the **extends** and **implements** clauses.
- ❖ `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`.
- ❖ So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`.
- ❖ So long as you **do not vary the type** argument, the subtyping relationship is preserved between the types.



```
interface PayloadList<E,P> extends List<E> {  
    void setPayload(int index, P val);  
    ...  
}
```

```
PayloadList<String,String>  
PayloadList<String,Integer>  
PayloadList<String,Exception>
```



Wildcards: Upper bounded

- ❖ In generic code, the question mark (?), called the **wildcard**, represents an unknown type.
- ❖ The wildcard can be used in a **variety of situations**: as the type of a parameter, field, or local variable; sometimes as a return type.
- ❖ The **upper bounded wildcard**, **<? extends Foo >**, where Foo is any type, matches Foo and any subtype of Foo .
- ❖ You can specify an upper bound for a wildcard, or you can specify a lower bound, but you **cannot** specify **both**.

```
public static void process(List<? extends Foo> list) {  
    for (Foo elem : list) {  
        // ...  
    }  
}
```

```
public static double sumOfList(List<? extends Number> list) {  
    double s = 0.0;  
    for (Number n : list)  
        s += n.doubleValue();  
    return s;  
}
```

Wildcards: Unbounded

- ❖ The **unbounded wildcard** type is specified using the wildcard character (**?**), for example, **List<?>**. This is called a list of unknown type.

```
public static void printList(List<Object> list) {  
    for (Object elem : list)  
        System.out.println(elem + " ");  
    System.out.println();  
}
```

It prints only a list of Object instances;
it **cannot** print List<Integer>, List<String>,
List<Double>, and so on

```
public static void printList(List<?> list) {  
    for (Object elem: list)  
        System.out.print(elem + " ");  
    System.out.println();  
}
```

To write a **generic** printList
method, use **List<?>**

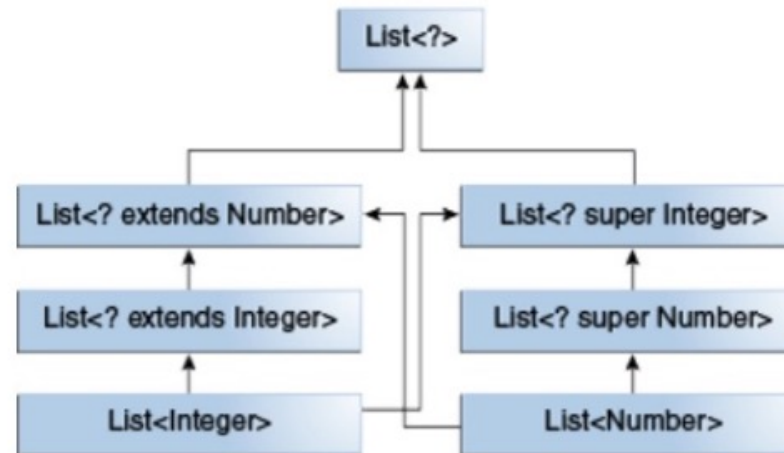
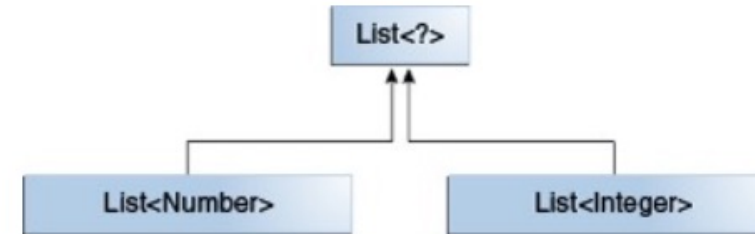
Wildcards: Lower Bounded

- ❖ An **upper bounded wildcard** restricts the unknown type to be a specific type or a subtype of that type and is represented using the **extends** keyword.
- ❖ A **lower bounded wildcard** is expressed using the wildcard character ('?'), following by the **super** keyword, followed by its lower bound: `< ? super A >`.
- ❖ To write the method that works on lists of Integer and the super types of **Integer**, such as **Integer**, **Number**, and **Object**, you would specify `List<? Super Integer>`.
- ❖ The term `List<Integer>` is more **restrictive** than `List<? super Integer>`.

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```


Wildcards and Subtyping

- ❖ Although Integer is a subtype of Number, List<Integer> is **not** a **subtype** of List<Number> and, these two types are **not related**.
- ❖ The **common parent** of List<Number> and List<Integer> is **List<?>**.



A hierarchy of several generic List class declarations.

Collections in Java

A **collections framework** is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.

All collections frameworks contain the following:

- ❖ **Interfaces**: allows collections to be manipulated independently of the details of their representation.
- ❖ **Implementations**: concrete implementations of the collection interfaces.
- ❖ **Algorithms**: the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
 - The algorithms are said to be **polymorphic**: that is, the same method can be used on many different implementations of the appropriate collection interface.

Core Collection Interfaces:

- ❖ The core collection interfaces encapsulate different types of collections
- ❖ The interfaces allow collections to be manipulated independently of the details of their representation.



The core collection interfaces.

The Collection Interface

- ❖ A **Collection** represents a group of objects known as its elements.
- ❖ The **Collection interface** is used to pass around collections of objects where maximum generality is desired.
- ❖ For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument.
- ❖ The **Collection interface** contains methods that perform basic operations, such as
 - `int size()`,
 - `boolean isEmpty()`,
 - `boolean contains(Object element)`,
 - `boolean add(E element)`,
 - `boolean remove(Object element)`,
 - `Iterator<E> iterator()`,
 -**many more** ...

More at : <https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>

Collection Implementations

- ❖ The general purpose **implementations** are summarized in the following table:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implemented Classes in the Java Collection,
Read their APIs.

- ❖ Overview of the *Collections Framework* at the following page:
<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

Wrappers for the Collection classes

❖ <https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html>

Demo: Collections Framework

Demo

End

Junit Testing in Java

COMP2511, CSE, UNSW



UNSW
SYDNEY

Software Testing

❖ Different types of testing:

- Object Oriented Design document describes responsibilities of classes and methods (APIs) → **Unit Testing**
- System Design Document → Integration Testing
- Requirements Analysis Document → System Testing
- Client Expectation → Acceptance Testing

❖ **Unit Testing** is also useful for **refactoring** tasks.

❖ In this course, we will focus on Unit testing.

JUnit

- ❖ **JUnit** is a popular unit testing (open source) framework for testing Java programs.
- ❖ Most popular IDEs facilitate easy integration of Junit.
- ❖ Basic Junit Terminology:
 - **Test Case** – Java class containing test methods
 - **Test Method** – a method that executes the test code, annotated with `@Test`, in a Test Case
 - **Asserts** - asserts or assert statements check an expected result versus the actual result
 - **Test Suites** – collection of several Test Cases

JUnit Example: assertEquals, assertTrue

```
14  /**
15   * Tests for Pineapple on Piazza
16   * @author Nick Patrikeos
17   */
18  public class PiazzaTest {
19
20      @Test
21      public void testExampleUsage() {
22          // Create a forum and make some posts!
23          PiazzaForum forum = new PiazzaForum("COMP2511");
24          assertEquals("COMP2511", forum.getName());
25
26          Thread funThread = forum.publish("The Real Question - Pineapple on Piazza", "Who likes pineapple on piazza?");
27
28          funThread.setTags(new String[] { "pizza", "coding", "social", "hobbies" });
29          assertTrue(
30              Arrays.equals(new String[] { "coding", "hobbies", "pizza", "social" }, funThread.getTags().toArray()));
31
32          funThread.publishPost("Yuck!");
33          funThread.publishPost("Yes, pineapple on pizza is the absolute best");
34          funThread.publishPost("I think you misspelled pizza btw");
35          funThread.publishPost("I'll just fix that lol");
36
37          assertEquals(5, funThread.getPosts().size());
38      }
39
40      @Test
41      public void testSearchByTag() {
42          PiazzaForum forum = new PiazzaForum("COMP2511");
43
44          Thread labThread = forum.publish("Lab 01", "How do I do the piazza exercise?");
45          Thread assignmentThread = forum.publish("Assignment", "Are we back in blackout?");
46          labThread.setTags(new String[] { "Java" });
47          assignmentThread.setTags(new String[] { "Java" });
48
49          List<Thread> searchResults = forum.searchByTag("Java");
50          assertEquals("Lab 01", searchResults.get(0).getTitle());
51          assertEquals("Assignment", searchResults.get(1).getTitle());
52      }
53  }
```

```
15 public class ArchaicFsTest {
16     @Test
17     public void testCdInvalidDirectory() {
18         ArchaicFileSystem fs = new ArchaicFileSystem();
19
20         // Try to change directory to an invalid one
21         assertThrows(UNSWNoSuchFileException.class, () -> {
22             fs.cd("/usr/bin/cool-stuff");
23         });
24     }
25
26     @Test
27     public void testCdValidDirectory() {
28         ArchaicFileSystem fs = new ArchaicFileSystem();
29
30         assertDoesNotThrow(() -> {
31             fs.mkdir("/usr/bin/cool-stuff", true, false);
32             fs.cd("/usr/bin/cool-stuff");
33         });
34     }
35
36     @Test
37     public void testCdAroundPaths() {
38         ArchaicFileSystem fs = new ArchaicFileSystem();
39
40         assertDoesNotThrow(() -> {
41             fs.mkdir("/usr/bin/cool-stuff", true, false);
42             fs.cd("/usr/bin/cool-stuff");
43             assertEquals("/usr/bin/cool-stuff", fs.cwd());
44             fs.cd("..");
45             assertEquals("/usr/bin", fs.cwd());
46             fs.cd("../bin/..");
47             assertEquals("/usr", fs.cwd());
48         });
49     }
50
51     @Test
52     public void testCreateFile() {
53         ArchaicFileSystem fs = new ArchaicFileSystem();
54
55         assertDoesNotThrow(() -> {
56             fs.writeFile("test.txt", "My Content", EnumSet.of(FileWriteOptions.CREATE, FileWriteOptions.TRUNCATE));
57             assertEquals("My Content", fs.readFromFile("test.txt"));
58             fs.writeFile("test.txt", "New Content", EnumSet.of(FileWriteOptions.TRUNCATE));
59             assertEquals("New Content", fs.readFromFile("test.txt"));
60         });
61     }
62 }
```

Junit: Dynamic and parameterized tests

For information on Dynamic and parameterized tests,

- see the tutorial at <https://www.vogella.com/tutorials/JUnit/article.html>

❖ For more information on JUnit, read the **user guide** at:

- <https://junit.org/junit5/docs/current/user-guide/>

End

Software Design Principles

COMP2511, CSE, UNSW



What Goes Wrong in Software Design?

- ❖ **Initial design** is clean and elegant, often well-structured.
- ❖ **Over time**, design **degrades** due to evolving requirements and rushed changes.
- ❖ Known as "**software rot**", this process makes code **hard to maintain and evolve**.

Symptoms:

- **Rigidity**: Small changes cause widespread impact.
- **Fragility**: One change breaks unrelated parts.
- **Immobility**: Useful components can't be reused easily.
- **Viscosity**: Environment or process encourages hacks over clean design.

Rigidity and Fragility

Rigidity: System resists change due to interdependencies.

Example: A login module change forces updates in unrelated reporting or database modules.

Impact: Managers hesitate to allow even minor fixes.

Fragility: Changes result in unexpected breakages.

Example: Fixing an email validator crashes the profile picture upload feature.

Impact: Developer trust and morale drop; testing becomes difficult.

Observation: The above are due to **poor dependency** management, not just evolving requirements.

Immobility and Viscosity

Immobility: Modules can't be reused due to tight coupling.

Example: A "*UserNotification*" class depends on web framework internals, so we cannot reuse in CLI app.

Design viscosity: Easier to do the wrong thing (hack) than the right thing.

Environmental viscosity: Long compile/test cycles encourage shortcuts.

Example: Hack a feature with global variables instead of refactoring due to 20-minute build time.

Observation: Most symptoms of rot are caused by **bad dependency structures**.

What Are Software Design Principles?

- ❖ They provide guidelines to develop systems that are **maintainable, flexible, reusable, and robust**.
- ❖ Adhering to these principles helps to **mitigate** common software engineering issues such as **design rot (degradation)** and ensures software remains scalable and adaptable over time.
- ❖ Changing requirements **don't have to ruin** design.
- ❖ **Good design anticipates change**, however, bad design breaks under it.

Importance of Software Design Principles

- ❖ **Maintainability**: Software should be easy to update and enhance without extensive refactoring (re-engineering).
- ❖ **Flexibility**: Systems should adapt smoothly to changing requirements.
- ❖ **Reusability**: Components and modules should be designed to be easily reusable across various parts of the application or even in different projects.
- ❖ **Robustness**: The software should handle errors gracefully and maintain functionality under different circumstances.

SOLID Principles (1)

An acronym that represents five crucial principles for object-oriented design:

Single Responsibility Principle (SRP):

- A class should have only one reason to change, focusing on a single functionality.

Open/Closed Principle (OCP):

- Software entities should be open for extension but closed for modification.

Liskov Substitution Principle (LSP):

- Objects of a superclass should be replaceable with objects of subclasses without affecting the correctness of the program.

SOLID Principles (2)

Interface Segregation Principle (ISP):

- Clients should not be forced to depend on interfaces they do not use; favor many specific interfaces over a single general-purpose one.

Dependency Inversion Principle (DIP):

- Depend on abstractions, not concrete implementations. Higher-level modules should not depend on lower-level modules but rather on abstractions.

Why Follow These Principles?

- ❖ **Preventing Software Rot:** Avoid the deterioration of the software design over time.
- ❖ **Ease of Maintenance:** Reduce the cost and effort involved in updating and managing code.
- ❖ **Enhanced Productivity:** Developers spend less time debugging and refactoring, more on innovation and delivering value.
- ❖ **Improved Collaboration:** Clear, principle-driven design aids team communication and collaboration.

Real-world Example

Consider an **online payment** system:

Without design principles:

- ❖ Payment methods (Credit Card, PayPal, Crypto, etc.) tightly coupled in the codebase, making additions or modifications challenging and error-prone.

Applying SOLID principles:

- ❖ Each payment method is encapsulated within its class (**SRP**).
- ❖ Adding new payment methods requires implementing a payment interface without altering existing code (**OCP, DIP**).
- ❖ Users of payment classes aren't exposed to methods irrelevant to them (**ISP**).

Good Design

“Change in software is constant, good design embraces it!”

- ❖ Following structured design principles ultimately results in **higher-quality, longer-lasting** software.

Software Cohesion and Coupling

- ❖ **Cohesion:** The degree to which elements of a module/class **belong together**.
- ❖ **Coupling:** The **degree of interdependence** between software modules.
- ❖ High cohesion and low coupling are hallmarks of good software design.

What is Cohesion?

- ❖ **Cohesion**: The degree to which elements of a module/class belong together.
- ❖ **High Cohesion**: Elements of the module work towards a single purpose.
- ❖ **Low Cohesion**: Elements are unrelated or loosely related.
- ❖ Aim for **high cohesion** for maintainability and readability.

Examples of Cohesion

High Cohesion

Class: InvoiceProcessor

- **Methods:** calculateTotal(), applyDiscount(), generateInvoice()
- All methods related to processing an invoice.

Benefits: Easier to understand and maintain, Reusable

Low Cohesion

Class: UtilityClass

- **Methods:** readFile(), sendEmail(), sortArray()
- Functions unrelated to one another.

Problems: Hard to maintain, Difficult to test, Not reusable as a unit

What is Coupling?

- ❖ **Coupling**: The degree of interdependence between software modules.
- ❖ **Tight Coupling**: Modules heavily dependent on each other.
- ❖ **Loose Coupling**: Modules operate independently with minimal dependencies.
- ❖ **Aim for** loose coupling to enable flexibility and reuse.

Types of Coupling

Some of the important types of coupling are:

- ❖ **Data** Coupling: Modules share data through parameters.
- ❖ **Control** Coupling: One module controls the flow of another (e.g., flags).
- ❖ **External** Coupling: Modules depend on externally imposed data formats.
- ❖ **Common** Coupling: Shared global variables.
- ❖ **Content** Coupling: One module modifies data of another.

Examples of Coupling

Low Coupling

Modules: UserInterface, BusinessLogic, DataAccess

- Each layer interacts through interfaces.

Benefits: Easy to change or replace components, Improved testability

High Coupling

Class A calls methods of **Class B** directly and modifies its state.

Problems: Difficult to reuse or refactor, Ripple effects from changes

Design Tips for High Cohesion

- ❖ Use the **Single Responsibility Principle** (SRP), as far as possible.
- ❖ **Group** related functionalities.
- ❖ **Avoid** “God classes”.
- ❖ **Refactor** when a class or method grows too large.

Design Tips for Low Coupling

- ❖ Minimize shared data
- ❖ Use interfaces and abstractions
- ❖ Apply Dependency Injection
- ❖ Use event-driven or observer patterns, for loosely dynamically coupled systems

When to Use Design Principles?

- ❖ Design principles **help to remove** design smells: needless complexity.
- ❖ However, they **should not be used** when there are **no** design **smells**.
- ❖ It is a **mistake to blindly accept** a principle just because it is one.
- ❖ **Avoid over-adherence**, it can create a new design smell: needless complexity.

Design Principle:

Principle of Least Knowledge (Law of Demeter)

- ❖ The **Principle of Least Knowledge** (also called the **Law of Demeter**) suggest that a module (or object) should **only talk to its immediate "friends"**, and **not to strangers**.
- ❖ In simpler terms: **“Only call methods on objects you directly know.”**

❖ Formal Rule

A method **M** of an object **O** may only invoke methods that belong to:

- 1) O itself
- 2) M's parameters
- 3) Any objects created/instantiated within M
- 4) O's direct fields (its own instance variables)

Design Principle:

Principle of Least Knowledge (Law of Demeter)

- ❖ **Minimises coupling:** Prevents objects from becoming overly dependent on others' internal structure.
- ❖ **Enhances maintainability:** Changes in one class are less likely to ripple through the system.
- ❖ **Improves encapsulation:** Objects hide their data better and expose minimal necessary interfaces.

Code Example – Violating LoD (Tightly Coupled)

```
class Engine {  
    public void start() { System.out.println("Engine started"); }  
}  
  
class Car {  
    private Engine engine = new Engine();  
    public Engine getEngine() { return engine; }  
}  
  
class Driver {  
    public void startCar(Car car) {  
        car.getEngine().start();  
    }  
}
```

Violates LoD,
accessing a “stranger” (engine)



Code Example – Respecting LoD (Loosely Coupled)

```
class Engine {  
    public void start() { System.out.println("Engine started"); }  
}  
  
class Car {  
    private Engine engine = new Engine();  
    public void start() { engine.start(); }  
}  
  
class Driver {  
    public void startCar(Car car) {  
        car.start();  
    }  
}
```

Car mediates access



Talks only to its direct friend



Definition of LSP (Liskov Substitution Principle)

"Objects of a superclass should be replaceable with objects of a subclass **without breaking** the application."

- *Barbara Liskov, 1987*

❖ This ensures a subclass behaves in ways that **do not** surprise or **violate** the expectations set by the parent class.

❖ Formally:

"Let S be a subtype of T. Then, objects of type T may be replaced with objects of type S **without altering** any of the desirable properties of the program."

Real-World Analogy: LSP

- ❖ Superclass: **Bird**
Subclass: **Penguin**
- ❖ Birds can fly, therefore fly() is in the base (super) class **Bird**.
- ❖ Penguins are birds, but they **cannot fly**.
- ❖ **Problem**: Substituting **Penguin** for **Bird** breaks the program!

Examples: LSP

```
class Bird {  
    void fly() {  
        System.out.println("Flying...");  
    }  
}
```

```
class Ostrich extends Bird {  
    @Override  
    void fly() {  
        throw new UnsupportedOperationException("Ostrich can't fly");  
    }  
}
```

Violating LSP

```
interface Bird {  
    void eat();  
}
```

```
interface FlyingBird extends Bird {  
    void fly();  
}
```

```
class Sparrow implements FlyingBird {  
    public void fly() { System.out.println("Sparrow flies"); }  
    public void eat() { System.out.println("Sparrow eats"); }  
}
```

```
class Ostrich implements Bird {  
    public void eat() { System.out.println("Ostrich eats"); }  
}
```

Fixing the Violation

Example: LSP (Shape Hierarchy)

```
class Rectangle {  
    int width, height;  
    void setWidth(int w) { width = w; }  
    void setHeight(int h) { height = h; }  
    int area() { return width * height; }  
}
```

```
class Square extends Rectangle {  
    void setWidth(int w) {  
        width = w;  
        height = w;  
    }  
    void setHeight(int h) {  
        width = h;  
        height = h;  
    }  
}
```

We cannot substitute Square for Rectangle,
may break logic expecting width != height.

Results in **incorrect** behavior!



```
Rectangle r1 = new Rectangle();  
r1.setWidth(50); // only changes Width, as expected  
r1 = new Square();  
r1.setWidth(50); // Unexpectedly, it also changes Height!
```

So, we cannot replace an object of Rectangle by an object of Square!

Example: LSP (Shape Hierarchy)

```
class Rectangle {  
    int width, height;  
    void setWidth(int w) { width = w; }  
    void setHeight(int h) { height = h; }  
    int area() { return width * height; }  
}
```

```
class Square extends Rectangle {  
    void setWidth(int w) {  
        width = w;  
        height = w;  
    }  
    void setHeight(int h) {  
        width = h;  
        height = h;  
    }  
}
```



We cannot substitute Square for Rectangle,
may break logic expecting width != height.

After refactoring



```
interface Shape {  
    int area();  
}
```



```
class Rectangle implements Shape { ... }  
class Square implements Shape { ... }
```

Why LSP Matters

- ❖ Encourages correct hierarchy modelling
- ❖ Enables safe polymorphism
- ❖ Reduces unexpected behaviour at runtime
- ❖ Facilitates reusability and maintainability
- ❖ Think of LSP as a contract: subclasses must honour the guarantees of their parents.

Introduction to Covariance and Contravariance

- ❖ Covariance and Contravariance describe how types behave in inheritance when method overriding.
- ❖ Covariance: Return type can be more specific (subtype)
- ❖ Contravariance: Parameter types can be more general (supertype)

Covariant Return Types

- ❖ Allows the **return type** in an overridden method to be a **subtype of the original**.
- ❖ Enables **more specific** results while remaining compatible.

```
class Animal {}  
class Dog extends Animal {}  
  
class AnimalShelter {  
    Animal adopt() { return new Animal(); }  
}  
  
class DogShelter extends AnimalShelter {  
    @Override  
    Dog adopt() { return new Dog(); }  
}
```

Contravariance in Parameters

- ❖ **Contravariant** parameters accept **supertypes** of the original type.
- ❖ This is **not allowed** in typical method overriding (**Java**, C++).

```
class Parent {  
    void process(Number n) { ... }  
}  
  
class Child extends Parent {  
    void process(Integer i) { ... }  
}
```

Not Overriding,
But results in Overloading!
Now there are two methods,
one each for *Number* and *Integer* types.

Rules Summary for Method Overriding

Aspect	Rule in OOP Overriding
Method Name	Must match
Parameters	Must be identical
Return Type	Covariant allowed
Exceptions	Can be narrower
Access Modifier	Can be more open

End

Refactoring

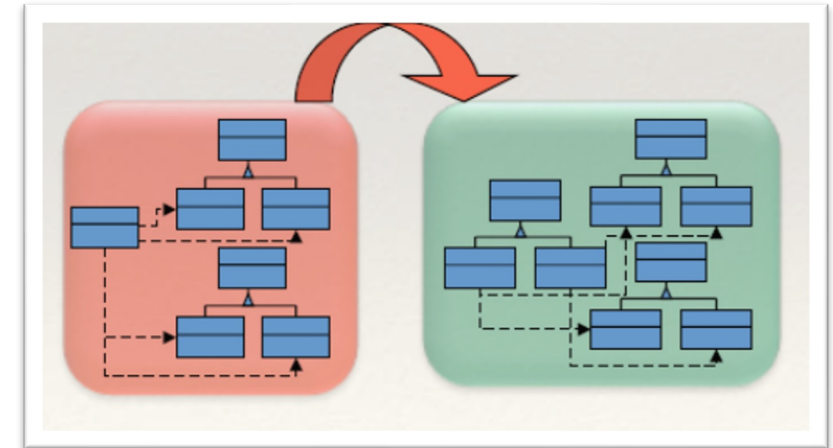
COMP2511, CSE, UNSW



UNSW
SYDNEY

Introduction to Refactoring

- ❖ Refactoring is the process of restructuring existing code **without changing** its **external behavior**.
- ❖ Aim is to:
 - **improve** internal structure/**design**, **readability**, and **maintainability**
 - help **detect bugs**.
 - increase **development speed**.
 - help conform to design principles and **eliminate** design/**code smells**.



When to Refactor

- ❖ Before adding **new features** if current structure is not suitable.
- ❖ While **fixing bugs**.
- ❖ During **code reviews**.


Code Smells

- ❖ **Code smells** are indicators of potential **design issues**.
- ❖ They hint at poor design but do **not** guarantee **defects**.
- ❖ **Refactoring** addresses code smells.

Common Code Smells:

Duplicated Code	Shotgun Surgery
Long Method	Feature Envy
Large Class	Lazy Classes
Long Parameter List	Data Classes
Divergent Change	

Refactoring Cycle

- 
- ❖ Step 1: Identify **code smell**.
 - ❖ Step 2: **Write tests** to confirm current behaviour.
 - ❖ Step 3: Apply small **refactoring** step.
 - ❖ Step 4: **Re-run tests**.
 - ❖ Step 5: **Repeat**.

Refactoring Technique: Extract Method

- ❖ Identify **logical chunks of code** and extract into separate methods.
- ❖ **Benefits:** improves readability, reduces duplication.

Before

```
void printOwing() {  
    printBanner();  
    // calculate outstanding  
    double outstanding = 0;  
    for (Order o : orders) {  
        outstanding += o.getAmount();  
    }  
    printDetails(outstanding);  
}
```

After

```
void printOwing() {  
    printBanner();  
    double outstanding = calculateOutstanding();  
    printDetails(outstanding);  
}  
  
double calculateOutstanding() {  
    double result = 0;  
    for (Order o : orders) {  
        result += o.getAmount();  
    }  
    return result;  
}
```

Refactoring Technique: Move Method

- ❖ Move methods to the class whose data they use most.

```
class Customer {  
    double getDiscount(Product product) {  
        return product.getBasePrice() * 0.1;  
    }  
}
```

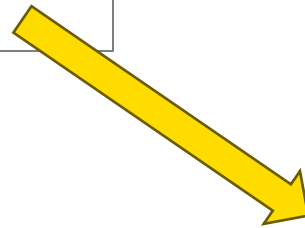
Move *getDiscount* to *Product* class.

```
class Product {  
    double getDiscount() {  
        return this.getBasePrice() * 0.1;  
    }  
}
```

Refactoring Technique: Replace Temp with Query

- ❖ Move expressions into methods instead of temporary variables.

```
double basePrice = quantity * itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;
```



```
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
  
double basePrice() {  
    return quantity * itemPrice;  
}
```

Refactoring Technique: Replace Conditional with Polymorphism

- ❖ **Switch** or **if-else chains** based on type codes are hard to maintain and violate OOP principles.
 - Adding a new type **requires changes** to every switch statement.
 - Increases rigidity and breaks **Open/Closed Principle**.

Solution:

- Replace **switch** statements with inheritance.
- Define a superclass with an abstract method and implement this method in subclasses, each representing a case of the switch.

Refactoring Technique: Replace Conditional with Polymorphism

❖ Use **polymorphism** instead of conditionals.

```
class Movie {  
    int getPriceCode();  
}  
  
class Rental {  
    double getCharge() {  
        switch(movie.getPriceCode()) {  
            case REGULAR: return daysRented * 2;  
            case CHILDRENS: return daysRented * 1.5;  
        }  
    }  
}
```



```
abstract class Movie {  
    abstract double getCharge(int daysRented);  
}
```

```
class RegularMovie extends Movie {  
    double getCharge(int daysRented) {  
        return daysRented * 2;  
    }  
}
```

```
class ChildrensMovie extends Movie {  
    double getCharge(int daysRented) {  
        return daysRented * 1.5;  
    }  
}
```

Refactoring Using Composition

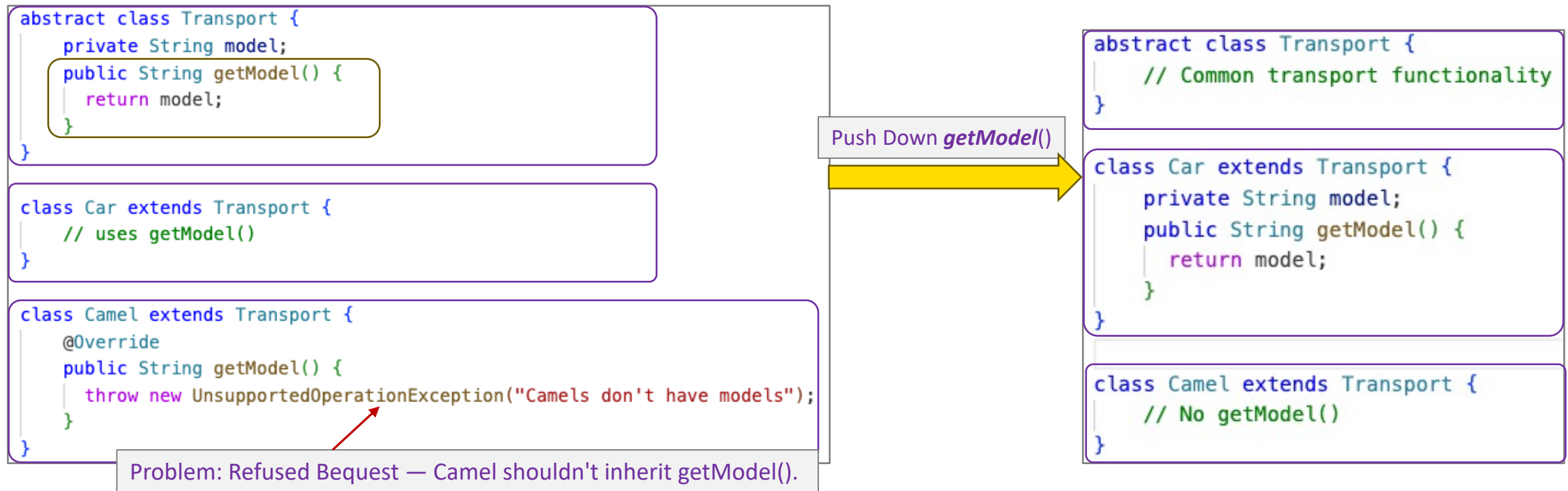
- ❖ Favor composition over inheritance.

Instead of extending *Logger* class,
use composition (has-a relation) and method forwarding.

```
class Application {  
    private Logger logger = new Logger();  
    void logInfo(String msg) {  
        logger.log(msg);  
    }  
}
```

Design Smell: Refused Bequest

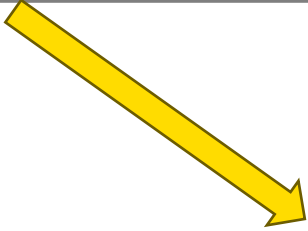
Refused Bequest: subclass inherits inappropriate behavior.



Smell: Long Parameter List

- ❖ To avoid long parameter lists, encapsulate related parameters into a data class and pass an instance of that class instead.

```
void createUser(String name, int age, String email, String phone)
```



```
class UserInfo {  
    String name;  
    int age;  
    String email;  
    String phone;  
}
```

```
void createUser(UserInfo user)
```

Smell: Large Method/Class

- ❖ **Large Method**: method with many lines doing multiple things.
- ❖ Refactor: use **Extract Method** to create new method(s)

- ❖ **Large Class**: Class with 20+ methods and many fields.
- ❖ Refactor: use **Extract Class** to separate concerns.

Smell: Similar Code Fragments

Case 1: Same code in multiple methods of the same class

- Use **Extract Method** and invoke it from each place.

Case 2: Same code in two subclasses of the same level

- Use **Extract Method** in both subclasses, Use **Pull Up Field** or **Pull Up Method** to unify code in the superclass.
- If inside constructors: use **Pull Up Constructor Body**.
- For similar but not identical code: use **Template Method**.
- If algorithms differ, use **Strategy Pattern**.

Case 3: Duplicate code in unrelated classes

- Use **Extract Superclass** to unify shared logic.

Smell: Feature Envy

- ❖ A method is more interested in **another class's data** than its own.

Symptoms

- The method invokes several methods on **another object** to calculate a value.
- Causes unnecessary coupling and **breaks encapsulation**.

Solution: Move the method to the class that owns the data (**Move Method**).

- If only part of the method accesses external data: use **Extract Method** followed by **Move Method**.
- If multiple external classes are involved: identify which one holds the majority of used data and move the method there.

Smell: Divergent Change

- ❖ A class is changed in many unrelated ways for different reasons.
- ❖ Violates Single Responsibility Principle.
- ❖ Increases risk of regression bugs due to unrelated modifications

Solution:

- Identify the reasons for change and separate them into cohesive classes.
- Use **Extract Class** to encapsulate each responsibility.

```
// Before
class DocumentManager {
    void print(Document doc) { ... }
    void save(Document doc) { ... }
    void exportToPDF(Document doc) { ... }
}

// After
class PrintService {
    void print(Document doc) { ... }
}

class PersistenceService {
    void save(Document doc) { ... }
}

class ExportService {
    void exportToPDF(Document doc) { ... }
}
```


Smell: Shotgun Surgery

- ❖ A small change requires updating many different classes.
- ❖ Makes code brittle and hard to maintain.

Solution:

- Consolidate related changes into a single class.
- Use **Move Method**, **Move Field**, or **Inline Class** to localize the change.

```
// Before: logic for logging exists in multiple classes
class Order {
    void logCreation() { Logger.log("Order created"); }
}
class Invoice {
    void logGeneration() { Logger.log("Invoice generated"); }
}
```



```
// After: Centralized logging
class LogService {
    static void logOrderCreation() { Logger.log("Order created"); }
    static void logInvoiceGeneration() { Logger.log("Invoice generated"); }
}
```

Divergent Change and Shotgun Surgery

- ❖ **Divergent Change** = One class changes for many unrelated reasons.
- ❖ **Shotgun Surgery** = One change spreads across many classes.
- ❖ Both can be addressed with refactoring to improve modularity and reduce fragility.

Useful Links

<https://refactoring.guru/refactoring/smells>

<https://www.refactoring.com/catalog/>

Demo

The Video Rental System

End

Introduction to Software Patterns and Strategy Pattern

COMP2511, CSE, UNSW



What Are Design Patterns?

- ❖ **Proven** solutions to common software design problems.
- ❖ **Reusable** templates that help structure software.
- ❖ Provide **shared vocabulary** for developers.

Why Use Design Patterns?

- ❖ Serve as a **template** or a **guide** for addressing important software design issues.
- ❖ Is **not a complete implementation**, but rather a **flexible guideline** for addressing recurring design challenges.
- ❖ **Captures design expertise**, making it easier to share and reuse across projects.
- ❖ Offers a **common vocabulary** that enhances communication among developers.
- ❖ Improve code **readability** and **reusability**
- ❖ Promote **best practices** and industry standards
- ❖ Facilitate **maintainability** and **scalability**

Mastering Design Patterns – An Art & Craft

- ❖ Develop a strong working **knowledge of** various **patterns**.
- ❖ **Understand clearly** the problems they can effectively solve.
- ❖ Recognize accurately when a specific problem can benefit from applying a pattern.

Origins and History of Design Patterns

- ❖ The concept stems from architecture, originally introduced by [Christopher Alexander](#) and colleagues, who identified around 250 design patterns for building construction.
- ❖ [Adapted](#) to software by the "[Gang of Four](#)" (GoF): Gamma, Helm, Johnson, Vlissides
- ❖ GoF Book (1994): *Design Patterns: Elements of Reusable Object-Oriented Software*

Key Elements of a Design Pattern:

- ❖ **Name:** Identifier for pattern
- ❖ **Problem:** Context and issue
- ❖ **Solution:** General design
- ❖ **Consequences:** Results and trade-offs

When NOT to Use Patterns

- ❖ When patterns add unnecessary complexity
- ❖ When simpler solutions suffice
- ❖ Avoid "pattern abuse" or "overengineering"

Design Patterns vs. Algorithms

- ❖ **Algorithms** solve computational problems
- ❖ **Design Patterns** solve design/architectural problems
- ❖ **Example:**
 - Algorithm: *QuickSort*
 - Pattern: Strategy to switch sorting algorithms

Design Patterns and Software Principles

❖ Closely tied to SOLID principles:

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

❖ Patterns tries to address SOLID principles

Problem Statement

Design Problem:

For simulation, represent a **car** with **varying types** of **engines** and **brakes**.

- ❖ A **Car** class should support, along with other behaviours:
 - **4 types of engines** (e.g., Petrol, Diesel, Electric, Hybrid)
 - **5 types of brakes** (e.g., Disc, Drum, Regenerative, ABS, Air Brakes)
- ❖ Requirements **may change** (add or modify engine/brake types)

Implementation with If-Else

```
public class Car {  
    private String engineType;  
    private String brakeType;  
  
    public void startEngine() {  
        if (engineType.equals("petrol")) {  
            // Petrol engine logic  
        } else if (engineType.equals("diesel")) {  
            // Diesel engine logic  
        } else if (engineType.equals("electric")) {  
            // Electric engine logic  
        } else if (engineType.equals("hybrid")) {  
            // Hybrid engine logic  
        }  
    }  
  
    public void applyBrakes() {  
        if (brakeType.equals("disc")) {  
            // Disc brake logic  
        } else if (brakeType.equals("drum")) {  
            // Drum brake logic  
        } else if (brakeType.equals("regenerative")) {  
            // Regenerative braking logic  
        }  
        // ...and so on  
    }  
}
```


Implementation with If-Else

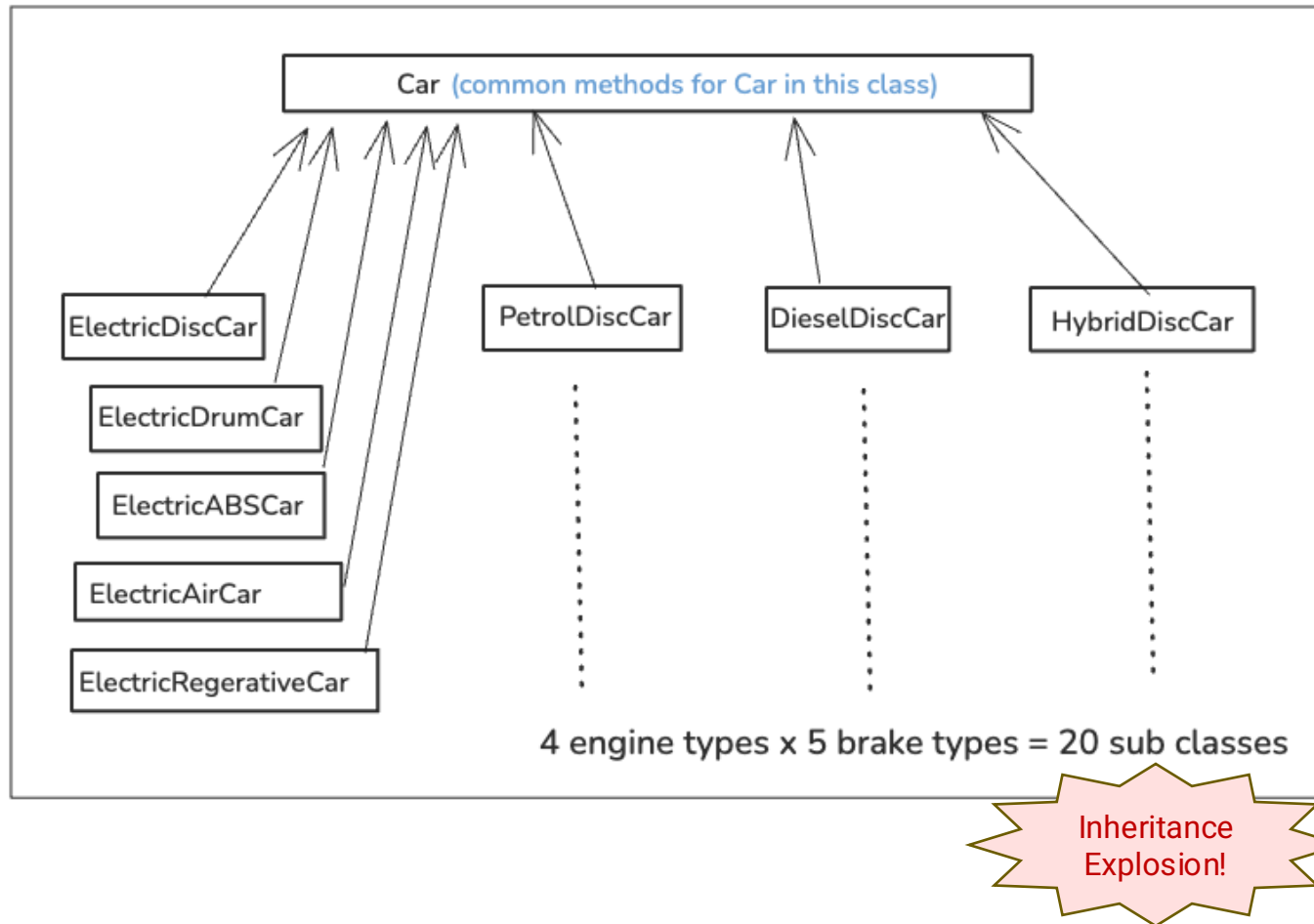
Problems with hardcoding logic, it is a **bad practice**:

- ❖ **Violates the Open-Closed Principle**: Class must be modified for every new brake or engine type.
- ❖ **Adding new behaviour** leads to code duplication and potential bugs.
- ❖ **Not scalable**: Explosion of if-else or switch blocks.
- ❖ **Code is hard to read and maintain.**

```
public class Car {  
    private String engineType;  
    private String brakeType;  
  
    public void startEngine() {  
        if (engineType.equals("petrol")) {  
            // Petrol engine logic  
        } else if (engineType.equals("diesel")) {  
            // Diesel engine logic  
        } else if (engineType.equals("electric")) {  
            // Electric engine logic  
        } else if (engineType.equals("hybrid")) {  
            // Hybrid engine logic  
        }  
    }  
  
    public void applyBrakes() {  
        if (brakeType.equals("disc")) {  
            // Disc brake logic  
        } else if (brakeType.equals("drum")) {  
            // Drum brake logic  
        } else if (brakeType.equals("regenerative")) {  
            // Regenerative braking logic  
        }  
        // ...and so on  
    }  
}
```

Bad
design!

Alternative: Inheritance-Based Design



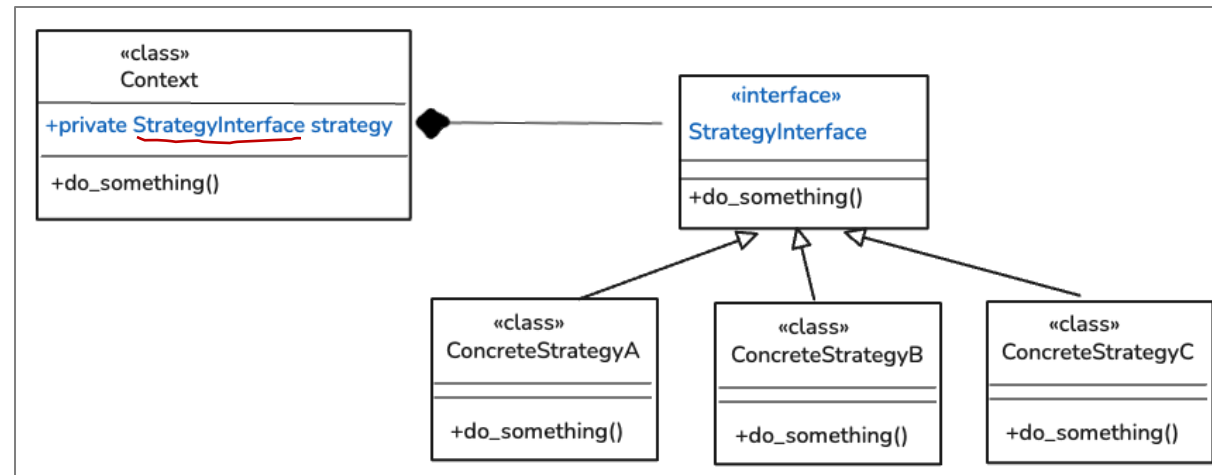
- ❖ Consider subclassing for **each combination**.
- ❖ With **M engines** types and **N brakes** types, we need **$M \times N$ subclasses**
- ❖ Adding a new engine type requires **N new classes**, for each brake type.
- ❖ Inheritance **Explosion** Problem!
Not scalable
- ❖ **Tightly couples** engine and brake behaviour
- ❖ Hard to test and **reuse** logic

Strategy Pattern: Motivation

- ❖ **Hardcoding** algorithm logic in a class makes it **inflexible**.
- ❖ *Example*: A **Car** class with **multiple** engine and brake **behaviours**.
- ❖ *Problems*:
 - What if we need to represent all possible unique combinations of brakes and engines?
 - What if we need to change engine/brake behaviour at runtime?

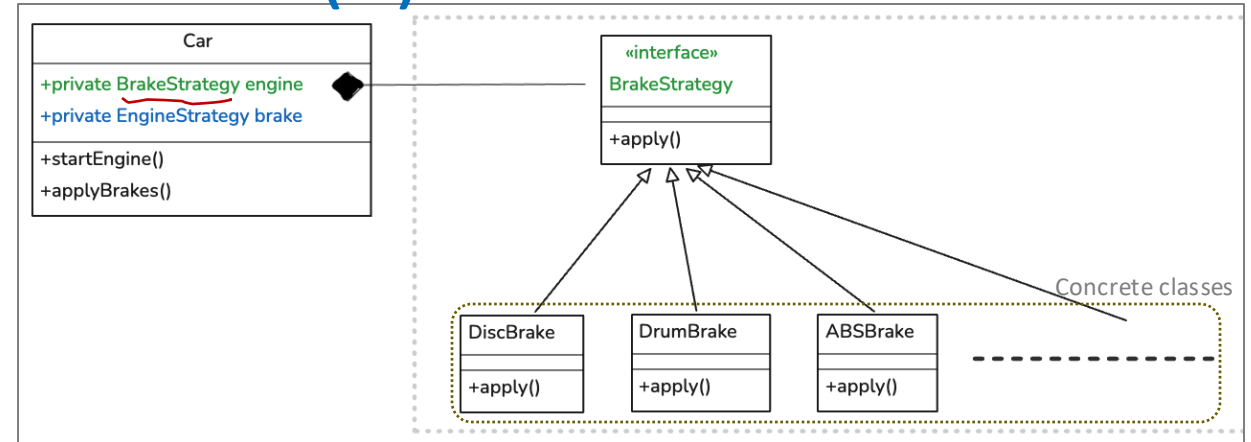
Strategy Pattern

- ❖ Define a **family of algorithms** (e.g. family of engine algorithms).
- ❖ Encapsulate each algorithm in a separate **strategy class** (e.g. a class for petrol engine, a class for electric engine, etc.).
- ❖ Make algorithms **interchangeable** in the context object (e.g. in a car object).
- ❖ Vary behaviour **without changing** the **context** class.



Alternative: Using Strategy Pattern (1)

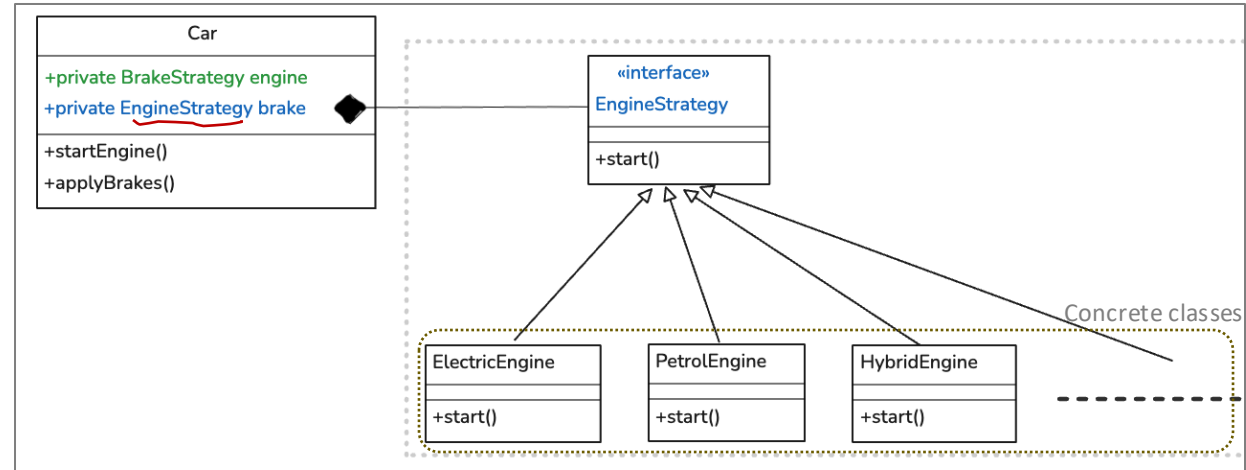
- ❖ A **Car** class contains an object of type **BrakeStrategy**.
- ❖ **BrakeStrategy** is an interface that defines a method such as **apply()** to **encapsulate brake behaviour**.
- ❖ Various concrete **classes** like **DiscBrake**, **ABSBrake**, etc. **implement** the **BrakeStrategy** interface to represent different braking strategies.
- ❖ The **Car** class **delegates** its **braking strategy** to the associated **BrakeStrategy** object/instance.



```
public class Car {  
    private EngineStrategy engine;  
    private BrakeStrategy brake;  
  
    public Car(EngineStrategy engine, BrakeStrategy brake) {  
        this.engine = engine;  
        this.brake = brake;  
    }  
  
    public void startEngine() { engine.start(); }  
    public void applyBrakes() { brake.apply(); }  
}
```

Alternative: Using Strategy Pattern (2)

- ❖ Similarly, a **Car** class contains an object of type **EngineStrategy**.
- ❖ EngineStrategy is an interface that defines a method such as **start()** to **encapsulate engine behaviour**.
- ❖ Various concrete **classes** like **ElectricEngine**, **PetrolEngine**, etc. **implement** the EngineStrategy interface to represent different engine strategies.
- ❖ The **Car** class **delegates** its **engine strategy** to the associated EngineStrategy object/instance.



```
public class Car {  
    private EngineStrategy engine;  
    private BrakeStrategy brake;  
  
    public Car(EngineStrategy engine, BrakeStrategy brake) {  
        this.engine = engine;  
        this.brake = brake;  
    }  
  
    public void startEngine() { engine.start(); }  
    public void applyBrakes() { brake.apply(); }  
}
```

Using the Strategy-Based Car

```
EngineStrategy engine = new ElectricEngine();  
BrakeStrategy brake = new RegenerativeBrake();  
Car car = new Car(engine, brake);  
car.startEngine();  
car.applyBrakes();
```

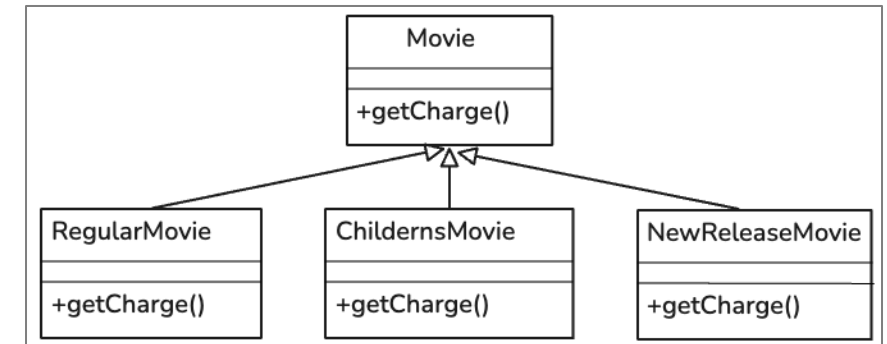
Strategy Pattern to the Rescue

Use **composition** to encapsulate engine and brake behaviour:

- ❖ **Encapsulate** variations
- ❖ Add **more classes** for new engine and brake types
- ❖ Use **method overriding** to change behaviour of the existing engine/brake
- ❖ Adheres to **Open-Closed Principle** (e.g. no need to change Car class for the above)

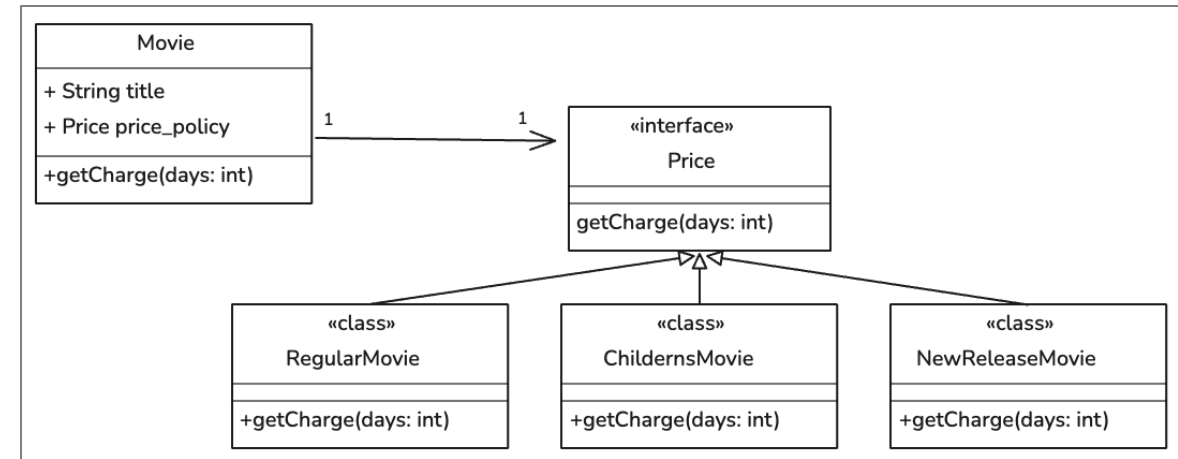
Video Rental Example: Using Inheritance

- ❖ The **Movie** is defined as an **interface**.
- ❖ Each concrete movie class (RegularMovie, ChildrenMovie, NewReleaseMovie) handles **both** the **movie class** and its **pricing logic**, resulting in **tight coupling**.
- ❖ However, a movie's **classification** or its **pricing can change** during its lifetime.
- ❖ Modifying a movie's class or pricing behaviour at runtime is **not straightforward** in this design.
- ❖ This approach is not ideal; we can **refactor and improve it using the Strategy Pattern**, which allows dynamic selection of pricing behaviour.



Video Rental Example: Using Strategy Pattern

- ❖ A **Movie** class contains a reference to a **Price** strategy object.
- ❖ **Price** is an interface that defines methods such as **getCharge(days)** to encapsulate pricing behaviour.
- ❖ Various **concrete classes** like **ChildrenPrice**, **RegularPrice**, and **NewReleasePrice** **implement the Price interface** to represent different pricing strategies.
- ❖ The **Movie** class **delegates** its **pricing logic** to the associated **Price strategy instance**.
- ❖ To change the pricing behaviour of a movie, simply assign a **different Price strategy object**, making the design flexible and maintainable.



Benefits of Strategy Pattern

- ❖ Promotes **Composition over Inheritance**: Allows behaviours to be combined and reused without deep inheritance hierarchies.
- ❖ Supports **Runtime Behaviour Change**: Strategies can be swapped dynamically at runtime to adapt to changing context (e.g., a hybrid car switching between electric and petrol engines).
- ❖ **Encourages Separation of Concerns**: Keeps the Car class focused on orchestration while delegating specific behaviours to strategy classes.
- ❖ **Enables Open-Closed Principle**: New strategies can be added without changing existing code, reducing the risk of introducing bugs.
- ❖ Encourages **modular** design.
- ❖ **Scalable** and **reusable** components

Composite Pattern

COMP2511, CSE, UNSW



UNSW
SYDNEY

Composite Pattern

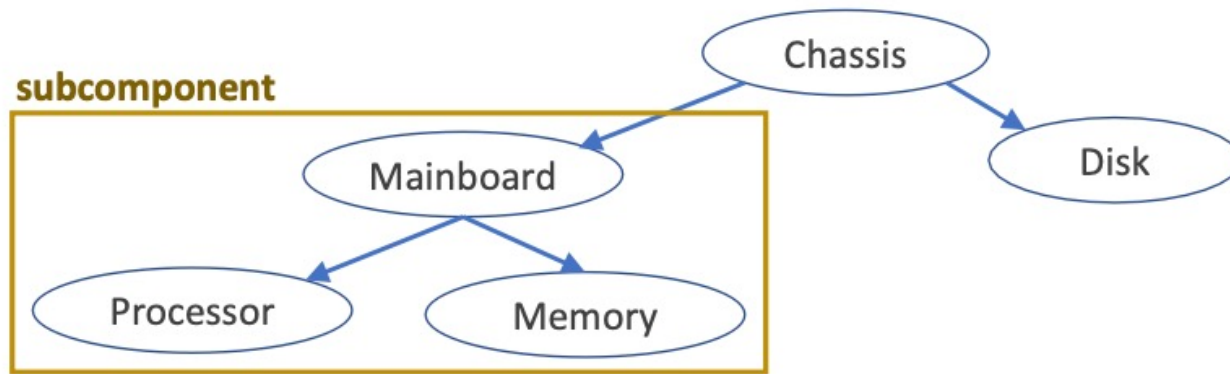
These lecture notes use material from the reference book “*Head First Design Patterns*”.

Composite Pattern: Motivation and Intent

- In OO programming, a **composite** is an object designed as a composition of one-or-more similar objects (exhibiting similar functionality).
- Aim is to be able to manipulate a single instance of the object just as we would manipulate a group of them. For example,
 - operation to resize a **group** of Shapes should be **same as** resizing a **single** Shape.
 - calculating size of a **file** should be **same as** a **directory**.
- **No discrimination** between a Single (leaf) Vs a Composite (group) object.
 - If we discriminate between a single object and a group of object, code will become more complex and therefore, more error prone.

Composite Pattern: More Examples

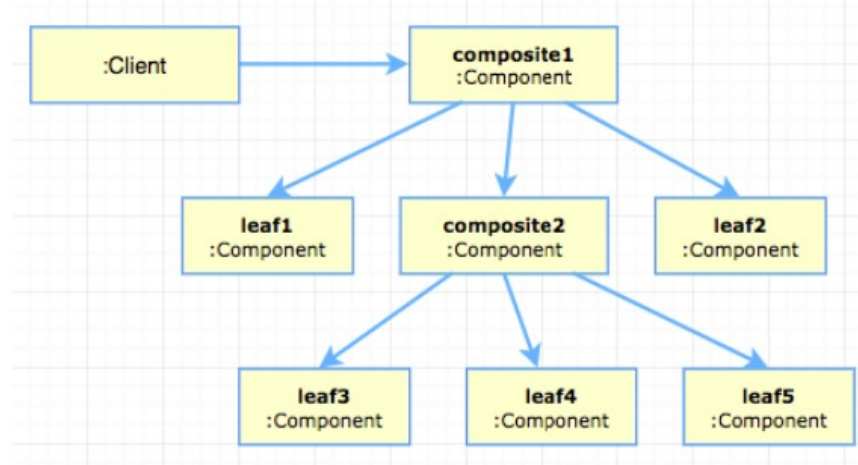
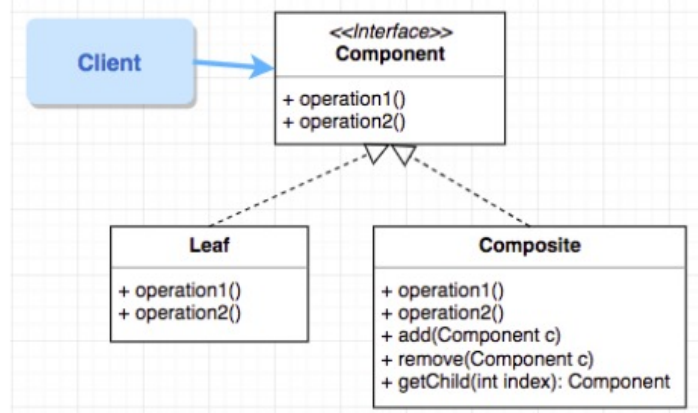
Calculate the total price of an **individual part** or a complete **subcomponent** (consisting of many parts) without having to treat part and subcomponent differently.



A **text document** can be organized as **part-whole hierarchy** consisting of

- characters, pictures, lines, pages, etc. (parts) and
- lines, pages, document, etc. (wholes).
- Display a line, page or the entire document (consisting of many pages) **uniformly** using the same operation/method.

Composite Pattern: Possible Solution



- Define a unified **Component** interface for both **Leaf** (*single / part*) objects and **Composite** (*Group / whole*) objects.
- A **Composite** stores a **collection of children** components (either **Leaf** and/or **Composite** objects).
- Clients can **ignore the differences** between compositions of objects and individual objects, this greatly **simplifies clients** of complex hierarchies and makes them easier to implement, change, test, and reuse.

Composite Pattern: Possible Solution

- **Tree structures** are normally used to represent part-whole hierarchies. A multiway tree structure stores a collection of say **Components** at each node (**children** below), to store **Leaf** objects and **Composite** (subtree) objects.
- A **Leaf** object performs operations directly on the object.
- A **Composite** object performs operations on its **children**, and if required, collects return values and derives the required answers.

Code Segment from the **Composite** class

```
ArrayList<Component> children = new ArrayList<Component>();

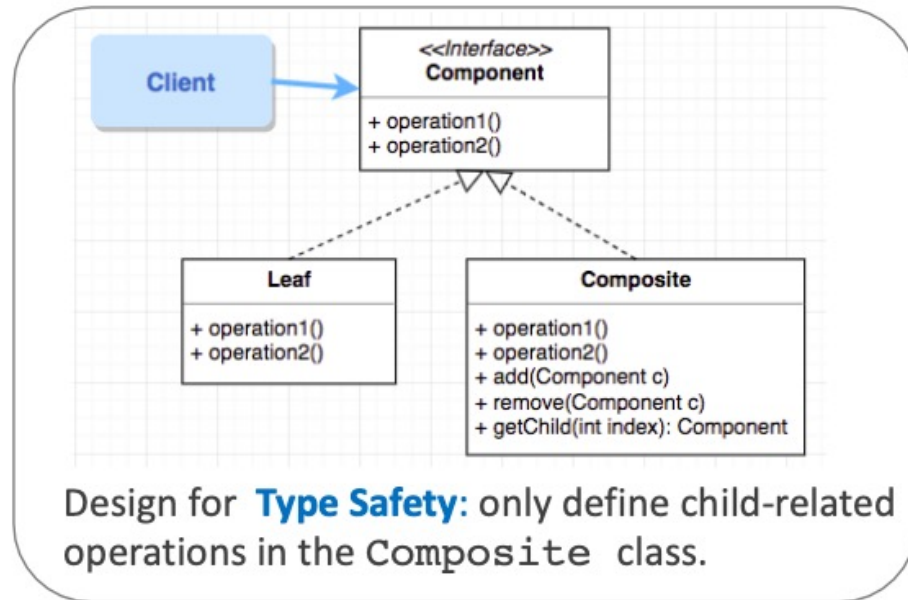
@Override
public double calculateCost() {
    double answer = this.getCost();
    for(Component c : children) {
        answer += c.calculateCost();
    }

    return answer;
}
```

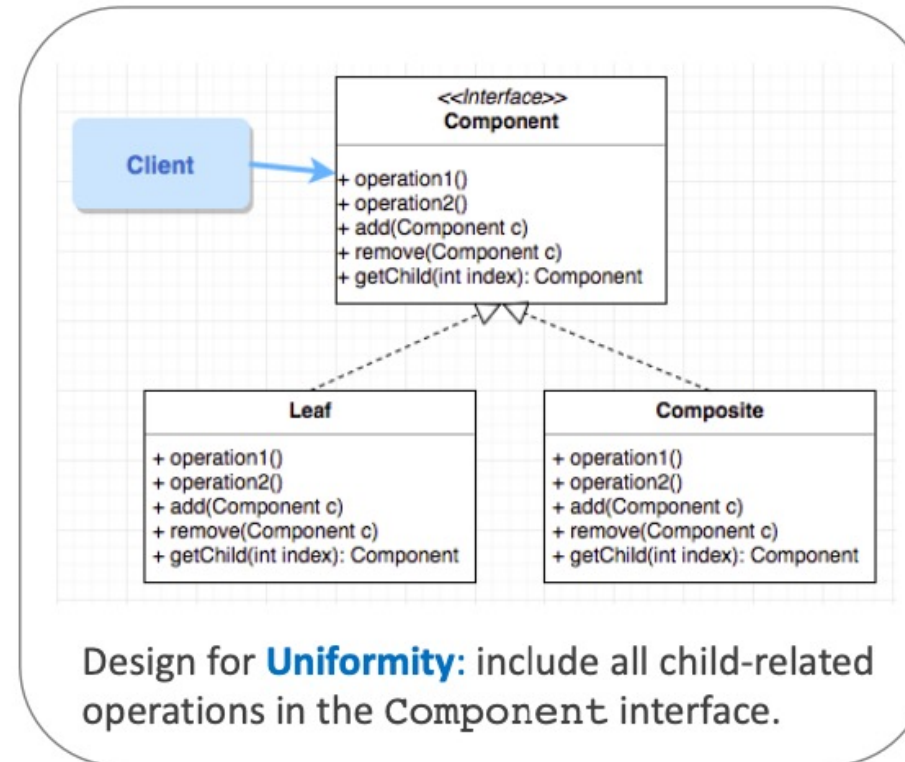
For more, read the example code provided for this week

Implementation Issue: Uniformity vs Type Safety

Two possible approaches to implement child-related operations (methods like add, remove, getChild, etc.):



See the [next slide](#) for more details.



Implementation Issue: Uniformity vs Type Safety

Design for **Uniformity**

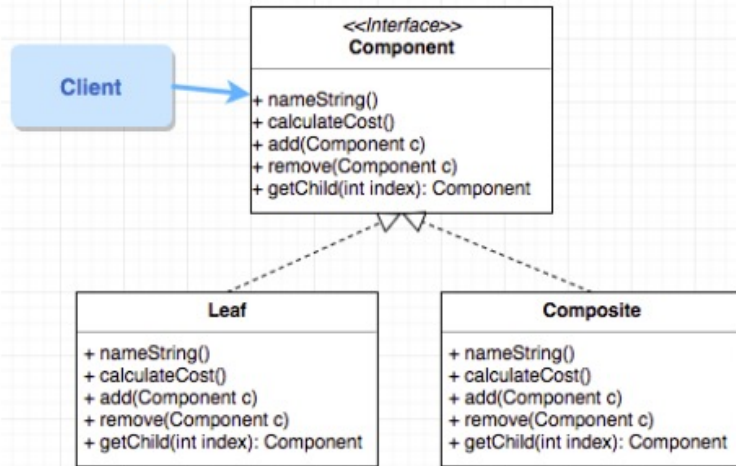
- include all child-related operations in the `Component` interface, this means the `Leaf` class needs to implement these methods with “do nothing” or “throw exception”.
- a client can treat both `Leaf` and `Composite` objects uniformly.
- we lose type safety because `Leaf` and `Composite` types are not cleanly separated.
- useful for dynamic structures where children types change dynamically (from `Leaf` to `Composite` and vice versa), and a client needs to perform child-related operations regularly. For example, a document editor application.

Design for **Type Safety**

- only define child-related operations in the `Composite` class
- the type system enforces type constraints, so a client cannot perform child-related operations on a `Leaf` object.
- a client needs to treat `Leaf` and `Composite` objects differently.
- useful for static structures where a client doesn't need to perform child-related operations on “unknown” objects of type `Component`.

Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week



```

Component mainboard = new Composite("Mainboard", 100);
Component processor = new Leaf("Processor", 450);
Component memory = new Leaf("Memory", 80);
mainboard.add(processor);
mainboard.add(memory);

```

```

Component chasis = new Composite("Chasis", 75);
chasis.add(mainboard);

```

```

Component disk = new Leaf("Disk", 50);
chasis.add(disk);

```

```

System.out.println("[0] " + processor.nameString());
System.out.println("[0] " + processor.calculateCost());

```

```

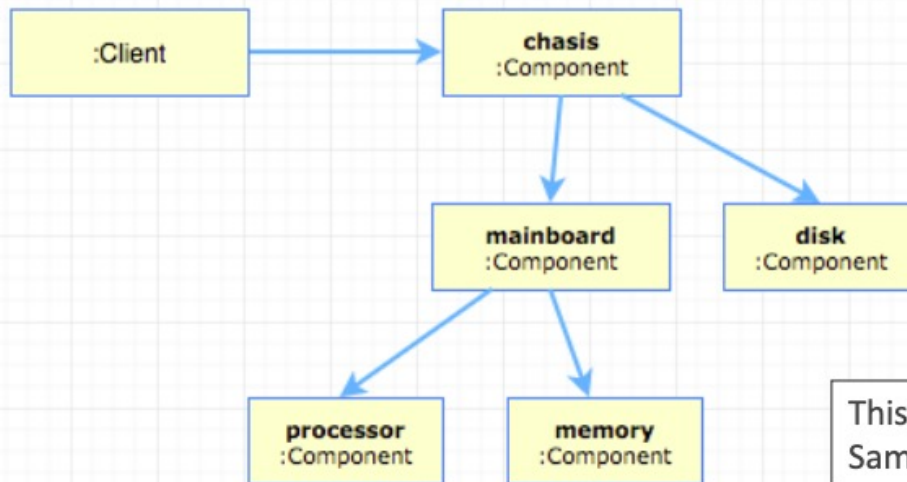
System.out.println("[1] " + mainboard.nameString());
System.out.println("[1] " + mainboard.calculateCost());

```

```

System.out.println("[2] " + chasis.nameString());
System.out.println("[2] " + chasis.calculateCost());

```



This example uses design for [Uniformity](#) (see composite.uniformity).
Sample code also includes design for [Type Safety](#) (see composite.typesafe).

Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week

```
public class MenuTestDrive {  
    public static void main(String args[]) {  
        MenuComponent pancakeHouseMenu =  
            new Menu("PANCAKE HOUSE MENU", "Breakfast");  
        MenuComponent dinerMenu =  
            new Menu("DINER MENU", "Lunch");  
        MenuComponent cafeMenu =  
            new Menu("CAFE MENU", "Dinner");  
        MenuComponent dessertMenu =  
            new Menu("DESSERT MENU", "Dessert of course!");  
        MenuComponent coffeeMenu = new Menu("COFFEE MENU", "S  
  
        MenuComponent allMenus = new Menu("ALL MENUS", "All m  
  
        allMenus.add(pancakeHouseMenu);  
        allMenus.add(dinerMenu);  
        allMenus.add(cafeMenu);  
  
        pancakeHouseMenu.add(new MenuItem(  
            "K&B's Pancake Breakfast",  
            "Pancakes with scrambled eggs, and toast",  
            true,  
            2.99));  
        pancakeHouseMenu.add(new MenuItem(  
            "Regular Pancake Breakfast",  
            "Pancakes with fried eggs, sausage",  
            false,  
            2.99));
```

```
allMenus.print();
```

```
ALL MENUS, All menus combined  
-----  
  
PANCAKE HOUSE MENU, Breakfast  
-----  
K&B's Pancake Breakfast(v), 2.99  
    -- Pancakes with scrambled eggs, and toast  
Regular Pancake Breakfast, 2.99  
    -- Pancakes with fried eggs, sausage  
Blueberry Pancakes(v), 3.49  
    -- Pancakes made with fresh blueberries, and blueberry syrup  
Waffles(v), 3.59  
    -- Waffles, with your choice of blueberries or strawberries  
  
DINER MENU, Lunch  
-----  
Vegetarian BLT(v), 2.99  
    -- (Fakin') Bacon with lettuce & tomato on whole wheat  
BLT, 2.99  
    -- Bacon with lettuce & tomato on whole wheat  
Soup of the day, 3.29  
    -- A bowl of the soup of the day, with a side of potato salad  
Hotdog, 3.05
```

Demos

- ❖ Live Demos ...
- ❖ Make sure you **properly understand** the demo example code available for this week.

Summary

- The Composite Pattern provides a structure to hold both individual objects and composites.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
- There are many design tradeoffs in implementing Composite. You need to balance transparency/uniformity and type safety with your needs.

Creational Patterns

COMP2511, CSE, UNSW



UNSW
SYDNEY

Creational Patterns

Creational patterns provide various **object creation** mechanisms, which increase flexibility and reuse of existing code.

❖ Factory Method

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

❖ Abstract Factory

- let users produce families of related objects without specifying their concrete classes.

❖ Builder

- let users construct complex objects step by step. The pattern allows users to produce different types and representations of an object using the same construction code.

❖ Singleton

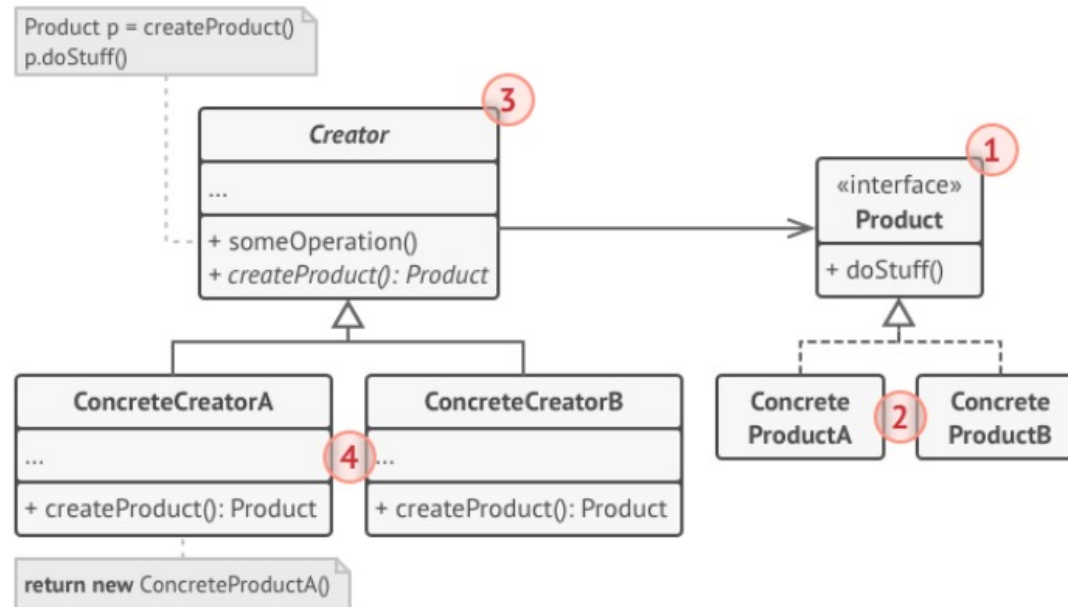
- Let users ensure that a class has only one instance, while providing a global access point to this instance.

Factory Method

Factory Method

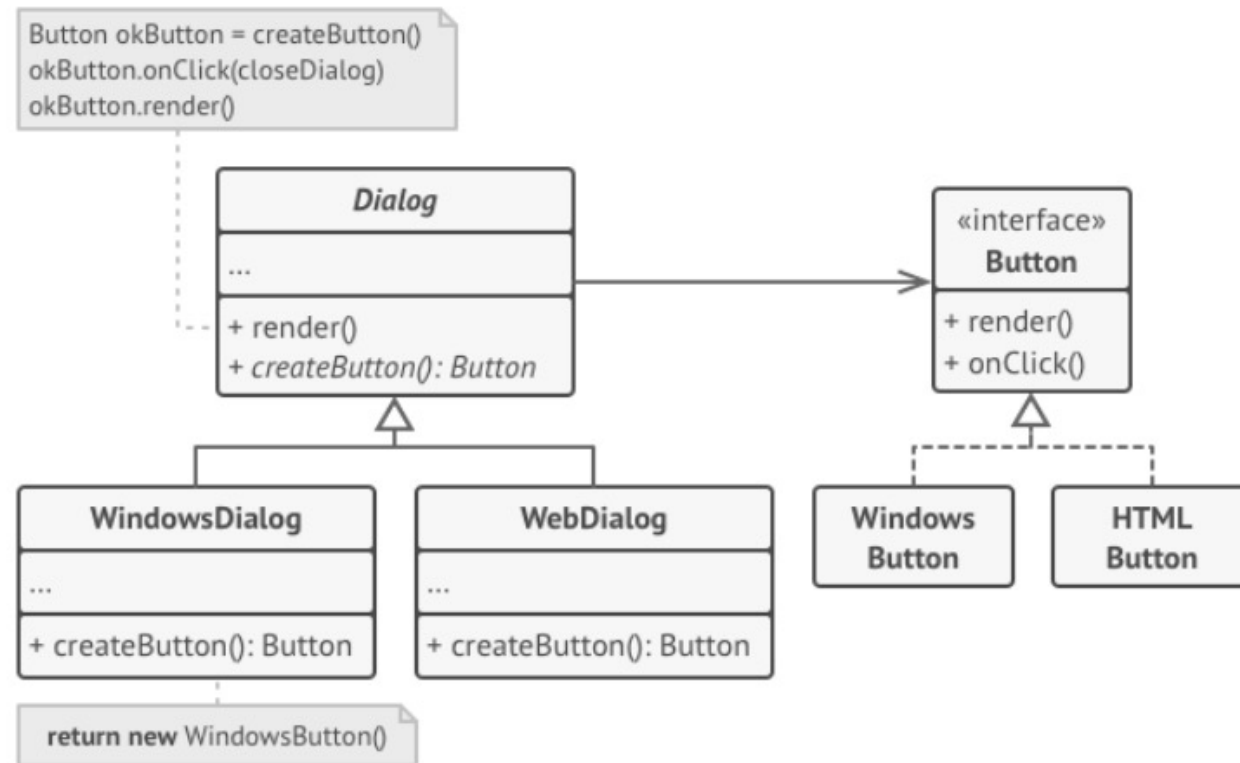
- ❖ **Factory Method** is a creational design pattern that uses factory methods to deal with the problem of creating objects **without** having to **specify the exact class** of the object that will be created.
- ❖ **Problem:**
 - creating an object directly within the class that requires (uses) the object is **inflexible**
 - it **commits** the class to a particular object and
 - makes it **impossible to change** the instantiation independently from (without having to change) the class.
- ❖ **Possible Solution:**
 - Define a **separate** operation (factory **method**) for creating an object.
 - Create an object by calling a **factory method**.
 - This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate).

Factory Method : Structure



1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.
2. **Concrete Products** are different implementations of the product interface.
3. The **Creator** class declares the factory method that returns new product objects.
4. **Concrete Creators** override the base factory method so it returns a different type of product.

Factory Method : Example



Example in Java (MUST read):

<https://refactoring.guru/design-patterns/factory-method/java/example>

Factory Method

For more, read the following:

<https://refactoring.guru/design-patterns/factory-method>

Abstract Factory Pattern

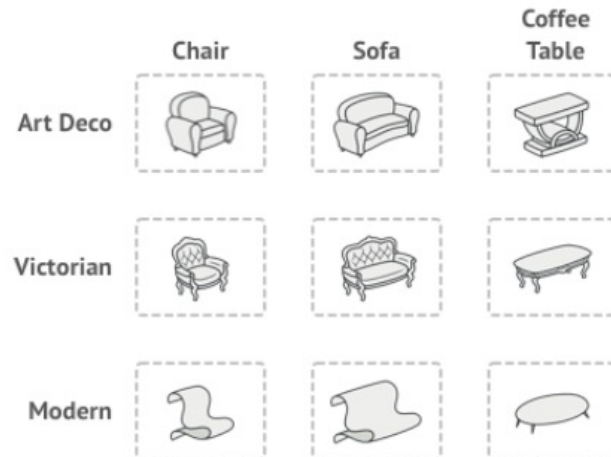
Abstract Factory Pattern

Intent: Abstract Factory is a creational design pattern that lets you produce **families of related objects** without specifying their concrete classes.

Problem:

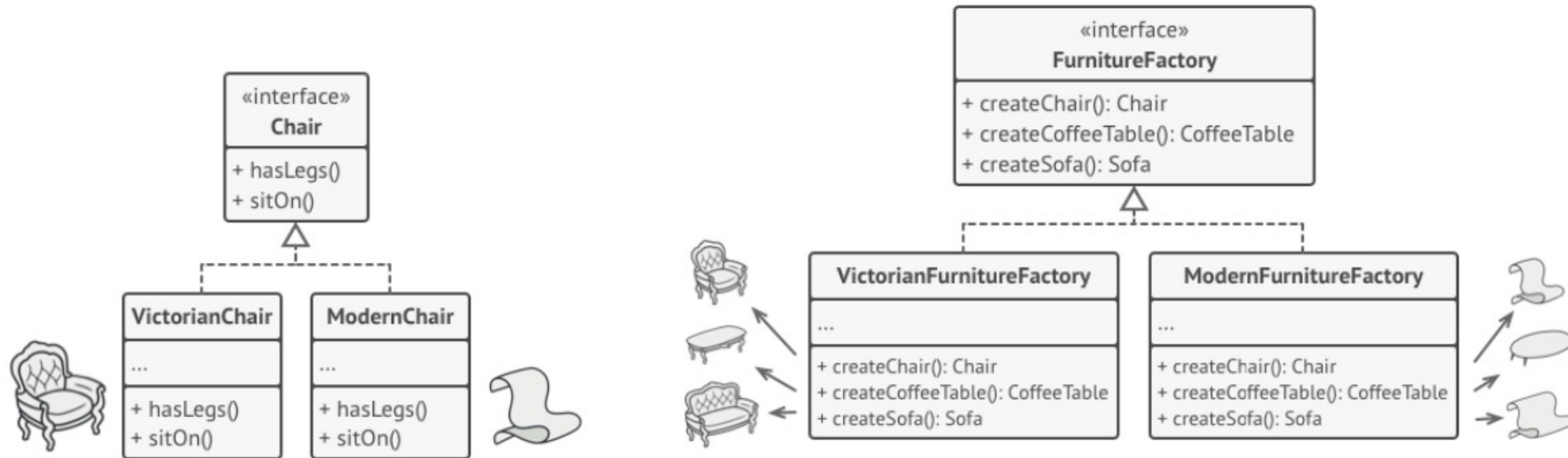
Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

- ❖ A family of related products, say: **Chair + Sofa + CoffeeTable**.
- ❖ Several variants of this family.
- ❖ For example, products **Chair + Sofa + CoffeeTable** are available in these **variants**:

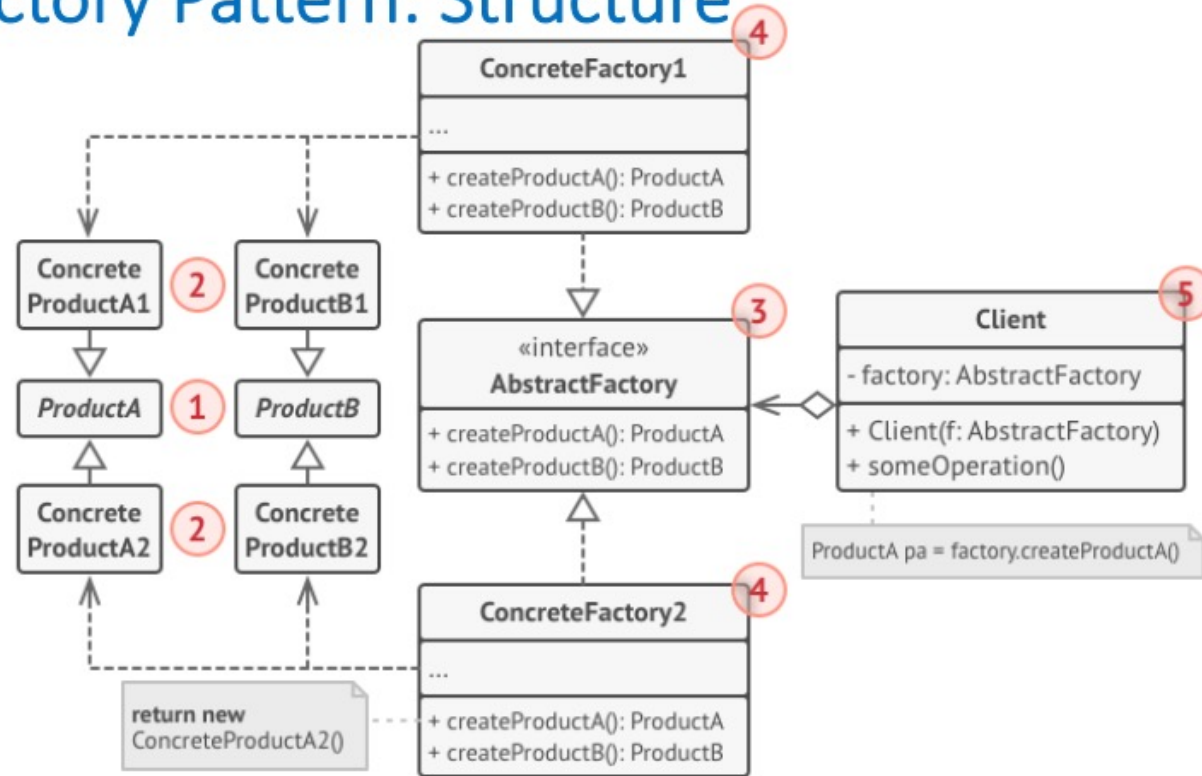


Abstract Factory Pattern:

Possible Solution:

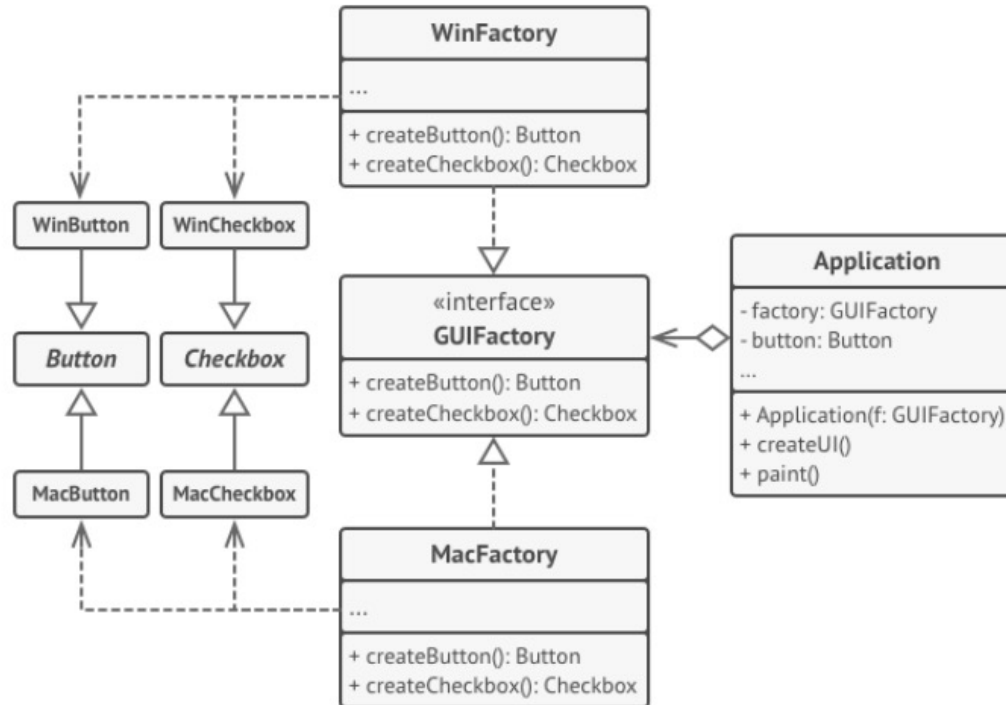


Abstract Factory Pattern: Structure



1. **Abstract Products** declare interfaces for a set of distinct but related products which make up a product family.
2. **Concrete Products** are various implementations of abstract products, grouped by variants. Each abstract product (chair/sofa) must be implemented in all given variants (Victorian/Modern).
3. The **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
4. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.
5. The **Client** can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

Abstract Factory Pattern: Example



Example in **Java (MUST read)**:

<https://refactoring.guru/design-patterns/abstract-factory/java/example>

Abstract Factory Pattern

For more, read the following:

<https://refactoring.guru/design-patterns/abstract-factory>

End

Observer Pattern

Observer Pattern

These lecture notes use material from the reference book “Head First Design Patterns”.

Observer Pattern

- The **Observer Pattern** is used to implement distributed **event handling** systems, in "event driven" programming.
- In the observer pattern
 - an object, called the **subject** (or **observable** or **publisher**) , maintains a list of its dependents, called **observers** (or **subscribers**), and
 - **notifies** the *observers* **automatically** of **any** state **changes** in the **subject**, usually by calling one of their methods.
- Many programming languages support the observer pattern, Graphical User Interface libraries use the observer pattern extensively.

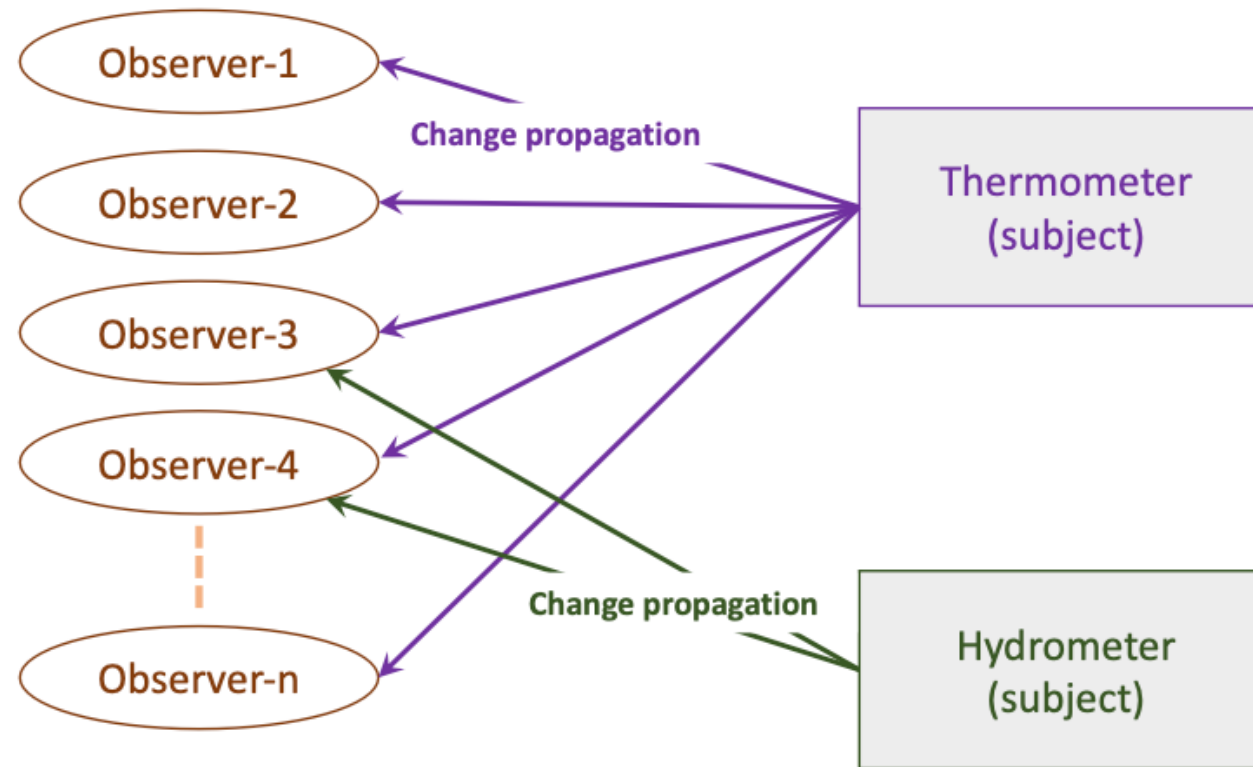
Observer Pattern

- The Observer Pattern defines a **one-to-many** dependency between objects so that when one object (*subject*) changes state, all of its dependents (*observers*) are notified and updated automatically.
- The aim should be to,
 - define a one-to-many dependency between objects **without** making the objects **tightly coupled**.
 - **automatically** notify/update an **open-ended** number of *observers* (dependent objects) when the *subject* changes state
 - be able to **dynamically** add and remove *observers*

Observer Pattern: Possible Solution

- Define *Subject* and *Observer* **interfaces**, such that when a subject changes state, all registered observers are notified and updated automatically.
- The **responsibility of**,
 - a ***subject*** is to maintain a list of observers and to notify them of state changes by calling their **`update()`** operation.
 - ***observers*** is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state when they are notified.
- This makes subject and observers **loosely coupled**.
- Observers can be **added** and **removed** independently **at run-time**.
- This notification-registration interaction is also known as **publish-subscribe**.

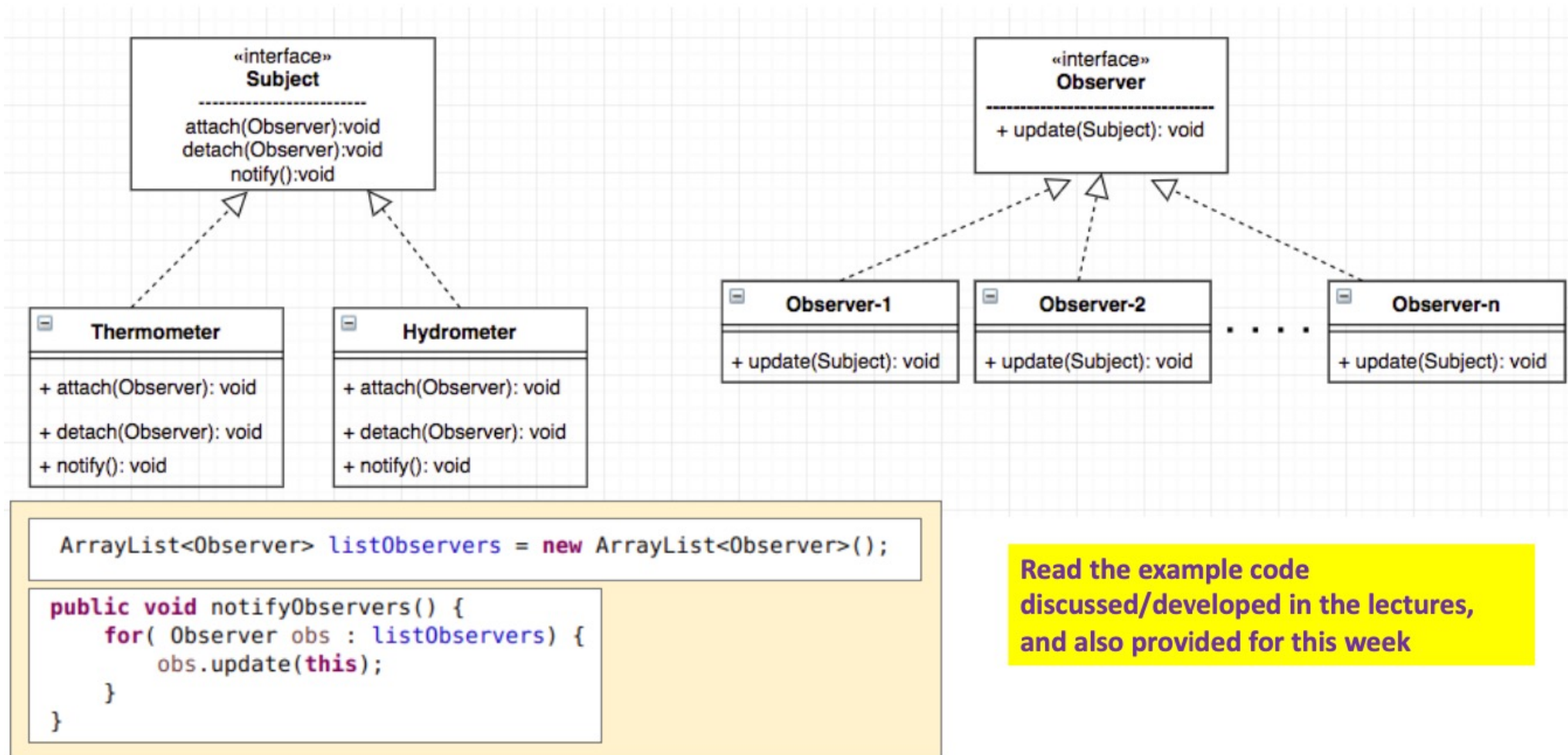
Multiple Observers and Subjects



Observers / Subscribers / Listeners

Observables / Subjects / Publishers

Observer Pattern: Possible Solution



Read the example code discussed/developed in the lectures, and also provided for this week

Passing data: Push or Pull

The *Subject* needs to pass (change) data while notifying a change to an *Observer*. Two possible options,

Push data

- *Subject* passes the changed data to its observers, for example:
`update(data1, data2, ...)`
- All *observers* must implement the above update method.

Pull data

- *Subject* passes reference to itself to its observers, and the observers need to get (pull) the required data from the subject, for example:
`update(this)`
- Subject needs to provide the required access methods for its observers.
For example, `public double getTemperature() ;`

```
public interface Subject {  
  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
  
}
```

Read the example code
discussed/developed in the lectures,
and also provided for this week

```
public class Thermometer implements Subject {  
  
    ArrayList<Observer> listObservers = new ArrayList<Observer>();  
    double temperatureC = 0.0;  
  
    @Override  
    public void registerObserver(Observer o) {  
        if(! listObservers.contains(o)) { listObservers.add(o); }  
    }  
  
    @Override  
    public void removeObserver(Observer o) {  
        listObservers.remove(o);  
    }  
  
    @Override  
    public void notifyObservers() {  
        for( Observer obs : listObservers) {  
            obs.update(this);  
        }  
    }  
  
    public double getTemperatureC() {  
        return temperatureC;  
    }  
  
    public void setTemperatureC(double temperatureC) {  
        this.temperatureC = temperatureC;  
        notifyObservers();  
    }  
  
}
```



**Notify Observers
after every update**


```
public interface Observer {

    public void update(Subject obj);

}
```

Update for
Multiple Subjects



Display after an update



Read the example code
discussed/developed in the lectures,
and also provided for this week

```
public class DisplayUSA implements Observer {
    Subject subject;
    double temperatureC = 0.0;
    double humidity = 0.0;

    @Override
    public void update(Subject obj) {

        if(obj instanceof Thermometer) {
            update( (Thermometer) obj);
        }
        else if(obj instanceof Hygrometer) {
            update((Hygrometer)obj);
        }
    }

    public void update(Thermometer obj) {
        this.temperatureC = obj.getTemperatureC();
        display();
    }

    public void update(Hygrometer obj) {
        this.humidity = obj.getHumidity();
        display();
    }

    public void display() {
        System.out.printf("From DisplayUSA: Temperature is %.2f F, "
            + "Humidity is %.2f\n", convertToF(), humidity);
    }

    public double convertToF() {
        return (temperatureC *(9.0/5.0) + 32);
    }
}
```

Read the example code
discussed/developed in the lectures,
and also provided for this week

```
public class Test1 {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
  
        Thermometer thermo = new Thermometer();  
        Observer usaDisplay = new DisplayUSA();  
        thermo.registerObserver(usaDisplay);  
  
        Observer ausDisplay = new DisplayAustralia();  
        thermo.registerObserver(ausDisplay);  
  
        System.out.println("\n----- thermo.setTemperatureC(30) ----- ");  
        thermo.setTemperatureC(30);  
        System.out.println("\n----- thermo.setTemperatureC(12) ----- ");  
        thermo.setTemperatureC(12);  
  
        Hygrometer hyg = new Hygrometer();  
        hyg.registerObserver(usaDisplay);  
  
        System.out.println("\n----- hyg.setHumidity(77) ----- ");  
        hyg.setHumidity(77);  
        System.out.println("\n----- hyg.setHumidity(96) ----- ");  
        hyg.setHumidity(96);  
        System.out.println("\n----- thermo.setTemperatureC(35) ----- ");  
        thermo.setTemperatureC(35);  
  
        thermo.removeObserver(usaDisplay);  
        System.out.println("\n----- thermo.removeObserver(usaDisplay) ----- ");  
  
        System.out.println("\n----- thermo.setTemperatureC(41) ----- ");  
        thermo.setTemperatureC(41);  
        System.out.println("\n----- ");  
    }  
}
```

add / register

change state

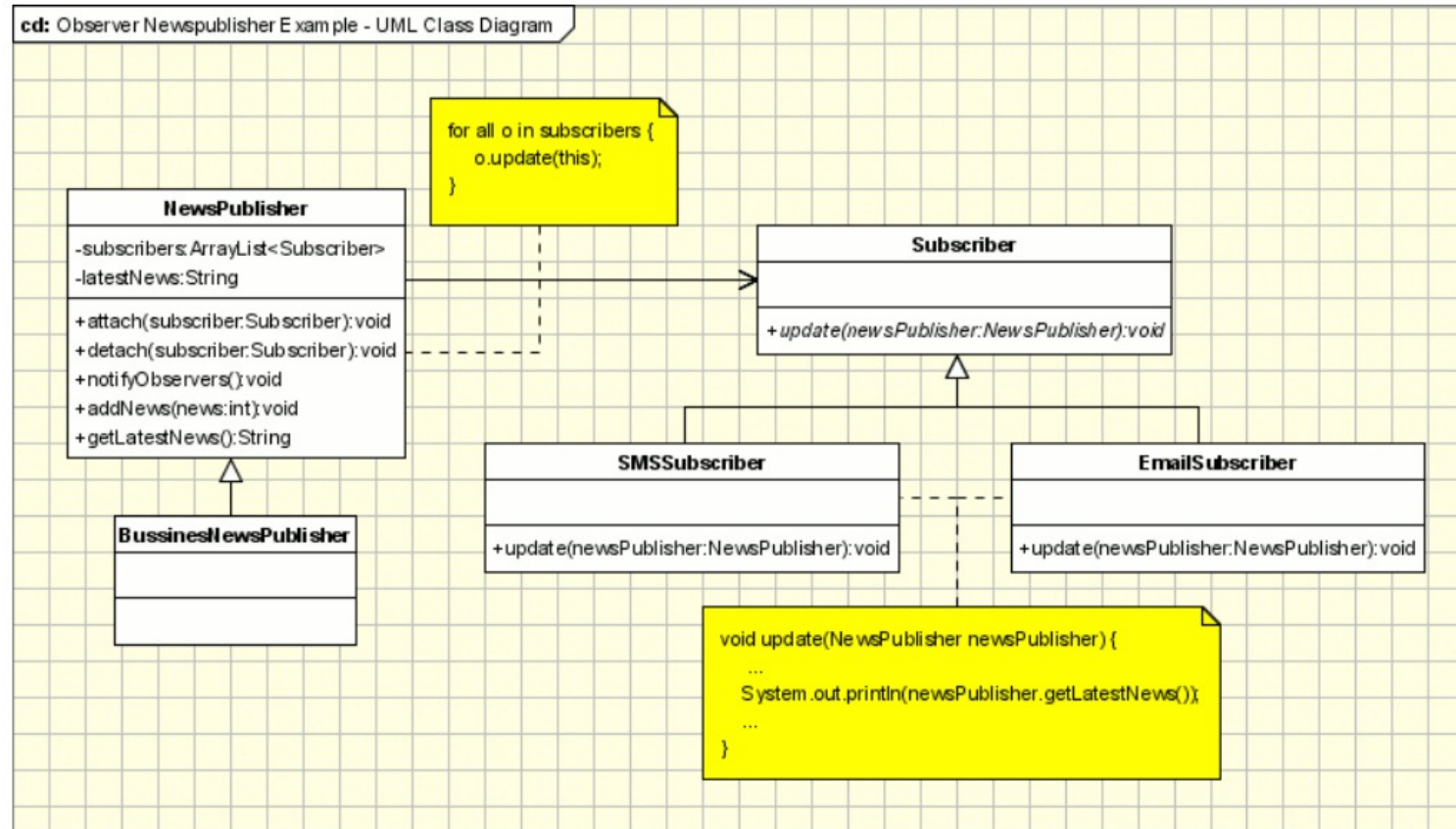
remove

Demos

Live Demos ...

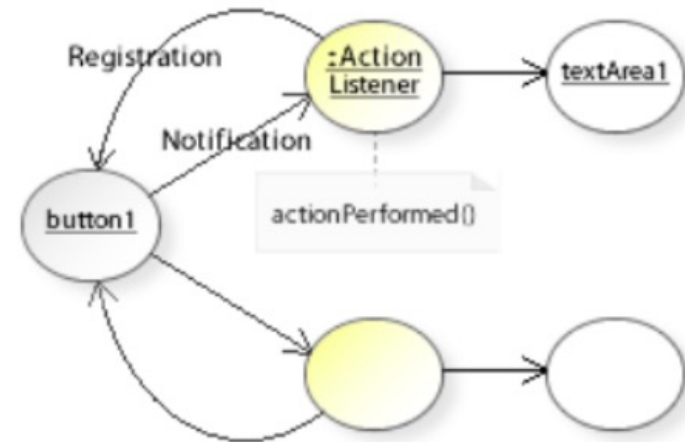
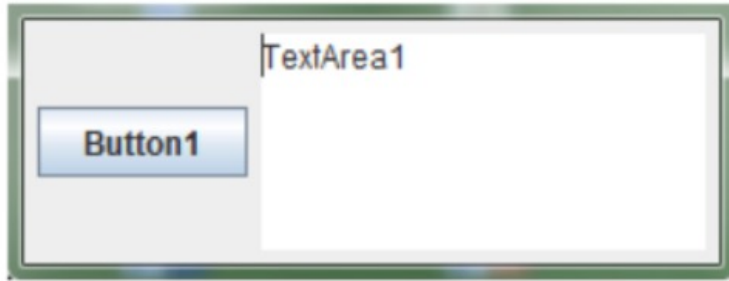
- ❖ Make sure you properly understand the demo example code available for this week.

Observer Pattern: Example



The above image is from <https://www.oodeign.com/observer-pattern.html>

Observer Pattern: UI Example



Summary

Advantages:

- Avoids tight coupling between *Subject* and its *Observers*.
- This allows the *Subject* and its *Observers* to be at different levels of abstractions in a system.
- **Loosely coupled** objects are easier to maintain and reuse.
- Allows **dynamic** registration and deregistration.

Be careful:

- A change in the subject may result in a chain of updates to its observers and in turn their dependent objects – resulting in a **complex update behaviour**.
- Need to properly manage such dependencies.

Summary

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).
- Don’t depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose `java.util.Observable`.
- Watch out for issues with the `java.util.Observable` implementation.
- Don’t be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You’ll also find the pattern in many other places, including JavaBeans and RMI.

From the reference book: “Head First Design Pattern”

Decorator Pattern

COMP2511, CSE, UNSW

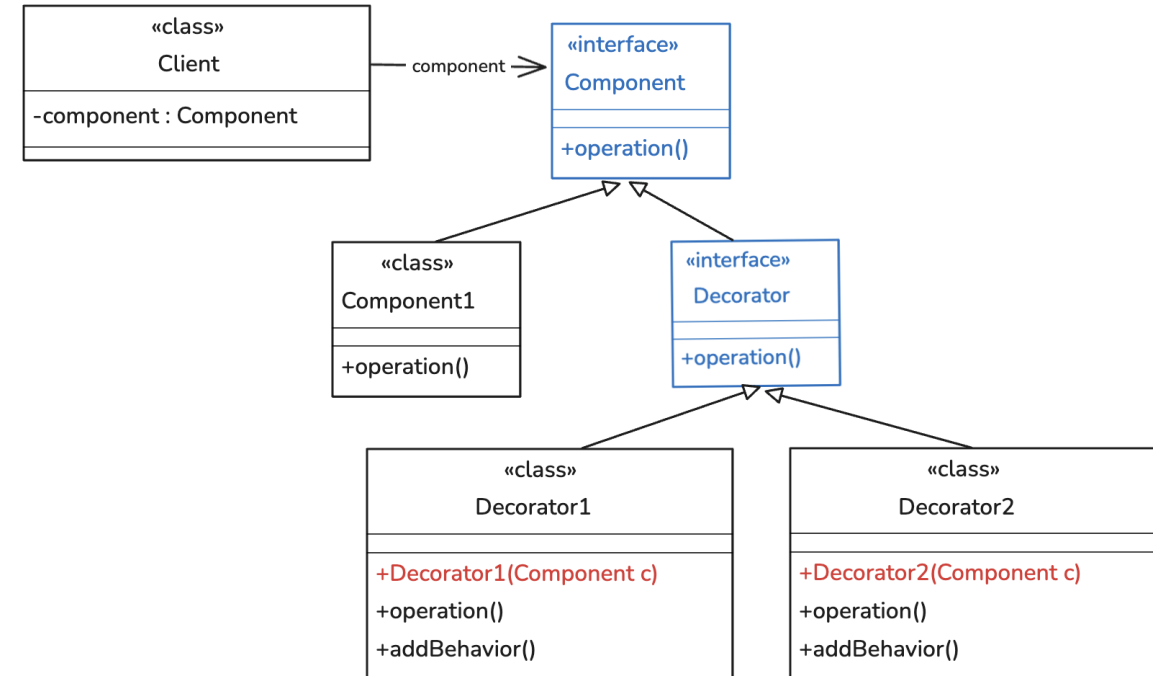


Decorator Pattern: Intent

- "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality." [GoF]
- Decorator design patterns allow us to selectively add functionality to an object (not the class) at runtime, based on the requirements.
- Original class is not changed (Open-Closed Principle).
- Inheritance extends behaviors at compile time, additional functionality is bound to all the instances of that class for their life time.
- The decorator design pattern prefers a composition over an inheritance. Its a structural pattern, which provides a wrapper to the existing class.
- Objects can be decorated multiple times, in different order, due to the recursion involved with this design pattern. See the example in the Demo.
- Do not need to implement all possible functionality in a single (complex) class.

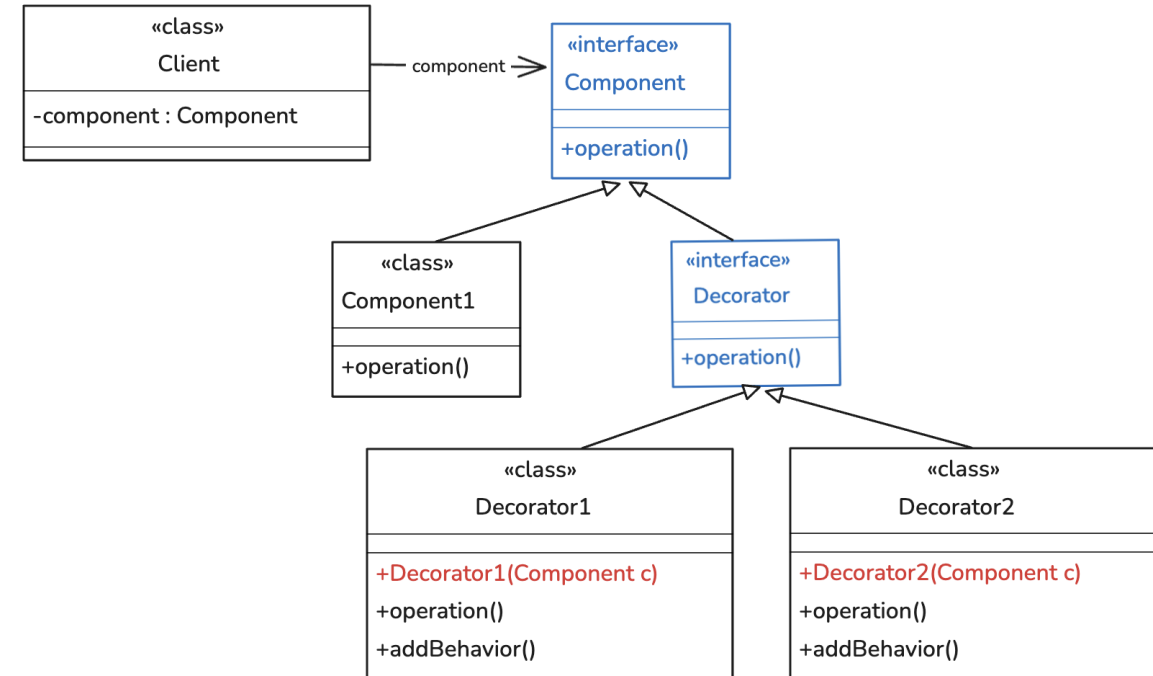
Decorator Pattern: Structure

- ❖ *Client* : refers to the Component interface.
- ❖ *Component*: defines a common interface for *Component1* and *Decorator* objects
- ❖ *Component1* : defines objects that get decorated.
- ❖ *Decorator*: maintains a reference to a *Component* object, and forwards requests to this component object (*component.operation()*)
- ❖ *Decorator1, Decorator2, ...* :
Implement additional functionality (*addBehavior()*) to be performed before and/or after forwarding a request.

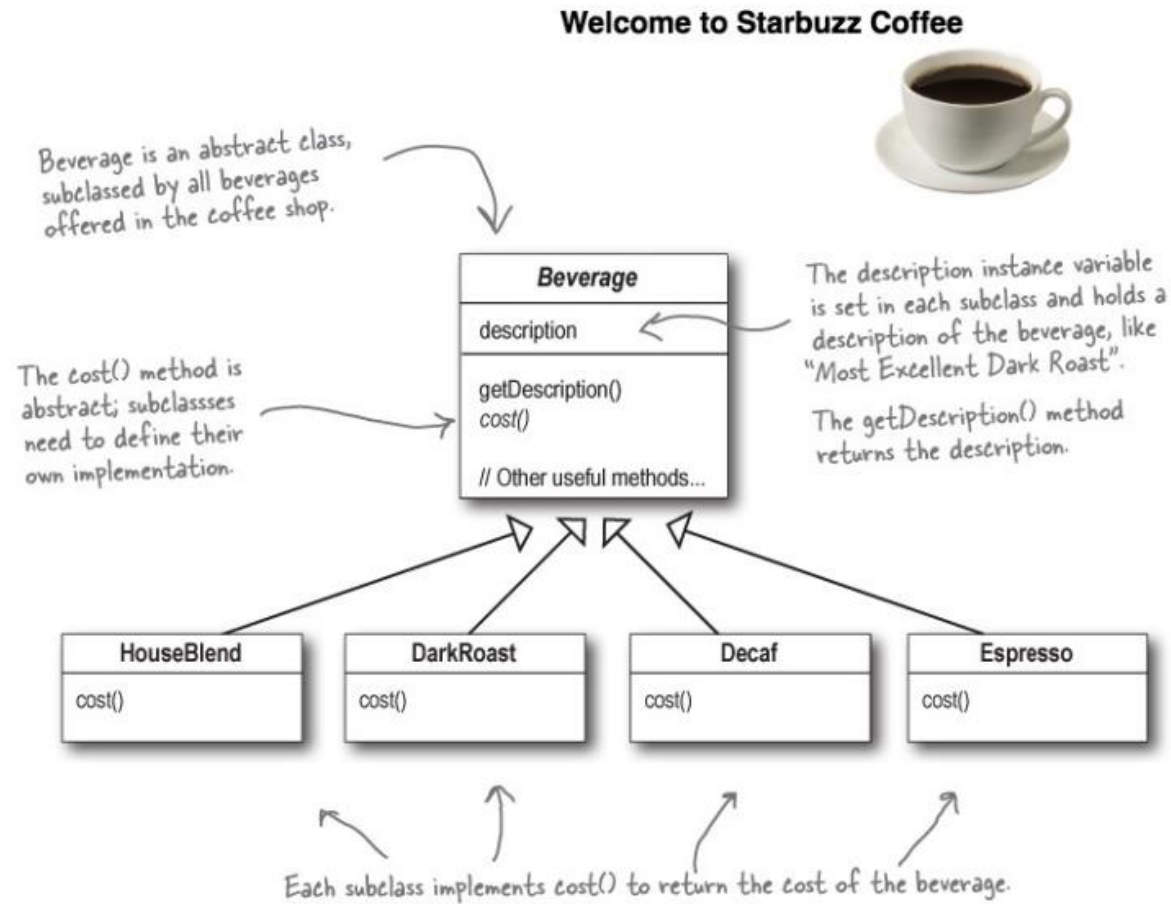


Decorator Pattern: Structure

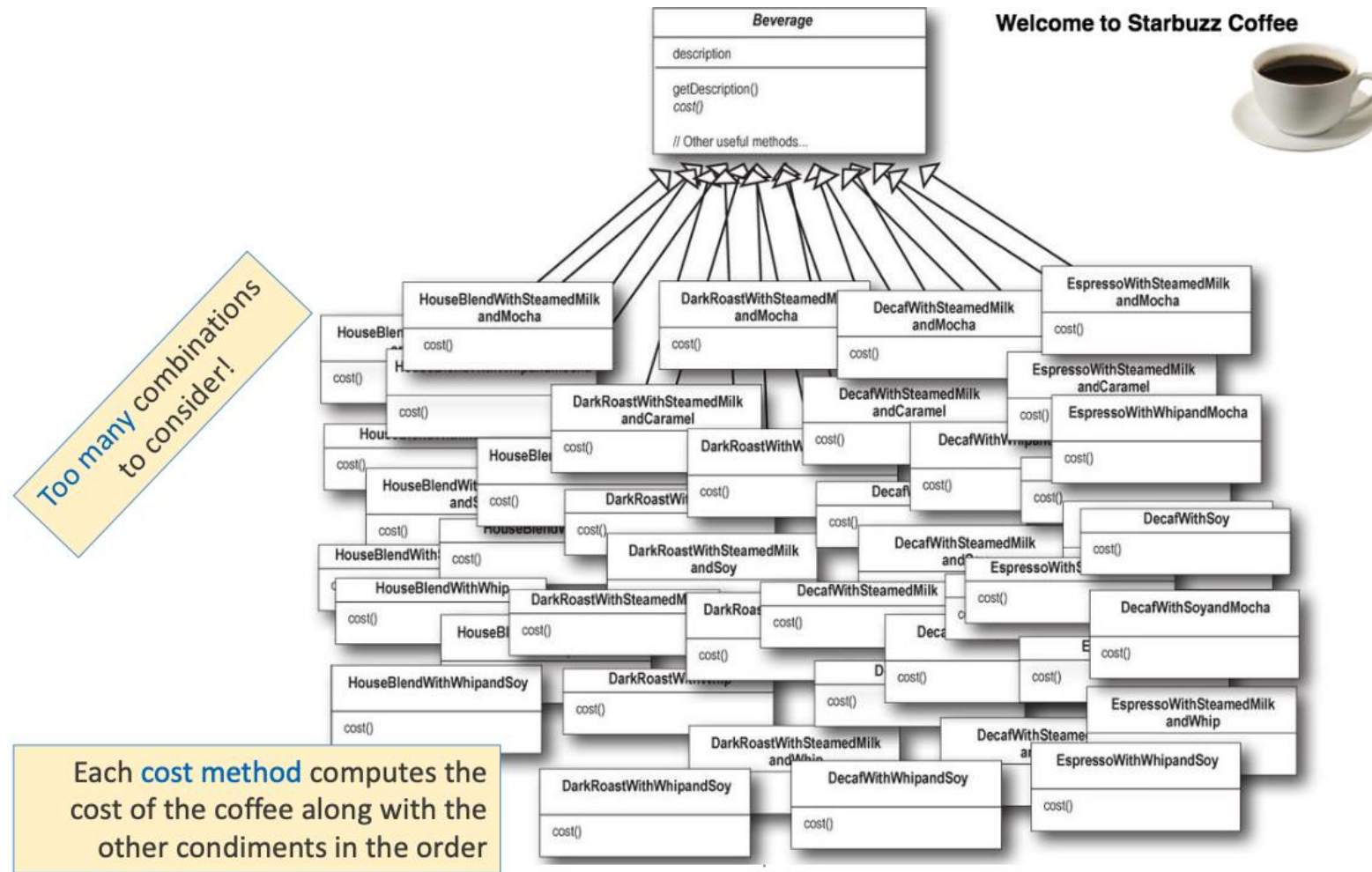
- ❖ Given that the decorator has the same supertype as the object it decorates, we can pass around a **decorated** object **in place** of the **original** (wrapped) object.
- ❖ The **decorator adds its own** behavior either before and/or after delegating to the object it decorates to do the rest of the job.



Decorator Pattern: Example

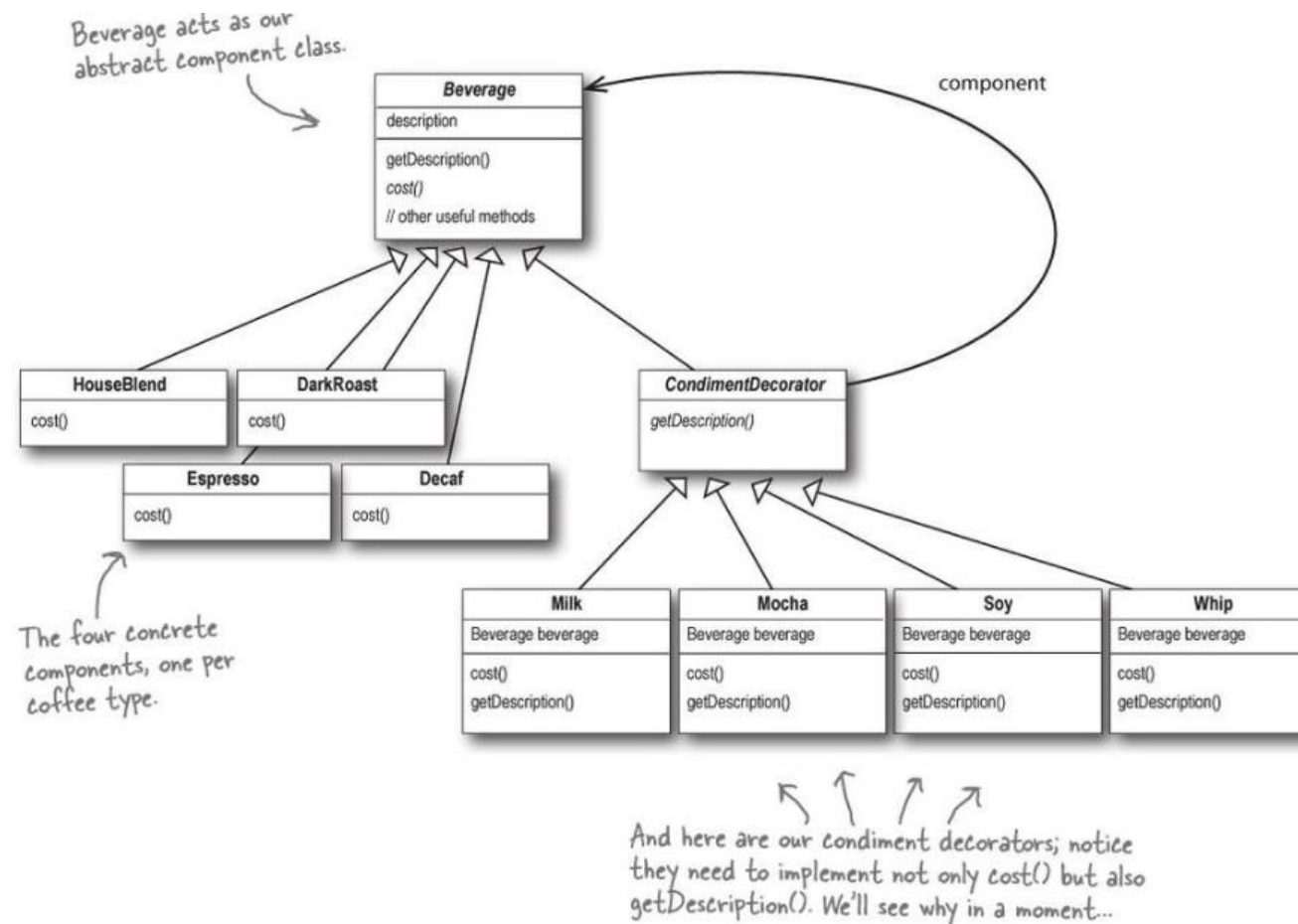


Decorator Pattern: Example



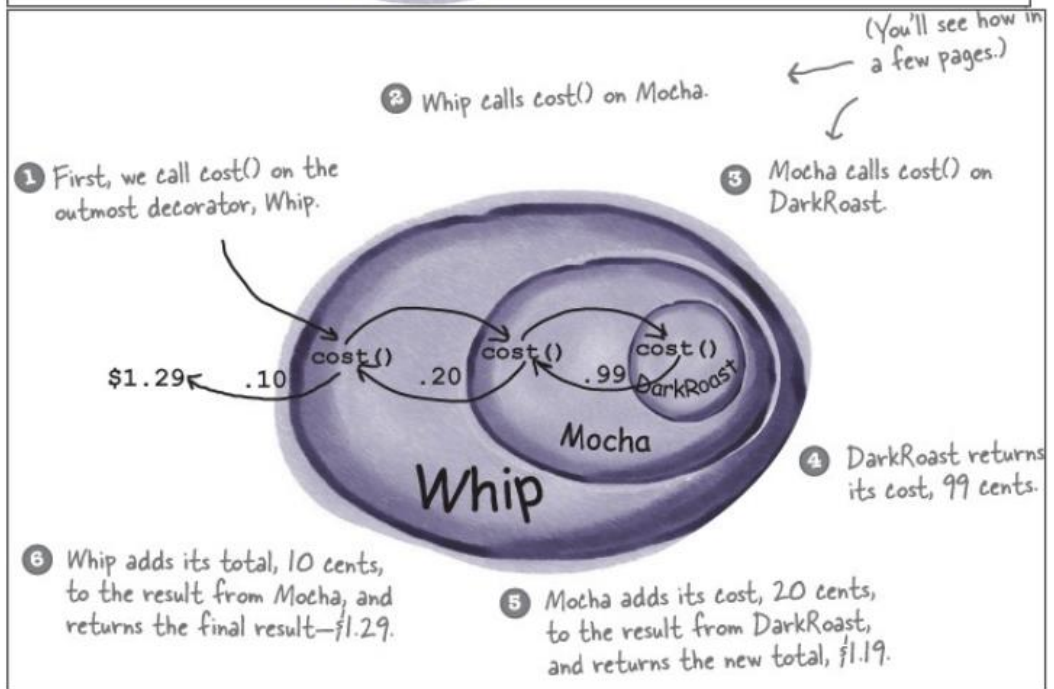
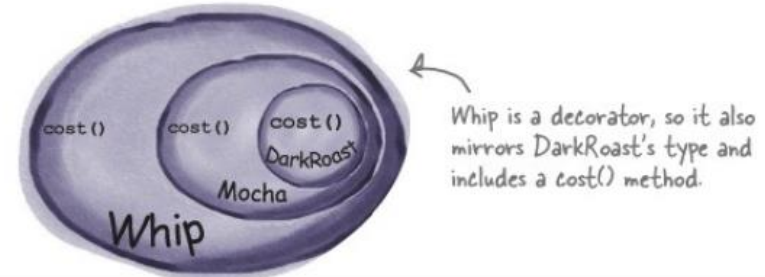
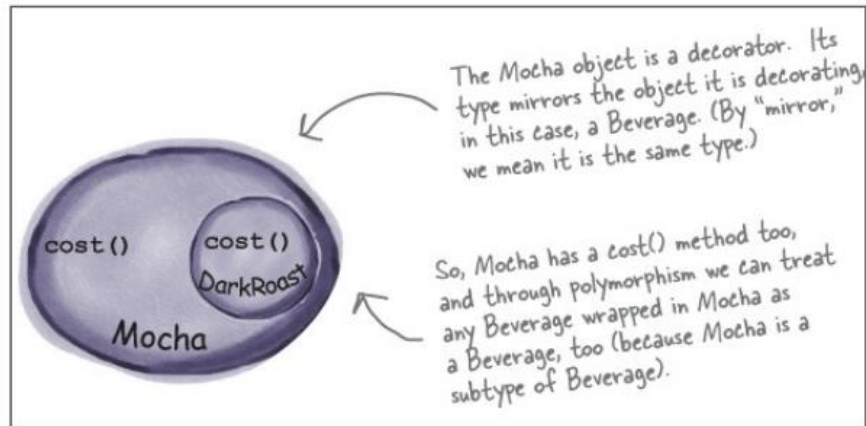
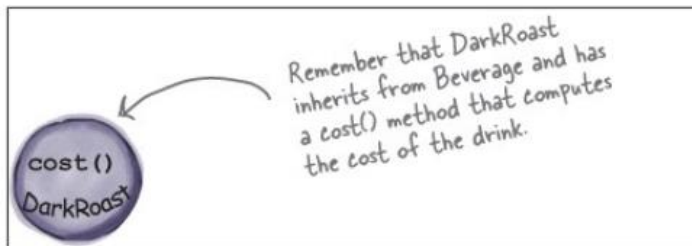
Decorator Pattern: Example

Welcome to Starbuzz Coffee



Decorator Pattern: Example

Constructing a drink order with Decorators



Decorator Pattern: Code

```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
    + " $" + beverage.cost());
System.out.println("-----");
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
    + " $" + beverage2.cost());

System.out.println("----- ");

Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
    + " $" + beverage3.cost());
System.out.println("-----");
```

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println(" - adding One Whip cost of 0.10c ");
    System.out.println(" - new cost is: " + (0.10 + beverage_cost ) );

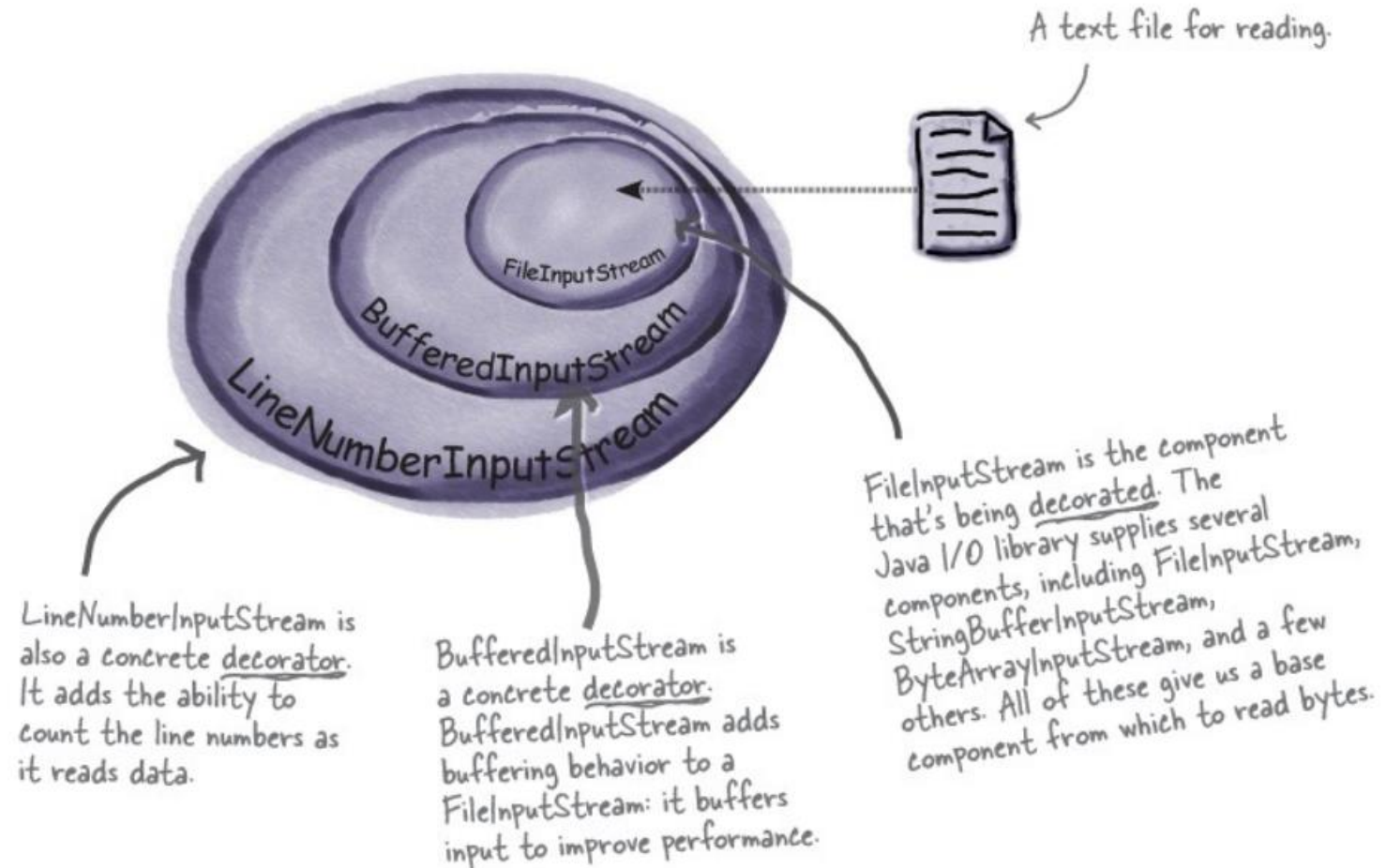
    return 0.10 + beverage_cost ;
}
```

Read the example code
discussed/developed in the lectures,
and also provided for this week

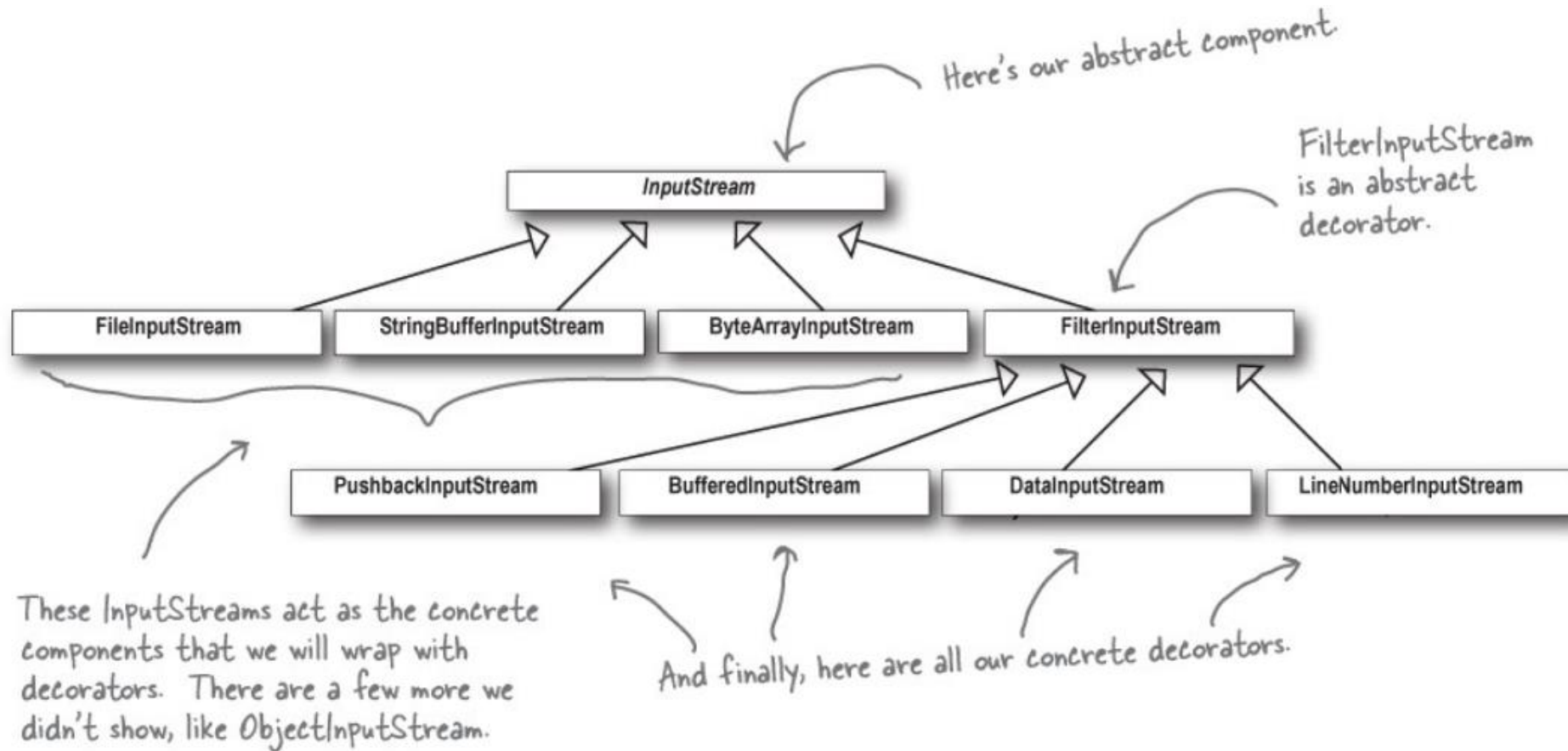
```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
    System.out.println(" - adding One Mocha cost of 0.20c ");
    System.out.println(" - new cost is: " + (0.20 + beverage_cost ) );

    return 0.20 + beverage_cost ;
}
```

Decorator Pattern: Java I/O Example



Decorator Pattern: Java I/O Example



Decorator Pattern: Code

```
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

Read the example code
discussed/developed in the lectures,
and also provided for this week

Decorator Pattern:

..... Demo

End

Functional Paradigm in Java

COMP2511, CSE, UNSW



UNSW
SYDNEY

Java Lambda Expressions

- ❖ Lambda expressions allow us to
 - ❖ easily define **anonymous methods**,
 - ❖ treat **code as data** and
 - ❖ pass **functionality** as method **argument**.
- ❖ An **anonymous** inner **class** with **only one method** can be replaced by a **lambda** expression.
- ❖ Lambda expressions can be used to implement an interface with **only one abstract method**. Such interfaces are called **Functional Interfaces**.
- ❖ Lambda expressions offer **functions as objects** - a feature from functional programming.
- ❖ Lambda expressions are less verbose and offers more flexibility.

Java Lambda Expressions - Syntax

A lambda expression consists of the following:

- ❖ A **comma-separated** list of formal **parameters** enclosed in parentheses. No need to provide data types, they will be inferred. For only one parameter, we can omit the parentheses.
- ❖ The **arrow** token, **->**
- ❖ A **body**, which consists of a single expression or a statement block.

```
public interface MyFunctionInterfaceA {  
    public int myCompute(int x, int y);  
}
```

```
public interface MyFunctionInterfaceB {  
    public boolean myCmp(int x, int y);  
}
```

```
public interface MyFunctionInterfaceC {  
    public double doSomething(int x);  
}
```

```
MyFunctionInterfaceA f1 = (x, y) -> x + y ;  
  
MyFunctionInterfaceA f2 = (x, y) -> x - y + 200;  
  
MyFunctionInterfaceB f3 = (x, y) -> x > y ;  
  
MyFunctionInterfaceC f4 = x -> {  
    double y = 1.5*x;  
    return y + 8.0;  
};  
  
System.out.println( f1.myCompute(10, 20) ); // prints 30  
System.out.println( f2.myCompute(10, 20) ); // prints 190  
System.out.println( f3.myCmp(10, 20) );      // prints false  
System.out.println( f4.doSomething(10) );    // prints 23.0
```

Method References

We can treat an existing method as an instance of a Functional Interface.

There are multiple ways to refer to a method, using `::` operator.

- ❖ A **static** method (`ClassName::methName`)
- ❖ An **instance** method of a particular object (`instanceRef::methName`) or (`ClassName::methName`)
- ❖ A class **constructor** reference (`ClassName::new`)
- ❖ Etc.


Function Interfaces in Java

- ❖ Functional interfaces, in the package `java.util.function`, provide predefined **target types** for **lambda expressions** and **method references**.
- ❖ Each functional interface has a **single abstract method**, called the functional method for that functional interface, to which the **lambda expression's parameter and return types are matched** or adapted.
- ❖ Functional interfaces can provide a target type in **multiple contexts**, such as assignment context, method invocation, etc. For example,

```
Predicate<String> p = String::isEmpty;

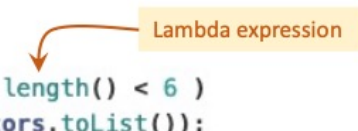
// Collect empty strings
List<String> strEmptyList1 = strList.stream()
    .filter( p )
    .collect(Collectors.toList());

System.out.println("Number of empty strings: " + strEmptyList1.size());
// prints 3
```



```
// Collect strings with length less than six
List<String> strEmptyList2 = strList.stream()
    .filter( e -> e.length() < 6 )
    .collect(Collectors.toList());

System.out.println("Number of strings with length < 6: " + strEmptyList2.size());
// prints 4
```



Function Interfaces in Java

- ❖ There are several basic *function shapes*, including
 - ❖ **Function** (unary function from T to R),
 - ❖ **Consumer** (unary function from T to void),
 - ❖ **Predicate** (unary function from T to boolean), and
 - ❖ **Supplier** (nilary function to R).
- ❖ More information at the package summary page
<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

Function Interfaces in Java: Examples

```
Function<String, Integer> func = x -> x.length();  
Integer answer = func.apply("Sydney");  
System.out.println(answer); // prints 6
```

```
Function<String, Integer> func1 = x -> x.length();  
Function<Integer, Boolean> func2 = x -> x > 5;  
Boolean result = func1.andThen(func2).apply("Sydney");  
System.out.println(result);
```

```
Predicate<Integer> myPass = mark -> mark >= 50 ;  
List<Integer> listMarks = Arrays.asList(45, 50, 89, 65, 10);  
List<Integer> passMarks = listMarks.stream()  
    .filter(myPass)  
    .collect(Collectors.toList());  
  
System.out.println(passMarks); // prints [50, 89, 65]
```

```
Consumer<String> print = x -> System.out.println(x);  
print.accept("Sydney"); // prints Sydney
```

Function Interfaces in Java: Examples

```
// Consumer to multiply 5 to every integer of a list
Consumer<List<Integer> > myModifyList = list -> {
    for (int i = 0; i < list.size(); i++)
        list.set(i, 5 * list.get(i));
};

List<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(1);
list.add(10);

// Implement myModifyList using accept()
myModifyList.accept(list);

// Consumer to display a list of numbers
Consumer<List<Integer>> myDispList = myList -> {
    myList.stream().forEach(e -> System.out.println(e));
};

// Display list using myDispList
myDispList.accept(list);
```

Comparator using Lambda Expression: Example

```
//Using an anonymous inner class  
Comparator<Customer> myCmpAnonymous = new Comparator<Customer>() {  
    @Override  
    public int compare(Customer o1, Customer o2) {  
        return o1.getRewardsPoints() - o2.getRewardsPoints();  
    }  
};  
custA.sort( myCmpAnonymous );
```

Only one line!

```
//Using Lambda expression - simple example (only one line)  
custA.sort((Customer o1, Customer o2)->o1.getRewardsPoints() - o2.getRewardsPoints());
```

```
custA.forEach( (cust) -> System.out.println(cust) );
```

Print using Lambda expression

Comparator using Lambda Expression: Another Example

```
//Using Lambda expression - Another example (with return)  
custA.sort( (Customer o1, Customer o2)-> {  
    if(o1.getPostcode() != o2.getPostcode()) {  
        return o1.getPostcode() - o2.getPostcode() ; }  
    return o1.getRewardsPoints() - o2.getRewardsPoints() ;  
});
```

Parameters – o1 and o2

Body

Pipelines and Streams

- ❖ A **pipeline** is a sequence of **aggregate** operations.
- ❖ The following example prints the male members contained in the collection **roster** with a **pipeline** that consists of the **aggregate** operations **filter** and **forEach**:

```
roster
    .stream()
    .filter( e -> e.getGender() == Person.Sex.MALE )
    .forEach( e -> System.out.println(e.getName()) );
```

Using pipeline and aggregate ops:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

Traditional approach,
using a for-each loop:

- ❖ Please note that, in a pipeline, **operations are loosely coupled**, they only rely on their incoming streams and can be easily rearranged/replaced by other suitable operations.
- ❖ Just to clarify, the **“.” (dot) operator** in the above syntax has a very different meaning to the **“.” (dot) operator** used with an **instance** or a **class**.

Pipelines and Streams

- ❖ A **pipeline** contains the following components:
 - A **source**: This could be a collection, an array, a generator function, or an I/O channel. Such as *roster* in the example.
 - Zero or **more intermediate operations**. An intermediate operation, such as **filter**, produces a new stream.
- ❖ A **stream** is a sequence of elements. The method **stream** creates a stream from a collection (*roster*).
- ❖ The **filter** operation returns a new stream that contains elements that **match** its **predicate**. The **filter** operation in the example returns a stream that contains all male members in the collection *roster*.
- ❖ A **terminal** operation. A terminal operation, such as **forEach**, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of **forEach**, no value at all.

```
roster
    .stream()
    .filter( e -> e.getGender() == Person.Sex.MALE )
    .forEach( e -> System.out.println(e.getName()) );
```

Pipelines and Streams: Example

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- ❖ The above example calculates the average age of all male members contained in the collection **roster** with a pipeline that consists of the aggregate operations **filter**, **mapToInt**, and **average**.
- ❖ The **mapToInt** operation returns a new stream of type **IntStream** (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream.
- ❖ As expected, the **average** operation calculates the average value of the elements contained in a stream of type **IntStream**.
- ❖ There are many **terminal operations** such as **average** that return one value by combining the contents of a stream. These operations are called **reduction operations**; see the section Reduction for more information at <https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

Pipelines and Streams: Another Example

```
double avgNonEmptyStrLen = strList.stream()  
    .filter( e -> e.length() > 0 )  
    .mapToInt(String::length)  
    .average()  
    .getAsDouble();
```

End

Singleton Pattern and Asynchronous Design

COMP2511, CSE, UNSW

Creational Pattern: Singleton Pattern

Creational patterns provide various **object creation** mechanisms, which increase flexibility and reuse of existing code.

❖ Factory Method

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

❖ Abstract Factory

- let users produce families of related objects without specifying their concrete classes.

❖ Singleton

- Let users ensure that a class has only one instance, while providing a global access point to this instance.

Singleton Pattern

Intent: **Singleton** is a creational design pattern that lets you ensure that a class has **only one instance**, while providing a global access point to this instance.

Problem: A client wants to,

- ❖ ensure that a class has just a **single instance**, and
- ❖ provide a **global** access point to that instance

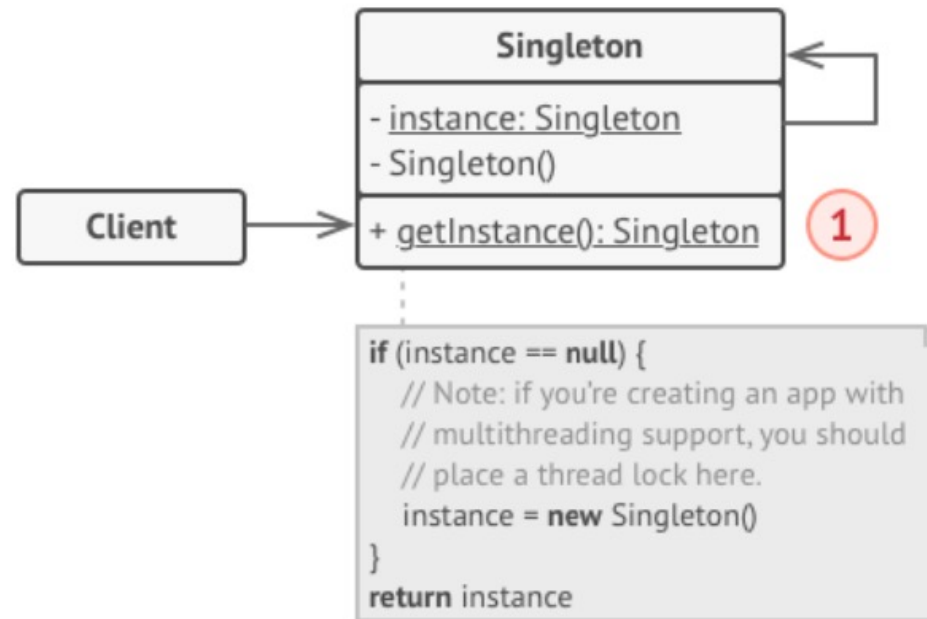
Solution:

All implementations of the Singleton have these two steps in common:

- ❖ Make the **default constructor private**, to prevent other objects from using the new operator with the Singleton class.
- ❖ Create a **static creation method** that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the **cached object**.
- ❖ If your code has access to the Singleton class, then it's able to **call** the **Singleton's static method**.
- ❖ Whenever Singleton's static method is called, the **same object** is always returned.

Singleton: Structure

- ❖ The **Singleton** class declares the **static** method ***getInstance*** (1) that returns the same instance of its own class.
- ❖ The Singleton's constructor should be hidden from the client code.
- ❖ Calling the ***getInstance*** (1) method should be the only way of getting the Singleton object.



Singleton: How to Implement

- ❖ Add a **private static field** to the class for storing the singleton instance.
- ❖ Declare a **public static creation method** for getting the singleton instance.
- ❖ Implement “lazy initialization” inside the static method.
 - It should create a **new object** on its first call and put it into the static field.
 - The method should always return that instance on all **subsequent calls**.
- ❖ Make the **constructor of the class private**.
 - The static method of the class will still be able to call the constructor, but not the other objects.
- ❖ **In a client**, call singleton’s static creation method to access the object.

For more information, read:

<https://refactoring.guru/design-patterns/singleton/java/example>

Synchronous vs Asynchronous Software Design



What is Synchronous programming?

- In *synchronous* programming, operations are carried out **in order**.
- The execution of an operation is **dependent upon** the completion of the **preceding** operation.
- Tasks (functions) A, B, and C are executed in a **sequence**, often using one thread.



What is Asynchronous programming?

- In *asynchronous programming*, operations are carried out **independently**.
- The execution of an operation is **not dependent upon** the completion of the **preceding** operation.
- Tasks (functions) A, B, and C are executed **independently**, can use multiple threads/resources.



Example: Synchronous vs Asynchronous programming

Synchronous

```
function getRecord(key) {  
    establish database connection  
    retrieve the record for key  
    return record;  
}
```

```
function display(rec){  
    display rec on the web page  
}
```

```
rec = getRecord('Rita');  
display(rec)
```

```
rec = getRecord('John');  
display(rec)
```

A

B

Asynchronous

```
function getRecord(key, callback) {  
    establish database connection  
    retrieve the record for key  
    callback(record);  
}
```

```
function display(rec){  
    display rec on the web page  
}
```

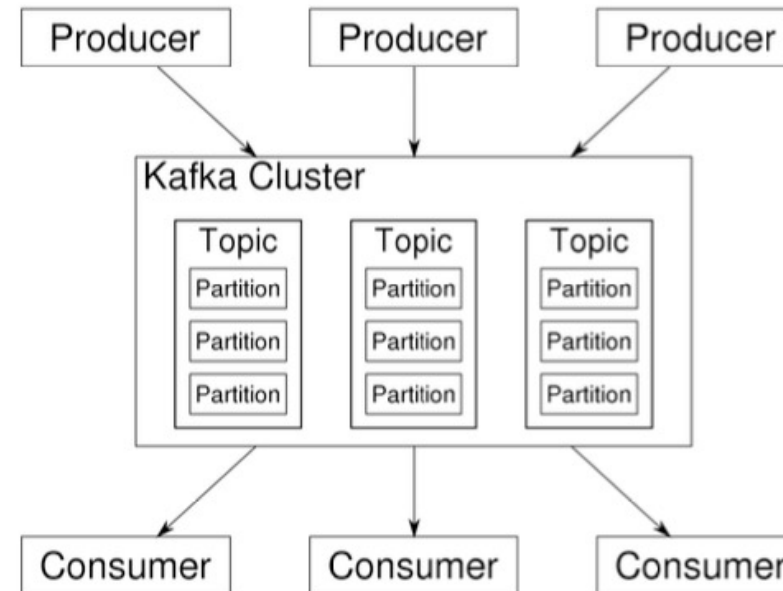
```
getRecord('Rita', display)  
getRecord('John', display)
```

A

B

Kafka: An Example of Asynchronous Software Design

- ❖ Today, streams of data records, including **streams of events**, are continuously generated by many online applications.
 - ❖ A **streaming platform** enables the development of applications that can continuously and easily consume and process streams of data and events.
 - ❖ Apache **Kafka** (Kafka) is a free and open-source distributed **streaming platform** useful for building, *real time* or *asynchronous*, **event-driven applications**.
 - ❖ Kafka offers **loose coupling** between *producers* and *consumers*.
 - ❖ Consumers have the option to either **consume** an event **in real time** or *asynchronously at a later time*.
 - ❖ Kafka maintains the **chronological order** of records/events, ensuring fault tolerance and durability.
 - ❖ To increase **scalability**, Kafka separates a topic and stores each **partition** on a different node.
- ❖ **Producer API** – Permits an application to **publish** streams of records/events.
 - ❖ **Consumer API** – Permits an application to **subscribe** to topics and processes streams of records/events.



END

Software Architecture

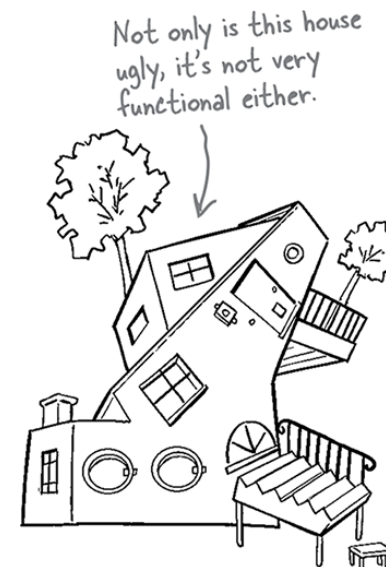
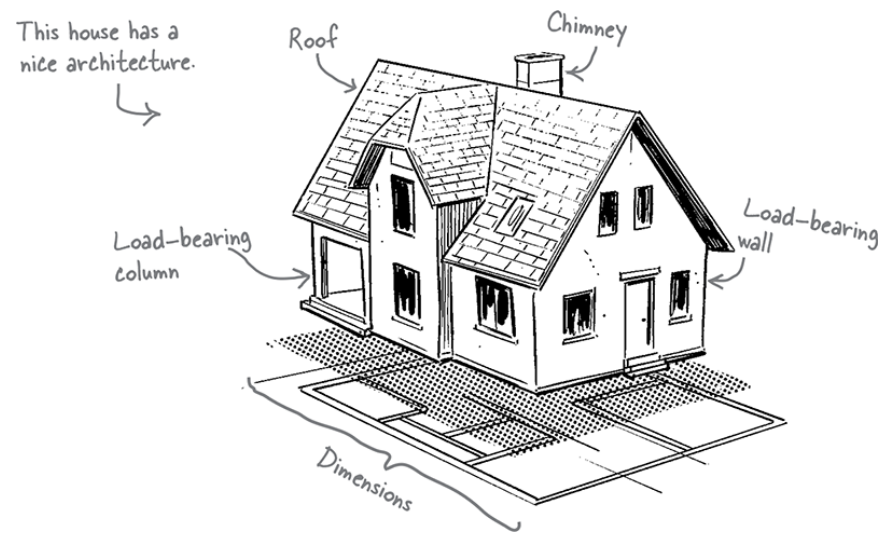
COMP2511, CSE, UNSW



These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

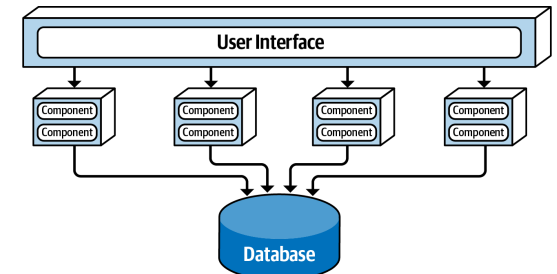
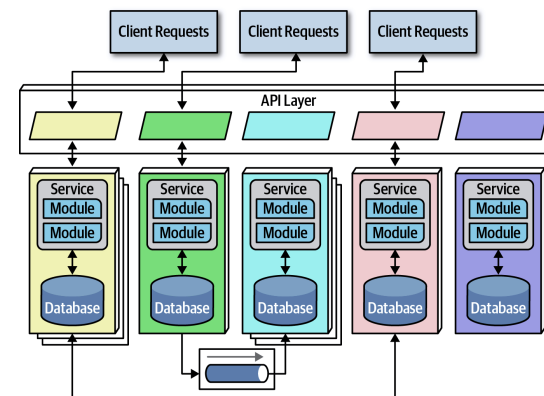
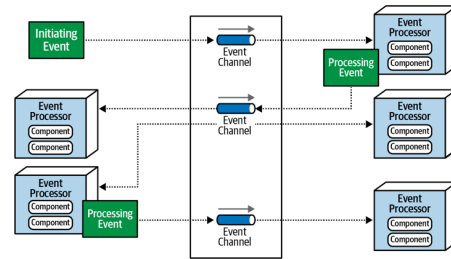
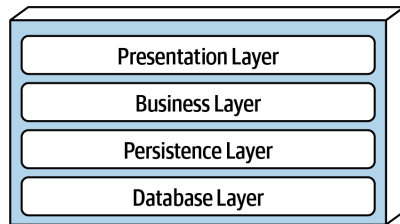
Software Architecture as a Metaphor

- ❖ While building a house, **architectural decisions** (rooms, floors, layout) are crucial and **costly to change** later.
- ❖ A **poorly architectural** house can lead to substandard and **uncomfortable** living conditions.



What is Software Architecture?

- ❖ Software architecture defines the **fundamental structure** of a software system.
- ❖ Influences how effectively the software can **adapt to changes**, scale, perform, and maintain its reliability.
- ❖ **Software Architecture** diagrams represent relationships between components (e.g. databases, services, interfaces).

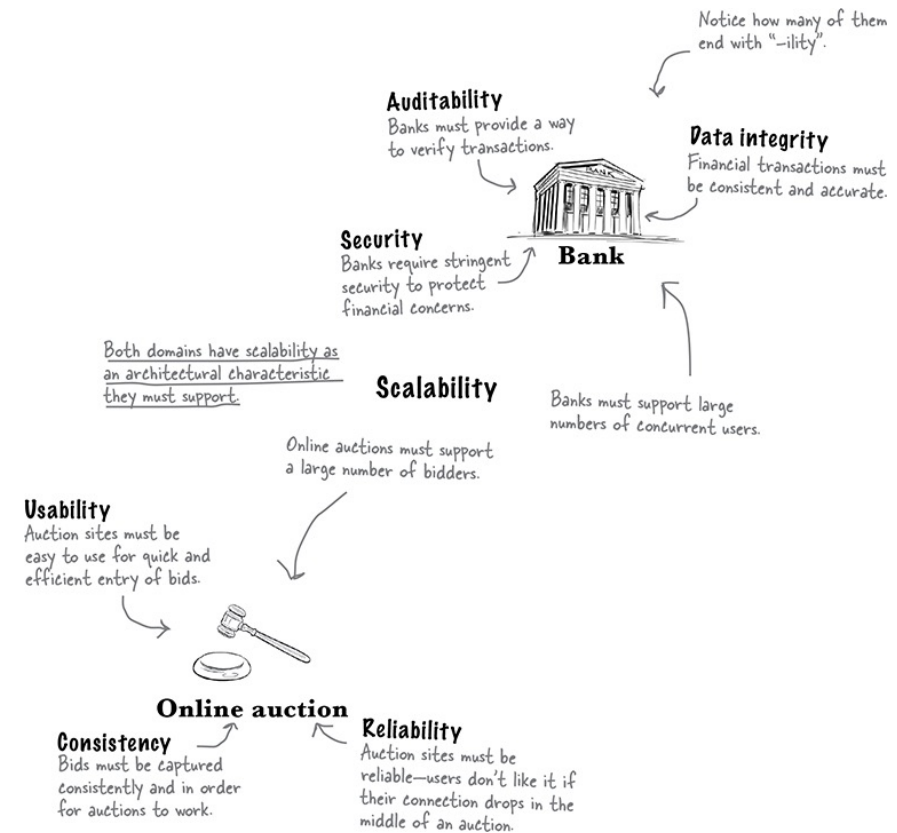
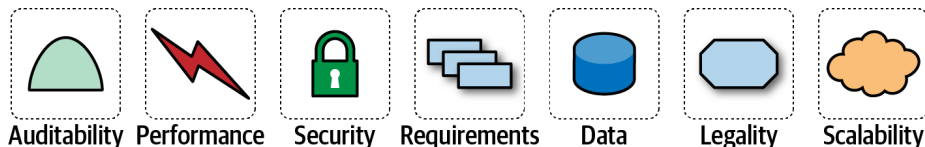


The Four Dimensions of Software Architecture

1. Architectural Characteristics
2. Architectural Decisions
3. Logical Components
4. Architectural Style

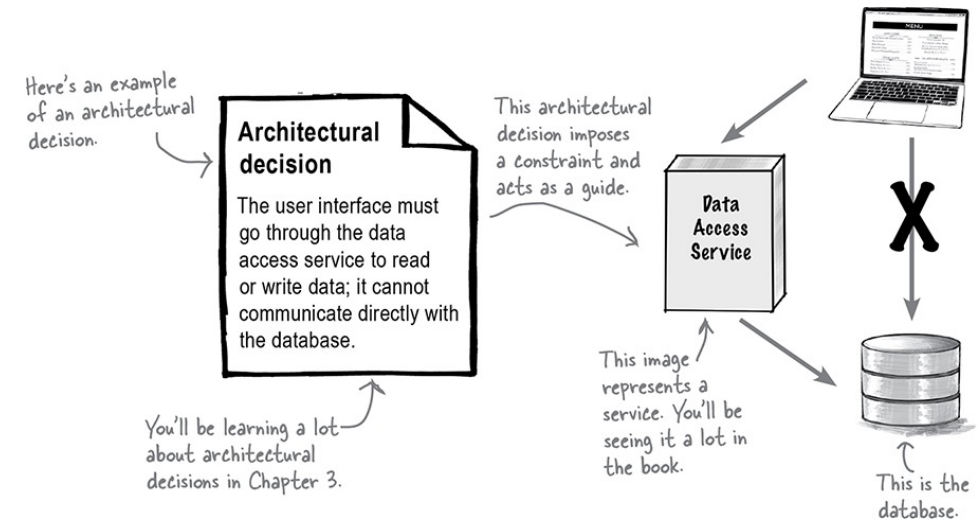
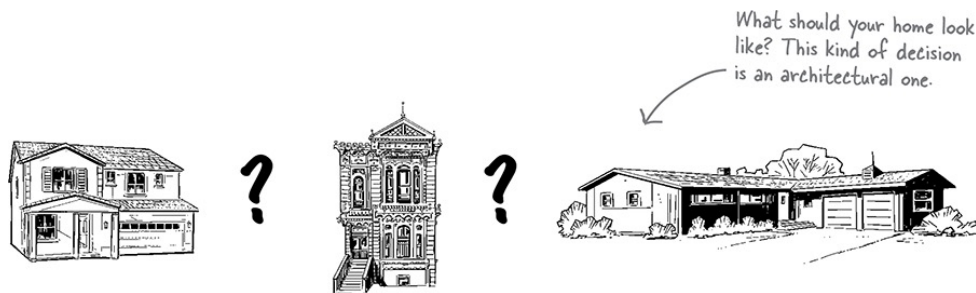
Dimension 1: Architectural Characteristics

- ❖ Architectural Characteristics define **fundamental qualities** software architecture must support.
- ❖ Commonly used Architectural Characteristics:
 - Scalability (support growth)
 - Reliability (consistent operation)
 - Availability (system uptime)
 - Testability (ease of testing components)
 - Security



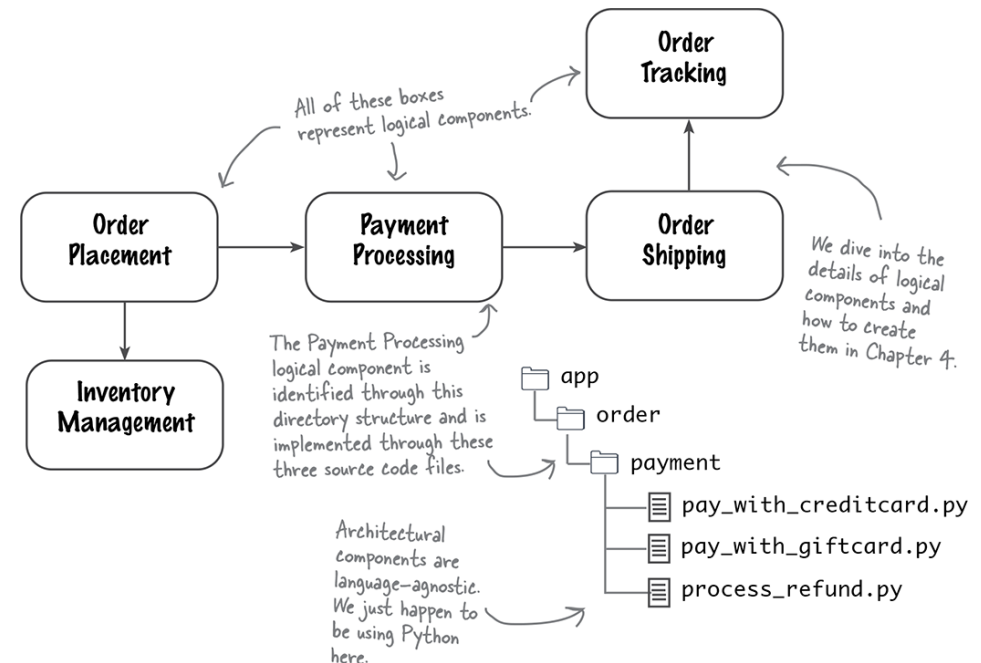
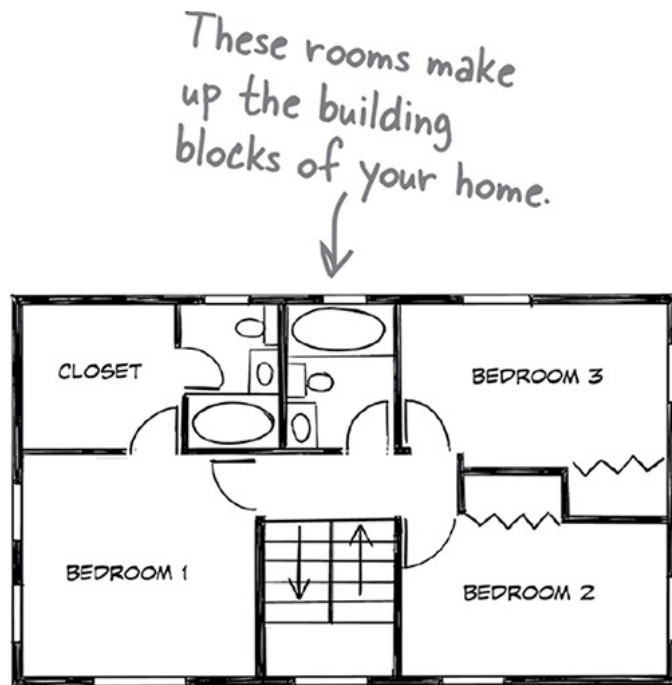
Dimension 2: Architectural Decisions

- ❖ Long-term **structural decisions** influencing software behaviour.
- ❖ Architectural Decisions **set constraints** guiding future development.



Dimension 3: Logical Components

- ❖ Functional **building blocks** representing business features.

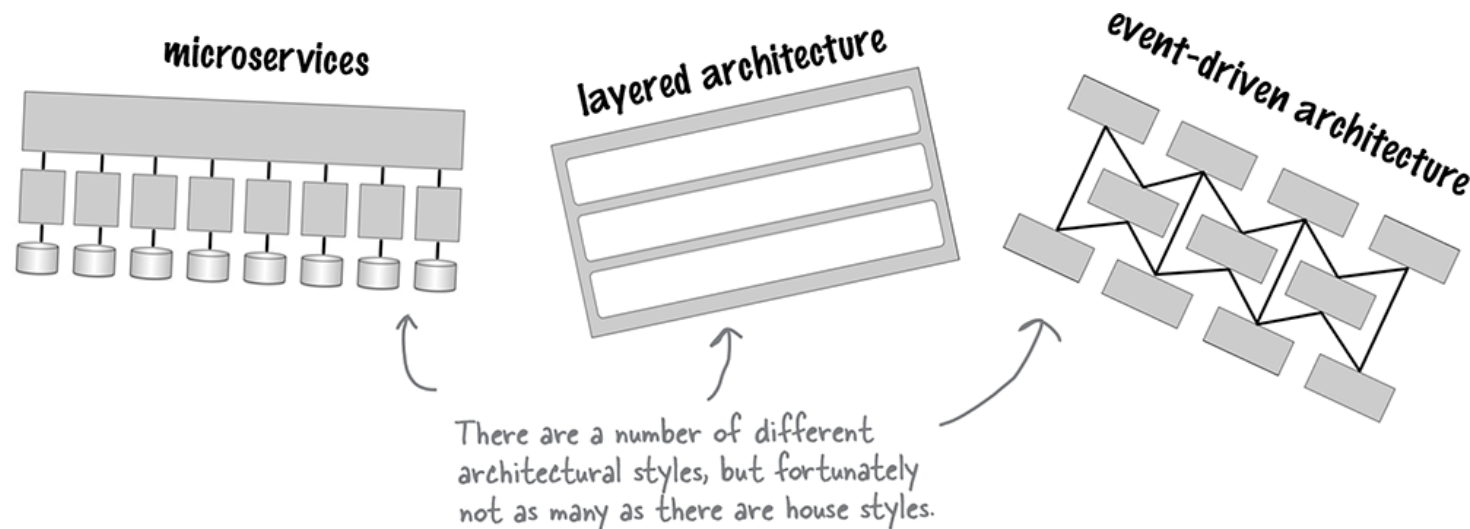


Dimension 4: Architectural Styles

- ❖ Overall **system** shape and **structural patterns**.
- ❖ Common styles:
 - Layered (clear separation of concerns)
 - Microservices (highly scalable and agile)
 - Event-driven (responsive and scalable)

- ❖ **Real-world** Examples:

- Netflix adopting microservices.
- Traditional enterprise apps using layered architecture.

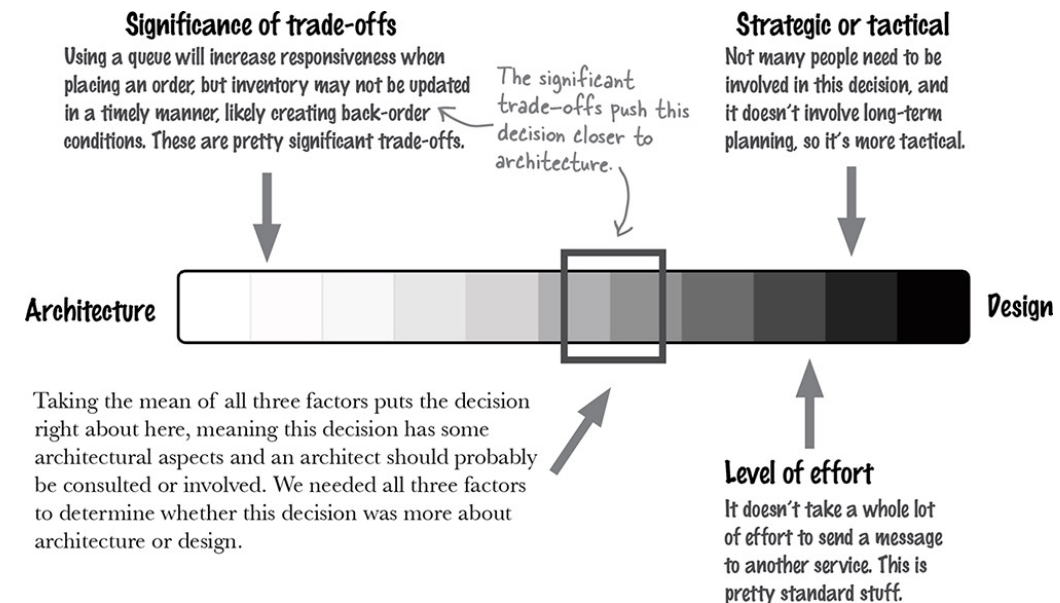


Architecture vs. Design

- ❖ **Architecture**: Structural decisions (hard to change).
- ❖ **Design**: Appearance and detailed decisions (easy to change).
- ❖ Decisions exist on a **spectrum** from pure architecture to pure design.
- ❖ **Strategic decisions** (architecture): Long-term, high impact, high effort.
- ❖ **Tactical decisions** (design): Short-term, low impact, low effort.

Example:

- ❖ Choosing databases (architecture) vs. UI button colour (design).

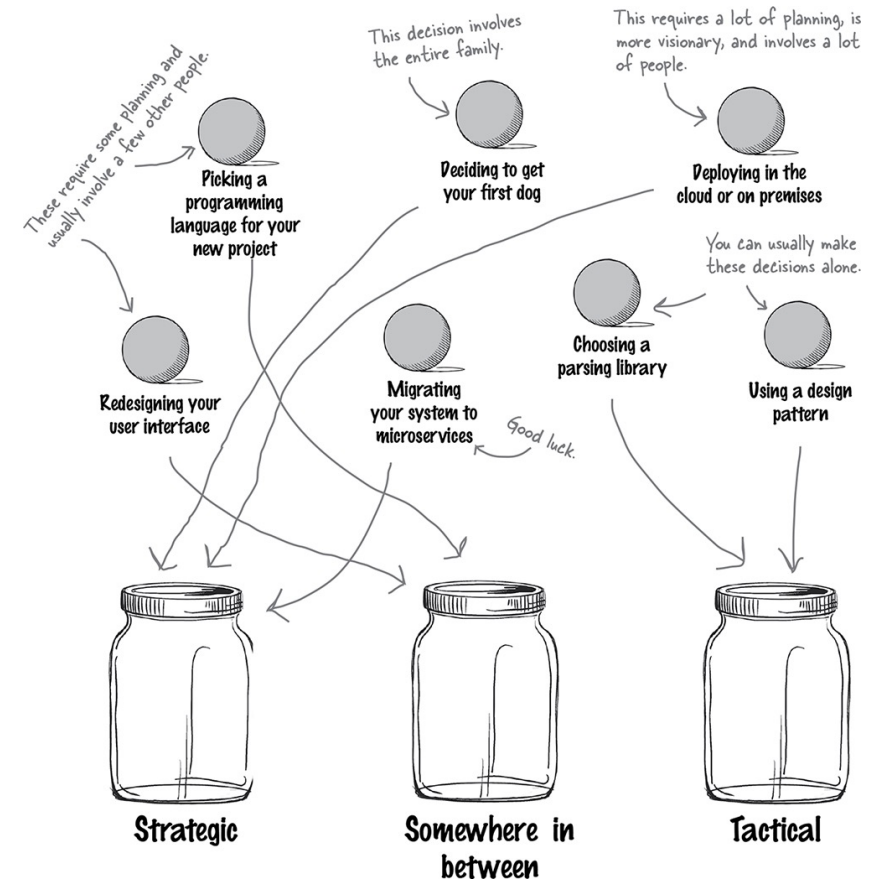


Identifying Architectural Decisions

- ❖ Questions to consider:
 - Is it **strategic** (long-term) or **tactical** (short-term)?
 - Effort to change: **high** or **low**?
 - Does it involve significant **trade-offs**?

Examples:

- Migrating from monolith to microservices (architecture, strategic).
- Changing background colour of login page (design, tactical).



Trade-offs in Decision Making

- ❖ Architectural decisions often involve **significant trade-offs**.

Example:

- **Cloud deployment**: scalability vs. cost.
- **Async messaging**: performance vs. complexity.
- Choosing between **performance** and data **consistency**.

- ❖ **Architects** handle strategic choices; **developers** manage detailed tactical choices

Significant Tradeoffs?

- | | |
|-----------------------------------------|----------------------------------------|
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input checked="" type="checkbox"/> Yes | <input type="checkbox"/> No |
| <input type="checkbox"/> Yes | <input checked="" type="checkbox"/> No |

Picking out what clothes to wear to work today

Choosing to deploy in the cloud or on premises

Selecting a user interface framework

Deciding on the name of a variable in a class file

Choosing between vanilla and chocolate ice cream

Deciding which architectural style to use

Choosing between REST and messaging

Using full data or only keys for the message payload

Selecting an XML parsing library

Deciding whether or not to break apart a service

Choosing between atomic or distributed transactions

Deciding whether or not to go out to dinner tonight

Are you getting hungry yet?

Okay, so maybe this is a difficult decision sometimes.

There are certainly trade-offs here, so this one could go either way.

These can impact scalability, performance, and overall maintainability.

This can impact data integrity and data consistency, but also scalability and performance.

Summary (1)

- ❖ Architecture focuses on structure and system-wide qualities; design is more about code-level appearance and organization.

- ❖ Four essential dimensions of software architecture:
 - Architectural **Characteristics** – Foundation traits like scalability, availability, security.
 - Architectural **Decisions** – Guideposts that define the system's constraints and trade-offs.
 - **Logical Components** – Functional building blocks implemented in code.
 - **Architectural Style** – High-level patterns like layered, event-driven, or microservices.

Summary (2)

- ❖ Software architecture is about making informed **structural decisions**, not just organising code.
- ❖ Understand and **prioritise** architectural **characteristics** for your system.
- ❖ Every architectural decision involves **trade-offs**, know the “why.”
- ❖ **Use ADRs** to document decisions and ensure long-term clarity.
- ❖ Choose an architectural style that supports your system’s **most critical characteristics**.
- ❖ Know when a decision is architectural (system-wide impact) or design-level (local impact).

“Good architecture **supports change**. Great architecture **explains why**.”

Architectural Characteristics

COMP2511, CSE, UNSW

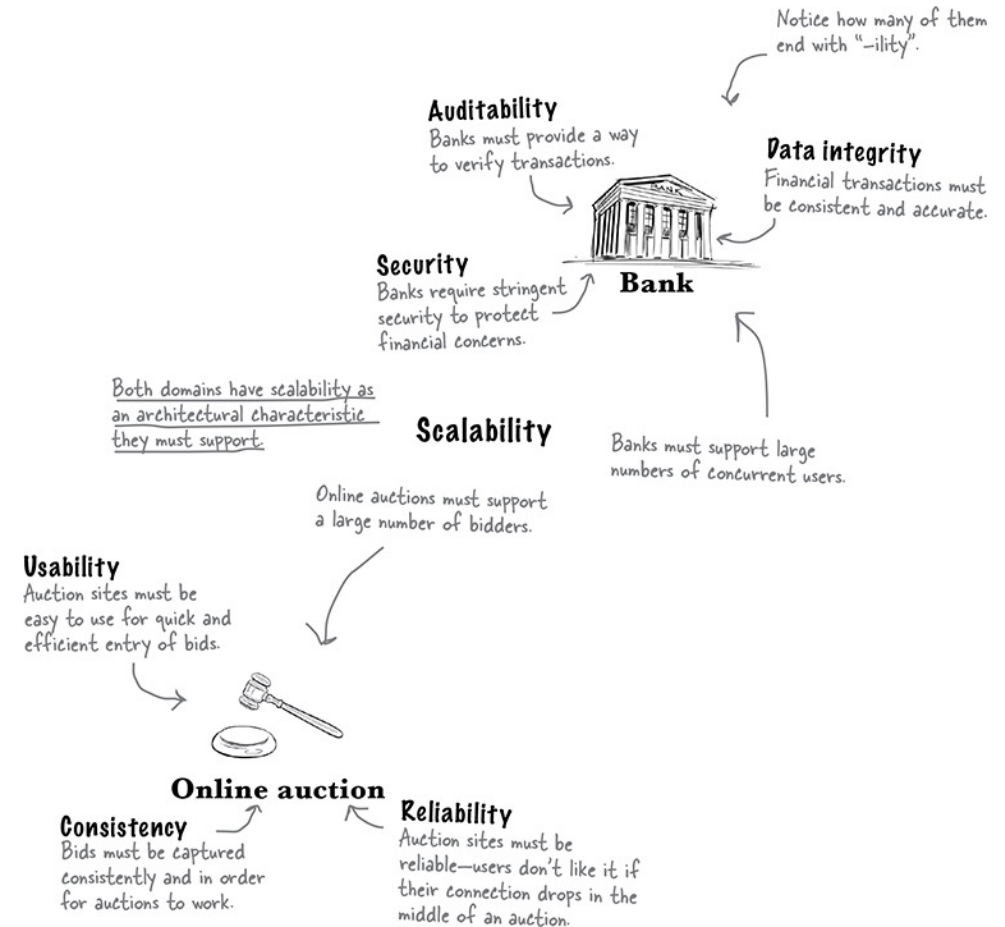


UNSW
SYDNEY

These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

What Are Architectural Characteristics?

- ❖ Architectural Characteristics define **fundamental qualities** software architecture must support.
- ❖ They are often **not explicitly defined**
- ❖ They **influence structure**, infrastructure, and quality of the system
- ❖ Architectural characteristics **guide decisions** like architectural style, deployment, and scalability.



Some of the Popular Architectural Characteristics

- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">○ Scalability: Handles growth in traffic or data size○ Availability: Ensures system uptime○ Maintainability: Easy to update, fix, or extend○ Security: Prevents unauthorized access | <ul style="list-style-type: none">○ Elasticity: Automatically adjusts resources based on load○ Deployability: Enables safe, frequent updates○ Responsiveness: Provides quick feedback to users |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- ❖ There is **no fixed** or exhaustive **list** — they evolve with time and context
- ❖ Many required characteristics are **not explicitly stated** in the requirements
- ❖ Proper **domain** and **contextual** understanding is essential **to identify** them
- ❖ They are **essential to align architecture** with real-world needs, for successful development, deployment, and future system resilience

Case Study - Lafter (Sillycon Symposia)

- ❖ A fictional tech-comedy conference platform to illustrate architectural thinking.
- ❖ **Use Case:** Participants post jokes/puns, react with "*HaHa*" or "*Giggle*," and engage with speakers.
- ❖ **Functional Needs:**
 - User registration
 - Posting content (jokes/puns)
 - Reaction buttons and speaker tools
 - Language support (international audience)
- ❖ **Architectural Constraints:**
 - **Scalability:** Must handle traffic bursts during peak conference hours.
 - **Availability:** Downtime would impact live sessions and user interaction.
 - **Security:** Accounts and speaker tools require access control.
 - **Maintainability:** Small team must support system with minimal overhead.

Applying Architectural Thinking to Lafter

❖ Design Decisions Influenced by Characteristics:

- Choose **microservices** for independently deployable features (e.g., joke-posting vs. notifications).
- Use **cloud-based hosting** with autoscaling to manage bursty traffic.
- Implement **CI/CD pipelines** to ensure rapid, reliable deployments.
- Enable **internationalisation** to support a multilingual audience.

❖ Examples of Characteristics Applied:

- **Elasticity** → Serverless functions for unpredictable joke-post surges
- **Responsiveness** → Real-time interactions using *WebSockets*
- **Testability** → Unit-tested microservices enable quick fixes

Architectural Characteristics *vs.* Logical Components

- ❖ **Architectural Characteristics:** How the system performs under various constraints
- ❖ **Logical Components:** What the software does (domain behavior)
- ❖ Both are essential for structural design

Component: User Registration

Characteristic: Scalability
(handles thousands of concurrent signups)

Component: Content Posting

Characteristic: Availability
(system is up when users post jokes)

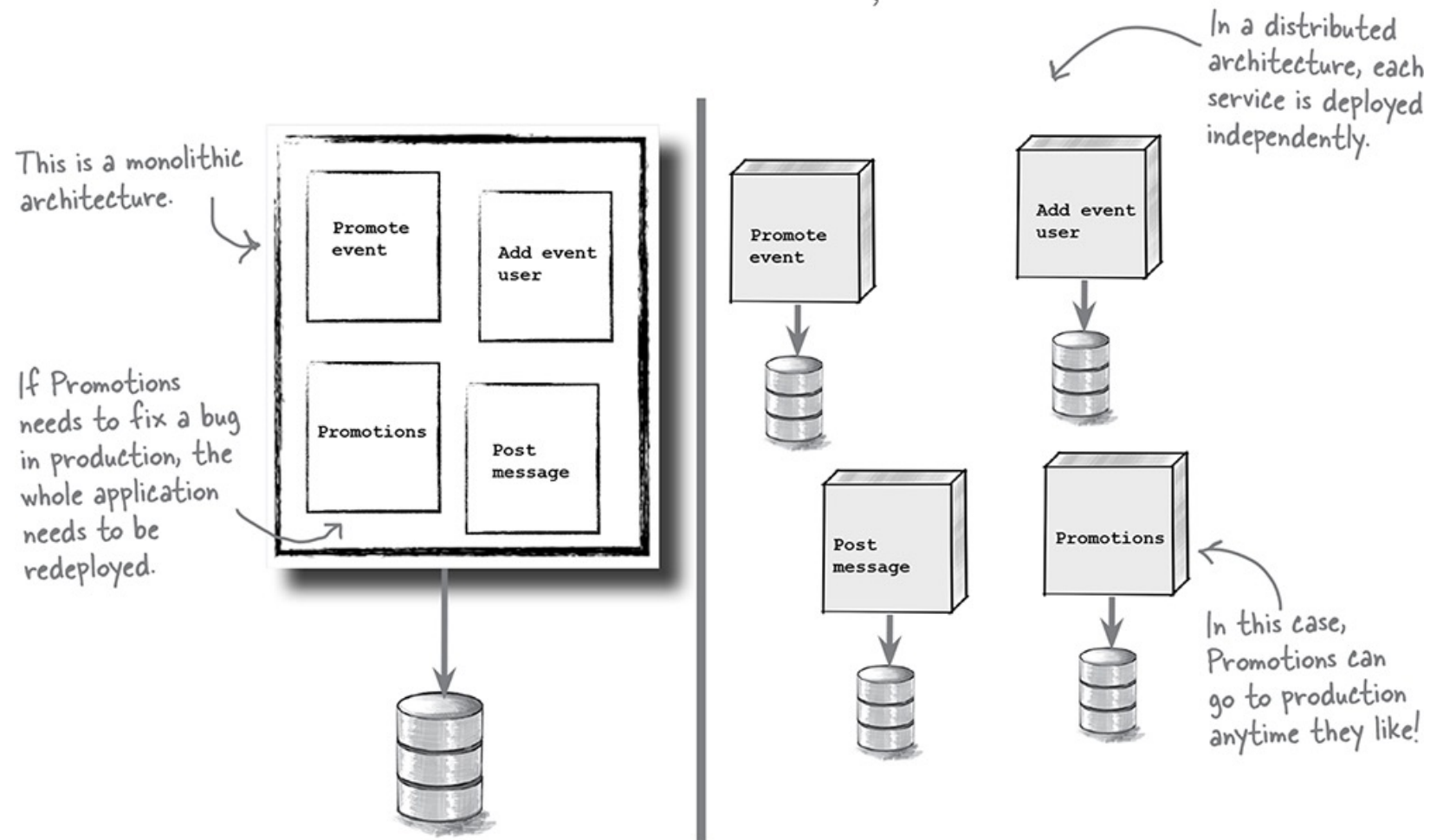
Component: Notifications

Characteristic: Responsiveness
(sends real-time updates with low latency)

Component: Admin Dashboard

Characteristic: Security
(restricts access to authorized users only)

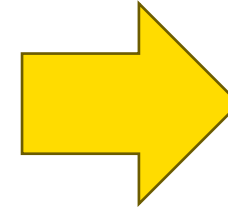
Characteristics Affect Structure



Characteristics Affect Structure

Architectural Characteristics

- **Security**: May require encryption layers, role-based access control, audit trails
- **Scalability**: May need load balancers, stateless services, database sharding
- **Availability**: May require failover mechanisms, replication, redundant systems
- **Responsiveness**: Might demand caching, asynchronous processing
- **Elasticity**: May need autoscaling and container orchestration



Architectural Impact

- **Monolithic vs Microservices**
- **Cloud vs On-Premise deployment**
- Etc.

Don't Overengineer!

❖ Too many characteristics = complexity

❖ They are:

- Synergistic (affect each other, improving *security* may result in low *performance*)
- Continuously evolving
- Impossible to standardize (*performance* and *responsiveness* might indicate the same behavior)

❖ Limit characteristics to prevent overengineering

- Identifying which characteristics are most critical acts as a filtering mechanism
- Helps eliminate "nice-to-have" features that add unnecessary complexity and cost
- Stay focused on traits essential for success
- Limit to around 7 key characteristics, humans best manage 7 ± 2 items!

Implicit vs Explicit Characteristics

- ❖ **Explicit:** Stated clearly in the requirements document
 - "The system must support French and Japanese" → Internationalization
 - "Allow only registered users to access admin panel" → Authorization
- ❖ **Implicit:** Not stated, but understood or expected (requires domain/context understanding)
 - Users expect their data to be secure even if not mentioned → Security
 - An app must perform well during high traffic without explicit mention → Performance/Scalability
 - A public website is expected to be available 24/7 → High Availability
- ❖ Architects must **read between** the lines to uncover hidden requirements

Explicit

- Packages are stacked outside a door.



Implicit

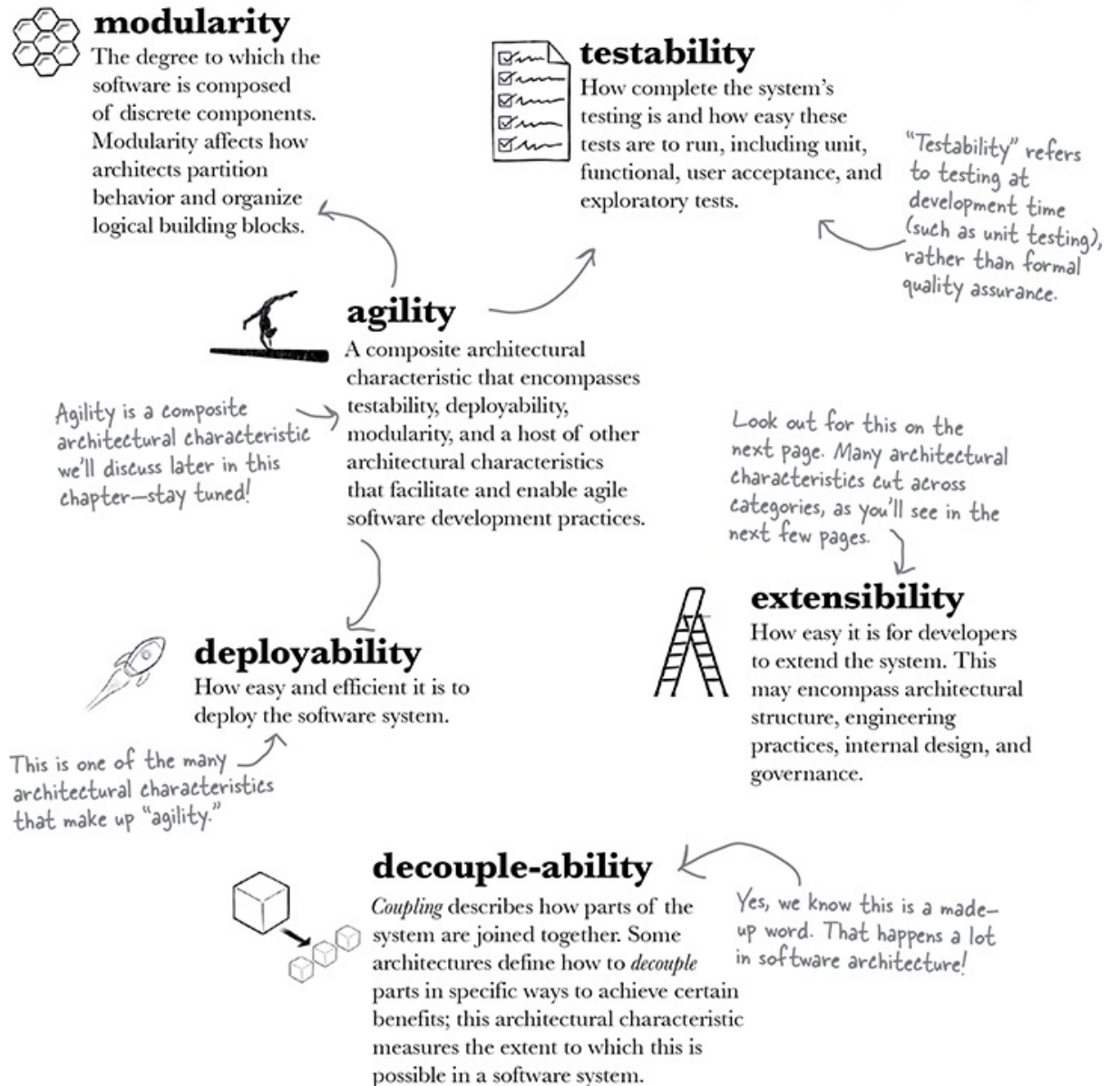
- No one is home.
- This family orders a bunch of stuff online.
- Is this family on vacation?

Types of Characteristics

- ❖ **Process** Characteristics: Deployability, Maintainability
- ❖ **Structural** Characteristics: Modularity, Coupling
- ❖ **Operational** Characteristics: Scalability, Availability
- ❖ **Cross-cutting** Characteristics: Accessibility, Security

Process Characteristics

- ❖ Represent the intersection between architecture and the **software development** process
- ❖ Reflect how the **system is built**, tested, deployed, and evolved
- ❖ **Guide decisions** related to engineering practices, automation, and team workflows



Structural Characteristics

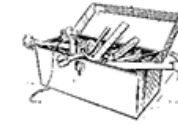
- ❖ Concerned with the **internal structure** and **composition** of the system
- ❖ Influence how **components** are coupled, **interact**, and evolve independently
- ❖ Impact design qualities like **modularity, cohesion, and adaptability**



security

How secure the system is, holistically. Does the data need to be encrypted in the database? How about for network communication between internal systems? What type of authentication needs to be in place for remote user access?

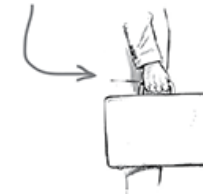
Security appears in every application, as an implicit or explicit architectural characteristic.



maintainability

How easy it is for architects and developers to apply changes to enhance the system and/or fix bugs.

Portability can apply to any part of the system, including the user interface and implementation platform.



portability

How easy it is to run the system on more than one platform (for example, Windows and macOS).



extensibility

How easy it is for developers to extend the system. This may encompass architectural structure, engineering practices, internal design, and governance.

This is one of those characteristics that belong to more than one category.

Some architectural characteristics cover development concerns rather than purely domain concerns.



localization

How well the system supports multiple languages, units of measurement, currencies, and other factors that allow it to be used globally.

Another flavor of localization is internationalization (i18n).

Operational Characteristics

- ❖ Represent how architectural decisions shape and support system **behavior at runtime**
- ❖ Define what the operations team can **monitor, control, or adapt** while the system is running
- ❖ Directly influence system **reliability, performance, and fault tolerance**



availability

What percentage of the time the system needs to be available and, if 24/7, how easy it is to get the system up and running quickly after a failure.

Usually represented as a number of "nines" (99.999% uptime = 5 nines, a bit under 6 minutes/year).



recoverability

How quickly the system can get online again and maintain business continuity in case of a disaster. This will affect the backup strategy and requirements for duplicated hardware.

A good example of the axiom that you can take any adjective and add "-ility" to make a new architectural characteristic!



robustness

The system's ability to handle errors and boundary conditions while running, such as if the power, internet connection, or hardware fails.



performance

How well the system achieves its timing requirements using the available resources.

As you will see shortly, "performance" has many different aspects.

When these are important, they are very important.



reliability/safety

Whether the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it endanger people's lives or cost the company large sums of money? Common for medical systems, hospital software, and airplane applications.

Some "-ilities" are easier to achieve than others. This one is often difficult.



scalability

How well the system performs and operates as the number of users or requests increases.

Our Laffer application will definitely need this!

Cross-Cutting Characteristics

- ❖ Span **multiple parts** of the system and affect other characteristics
- ❖ Often enforced through design, tooling, and governance



Composite Characteristics

- ❖ Formed from **multiple** simpler traits
- ❖ These high-level characteristics reflect complex system qualities that require multiple dimensions to evaluate.
- ❖ Examples:
 - **Reliability** = Availability + Consistency + Data Integrity
 - **Resilience** = Robustness + Fault Tolerance + Recoverability
- ❖ Must **break down** into measurable parts

Sources of Characteristics

- ❖ **Problem Domain (Feature-Driven)**: Driven by product goals, system features, and expected usage patterns.
 - A real-time multiplayer game must be highly responsive and scalable due to concurrent users → Responsiveness, Scalability
 - An e-commerce site must support flash sales and high traffic → Elasticity, Performance, Availability
- ❖ **Environmental Awareness (Organizational Constraints)**: Driven by company's culture, budget, and capabilities
 - A startup must deliver features fast → favors Deployability, Agility
 - A globally distributed team → needs Testability, Modularity for asynchronous collaboration
 - Legacy-heavy organizations → may prioritize Integrability with existing systems

Sources of Characteristics

❖ **Holistic Domain Knowledge (Industry and Compliance Expectations):** Driven by regulatory standards, industry best practices, and user trust factors.

- A banking app must comply with regulations → Security, Auditability, Availability
- A healthcare platform must protect patient data → Confidentiality, Data Integrity, Compliance
- Government services need to meet accessibility and privacy laws → Accessibility, Security, Maintainability

❖ **Why It Matters:**

- Ignoring a source can lead to critical failures later
- Architects must triangulate across all three to identify the most important characteristics

Translating Business Goals into Architecture

- ❖ A core responsibility of the architect is to **translate** high-level business goals into concrete architectural characteristics and decisions.

Examples:

- ❖ Business: “**The system must always work.**”
 - → High Availability, Fault Tolerance, Robustness
- ❖ Business: “**Customers shouldn’t wait.**”
 - → Responsiveness, Performance, Latency Budgeting
- ❖ Business: “**We need to move fast and innovate.**”
 - → Deployability, Modularity, Testability
- ❖ Business: “**We need to meet compliance and regulation.**”
 - → Security, Auditability, Traceability, Accessibility

Trade-offs Between Architectural Characteristics (1)

- ❖ Architectural characteristics frequently **compete** or **conflict**
- ❖ Enhancing one trait can reduce or compromise another

Examples:

- **Security vs. Performance** [More security controls (e.g., encryption, validation) add processing overhead]
- **Scalability vs. Simplicity** [Scalable systems often introduce distributed complexity (e.g., microservices, load balancers)]
- **Availability vs. Maintainability** [High availability may require complex failover and redundancy, complicating upgrades and maintenance]

Key Insight:

- Trade-offs are **not flaws**, but **conscious** architectural **decisions**
- There are **no universally right** answers—*"It depends."*

Trade-offs Between Architectural Characteristics (2)

More Examples:

- ❖ **Deployability** vs. **Robustness**: Rapid deployments can reduce test cycles and stability
- ❖ **Responsiveness** vs. Data **Consistency**: Fast user interactions might rely on eventually consistent models
- ❖ **Flexibility** vs. **Performance**: Designing for plugin support or configuration often introduces performance bottlenecks

Best Practices:

- ❖ **Engage stakeholders** early to understand what matters most
- ❖ **Prioritise** characteristics based on system goals, user needs, and domain constraints
- ❖ **Trade-offs** reflect organizational priorities and constraints
- ❖ Use **Architectural Decision Records (ADRs)** to document trade-offs and rationale

Summary

- ❖ Architecture is about making **conscious trade-offs**
 - ❖ Every decision comes with **upsides** and **downsides**
 - ❖ Capture both the decision and the **reason behind** it
 - ❖ Embrace change—**architecture evolves** with the system
-
- ❖ Architecture isn't about right answers—it's about **right reasoning**

Architectural Decision Records (ADRs)

These lecture slides are from the books:

- *“Head First Software Architecture”*, by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024
- *“Fundamentals of Software Architecture”*, 2nd Edition, by Mark Richards, Neal Ford

Architectural Decision Records (ADRs)

- ❖ "Why is more **important** than how."
- ❖ Architectural decisions must be **justified**
- ❖ Future team members need to **understand the rationale**
- ❖ Without context, good decisions **may seem arbitrary** or incorrect
- ❖ Architectural Decision Records (**ADRs**) capture the **what**, **why**, and **impact**
- ❖ An ADR has **seven sections**: Title, Status, Context, Decision, Consequences, Governance, and Notes.
- ❖ Important aspects of an architectural decision are documented, including the decision itself.



ADR

ADR Structure Overview

Main Sections:

- ❖ **Title**: Numbered and concise
- ❖ **Status**: Proposed, Accepted, Superseded, or Request for Comments (RFC)
- ❖ **Context**: Forces and constraints
- ❖ **Decision**: What was chosen and why
- ❖ **Consequences**: Trade-offs and impacts
- ❖ **Compliance**: Governance and enforcement
- ❖ **Notes**: Metadata (author, date, approval)

ADR Section - Title

❖ **Purpose:** Identify and summarize the decision

❖ **Best Practices:**

- Number sequentially (e.g., ADR 001)
- Short, descriptive, and unambiguous

❖ **Example:**

- “ADR 17: Asynchronous Messaging Between Order and Payment Services”
- “ADR 21: Transition to PostgreSQL for Inventory Management”
- “ADR 34: Enable OAuth 2.0 for Internal APIs”

ADR Section - Status

❖ Types:

- Proposed: Pending approval
- Accepted: Approved and active
- Superseded: Replaced by another ADR
- RFC: Open for feedback until a deadline

❖ Examples:

- ADR 12: Status: Accepted
- ADR 17: Status: Superseded by ADR 21
- ADR 34: Status: RFC, Deadline 30 May 2025

ADR Section - Context

❖ **Purpose:** Explain what situation led to this decision

❖ **Include:**

- Problem or force requiring a decision
- Alternatives under consideration

❖ **Examples:**

- “The Order service must transmit payment info. Options include REST or messaging.”
- “Inventory updates are inconsistent across services; central DB vs. event-based sync considered.”
- “Increased phishing attempts require re-evaluating access token validation approach.”

ADR Section - Decision

❖ **Purpose:** Describe what was chosen

❖ **Best Practices:**

- Use clear, assertive language: “We will use...”
- Justify with rationale

❖ **Examples:**

- “We will use asynchronous messaging due to latency and decoupling benefits.”
- “We will migrate inventory management to PostgreSQL to ensure consistency and performance.”
- “We will adopt OAuth 2.0 using *IdentityServer* for access control.”

ADR Section - Consequences

❖ **Purpose:** Describe outcomes and trade-offs

❖ **Include:**

- Positive and negative impacts
- Known limitations

❖ **Examples:**

- “Improves performance, however adds complexity in error handling.”
- “Enables real-time updates; requires Kafka infrastructure.”
- “Strengthens security, but introduces user reauthentication challenges.”

ADR Section - Compliance

❖ **Purpose:** Define how decision enforcement is measured

❖ **Types:**

- Manual review
- Automated tests (e.g., fitness functions)

❖ **Examples:**

- Static code analysis rules for package structure compliance
- Integration test that validates token expiration and renewal workflows

ADR Section - Notes

❖ Purpose: Capture metadata

❖ Typical Fields:

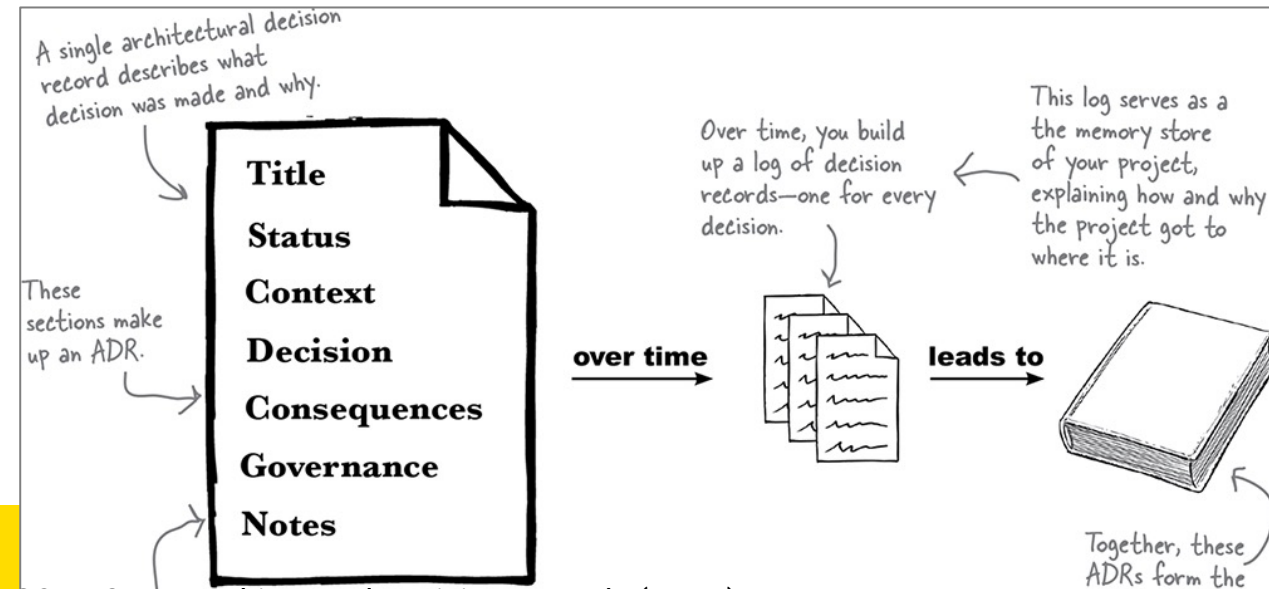
- Author, approval date, approver
- Last modified, superseded reference

❖ Examples:

- Author: A. Johnson, Approved by: Arch Review Board, 15 May 2025
- Author: M. Singh, Modified on: 10 May 2025, Supersedes ADR 12
- Author: L. Chen, RFC Deadline: 30 May 2025

Benefits of Using ADRs

- ❖ Serves as a memory **log for decisions**
- ❖ Helps new team members **understand context**
- ❖ Improves **consistency** and governance
- ❖ Supports **continuous** evolution and **learning**



Example: ADR

Title

012: Use of queues for asynchronous messaging between order and downstream services

Status

Accepted

Context

The trading service must inform downstream services (namely the notification and analytics services, for now) about new items available for sale and about all transactions. This can be done through synchronous messaging (using REST) or asynchronous messaging (using queues or topics).

Decision

We will use queues for asynchronous messaging between the trading and downstream services.

Using queues makes the system more extensible, since each queue can deliver a different kind of message. Furthermore, since the trading service is acutely aware of any and all subscribers, adding a new consumer involves modifying it—which improves the security of the system.

Consequences

Queues mean a higher degree of coupling between services.

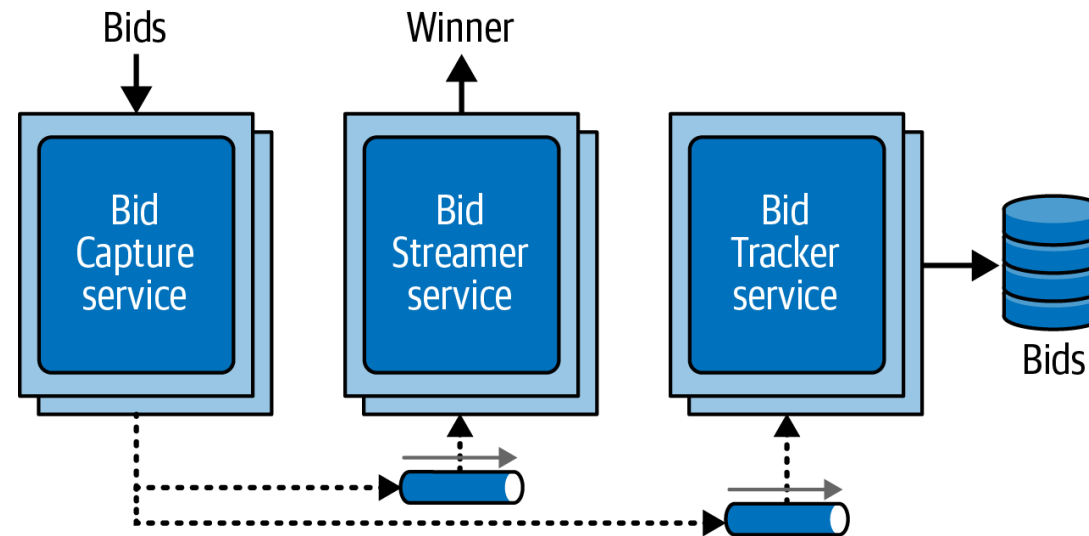
We will need to provision queuing infrastructure. It will require clustering to provide for high availability.

If additional downstream services (in addition to the ones we know about) need to be notified, we will have to make modifications to the trading service.

ADR – Auction System Example

Architectural Decision in the following auction system:

- ❖ use separate point-to-point queues between the bid capture, bid streamer, and bid tracker services **instead of** a single publish-and-subscribe topic (or even REST, for that matter)
- ❖ ADR needs to **justify the choice** to prevent confusion or disagreements among other designers or developers.



ADR – Auction System Example

ADR 76. Separate Queues for Bid Streamer and Bidder Tracker Services

STATUS

Accepted

CONTEXT

*The **Bid Capture** service, upon receiving a bid, must forward that bid to the **Bid Streamer** service and the **Bidder Tracker** service. This could be done using a single topic (pub/sub), separate queues (point-to-point) for each service, or REST via the Online Auction API layer.*

ADR – Auction System Example

DECISION

We will use separate queues for the `Bid Streamer` and `Bidder Tracker` services.

The `Bid Capture` service does not need any information from the `Bid Streamer` service or `Bidder Tracker` service (communication is only one-way).

The `Bid Streamer` service must receive bids in the exact order they were accepted by the `Bid Capture` service. Using messaging and queues automatically guarantees the bid order for the stream by leveraging first-in, first out (FIFO) queues.

Multiple bids come in for the same amount (for example, “Do I hear a hundred?”). The `Bid Streamer` service only needs the first bid received for that amount, whereas the `Bidder Tracker` needs all bids received. Using a topic (pub/sub) would require the `Bid Streamer` to ignore bids that are the same as the prior amount, forcing the `Bid Streamer` to store shared state between instances.

The `Bid Streamer` service stores the bids for an item in an in-memory cache, whereas the `Bidder Tracker` stores bids in a database. The `Bidder Tracker` will therefore be slower and might require backpressure. Using a dedicated `Bidder Tracker` queue provides this dedicated backpressure point.

ADR – Auction System Example

CONSEQUENCES

We will require clustering and high availability of the message queues.

This decision will require the `Bid Capture` service to send the same information to multiple queues.

Internal bid events will bypass security checks done in the API layer.

UPDATE: Upon review at the January 14, 2025, ARB meeting, the ARB decided that this was an acceptable trade-off and that no additional security checks are needed for bid events between these services.

COMPLIANCE

We will use periodic manual code reviews to ensure that asynchronous pub/sub messaging is being used between the `Bid Capture` service, `Bid Streamer` service, and `Bidder Tracker` service.

NOTES

Author: Subashini Nadella

Approved: ARB Meeting Members, 14 JAN 2025

Last Updated: 14 JAN 2025

Summary of ADR

- ❖ Each section contributes to clarity and traceability
- ❖ Together they provide context, rationale, and continuity
- ❖ Encourage consistent use across all teams and domains

Behavioural Modelling

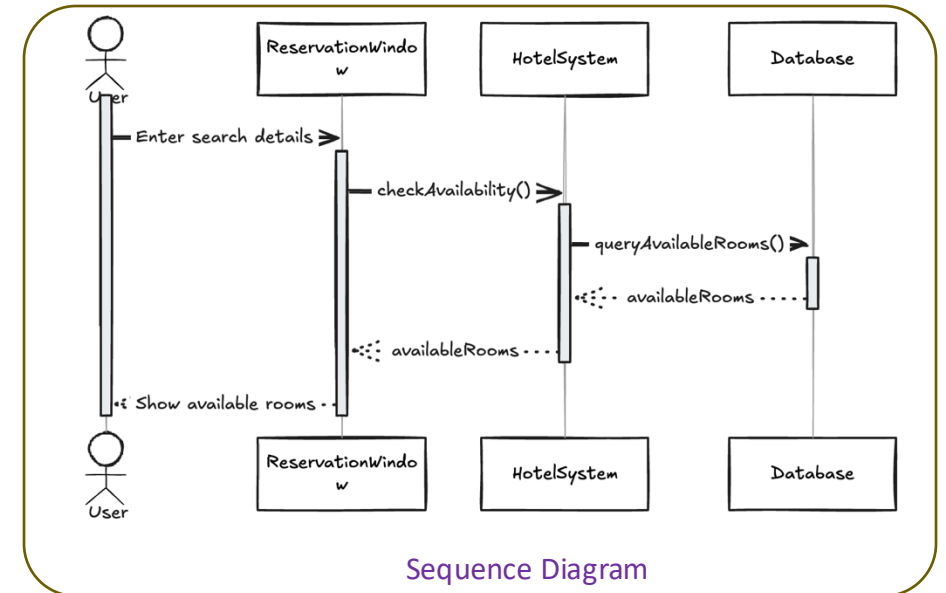
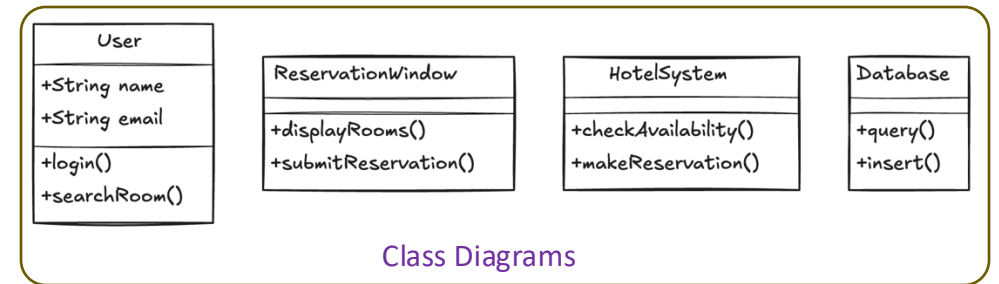
COMP2511, CSE, UNSW



UNSW
SYDNEY

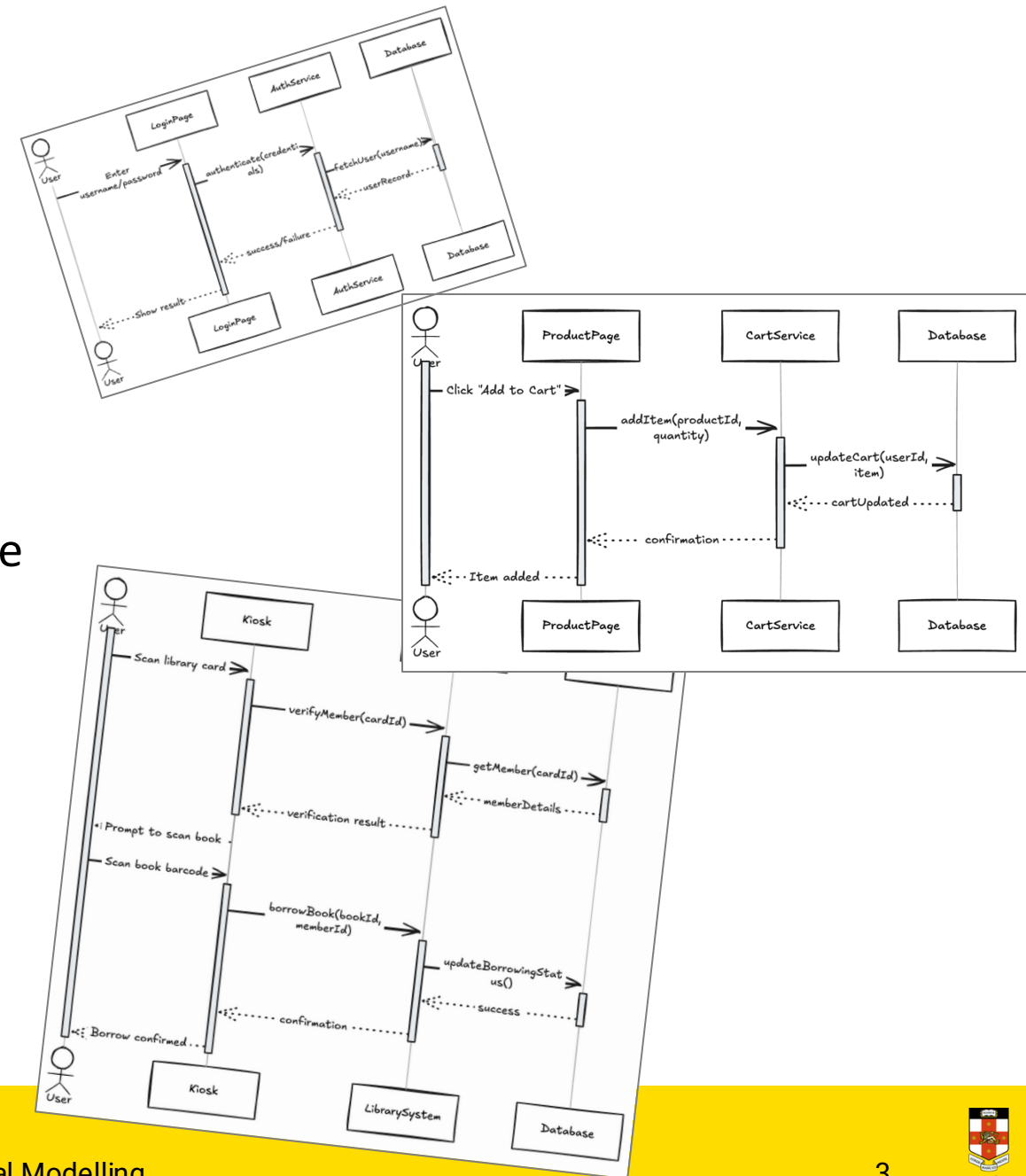
What is Behavioural Modelling

- ❖ **Behavioural modelling** captures *how* the system behaves in response to events or interactions *over time*.
- ❖ Software Design and Architecture **do not** tell us how components behave or interact over time.
- ❖ Different notations for expressing behaviour:
 - **Sequence diagrams**
 - Activity diagrams
 - State charts

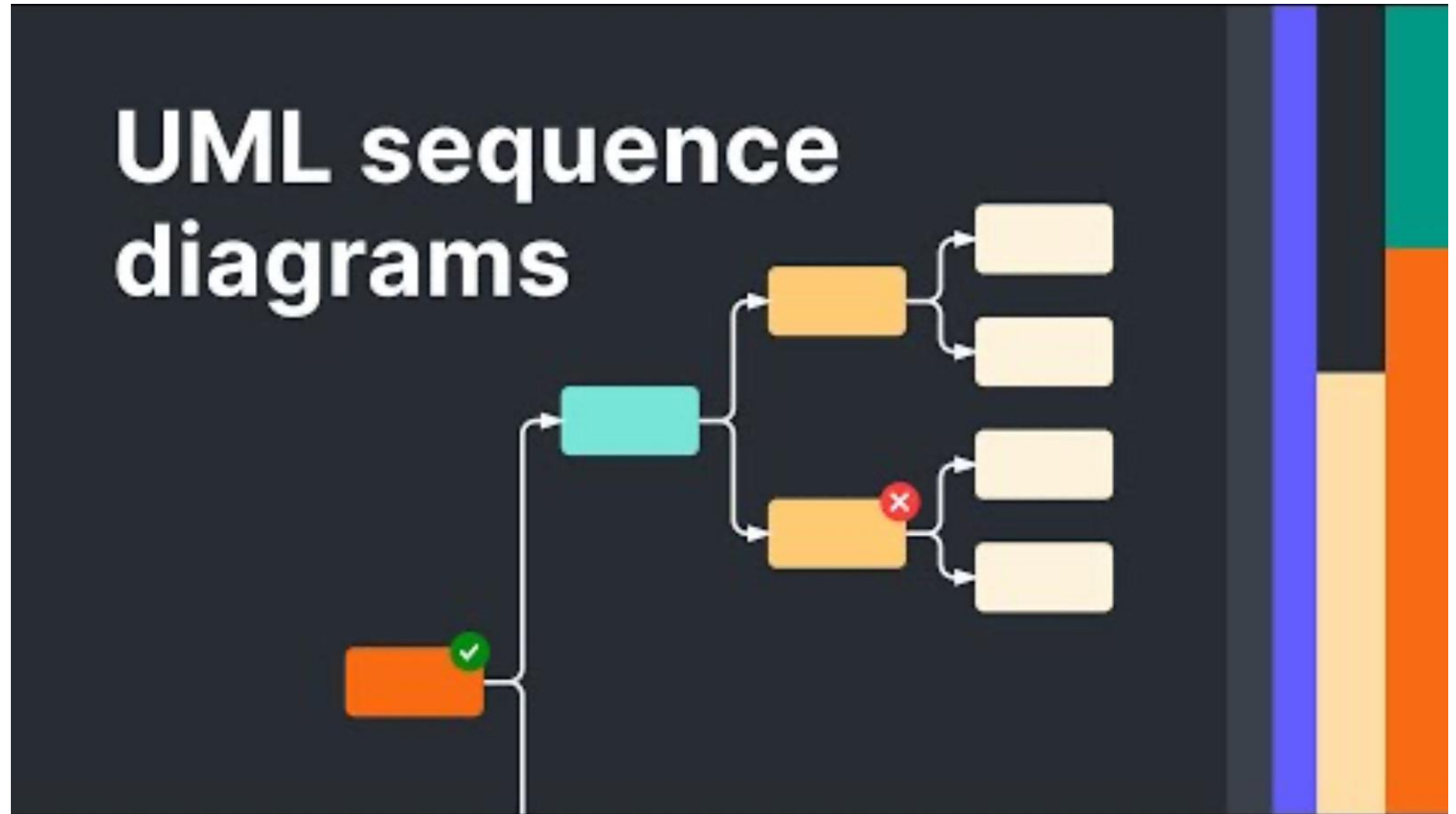


Sequence Diagrams

- ❖ A sequence diagram is an **interaction diagram** showing how objects interact in a **time-sequenced manner**.
- ❖ **Clarify interactions** among objects and improve system behaviour understanding.
- ❖ **Show** how operations are carried out through **message exchanges**.
- ❖ Emphasize the **temporal order** of interactions.

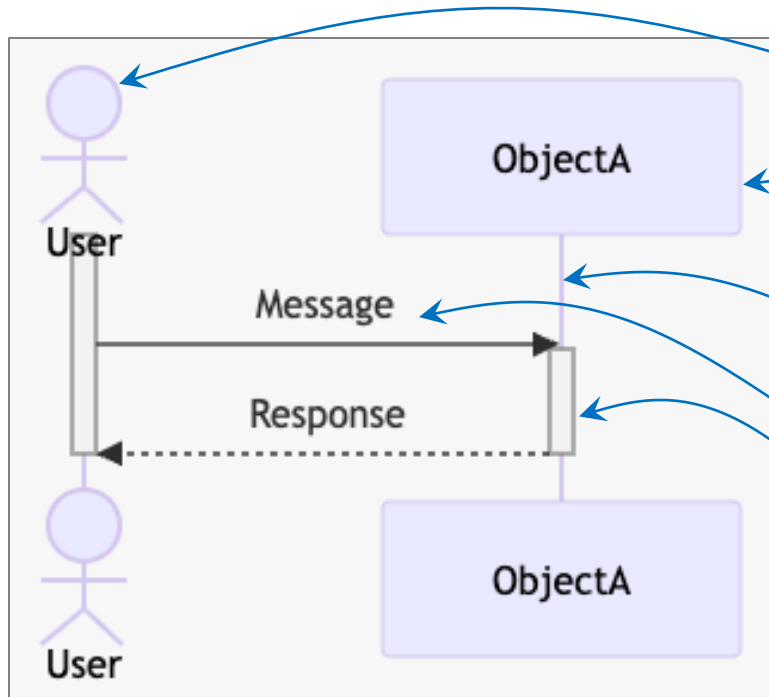


More on sequence diagrams



<https://www.youtube.com/watch?v=pCK6prSq8aw> (8 mins)

Key Components of a Sequence Diagram



Actor: External user or system

Objects: Entities involved, represented by rectangles.

Lifelines: Lines (or dashed lines) showing object existence during interactions.

Messages: Communication between objects.

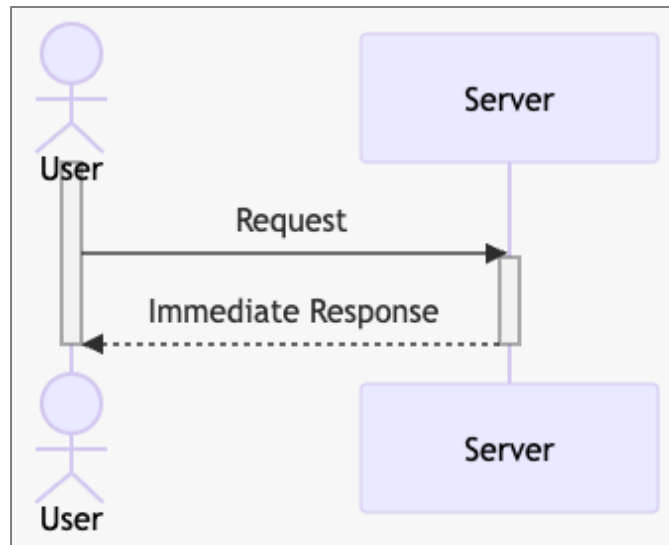
Activation Boxes: Indicate active processing of messages.

- User sends a message to ObjectA, activating ObjectA's processing.
- ObjectA responds, deactivating afterwards.

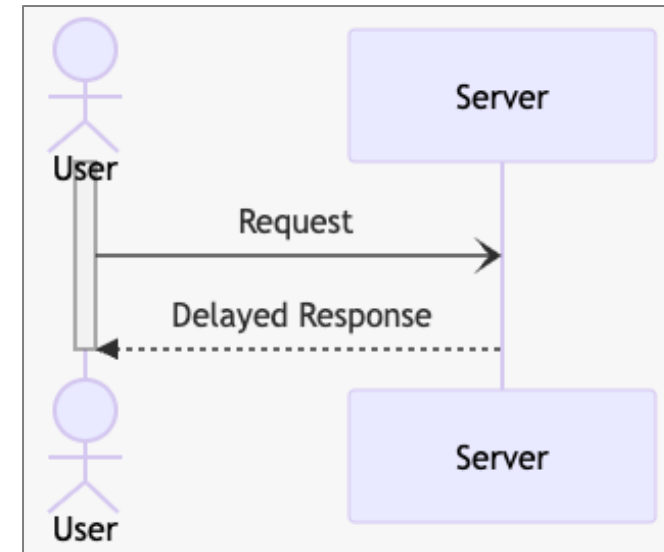
Types of Messages

Synchronous: Sender waits for a response.

Asynchronous: Sender does not wait for an immediate response.



Synchronous: User waits for Server to complete processing before proceeding.

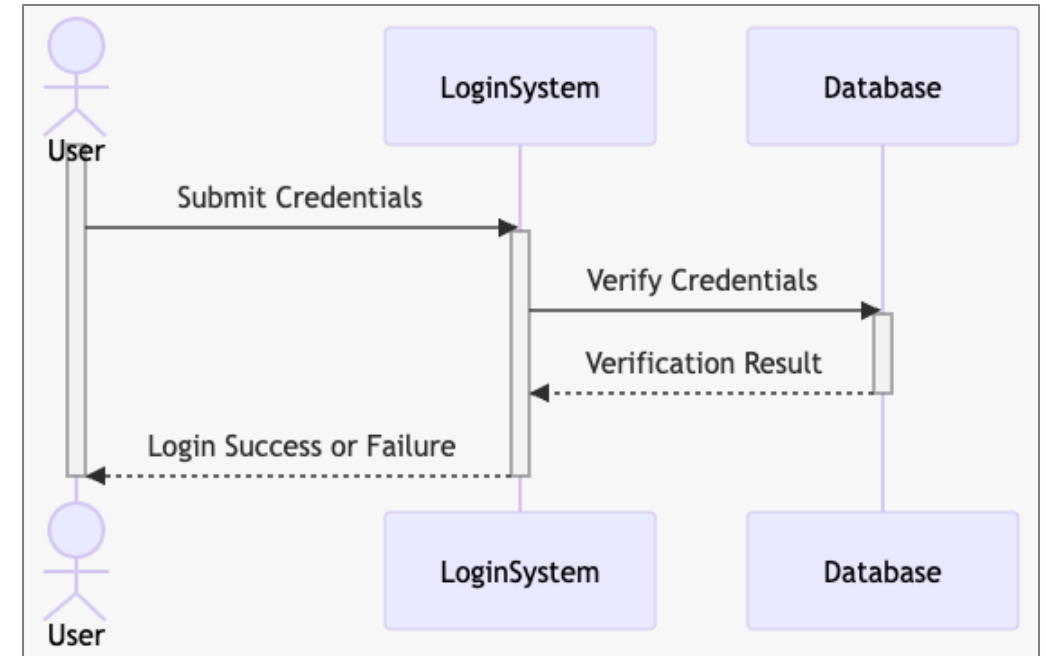


Asynchronous: User continues immediately without waiting for Server's response.

Sequence Diagram Overview

- ❖ **Horizontal axis** captures participating objects.
 - Objects are placed from left to right.
 - Order reflects participation in message sequence.
 - Horizontal layout is flexible but typically chronological.
- ❖ **Vertical axis** represents time (top to bottom).
 - Time flows downward.
 - Sequence diagrams focus on *order, not duration*.
 - Vertical spacing is not indicative of actual time intervals.
- ❖ **Messages** are shown as horizontal arrows. Messages can be calls/invocations for some methods in a component, or results given by that component.
- ❖ **Execution** shown using rectangles (**activations boxes**).

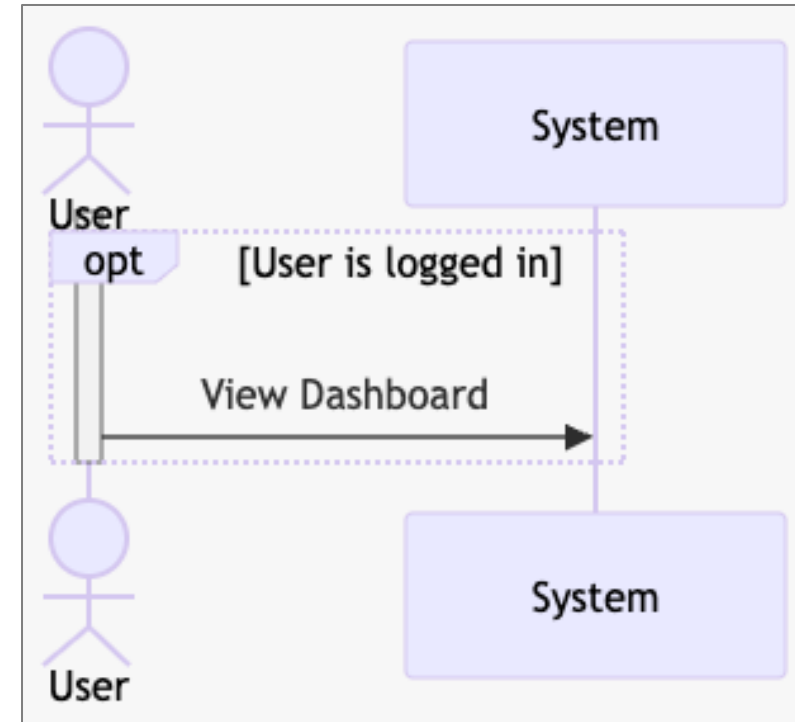
Example: User Login



User's credentials are checked against a database, resulting in either login success or failure.

Optional Interaction

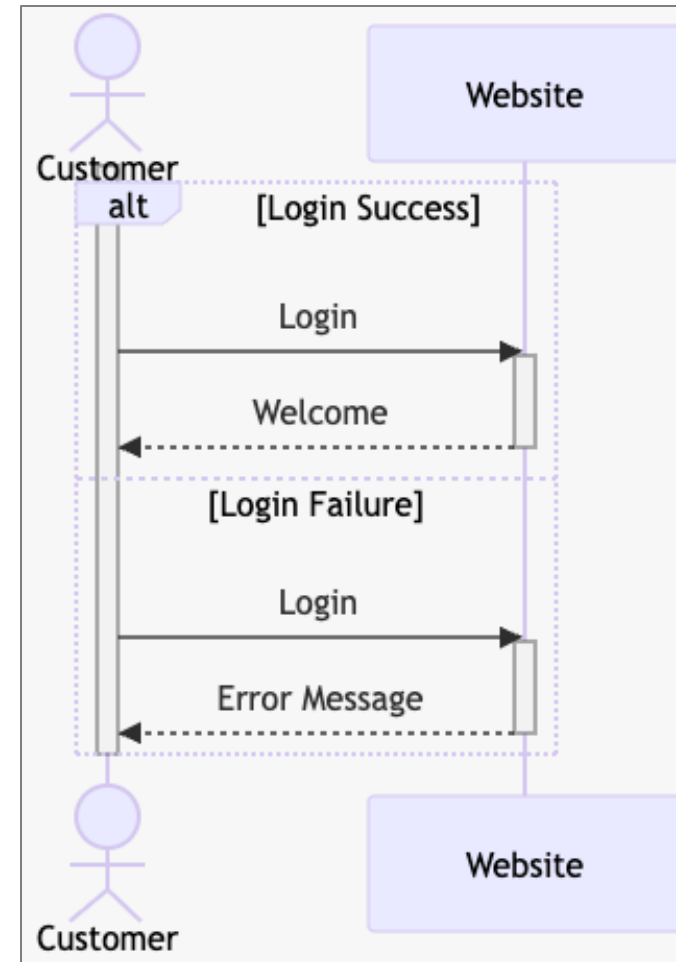
❖ **opt** represents **optional** scenarios.



Illustrates **optional logic** based on condition result (success).

Conditional Interaction

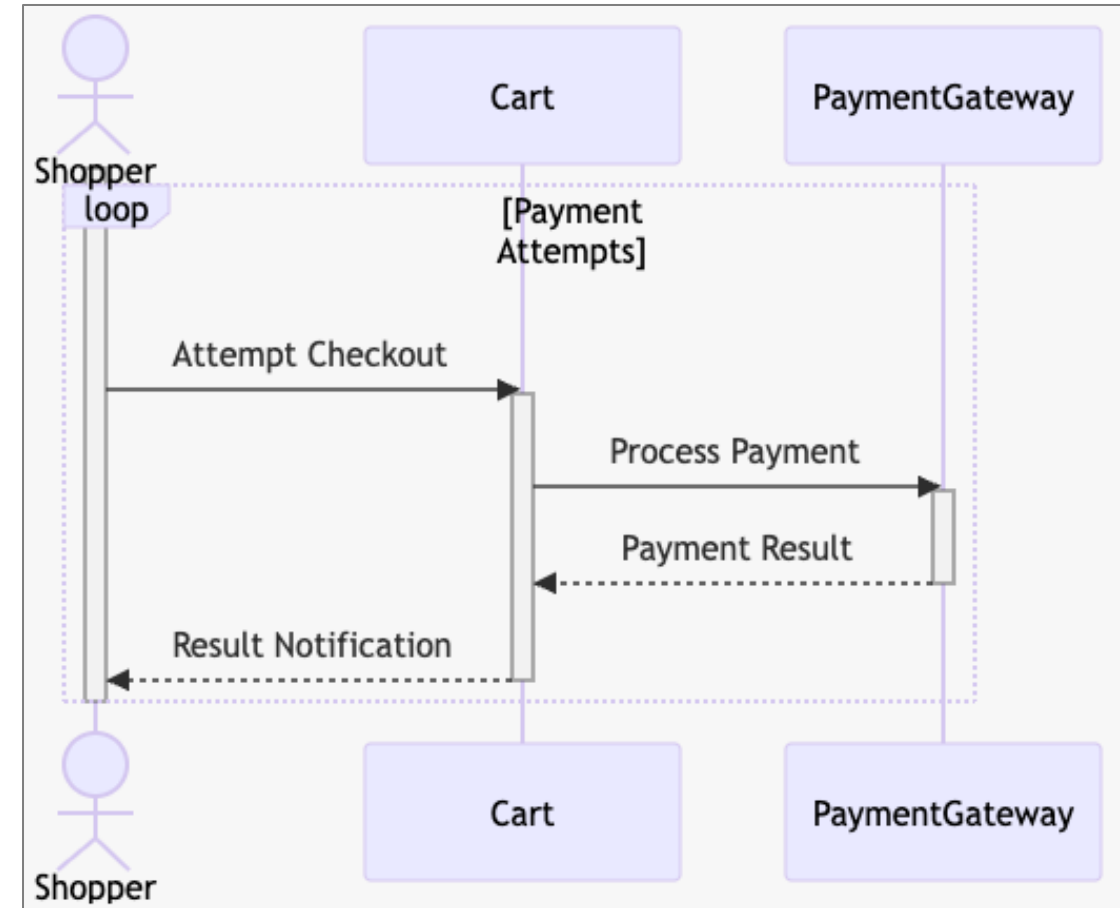
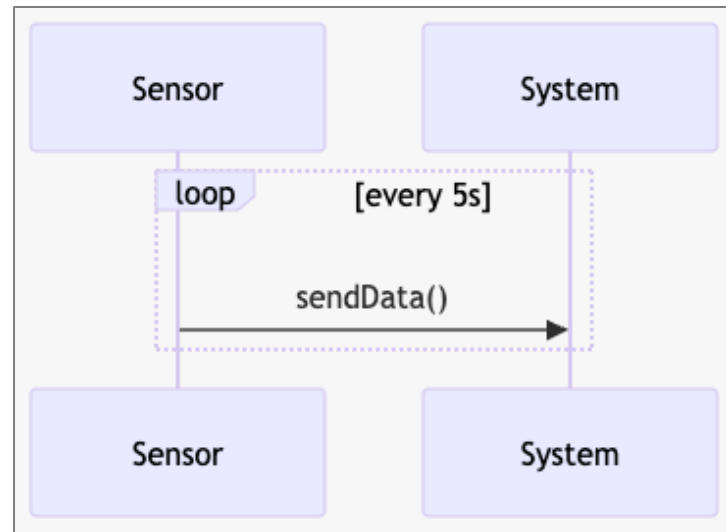
❖ **alt** represents **alternate** scenarios.



Illustrates **branching logic** based on condition results (success or failure).

Looping Interaction

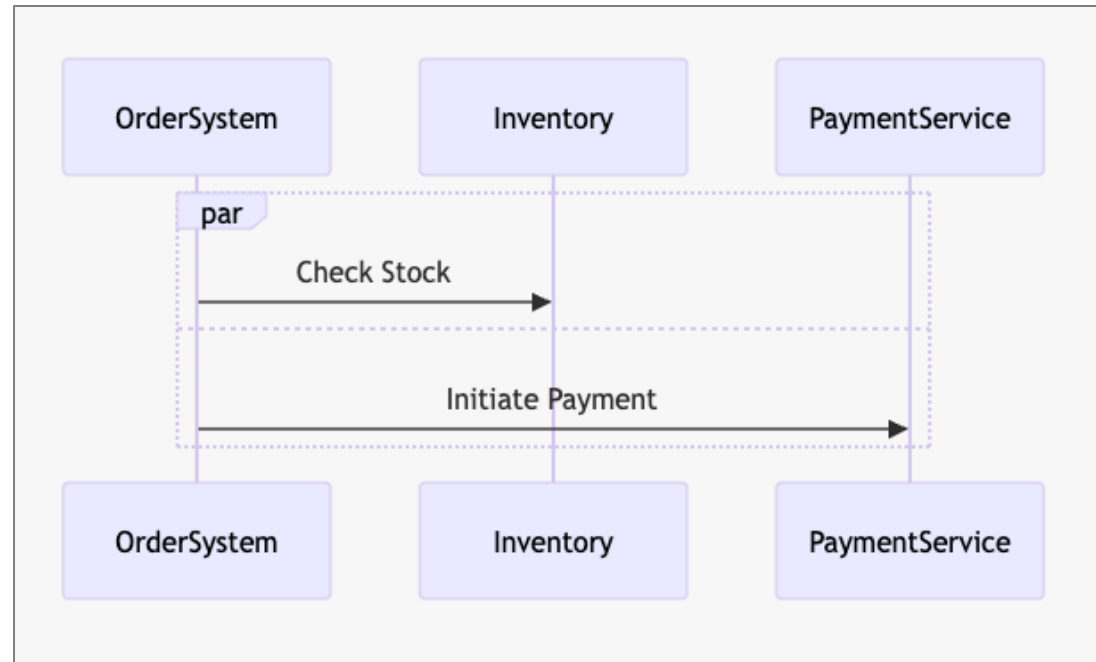
❖ **loop** represent **repeated** actions.



The loop continues until payment is successful, emphasizing **iterative** process.

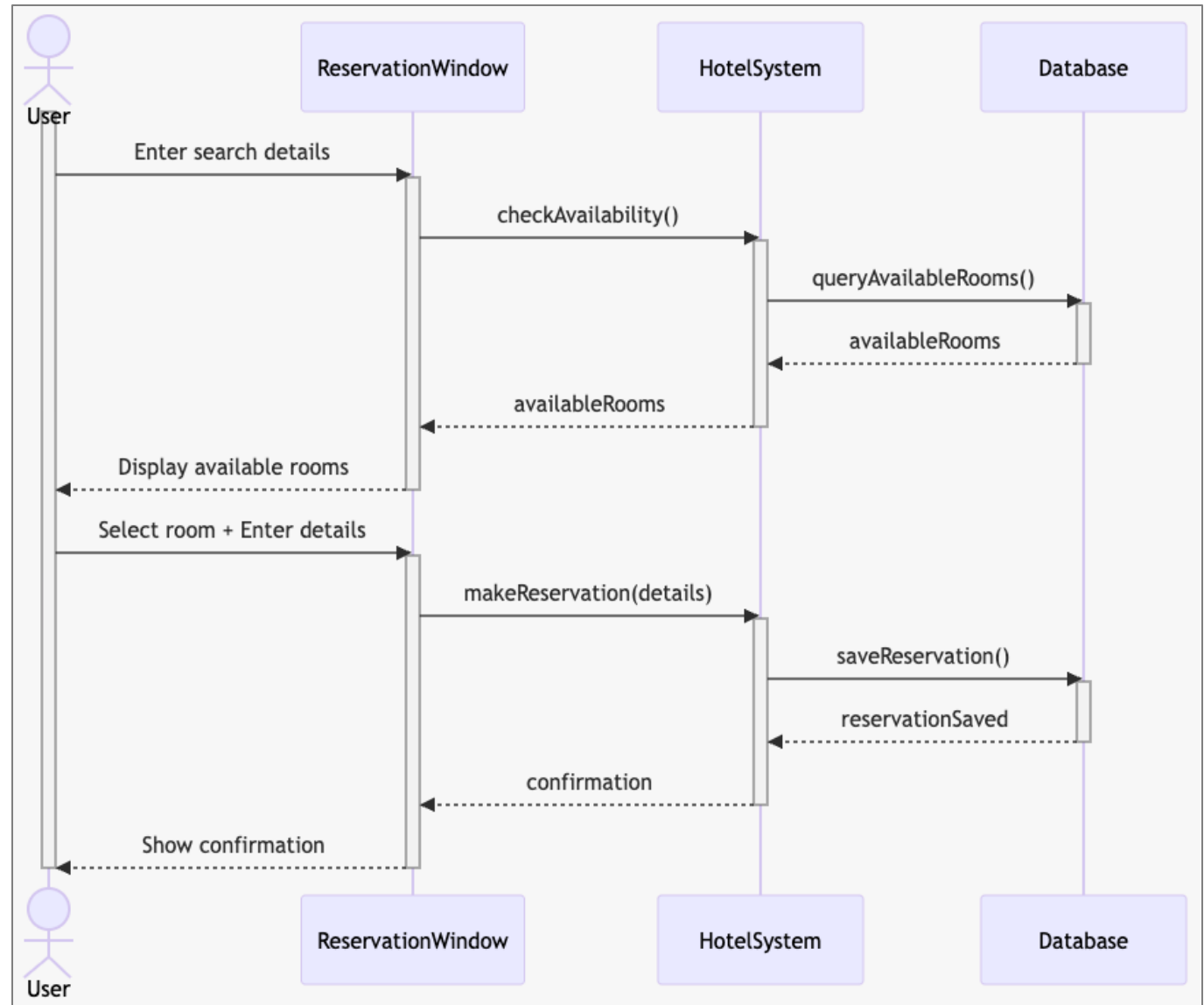
Parallel Processes

par represents **concurrent** processes.

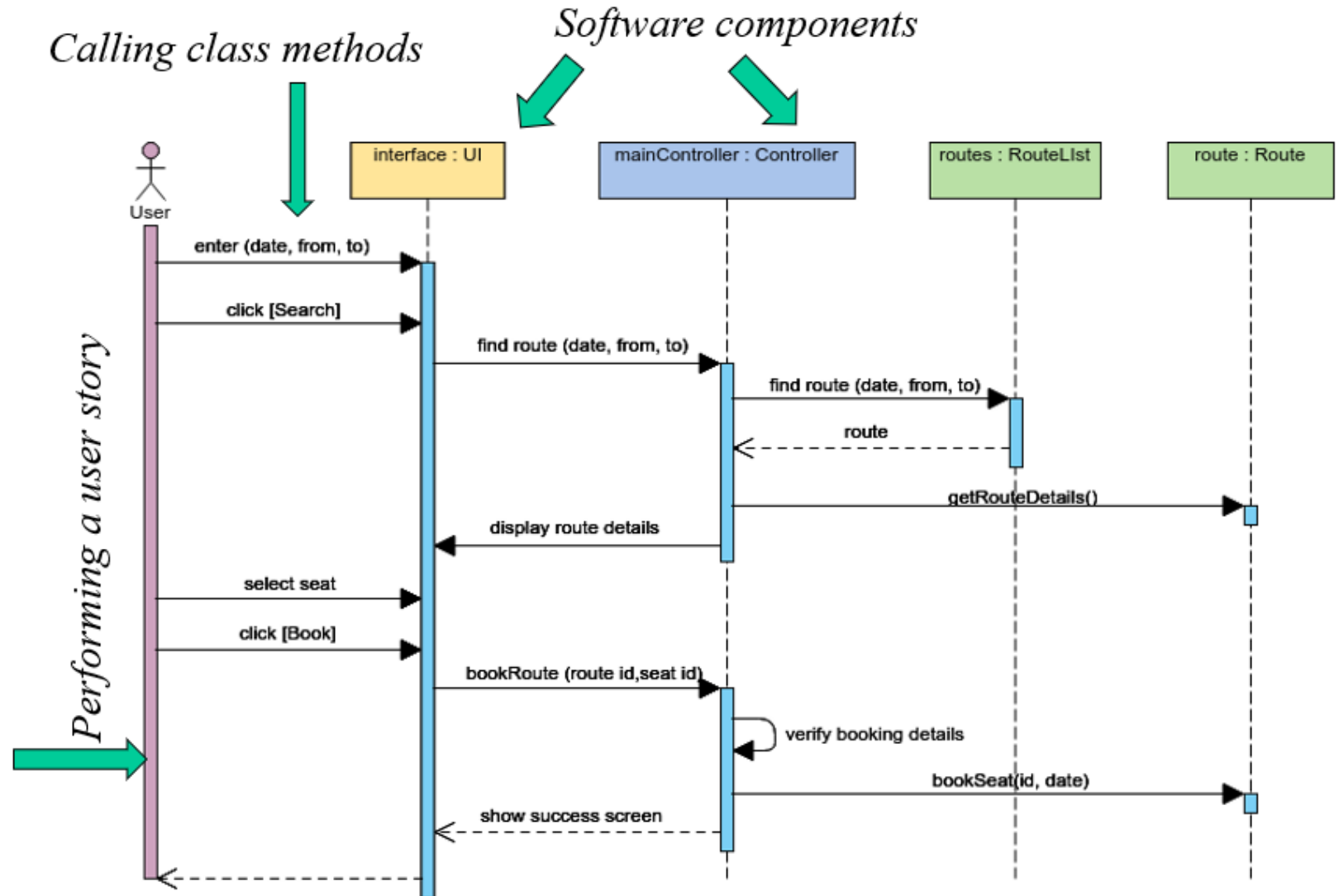


Multiple processes, such as inventory checking and payment, occur **simultaneously**.

Example: Hotel Reservation

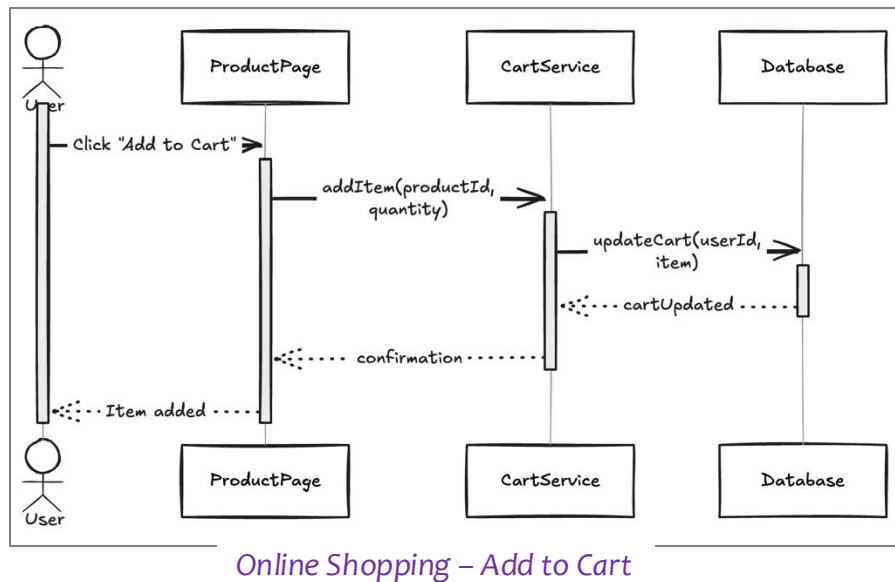
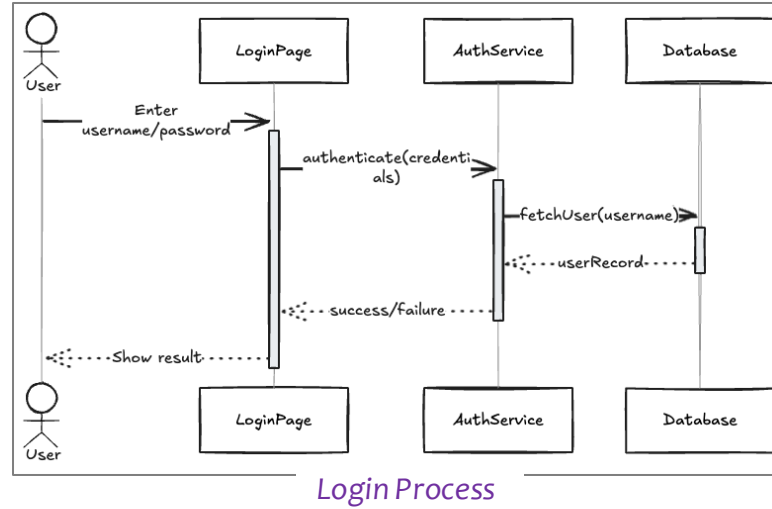


Example: Airline Booking

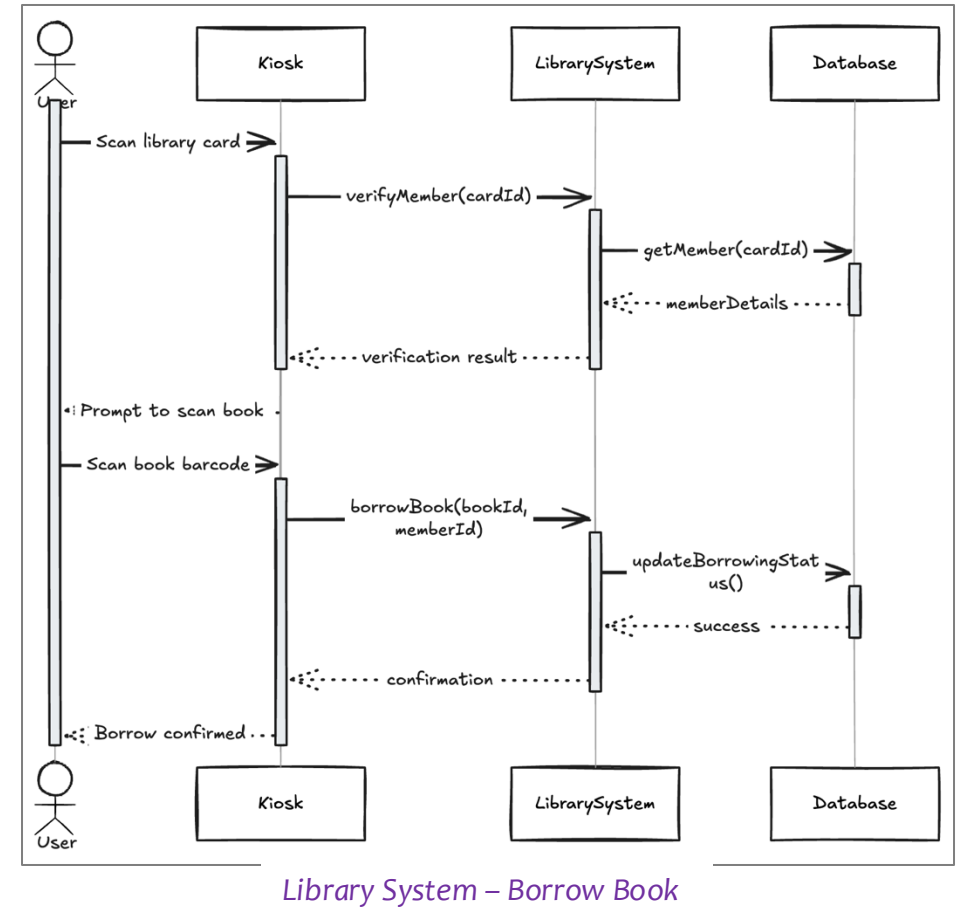


❖ More information in [The sequence diagram – IBM Developer](#)

Examples



Online Shopping – Add to Cart



Library System – Borrow Book

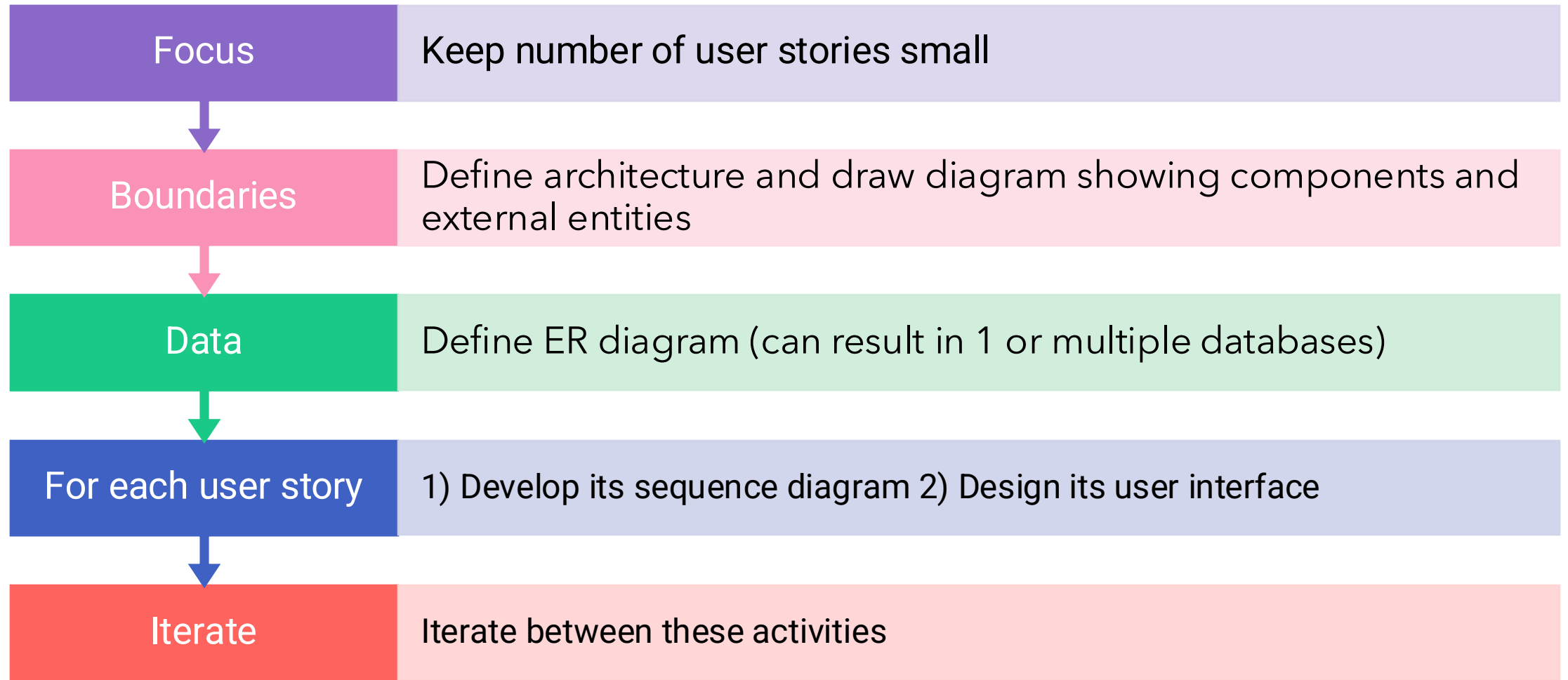
Benefits of Sequence Diagrams

- ❖ Clarifies interaction order and logic.
- ❖ Identifies inefficiencies and redundancies.
- ❖ Enhances team communication.
- ❖ Aids debugging and improves process clarity.
- ❖ Improves collaboration and understanding.

Common Mistakes

- ❖ Overcomplicating diagrams.
- ❖ Undefined roles and interactions.
- ❖ Incorrect message ordering.

Suggested Design Process in Software Engineering



Good Software Design Practices

Things to do

- Keep design documents “live” and shared between team members
- Use design as a way to decompose work
- Discuss design changes as a team

Things to avoid

- Too much focus on notation
- Quantity over quality
- Creating something for other manager (tick boxes) and forgetting design is for team



Web resources

Sequence diagrams

- [Sequence Diagram Tutorial - Complete Guide with Examples \(creately.com\)](https://creately.com/sequence-diagram-tutorial/)
- [Sequence Diagram Tutorial \(visual-paradigm.com\)](https://visual-paradigm.com/sequence-diagram-tutorial/)
- [UML Sequence Diagram Tutorial | Lucidchart](https://lucidchart.com/sequence-diagram-tutorial/)

Software design principles

- [Software Design Principles | Top 5 Principles of Software Development \(educba.com\)](https://educba.com/software-design-principles-top-5-principles-of-software-development/)

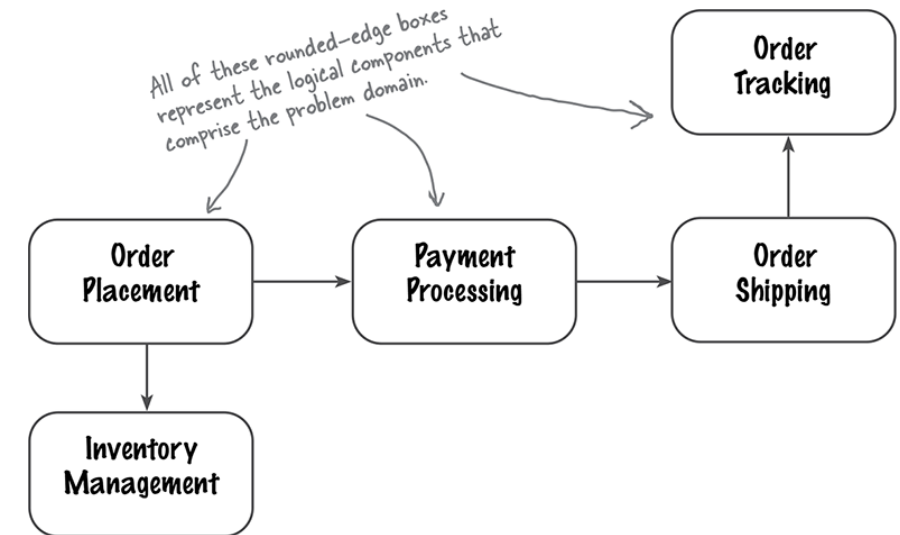
Logical Components and Modelling Using C4

COMP2511, CSE, UNSW

These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

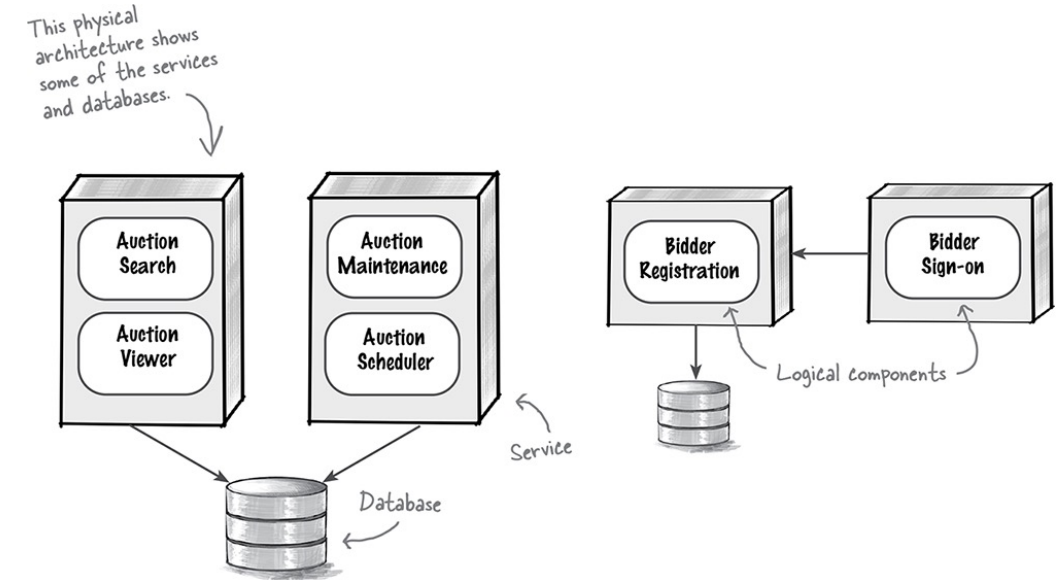
What Are Logical Components?

- ❖ **Functional building** blocks of the system
- ❖ Represent **major features** or responsibilities
- ❖ Typically **map to** folders or **modules** in the codebase



Logical vs Physical Architecture

- ❖ **Logical Architecture:** Describes what the system does (functional perspective)
- ❖ **Physical Architecture:** Describes how the system is built and deployed (technical perspective)
- ❖ Example:
 - **Logical:** Bidding, Registration, Payment
 - **Physical:** APIs, databases, gateways, services



Creating a Logical Architecture

Follow a 4-step process:

- ❖ Identify core components
 - ❖ Assign requirements
 - ❖ Analyse roles & responsibilities
 - ❖ Align with architectural characteristics
- Revisit this cycle whenever system changes are introduced

Align with Architectural Characteristics

❖ Break down or merge components based on:

- Scalability
- Availability
- Performance

❖ Example: Move bid logging to separate Bid Tracker to improve speed and availability

Component Coupling

- ❖ **Afferent** (incoming): How many depend on this component
- ❖ **Efferent** (outgoing): How many this component depends on
- ❖ **Total Coupling** = Afferent + Efferent

Goal: Keep coupling low for flexibility and maintainability

The Law of Demeter

- ❖ Also known as the **Principle of Least Knowledge**
- ❖ Each component should only interact with its **immediate neighbors**
- ❖ **Avoid tight coupling** caused by too much knowledge about the system

Coupling Trade-offs

- ❖ **Tightly Coupled** System: Easier to trace workflow, harder to change
- ❖ **Loosely Coupled** System: More maintainable, but harder to understand in one place

Remember: Everything is a **trade-off**

Logical Components: Summary

- ❖ Logical components are your system's **functional map**
- ❖ Use **descriptive names** based on responsibilities
- ❖ **Avoid** entity trap and generic components
- ❖ **Reduce** coupling using the Law of Demeter
- ❖ Regularly **reevaluate** components as requirements evolve

Introduction to C4 Architectural Modelling



Challenges in Architecture modelling

Its all about tradeoffs

- Addressing functional requirements
- Balancing non-functional requirements
- Finding a balance between “understandability” (by humans) and “correctness” (the code) is a complex undertaking, especially in cross-functional teams, where you’re explaining to a mixed group of technical and non-technical people

Iterative process

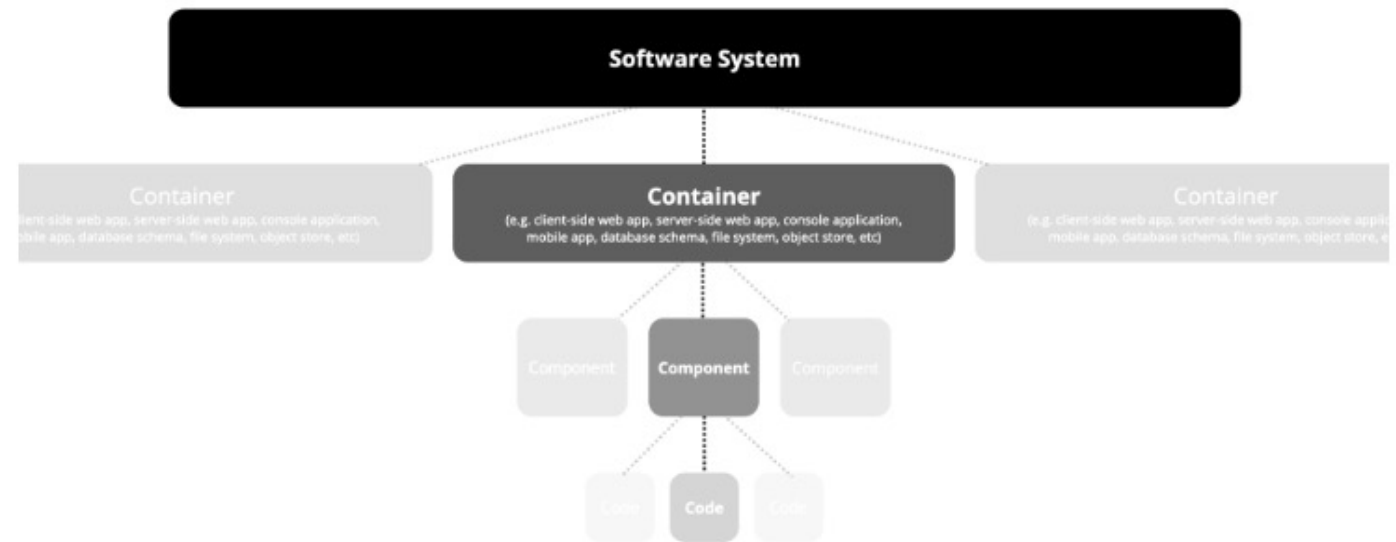
- There isn’t one but multiple software architectures
- High level architectures: closer to requirements
- Low level architectures: closer to implementations

Lack of standardization in modelling architectures

- Simple/informal => Ambiguity in meaning
- Formal (e.g. UML) => Learning curve / understandability

What is C4 ?

- ❖ Gives **names** to different design concepts
- ❖ Focuses on **intuitive visual** representations of these concepts
- ❖ Defines a set of **hierarchical diagram** arranged by levels
- ❖ **Lightweight** methodology for visual and verbal communication
- ❖ Allows more efficient conversations
- ❖ Notation independent
- ❖ Tooling independent



C4 Levels



System context level

Showing overall system + users + external systems.
Useful for Business stakeholders, execs and non-tech users



Containers level

Showing major application/components like web apps, APIs, DBs
Useful for developers, tech leads and architects.



Components level

Showing modules/services/classes within a container (e.g. routes, services, repositories)
Mainly for developers.



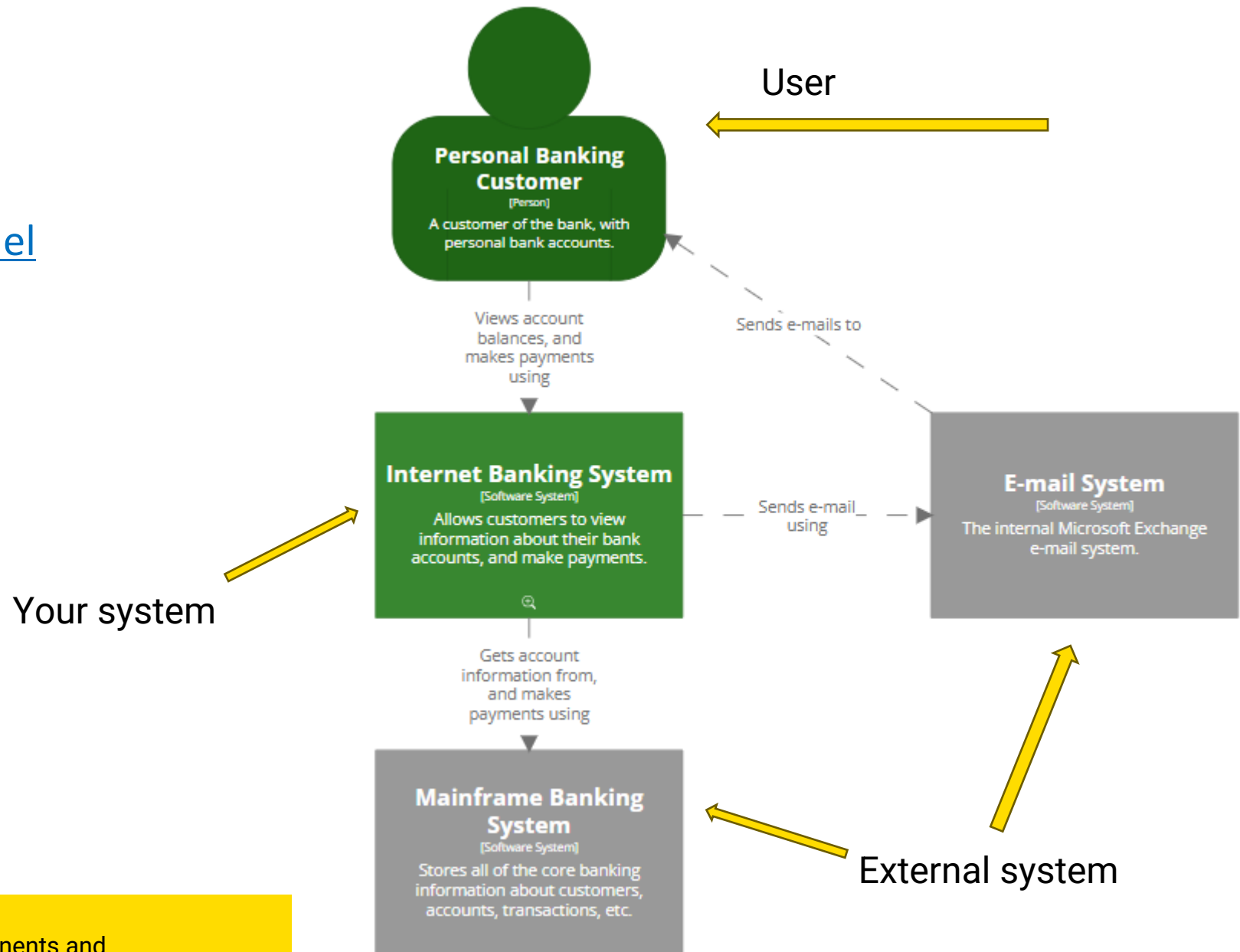
Code level

Level 1

- ❖ A context diagram is the most **general description** of what your system does
- ❖ Shows who will use it, and what other **systems it will interact with**.
- ❖ Will help you describe the **scope of your project** and help you pinpoint who the user is and what problem you're going to solve

Example

From [Example | C4 model](#)

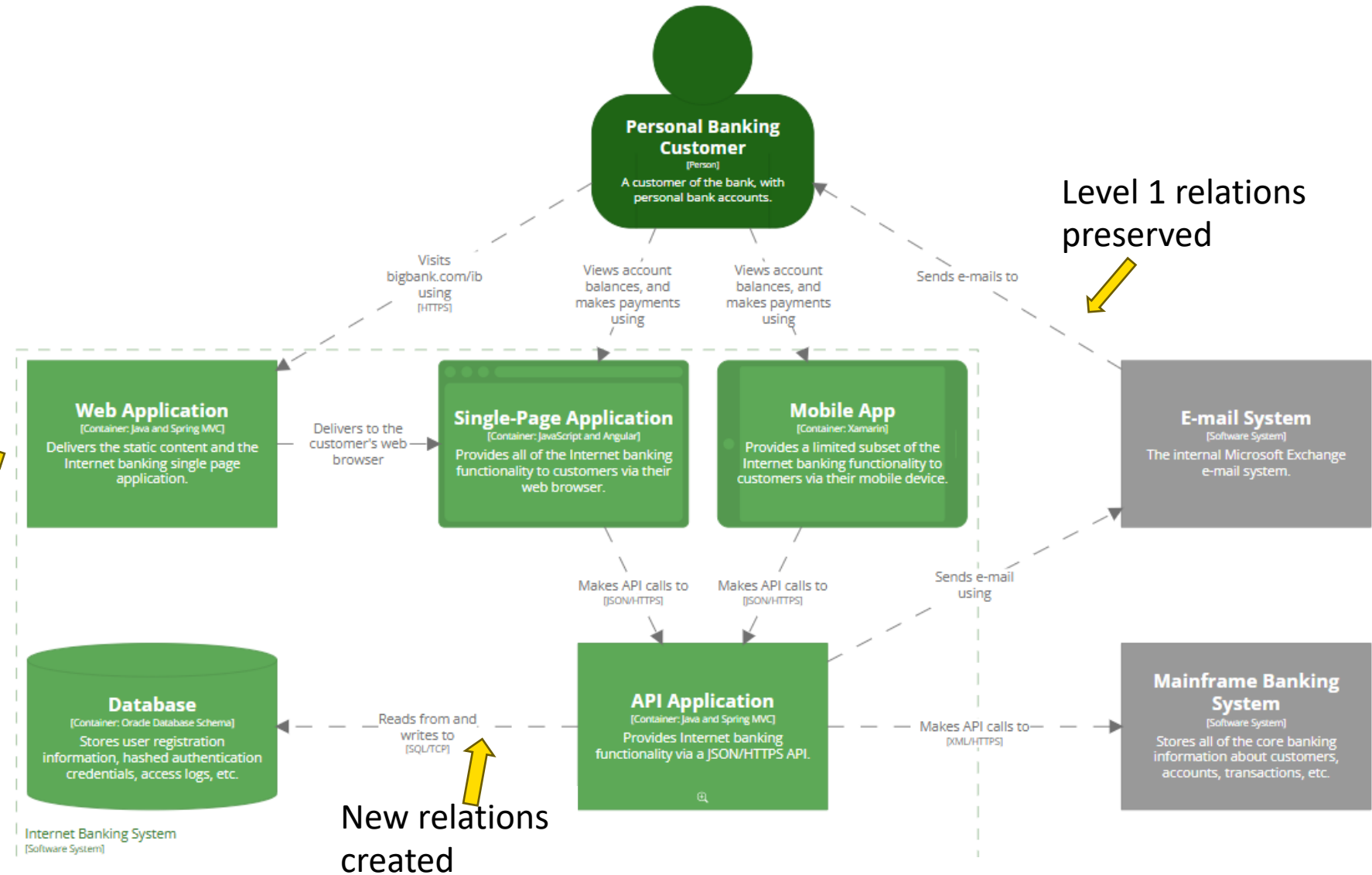


Level 2

- ❖ Container diagram takes the first step into **describing the software system** and shows the APIs, applications, databases, and microservices that the system will use.
- ❖ Each of these applications or services is represented with a **container** and the **interactions** between them are shown **at a high level**.

Example

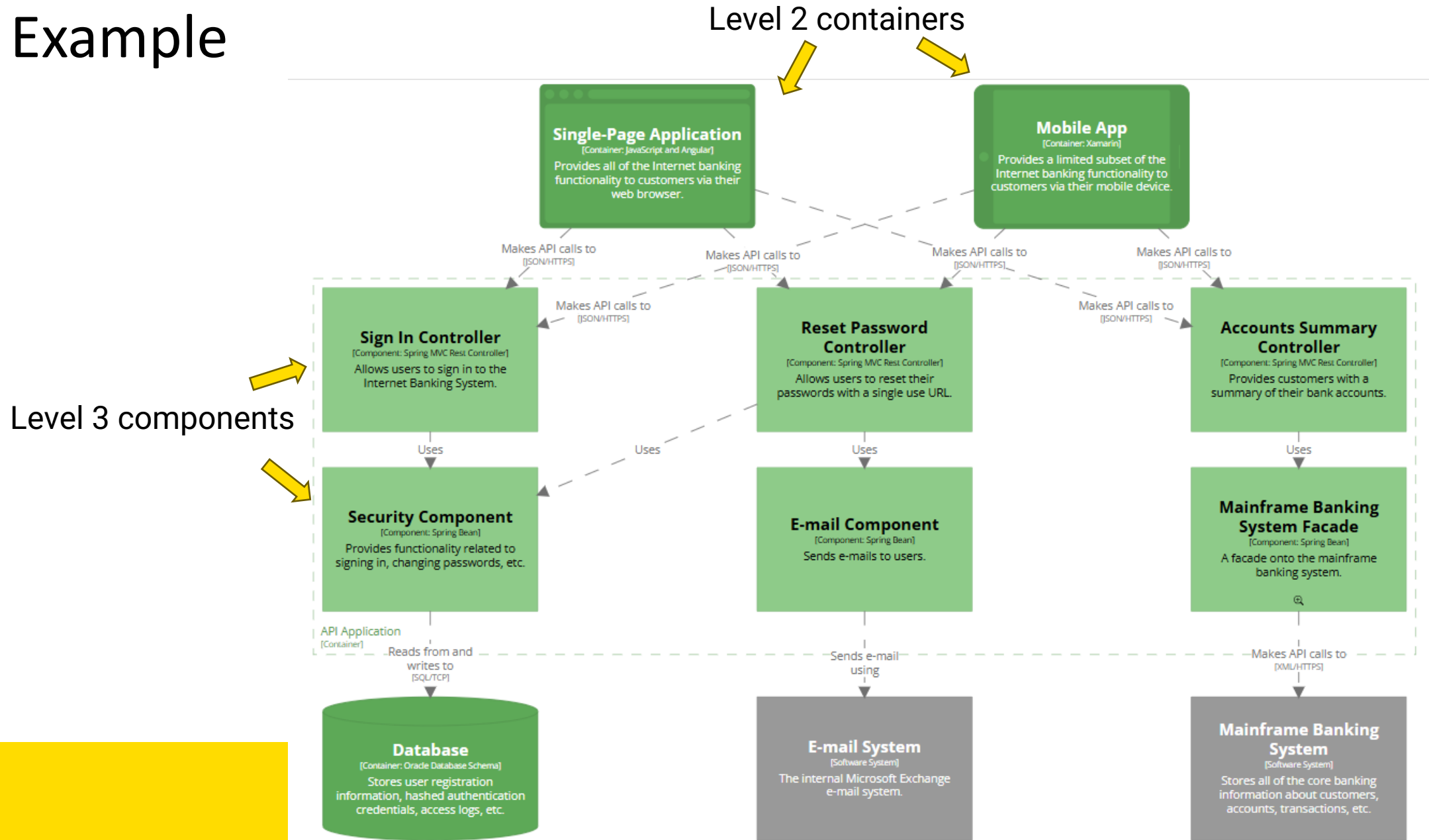
Your system decomposed into several containers



Level 3

- ❖ One step **deeper than the container** diagram, the component diagram details groups of code within a single container.
- ❖ These components represent **abstractions of your codebase**.
- ❖ Comparable to a UML component diagram but follows a **less-strict set of “rules”** in order to create the software architecture diagram.

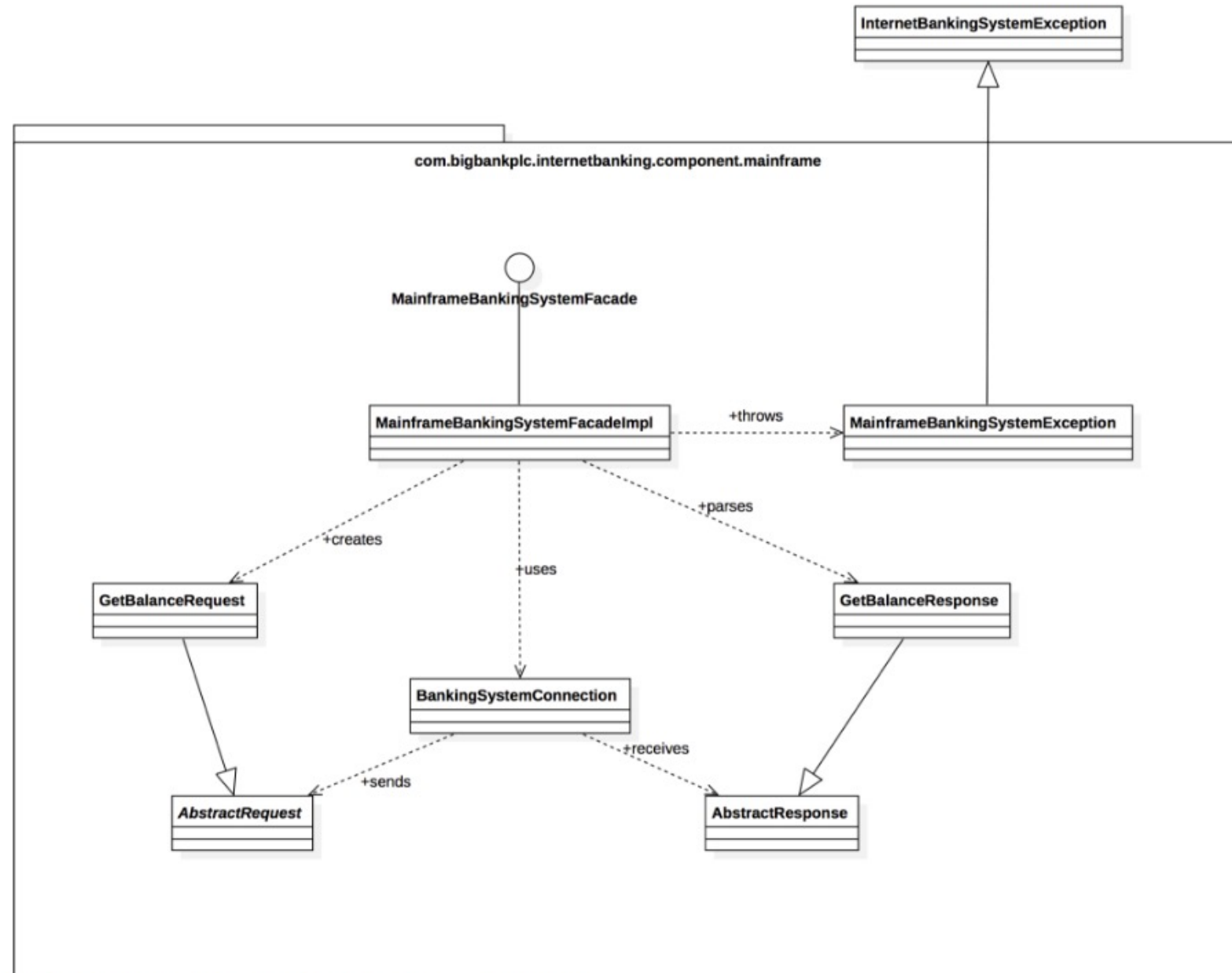
Example



Level 4

- ❖ Has **lots of detail** to show how the code of a single component is actually implemented.
- ❖ Can use a **UML** class diagram **or entity relationship** diagram that describes the component.

Example

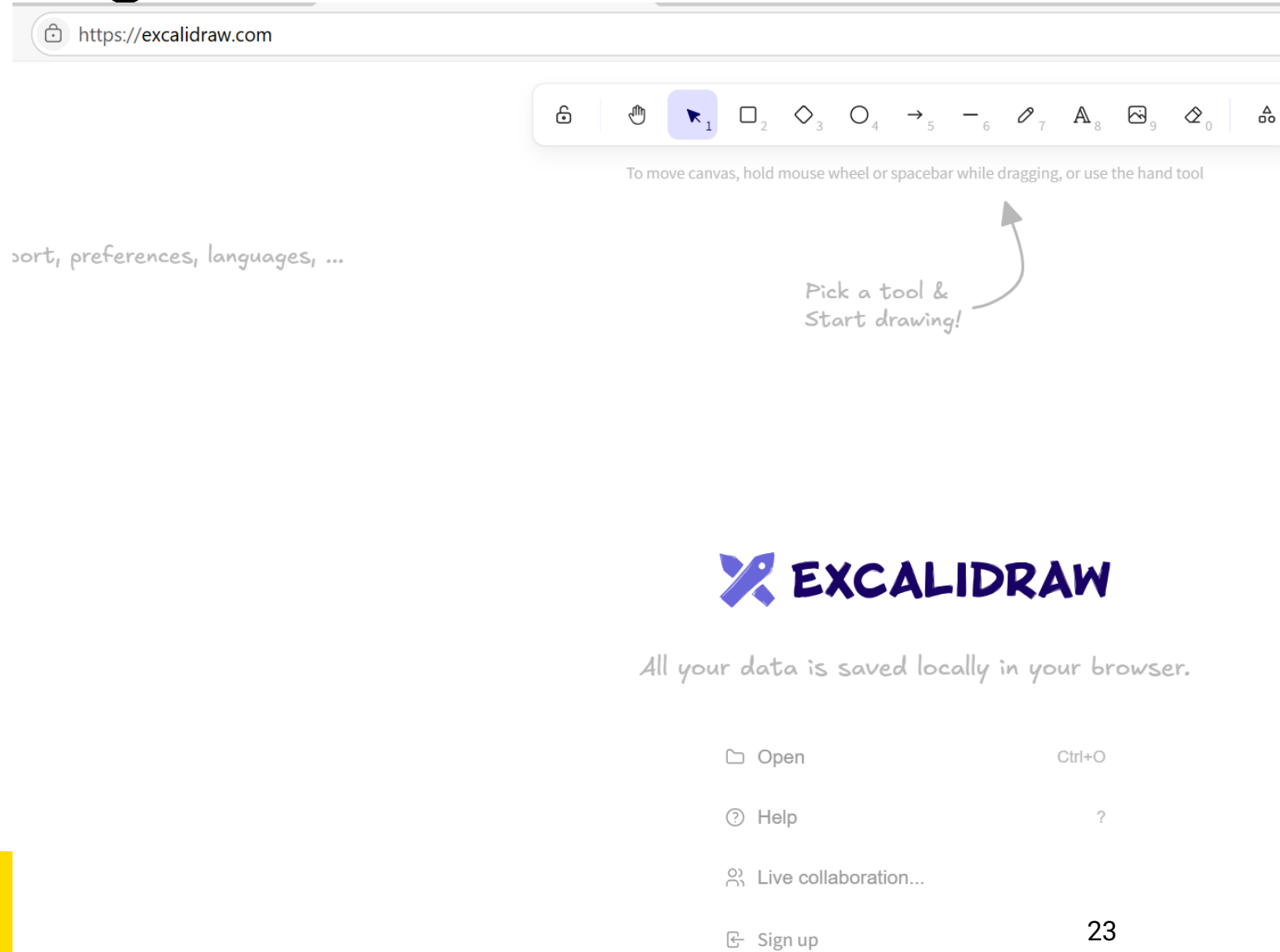


Class diagram for the Mainframe Banking System Facade component

Recommended Modelling tool

Simple one

<https://excalidraw.com/>



Excalidraw Libraries

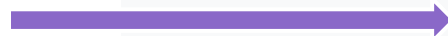
A directory of public libraries that you can easily add to [Excalidraw](#).

Follow the [instructions](#) if you want to **add your own library** into this list.

All the following libraries are under [MIT License](#).

Sort By · [New](#) · [Updated](#) · [Total Downloads](#) · [Downloads This Week](#) · [Author](#) · [Name](#)

1. Enter C4



(tip: you can type anywhere to start searching)

Hexagonal Architecture

@Armando Cordova Pelaez

↓ 7582

Created: 24 Sep 2021

Useful to diagram and learn more about Hexagonal (aka Ports and Adapters) Architecture by Alistair Cockburn and implementation by Jakub Nabrdalik. More information: <https://gist.github.com/corlaez/32707a1c41485d056c00251206435c89>



➔ Add to Excalidraw

↓ Download

C4 Architecture

@Dmitry Burnyshev

↓ 3080

Created: 24 May 2022

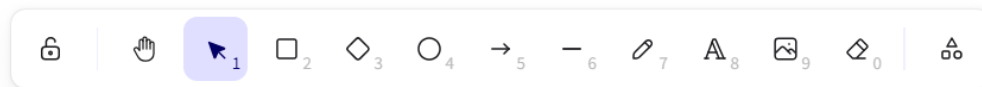
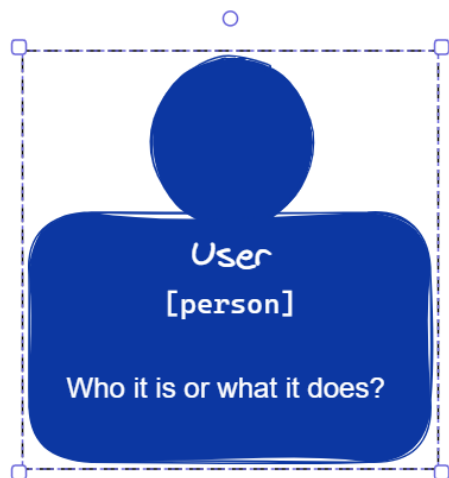
C4 Simon's Brown concept elements based on <https://c4model.com/>

Items: C4 elements, Person, Web App, Mobile App, Component, System, Existing System, Database, Group, Relation

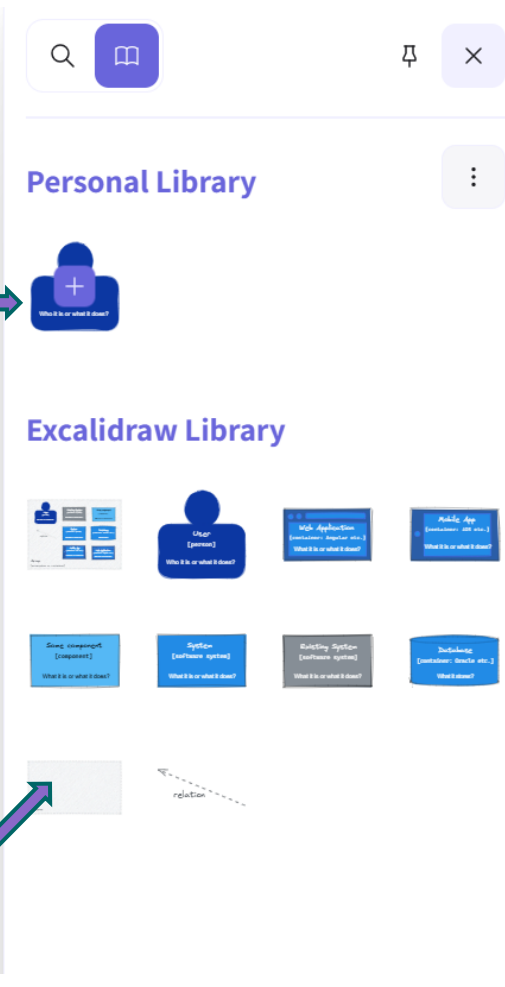


2. Add to Excalidraw

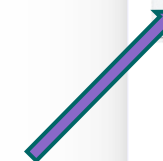




You can create
your own
shapes



You can drag
shapes into the
canvas



You can use
these shapes



Open Ctrl+O

Save to...

Export image... Ctrl+Shift+E

Live collaboration...

Command palette Ctrl+/
Find on canvas Ctrl+F

Help ?

Reset the canvas

Excalidraw+

GitHub

Follow us

Discord chat

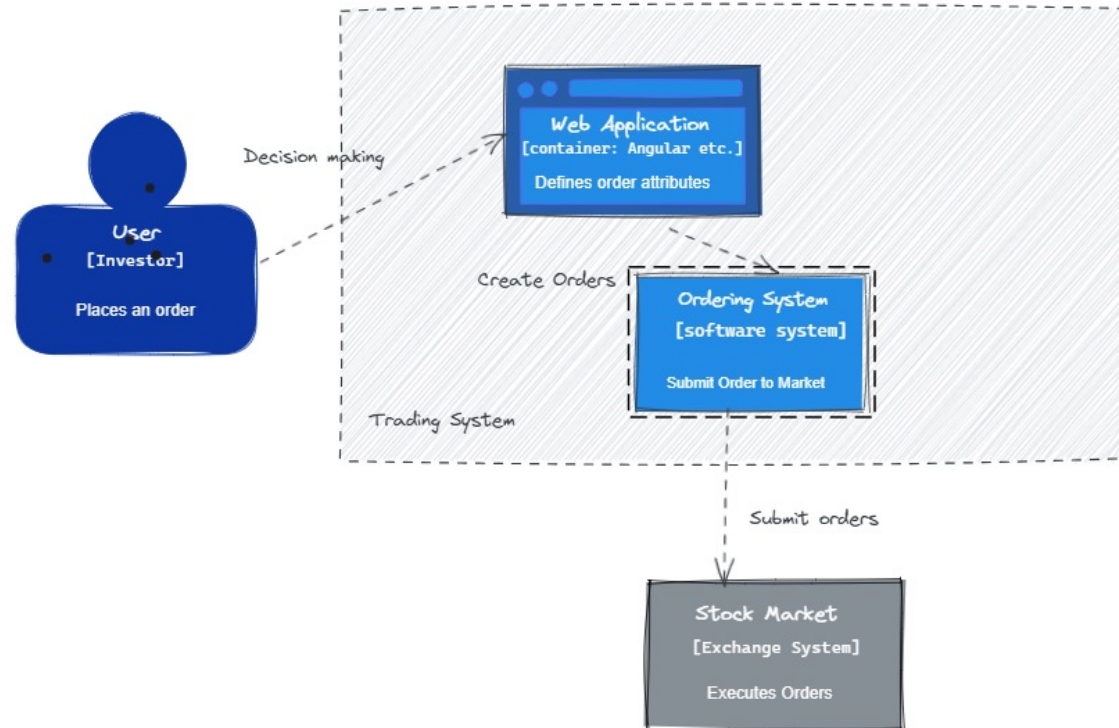
[Sign up](#)

Theme



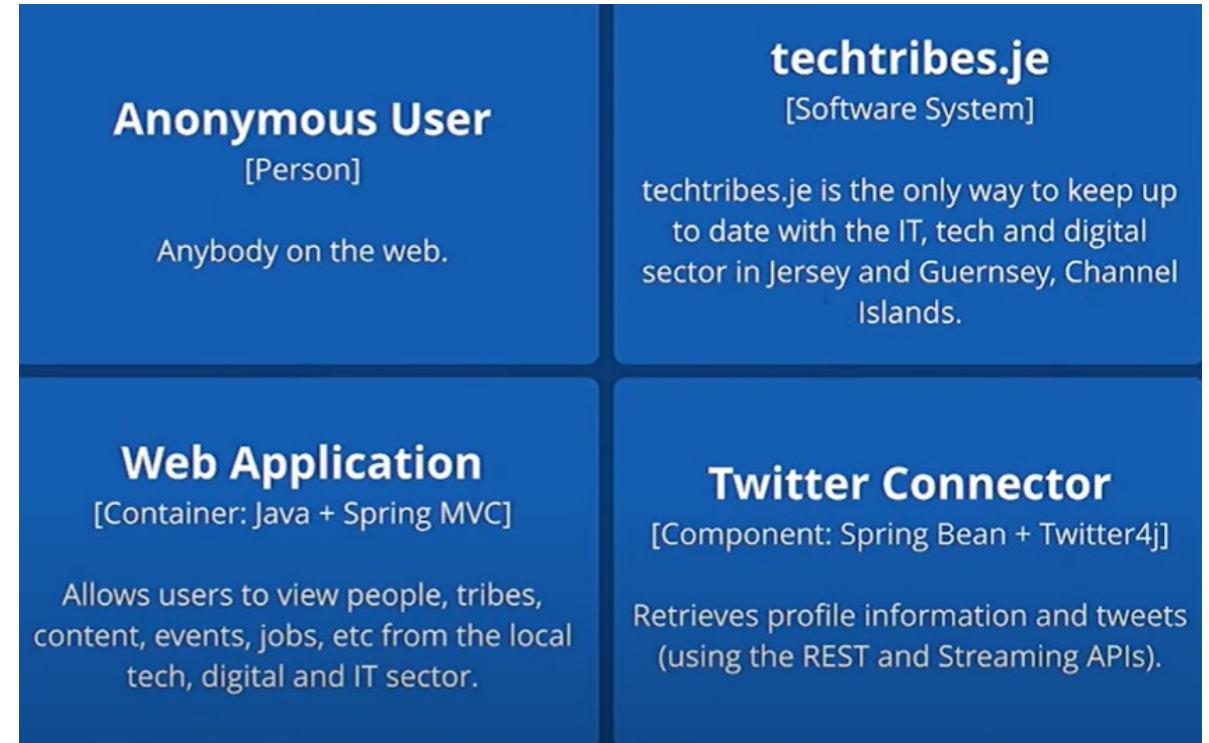
English

Canvas background



Good modelling practices

- ❖ Add a title to your diagram
- ❖ Avoid acronyms for business terms
- ❖ Consistent naming of components



Good modelling practices (cont.)

❖ Lines

- Clearly labelled
- Undirectional (follows words in boxes)

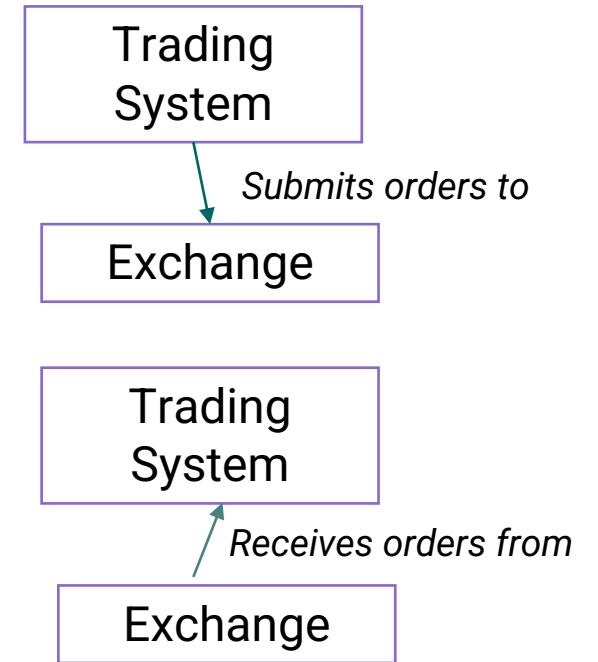
❖ Legend

- Use it for additional shapes/colours you introduce
- Also additional icons that describe components (AWS-style)
- Use it to enhance only (if remove them, diagram still makes sense)

❖ A good diagram should be self-explanatory

❖ More details in Simon Brown's video at:

<https://www.youtube.com/watch?v=x2-rSnhpw0g&t=785s>



[Review checklist | C4 model](#)

C4 Model Checklist

Software architecture diagram review checklist

General

Does the diagram have a title?	Yes	No
Do you understand what the diagram type is?	Yes	No
Do you understand what the diagram scope is?	Yes	No
Does the diagram have a key/legend?	Yes	No

Elements

Does every element have a name?	Yes	No
Do you understand the type of every element? (i.e. the level of abstraction; e.g. software system, container, etc)	Yes	No
Do you understand what every element does?	Yes	No
Where applicable, do you understand the technology choices associated with every element?	Yes	No
Do you understand the meaning of all acronyms and abbreviations used?	Yes	No
Do you understand the meaning of all colours used?	Yes	No
Do you understand the meaning of all shapes used?	Yes	No
Do you understand the meaning of all icons used?	Yes	No

Resources

C4 Model: <https://c4model.com/abstractions>

Tutorial video: https://www.youtube.com/watch?v=x2-rSnhpw0g&t=785s&ab_channel=AgileontheBeach

Articles

- Should you use the C4 model for system architecture design? <https://icepanel.medium.com/c4-model-for-system-architecture-design-225e00ebbd9>
- C4 model for system architecture design <https://icepanel.medium.com/c4-model-for-system-architecture-design-225e00ebbd9>

Other tools

- Flowchart maker <https://app.diagrams.net/>
- Open source tool <https://plantuml.com/>
- Lucid Charts <https://www.lucidchart.com/blog/c4-model>
- Gliffy <https://www.gliffy.com/blog/c4-model>

END

Architectural Styles

COMP2511, CSE, UNSW



UNSW
SYDNEY

These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

Introduction to Architectural Styles

❖ Architectural Styles:

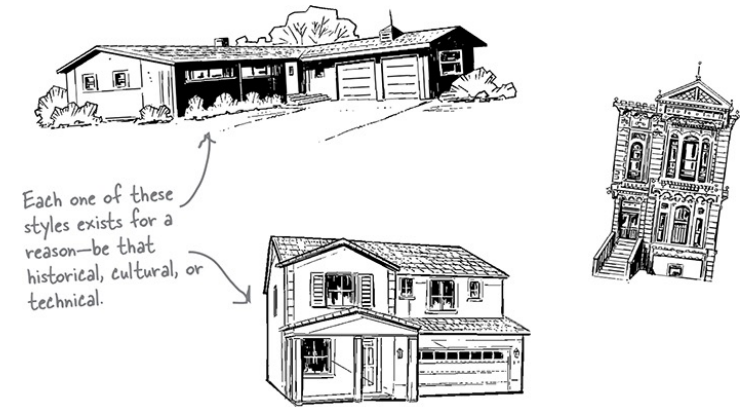
- Predefined **patterns** and philosophies guiding how software systems are structured and deployed.

❖ Importance of Understanding Styles:

- **Facilitates** better design decisions.
- **Aligns** software architecture with project needs.

❖ Example:

- Residential housing styles influenced by geography, climate, personal preference. Similarly, software architecture varies by project requirements.



Categorizing Architectural Styles

Two main categories for architectural styles:

1. Partitioning

- Technical vs. Domain-based.

2. Deployment

- Monolithic vs. Distributed.

❖ Why Categorize?

- Helps systematically analyse and select appropriate architecture.

		Partitioning	
		Technical	Domain
Deployment model	Monolith	Layered Microkernel	Modular monolith
	Distributed	Event-driven	Microservices

Partitioning by Technical Concerns

Technical Partitioning:

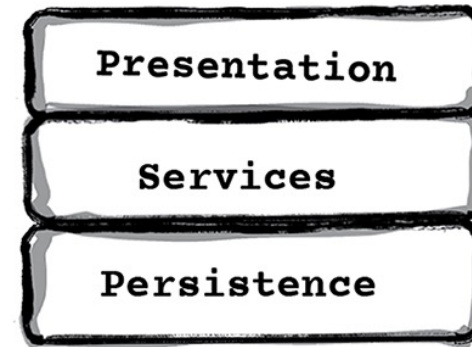
- Code organized by functional roles or technical layers.

Characteristics:

- Clear separation of responsibilities.
- Easier specialization of teams.

Example: A standard web application:

- Presentation Layer (UI);
- Business Logic Layer (Services)
- Data Persistence Layer (Database)



- Real-world Analogy:
Roles in a fancy restaurant (host, server, chef, busser) clearly divided by technical concerns (greeting, cooking, cleaning).

Partitioning by Domain Concerns

Domain Partitioning:

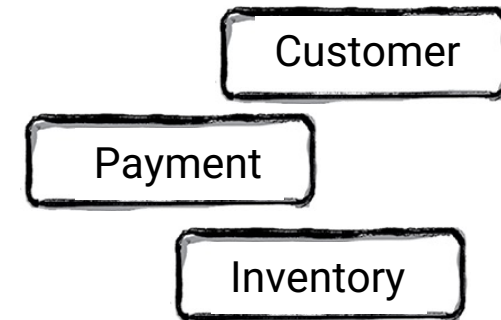
- Code organized around business domains or problem areas.

Characteristics:

- Alignment with business goals.
- Easier maintenance of related features.
- Strong domain modeling.

Example: An e-commerce platform:

- Customer Domain (user accounts, user interface)
- Inventory Domain (product catalog, stock management)
- Payment Domain (billing, transactions)



➤ **Real-world Analogy:**
Food court restaurants, each specialised in distinct cuisines (pizza, salads, burgers).

Comparing Technical vs. Domain Partitioning

Technical Partitioning	Domain Partitioning
Layered by technical roles	Organized by business areas
Easier for specialised teams	Aligned closely with business needs
Risk of over-generalisation	Risk of duplicating common functionalities

Example Scenario: A banking application:

- **Technical:** Separate teams for frontend, backend, DB administration.
- **Domain:** Separate teams for loans, investments, account management.

Deployment Models Overview

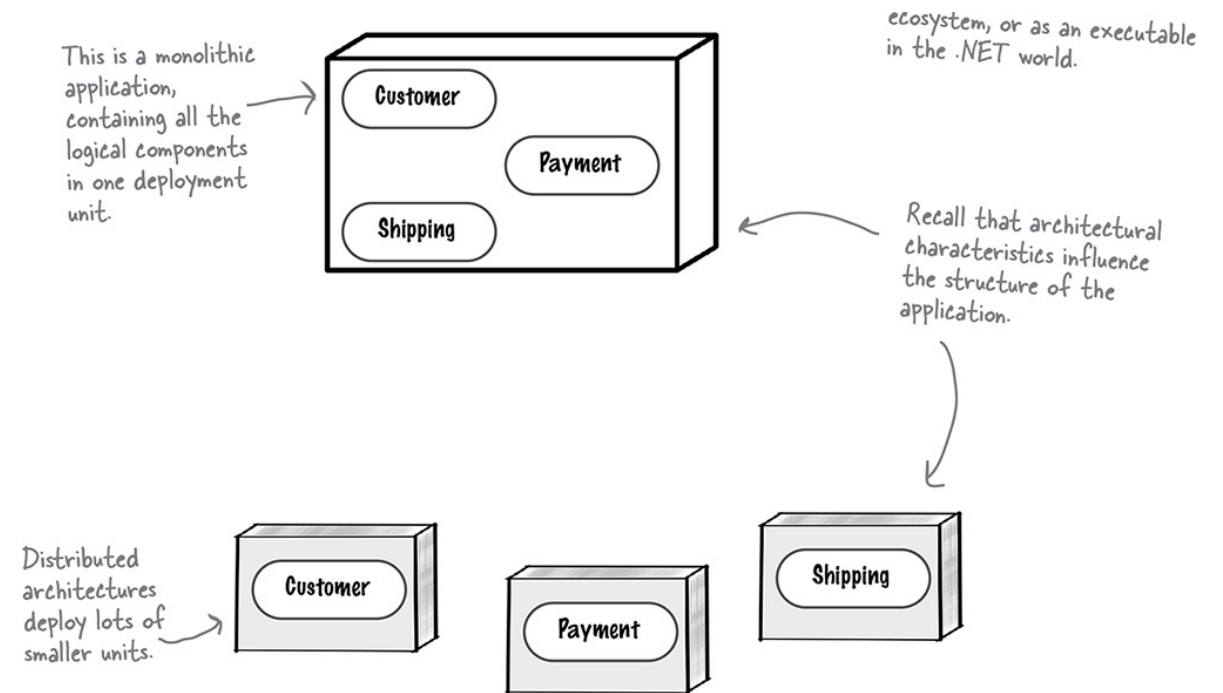
1. Monolithic Architecture

- Single deployable unit.

2. Distributed Architecture

- Multiple deployable units communicating over networks.

Choice affects scalability, complexity, and cost.



Monolithic Architecture – Overview and Pros

Monolithic:

- Entire application deployed as one single executable or package.

Pros:

- Easier initial development.
- Simplified debugging.
- Lower initial deployment cost.

Examples:

- A single .jar (Java) or .exe (.NET) containing all app logic and resources.
- Smartphone as a single device doing many functions (calling, browsing, tracking).



simplicity

Typically, monolithic applications have a single codebase, which makes them easier to develop and to understand.



cost

Monoliths are cheaper to build and operate because they tend to be simpler and require less infrastructure.



feasibility

Rushing to market? Monoliths are simple and relatively cheap, freeing you to experiment and deliver systems faster.



reliability

A monolith is an island. It makes few or no network calls, which usually means more reliable applications.



debuggability

If you spot a bug or get an error stack trace, debugging is easy, since all the code is in one place.

These are just a few of the many things monoliths are good at.

Keep an eye out for this point when we discuss cons on the next page.

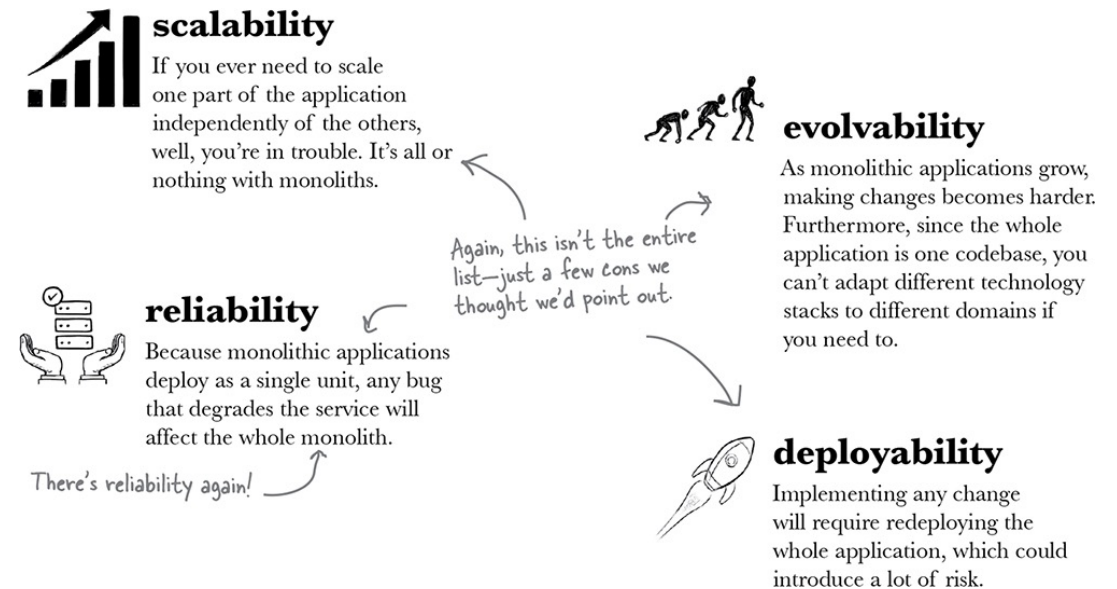
Monolithic Architecture - Limitations

Cons:

- Difficult to scale independently.
- Single bug can disrupt entire system.
- Inflexible when adapting to changing demands.

Example:

- Scaling a monolithic online store application
- Scaling means duplicating the entire application, increasing resource consumption significantly.



Distributed Architecture - Overview

Distributed:

- Application components deployed separately, each as individual processes/services.

Pros:

- Independent scalability of components.
- Encourages modular design.
- Fault isolation—failures affect only single units.

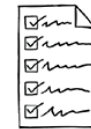
Example:

- Microservices architecture for Netflix or Amazon, allowing independent scaling of services like user management, video streaming, and recommendation systems.



scalability

Distributed architectures deploy different logical components separately from one another. Need to scale one? Go ahead!



testability

Each deployment only serves a select group of logical components. This makes testing a lot easier—even as the application grows.

Distributed architectures are a lot more testable than monolithic applications.



fault tolerance

Even if one piece of the system fails, the rest of the system can continue functioning.



modularity

Distributed architectures encourage a high degree of modularity because their logical components must be loosely coupled.



deployability

Distributed architectures encourage lots of small units. They evolved after modern engineering principles like continuous integration, continuous deployments, and automated testing became the norm.

Having lots of small units with good testability reduces the risk associated with deploying changes.

Distributed Architecture - Challenges

Cons:

- High complexity due to network dependence.
- Increased maintenance and debugging complexity.
- Higher infrastructure and operational costs.

Example:

- Managing distributed transactions across services—complex coordination required, increased risk of partial failures.

Real-world Analogy:

- Earlier days—separate devices for GPS, web browsing, and phone calls each required separate maintenance and integration.



performance

Distributed architectures involve lots of small services that communicate with each other over the network to do their work. This can affect performance, and although there are ways to improve this, it's certainly something you should keep in mind.



cost

Deploying multiple units means more servers. Not to mention, these services need to talk to one another—which entails setting up and maintaining network infrastructure.



simplicity

Distributed systems are the *opposite* of simple. Everything from understanding how they work to debugging errors becomes challenging.

We cannot emphasize enough how complex distributed architectures can be!

Debugging distributed systems involves thinking deeply about logging, and usually requires aggregating logs. This also adds to the cost.



debuggability

Errors could happen in any service involved in servicing a request. Since logical components are deployed in separate units, tracing errors can get very tricky.

Introduction to Fallacies of Distributed Computing

- ❖ Originated at Sun Microsystems in 1994
- ❖ Common false assumptions about networks
- ❖ Crucial for architects of distributed systems
- ❖ 11 total fallacies (8 classical + 3 additional)

Fallacy #1 - The Network Is Reliable

- ❖ Reality: Networks can and do fail
- ❖ Impact: Services might be healthy but unreachable
- ❖ Mitigation:
 - Use timeouts
 - Retry policies
- ❖ Example: Service A sends request to Service B → no response due to intermittent network issue

Fallacy #2 - Latency Is Zero

- ❖ Reality: Remote calls take milliseconds, not microseconds
- ❖ Impact: Chained service calls can add significant delay
- ❖ Mitigation:
 - Monitor 95th-99th percentile latency
 - Minimise unnecessary calls
- ❖ Example: 10 chained calls with 100ms each = 1s delay

Fallacy #3 - Bandwidth Is Infinite

- ❖ Reality: Bandwidth is limited, especially under load
- ❖ Impact: Excessive inter-service communication slows the system
- ❖ Mitigation:
 - minimizing the passing of large, complex data structures
- ❖ Example: Returning 500KB when only 200B needed → 1Gbps load for 2k req/s

Fallacy #4 - The Network Is Secure

- ❖ Reality: More endpoints = higher attack surface
- ❖ Impact: Inter-service communication can be vulnerable
- ❖ Mitigation:
 - Zero-trust architecture
 - Secure each endpoint
- ❖ Example: Internal services hacked due to open port

Fallacy #5 - Topology Never Changes

- ❖ Reality: Network topology evolves frequently
- ❖ Impact: Latency assumptions break
- ❖ Mitigation:
 - Coordinate with network teams
 - Use adaptive timeout policies
- ❖ Example: Sunday network upgrade → production timeouts Monday

Fallacy #6 - There Is Only One Administrator

- ❖ Reality: Multiple admins across departments
- ❖ Impact: Miscommunication and missed changes
- ❖ Mitigation:
 - Maintain a clear contact directory
 - Standardize change coordination
- ❖ Example: Change in one subnet unknowingly affects dependent service

Fallacy #7 - Transport Cost Is Zero

- ❖ Reality: Infrastructure and routing costs add up
- ❖ Impact: Distributed systems are more expensive
- ❖ Mitigation:
 - Assess total cost of ownership (TCO)
 - Consider hybrid designs
- ❖ Example: Simple REST call needs new proxies, firewalls, gateway

Fallacy #8 - The Network Is Homogeneous

- ❖ Reality: Different vendors, firmware, configurations
- ❖ Impact: Compatibility and packet loss
- ❖ Mitigation:
 - Test network assumptions regularly
 - Avoid hard dependencies on vendor features
- ❖ Example: Packet loss between Cisco and Juniper segments

Fallacy #9 - Versioning Is Easy

- ❖ Reality: Supporting multiple versions is hard
- ❖ Impact: Contract proliferation, test complexity
- ❖ Mitigation:
 - Limit concurrent versions
 - Use deprecation plans
- ❖ Example: Team supports 7 versions of same API endpoint

Fallacy #10 - Compensating Updates Always Work

- ❖ Reality: Rollbacks can fail too
- ❖ Impact: Data inconsistency
- ❖ Mitigation:
 - Design for idempotency
 - Include recovery mechanisms
- ❖ Example: Order placed, and rollback fails → duplicated state

Fallacy #11 - Observability Is Optional

- ❖ Reality: Without observability, debugging is impossible
- ❖ Impact: Silent failures across services
- ❖ Mitigation:
 - Centralized logging
 - Distributed tracing
 - E.g., *OpenTelemetry*: open-source framework for collecting, processing, and exporting telemetry data (traces, metrics, and logs) from cloud-native applications and infrastructure.
- ❖ Example: Request times out without any log trail

Fallacy - Summary and Implications

- ❖ Fallacies reveal key weaknesses in distributed systems
- ❖ Addressing them improves resilience and clarity
- ❖ Must be communicated to development and operations teams
- ❖ Good architecture anticipates and mitigates these assumptions

Comparing Monolithic vs. Distributed

Monolithic	Distributed
Simpler development & debugging	Complex system integration
Lower initial costs	Higher upfront infrastructure cost
Scaling is all-or-nothing	Individual services scalable
Single failure disrupts whole system	Fault tolerance through isolation

Discussion - Regulatory and Compliance Needs

Consider special needs like:

- Regulatory compliance (e.g., financial industry).
- Security requirements.

Monolithic:

- Easier control and monitoring in regulated environments.

Distributed:

- Can complicate compliance but increases modularity and maintainability.

Example:

- Banking systems might use **monolithic** for core banking due to tight regulatory controls, however **distributed** services for customer engagement modules.

Key Takeaways

- ❖ Numerous architectural styles exist; each with **unique** characteristics and **trade-offs**.
- ❖ **Partitioning styles**: Technical vs. Domain.
- ❖ **Deployment models**: Monolithic vs. Distributed.
- ❖ Choice of style **influenced by**:
 - Project goals.
 - Scalability requirements.
 - Complexity management.
 - Cost implications.

Layered Architecture

COMP2511, CSE, UNSW



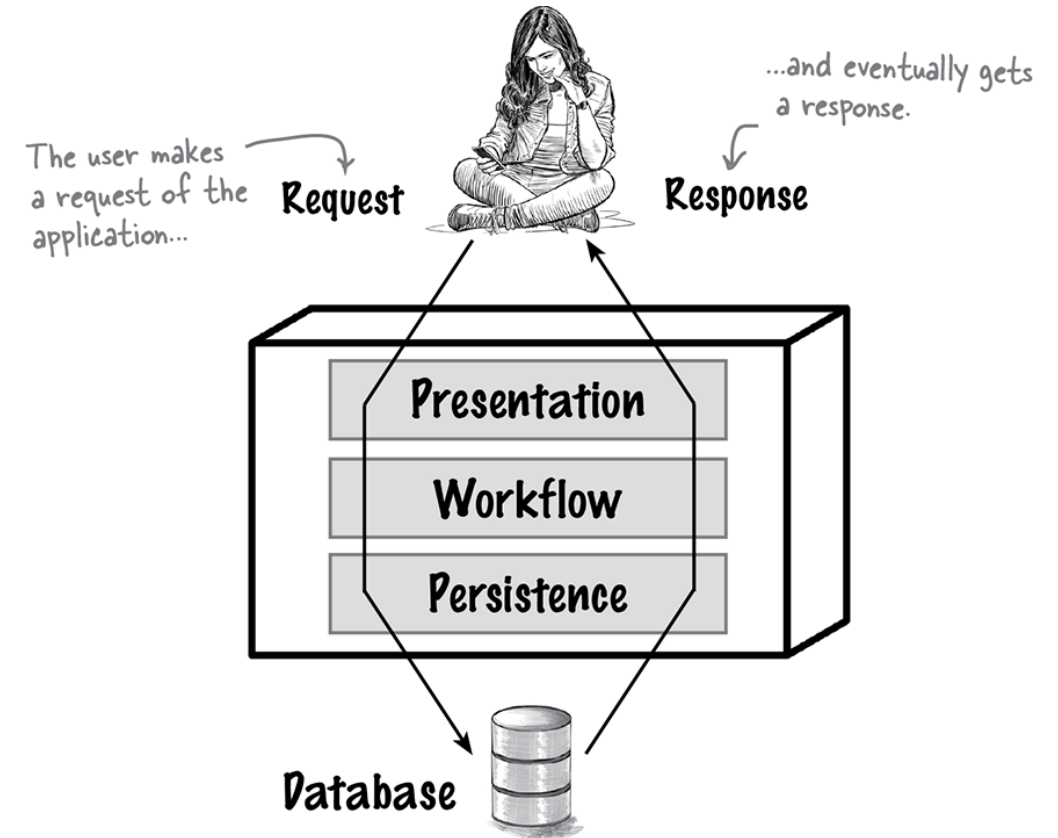
UNSW
SYDNEY

Introduction to Layered Architecture

- ❖ Layered Architecture **separates** technical **responsibilities** into **distinct layers**.
- ❖ **Simplifies** the design by dividing the system into manageable, **logical parts**.

Key benefits:

- **Easy** to understand and implement.
- Promotes reuse and **separation of concerns**.

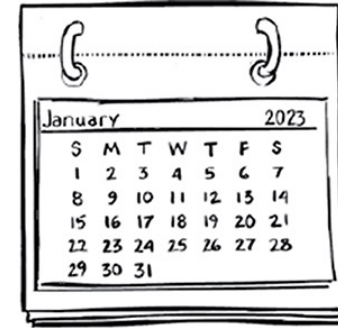


Case Study: Naan & Pop Restaurant

- ❖ **Startup** restaurant serving Indian-inspired flatbread sandwiches.
- ❖ Needs a **simple website** for online ordering quickly.

Requirements:

- **Time to market:** Quick launch.
- **Separation of responsibilities:** Clear division for UI specialists and database administrators.
- **Extensible:** Allow future enhancements easily.

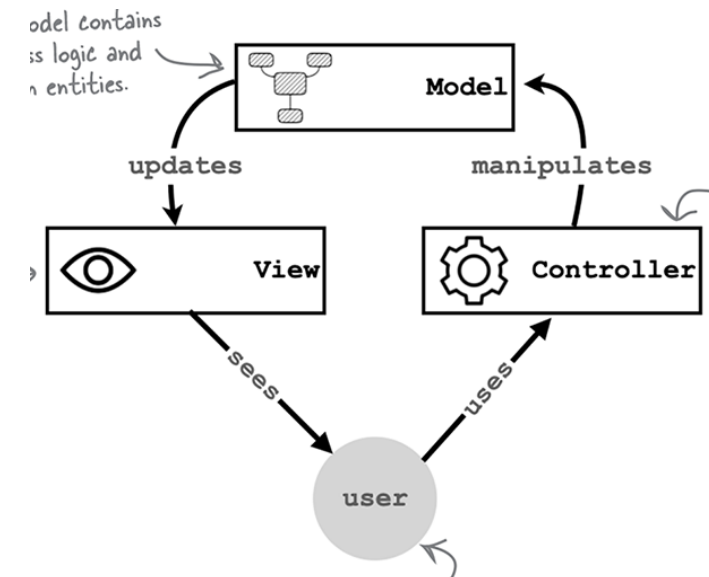


Why Choose Layered Architecture?

- ❖ Matches Naan & Pop's **needs**: simplicity, fast delivery, separation of technical roles.
- ❖ Aligns closely with familiar **design patterns** like **MVC**.

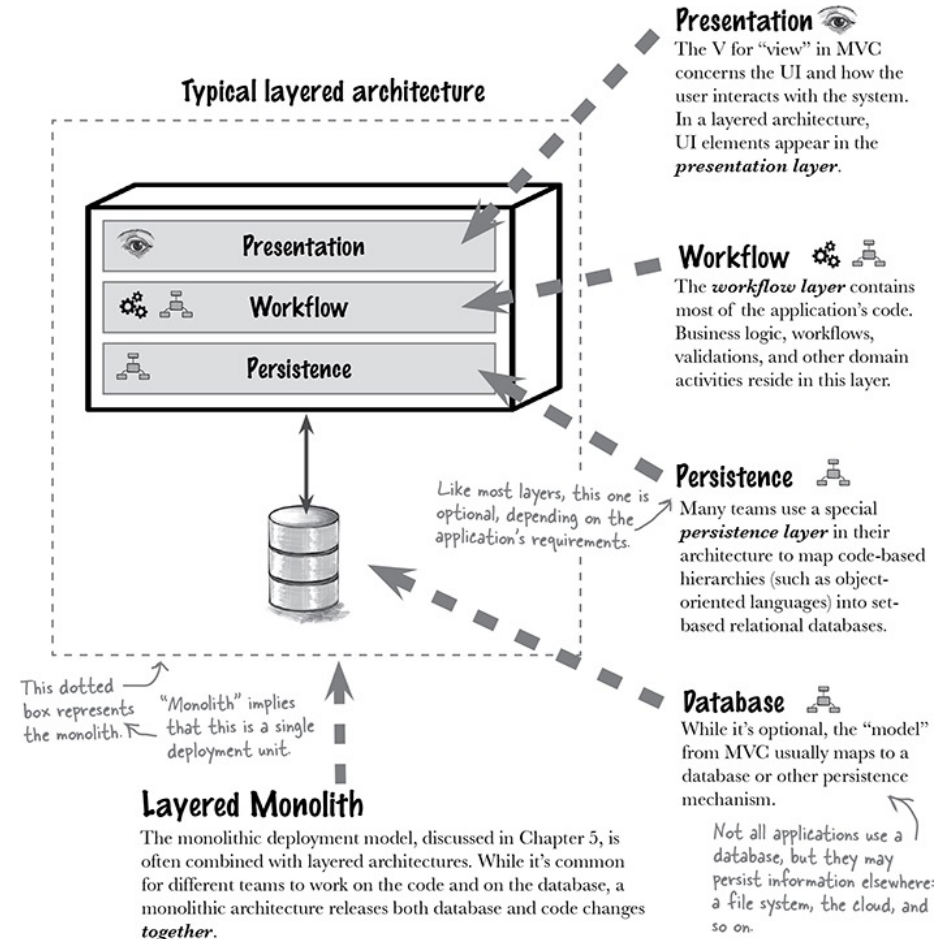
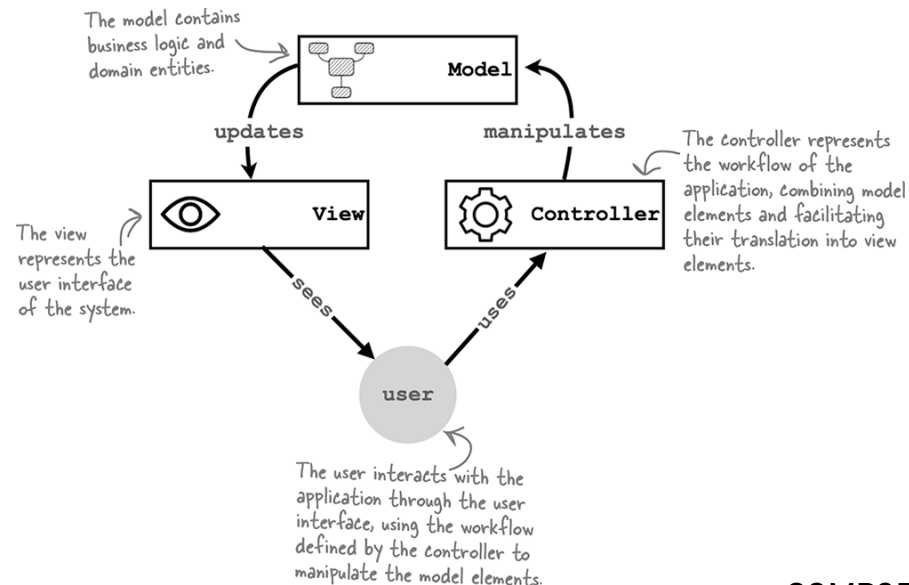
Trade-offs involved:

- Simplicity vs. extensibility.
- Speed vs. maintainability.



Mapping MVC to Layered Architecture

- ❖ MVC concepts **translate** naturally into architectural layers.
- ❖ **Additional layers** may be introduced based on real-world constraints (e.g., integration).

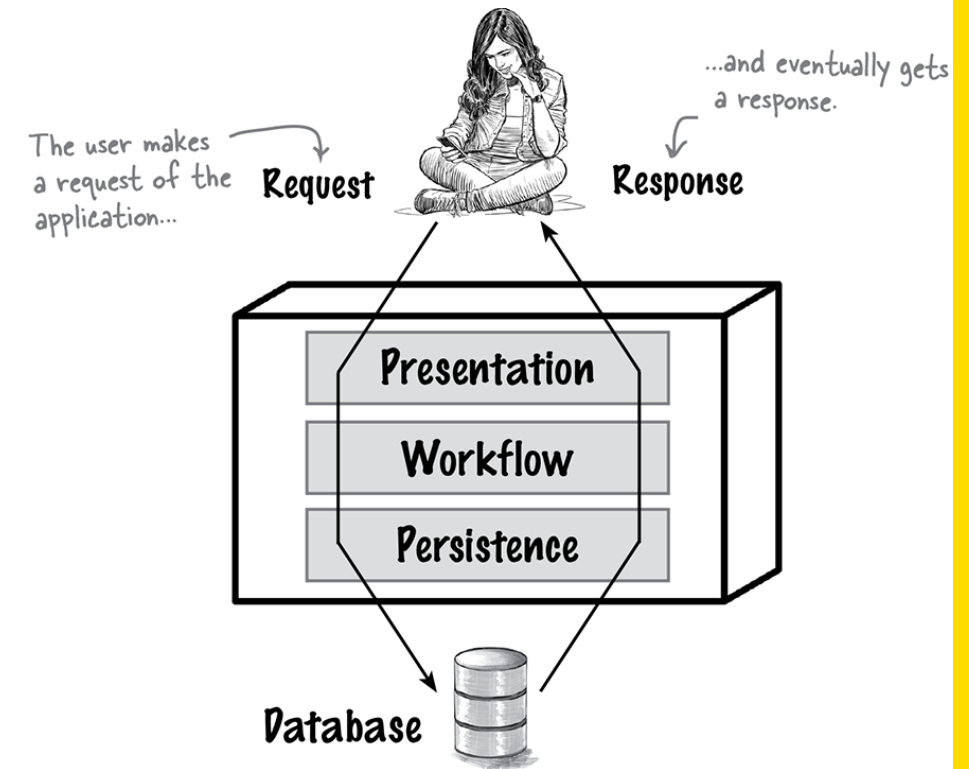


Layered Architecture – Philosophy

- ❖ Technically partitioned and usually **monolithic**.
- ❖ Domain logic **spans multiple layers**:
 - Presentation (UI components).
 - Workflow (business logic components).
 - Persistence (database schemas and operations).

Implication:

- Domain **changes** affect **multiple layers**.



Drivers for Layered Architecture

Why choose layered architecture?

- ❖ **Specialization**: Separates UI, business logic, and database, allowing team specialisation.
- ❖ **Physical separation**: Matches real-world technology separation (frontend/backend/database).
- ❖ **Ease of reuse**: Technical reuse across multiple projects.
- ❖ **Familiarity**: Mirrors MVC, easy for developers to grasp.

Physical Architectures in Layered Systems

Common physical architectures:

❖ Two-tier (Client/Server):

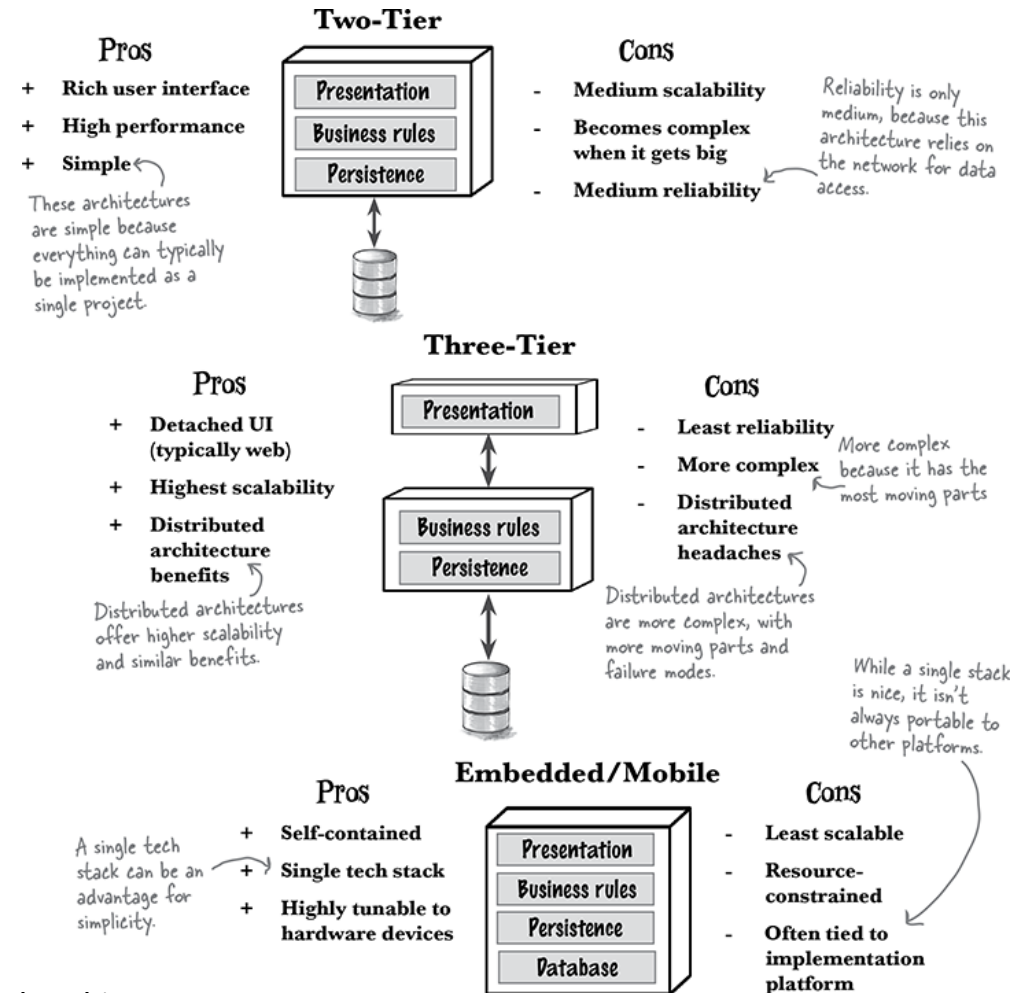
- Client UI directly accesses the database.

❖ Three-tier (Web):

- Browser (presentation),
- App server (business logic)
- Database server (persistence)

❖ Embedded/Mobile:

- All layers bundled into one deployable unit.



Physical Architecture – Pros and Cons

Physical Architecture	Pros	Cons
Two-tier (Client/Server)	Simple, quick to build	Less secure, poor scalability
Three-tier (Web)	Scalable, flexible	Complex infrastructure
Embedded/Mobile	High performance, simple deployment	Limited scalability

Adding Layers – Integration Layer Example

- ❖ **Additional layers** can be introduced for **specialised tasks** (e.g., Integration layer for delivery partners).
- ❖ Clearly **isolates** integration code from core business logic.

Example:

- Integration with Uber Eats API resides entirely within an Integration Layer.

Caveats – Domain Changes Impact Multiple Layers

- ❖ Layered architecture easily supports changes in technical capabilities.
- ❖ **However**, changes in the domain (e.g., adding pizzas to menu) will **affect multiple layers**:
 - Presentation layer (new UI)
 - Workflow layer (processing new item)
 - Persistence layer (storing item data)

Trade-off:

- Ease of technical changes **vs.** difficulty of domain-wide changes.

Layered Architecture: Strengths

- ❖ **Feasibility:** Quick, cost-effective solutions.
- ❖ **Technical partitioning:** Easy technical reuse.
- ❖ **Data-intensive operations:** Efficient local data processing.
- ❖ **Performance:** High internal performance without network overhead.
- ❖ **Fast development:** Ideal for MVPs and small systems.

Layered Architecture: Weaknesses

- ❖ **Deployability:** Monolith deployments become cumbersome as systems grow.
- ❖ **Coupling:** High risk of tight coupling (“big ball of mud”).
- ❖ **Scalability:** Difficult to scale individual functionalities independently.
- ❖ **Elasticity:** Poor performance under bursty traffic conditions.
- ❖ **Testability:** Increasingly difficult testing as codebase grows.

Layered Architecture – Rating Chart (Example)

Architectural Characteristic	Star Rating
Maintainability	★
Testability	★ ★
Deployability	★
Simplicity	★ ★ ★ ★ ★
Evolvability	★
Performance	★ ★ ★
Scalability	★
Elasticity	★
Fault Tolerance	★
Overall Cost	\$

Layered architectures are nice and simple.

Monoliths in general don't handle scalability and elasticity well, and layered ones even less so.

Testing isn't especially easy, but the team has been dealing with layered architectures so long that they've built up many techniques.

Well-designed layered architectures can boast quite high performance.

Simplicity, in this case, leads to affordability.

Layered Architecture – Exercises

An online auction system where users can bid on items

Why? _____

- ☐ Well suited for layered monolith
- ☐ Might be a fit for layered monolith
- ☐ Not well suited for layered monolith

A large backend financial system for processing and settling international wire transfers overnight

Why? _____

- ☐ Well suited for layered monolith
- ☐ Might be a fit for layered monolith
- ☐ Not well suited for layered monolith

A company entering a new line of business that expects constant changes to its system

Why? _____

- ☐ Well suited for layered monolith
- ☐ Might be a fit for layered monolith
- ☐ Not well suited for layered monolith

A small bakery that wants to start taking online orders

Why? _____

- ☐ Well suited for layered monolith
- ☐ Might be a fit for layered monolith
- ☐ Not well suited for layered monolith

A trouble ticket system for electronics purchased with a support plan, in which field technicians come to customers to fix problems

Why? _____

- ☐ Well suited for layered monolith
- ☐ Might be a fit for layered monolith
- ☐ Not well suited for layered monolith

Suitable Scenarios for Layered Architecture

Ideal Use Cases:

- ❖ Small, simple systems requiring quick delivery (e.g., small business websites).
- ❖ Data-intensive applications with local database storage (e.g., desktop CRM apps).
- ❖ Applications needing clear specialization boundaries (e.g., separate UI, backend, DB teams).

Summary of Layered Architecture

Key points:

- ❖ Simple, fast to implement.
- ❖ Clearly separates technical concerns.
- ❖ Ideal for stable domains with minimal changes.
- ❖ Challenging to adapt when domain changes significantly.

Modular Monoliths Architecture

COMP2511, CSE, UNSW



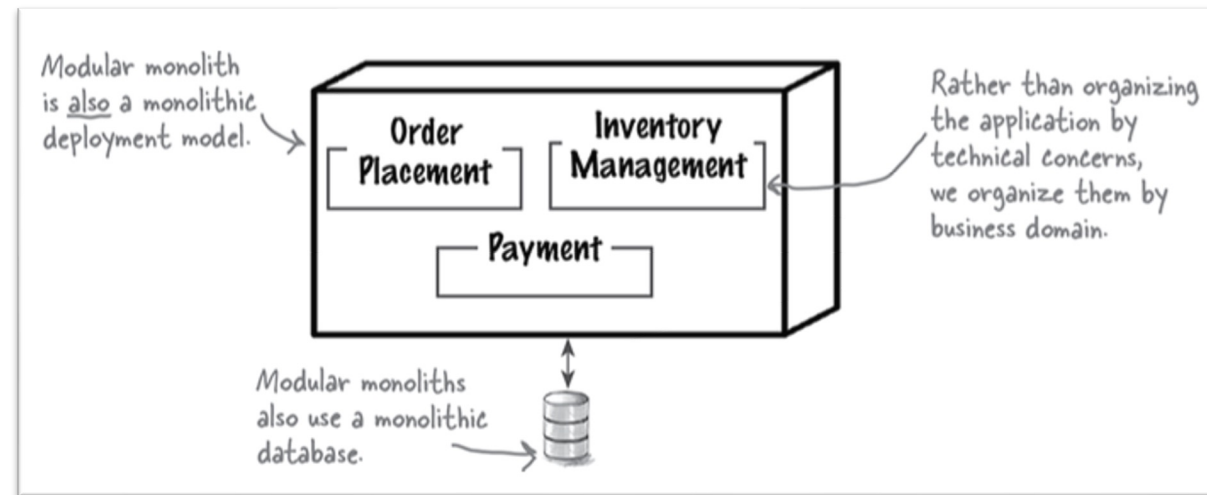
UNSW
SYDNEY

These lecture slides are from the books:

- “*Head First Software Architecture*”, by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024
- “*Fundamentals of Software Architecture*”, 2nd Edition, by Mark Richards, Neal Ford

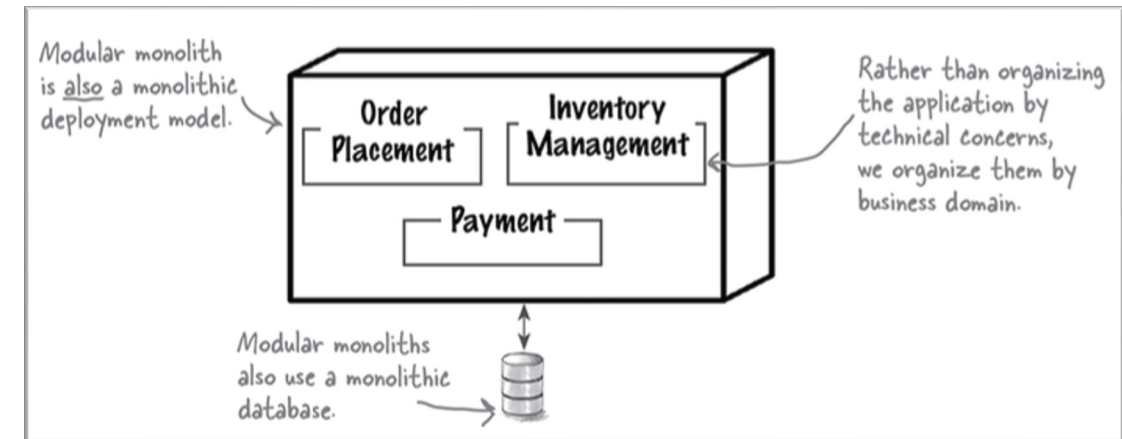
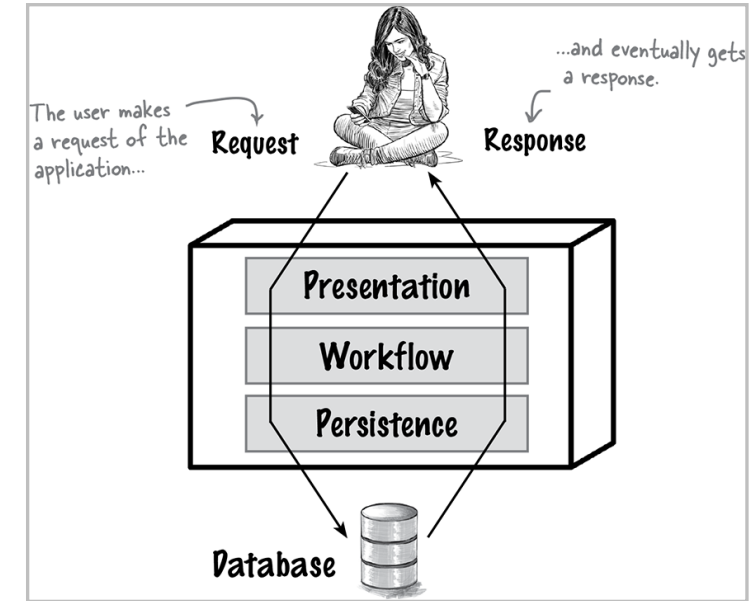
Introduction to Modular Monoliths

- ❖ **Definition:** A monolithic architecture organized by domain, not technical layers.
- ❖ **Goal:** Align code and teams around business capabilities.
- ❖ **Key Trait:** Deployed as a single unit, with domain-based modular structure



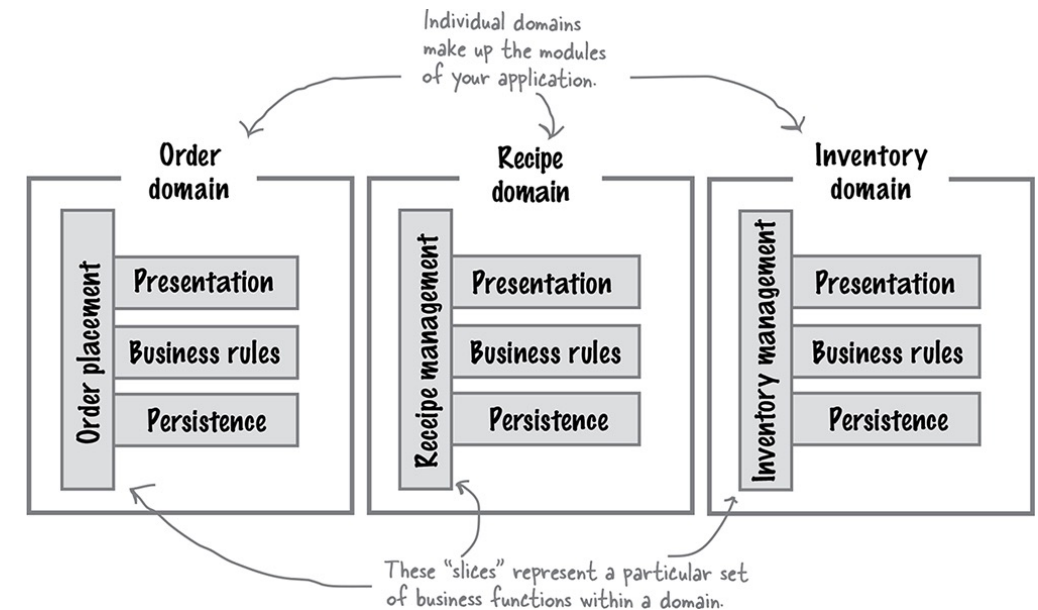
Layered vs. Modular Monolith

- ❖ **Layered**: Organized by technical concerns (UI, services, DB).
- ❖ **Modular**: Organized by domain (Order, Payment, Inventory).
- ❖ **Problem with Layered**: Changes often touch many teams.
- ❖ **Benefit of Modular**: Changes are isolated within a domain.



What Is a Module?

- ❖ Independent unit within a domain.
- ❖ Contains all business logic for its domain.
- ❖ Examples:
 - *OrderPlacement* module handles order lifecycle
 - *Recipe* module contains ingredients and cooking steps
 - *Inventory* module tracks stock levels and alerts
 - *UserManagement* module handles user accounts and roles



Why Choose a Modular Monolith?

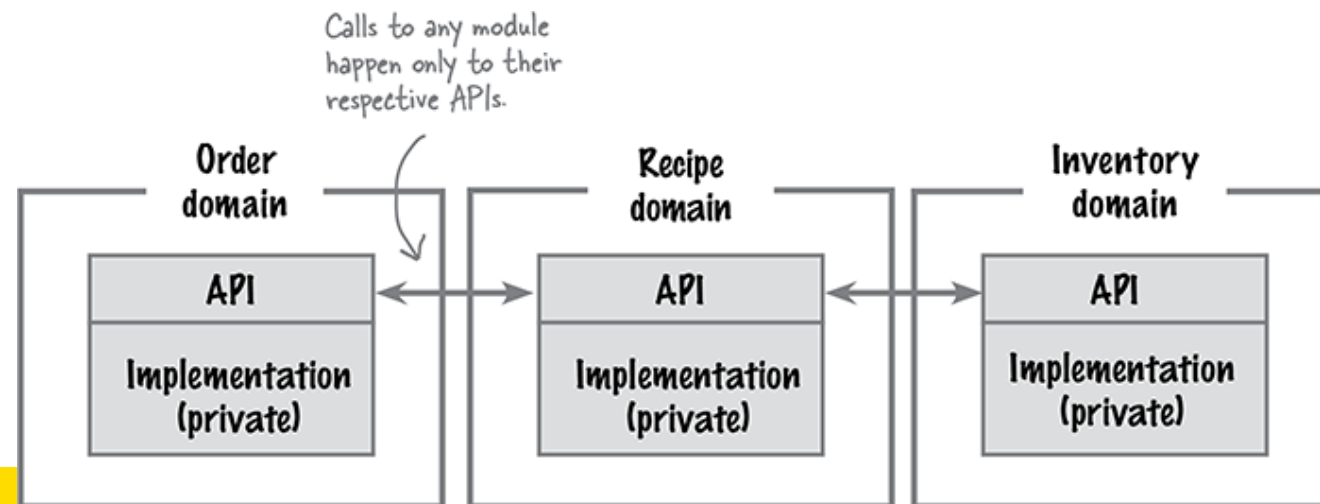
- ❖ **Business alignment:** Modules map to subdomains
- ❖ **Team ownership:** Cross-functional teams per domain
- ❖ **Faster changes:** Changes isolated to one module
- ❖ **High performance:** No inter-service network latency
- ❖ **Easier testing:** Scoped test suites per module

Code Organization in a Modular Monolith

- ❖ **Single** deployment
- ❖ **Separate** namespaces/packages for each module
- ❖ Each module has:
 - **Public** API
 - **Private** internals
- ❖ **Example** (namespace):
 - com.naanpop.order
 - com.naanpop.inventory
 - com.naanpop.reports

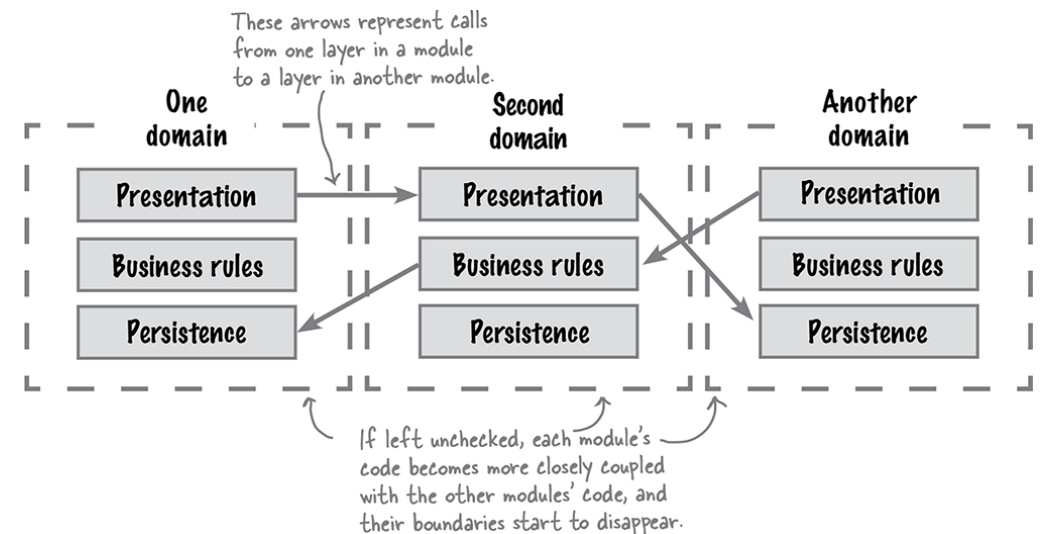
Managing Inter-Module Communication

- ❖ **Don't:** Direct calls between modules (tight coupling)
- ❖ **Do:** Use public APIs
- ❖ **Risk:** Big ball of mud from uncontrolled access
- ❖ **Solution:** Interface-based interaction only



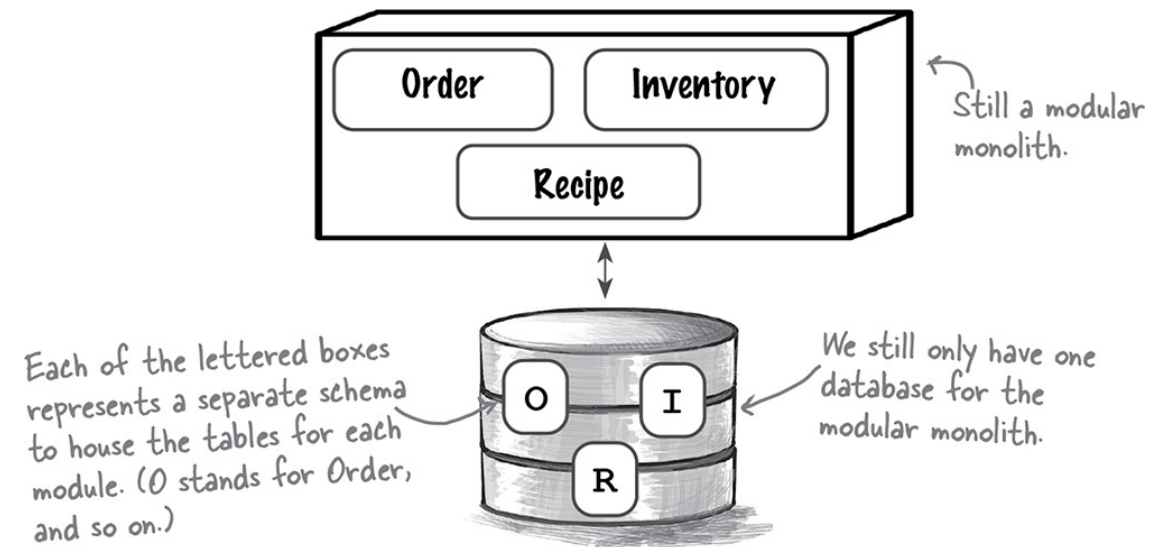
Keeping Modules Modular

- ❖ IDE features (e.g. auto-import) can break boundaries
- ❖ Separate folders/repositories
- ❖ Use build tools (e.g., Gradle subprojects)
- ❖ Use language features:
 - Java: JPMS
 - .NET: internal keyword



Modularizing the Database

- ❖ One DB per monolith, but **partitioned** by schema
- ❖ **Rule:** *Each module accesses only its own tables*
- ❖ **No foreign keys** between modules
- ❖ **Use ID references** and **API calls**



Avoiding Coupling in Data Access

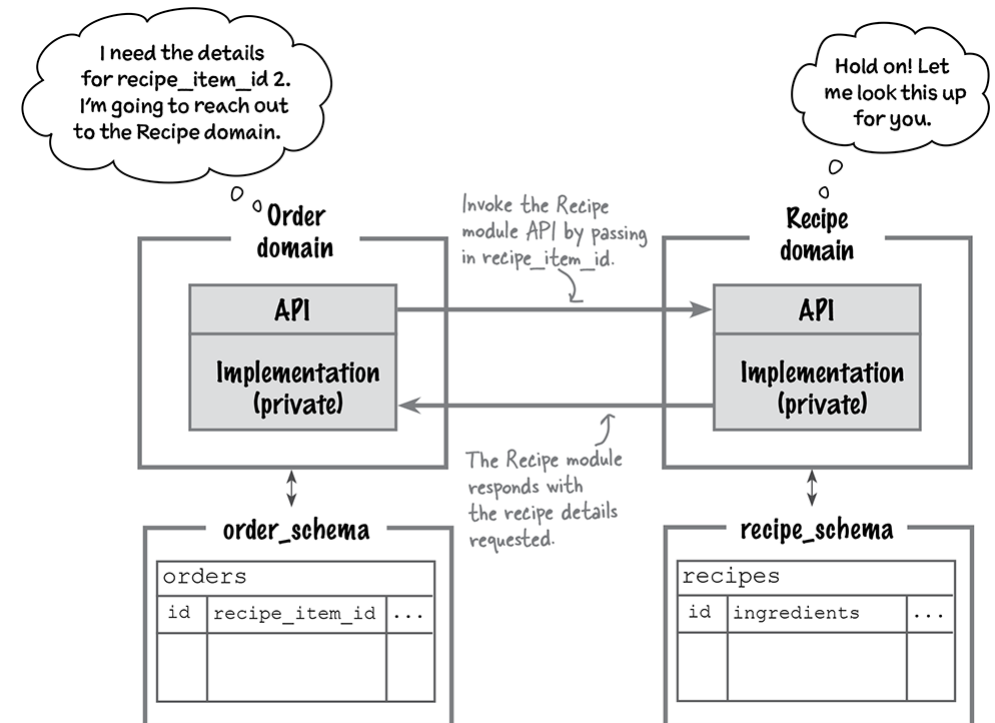
❖ **Risk:** JOINS across module tables reintroduce coupling

❖ **Solution:**

- Store IDs, not foreign keys
- Retrieve info via module API

❖ **Example:**

- Order module stores *RecipeItemID*
- Calls Recipe API when needed



Extending Modularity to Teams

- ❖ **Align** teams with subdomains (modular ownership)
- ❖ Foster **domain expertise** and autonomy
- ❖ **Minimize** coordination overhead
- ❖ **Example**: Inventory team owns inventory module and tests

Example – Expense Tracking App

❖ Requirements:

- Users add expenses
- Auditors review reports
- Audit trail for traceability

❖ Modules:

- ExpenseEntry
- AuditReview
- UserManagement

Example – Educational LMS

❖ Requirements:

- Instructors upload courses
- Students enroll and complete assessments
- Admins manage roles and reports

❖ Modules:

- CourseContent
- Enrollment
- AssessmentEngine
- UserAdministration

Benefits of Modular Monoliths

- ❖ **Domain Partitioning**: Better team alignment
- ❖ **Performance**: No inter-service latency
- ❖ **Maintainability**: Domain-local changes
- ❖ **Testability**: Scoped, isolated testing
- ❖ **Deployability**: Single unit, easier CI/CD

Limitations of Modular Monoliths

- ❖ **Reuse**: Harder to share utilities
- ❖ **One set of characteristics**: No per-module customization
- ❖ **Fragile modularity**: Easy to break boundaries
- ❖ **Operational limits**: Harder to scale or isolate faults

Governance and Discipline

❖ Modular monoliths require:

- Discipline in access control
- Codebase enforcement (tools, practices)
- Database discipline (modular schemas)

❖ Governance tools help but don't eliminate the need for vigilance

When to Use Modular Monoliths

- ❖ Teams **aligned** to **business domains**
- ❖ Applications that must remain **performant**
- ❖ Systems needing **easy testability** and deployment

Transition Path – Layered to Modular

- ❖ **Start** with layered → modularize by domain over time
- ❖ Introduce governance and APIs **gradually**
- ❖ **Split** database logically first, physically later

Modular Monolith Advantages

- ❖ Better domain alignment than layered monoliths
- ❖ Single deployment with domain modularity
- ❖ Enables domain-oriented teams
- ❖ Maintains runtime performance of monoliths
- ❖ Fewer operational headaches than microservices

Common Pitfalls in Modular Monoliths

- ❖ **Bypassing** module APIs (direct access)
- ❖ Database **JOINS** across modules
- ❖ **Overusing** shared libraries (tight coupling)
- ❖ **Lack of observability** into module interactions

Techniques for Success

- ❖ Define strong **module boundaries**
- ❖ Maintain **minimal** public API surface
- ❖ Invest in automated testing and monitoring
- ❖ **Review architecture** regularly for erosion

Modular Monolith Star Ratings

		Architectural Characteristic	Star Rating
These fare better than in the layered architectural style.	{	Maintainability	★ ★ ★
		Testability	★ ★ ★
		Deployability	★ ★ ★
		Simplicity	★ ★ ★ ★
Most monolithic architectures perform well, especially if well designed.	→	Evolvability	★ ★ ★
		Performance	★ ★ ★
		Scalability	★
		Elasticity	★
Overall, more expensive than layered architectures. Modular monoliths require more planning, thought, and long-term maintainance.	→	Fault Tolerance	★
		Overall Cost	\$ \$

Exercise

Which of the following systems might be well suited for the modular monolith architectural style, and why?

An online auction system where users can bid on items

Why? _____

- ☐ Well suited for modular monoliths
- ☐ Might be a fit for modular monoliths
- ☐ Not well suited for modular monoliths

A large backend financial system for processing and settling international wire transfers overnight

Why? _____

- ☐ Well suited for modular monoliths
- ☐ Might be a fit for modular monoliths
- ☐ Not well suited for modular monoliths

A company entering a new line of business that expects constant changes to its system

Why? _____

- ☐ Well suited for modular monoliths
- ☐ Might be a fit for modular monoliths
- ☐ Not well suited for modular monoliths

A small bakery that wants to start taking online orders

Why? _____

- ☐ Well suited for modular monoliths
- ☐ Might be a fit for modular monoliths
- ☐ Not well suited for modular monoliths

A trouble ticket system for electronics purchased with a support plan, in which field technicians come to customers to fix problems

Why? _____

- ☐ Well suited for modular monoliths
- ☐ Might be a fit for modular monoliths
- ☐ Not well suited for modular monoliths

Microservice Architecture

COMP2511, CSE, UNSW



UNSW
SYDNEY

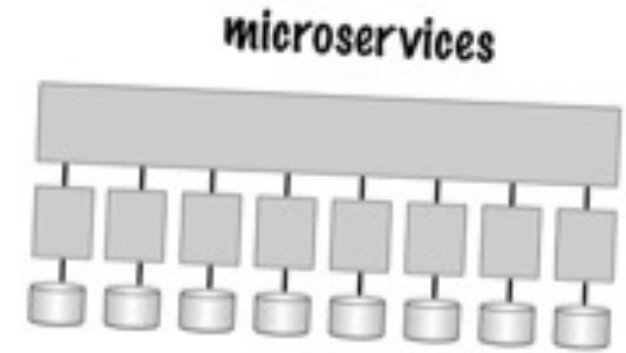
These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

Introduction to Microservices

- ❖ Microservices are **single-purpose**, independently deployed units.
- ❖ Ideal for environments requiring **frequent changes** and **scalability**.

Examples:

- Netflix's streaming services
- Amazon's product catalogue.



Defining Microservices

❖ Performs **one specific function** exceptionally **well**.

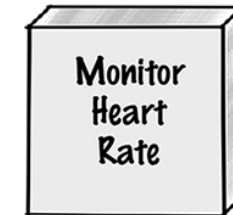
Examples:

- Dedicated microservice like "*Monitor Heart Rate*."
- "Authenticate User" service, "*Generate Invoice*" service.
- "User Profile Management" service.
- "Shopping Cart" service.
- "Notification and Alert" service.
- "Recommendation Engine" service (e.g., Netflix recommendations).



↖ This large service monitors all of a patient's vital signs.

This is quite a small service because it only performs a single function—let's call it a "micro"-service.



Exercise: Define Microservices

Identify single-purpose microservices below:

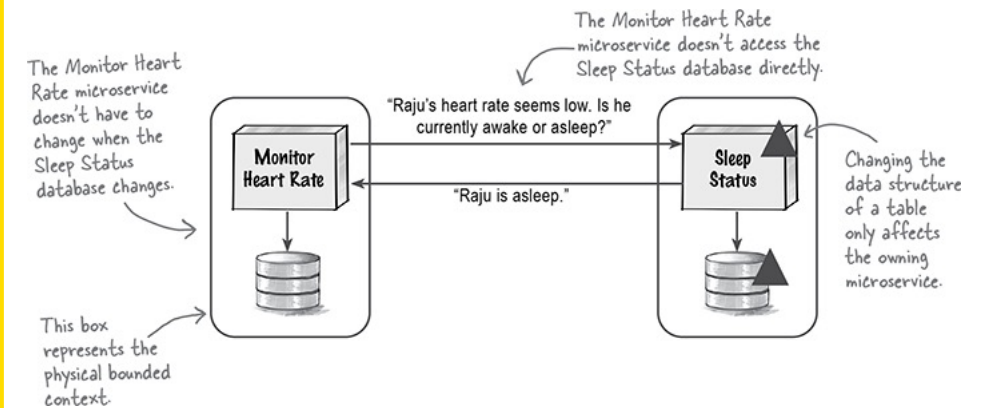
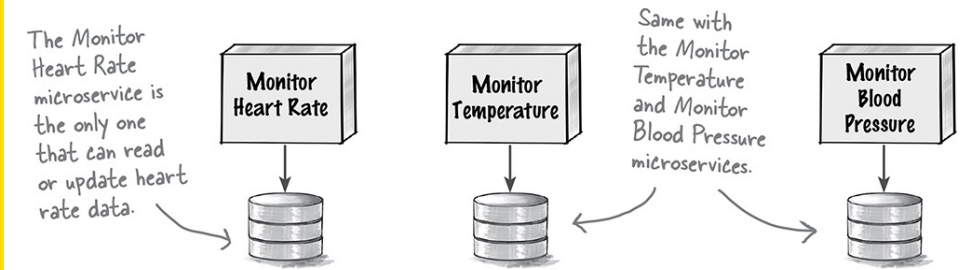
- ☐ Add a movie to your personal “to watch” list
- ☐ Pay for an order using your credit card
- ☐ Generate sales-forecasting and financial-performance reports
- ☐ Submit and process a loan application to get that new car you’ve always wanted
- ☐ Determining the shipping cost for an online order

Key Characteristics of Microservices

- ❖ Own their **own data** (Physical bounded contexts).
- ❖ Direct data **access restricted** to owning microservice.

Examples:

- Order service maintains its own order history database.
- Inventory service owns and manages product availability data.
- Payment service manages transaction records independently.
- User Authentication service securely stores user credentials separately.

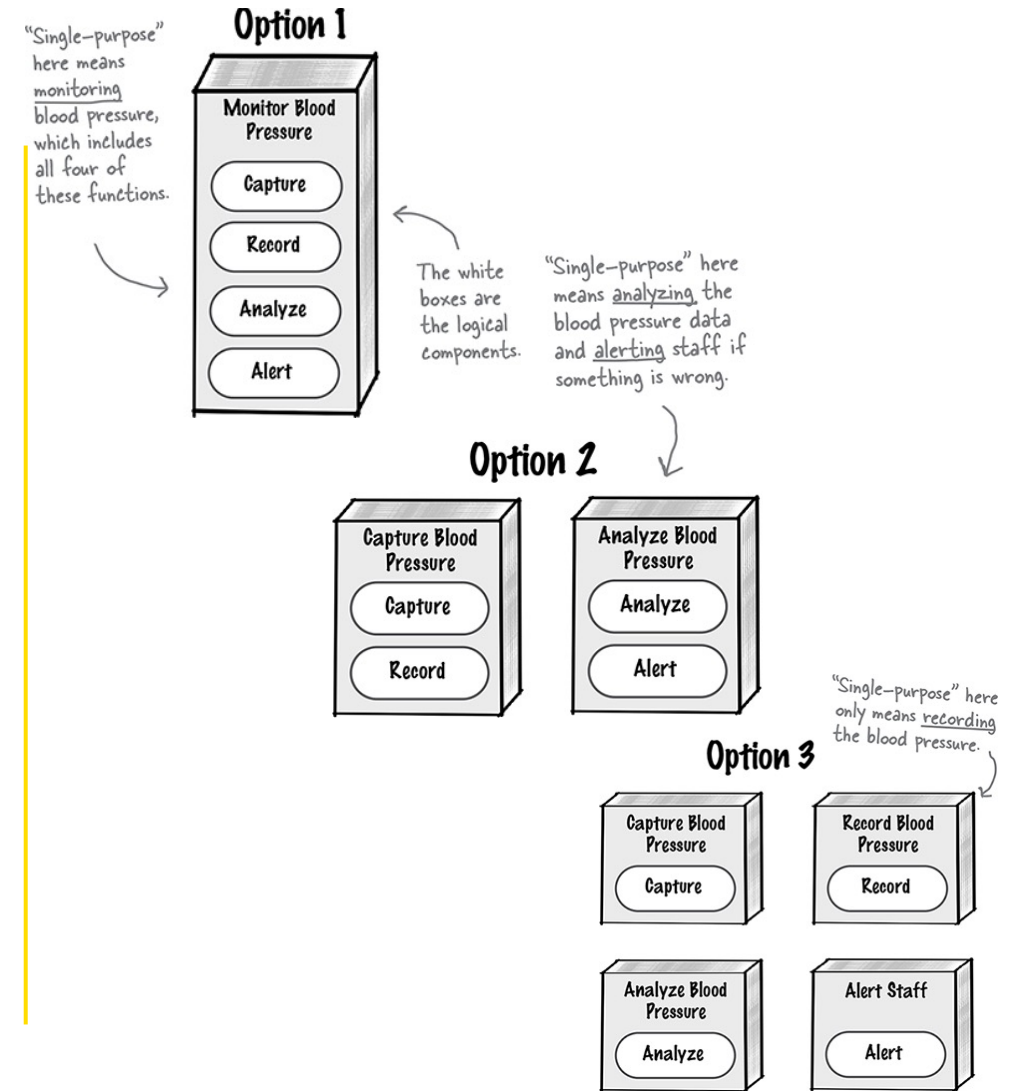


Determining Granularity

- ❖ Granularity: The scope of a microservice's responsibility.
- ❖ Avoid too fine-grained ("Grains of Sand" antipattern).

Examples:

- Single microservice handling payment transactions.
- A microservice dedicated to shipping and tracking orders.
- Product review and rating as a distinct service separate from product information.
- User notification service isolated from user profile management



Granularity Disintegrators

(Reasons to Make Services Smaller)

Cohesion: Functions within a service should be closely related.

- Payment processing separate from user authentication.

Fault Tolerance: Separating unstable functions for better reliability.

- Isolating an unstable email notification service.

Access Control: Easier management of sensitive data.

- Isolating financial data access.

Code Volatility: Isolating frequently changing parts.

- User interface components separated from stable backend logic.

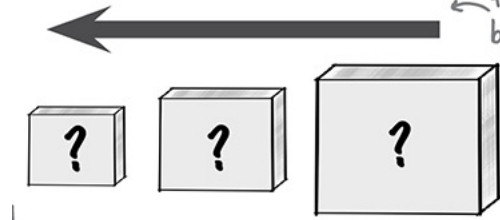
Scalability: Independent scaling for high-demand components.

- High-traffic "search" feature isolated for scaling.

Granularity Disintegrators

When should you consider making your services smaller, with less functionality?

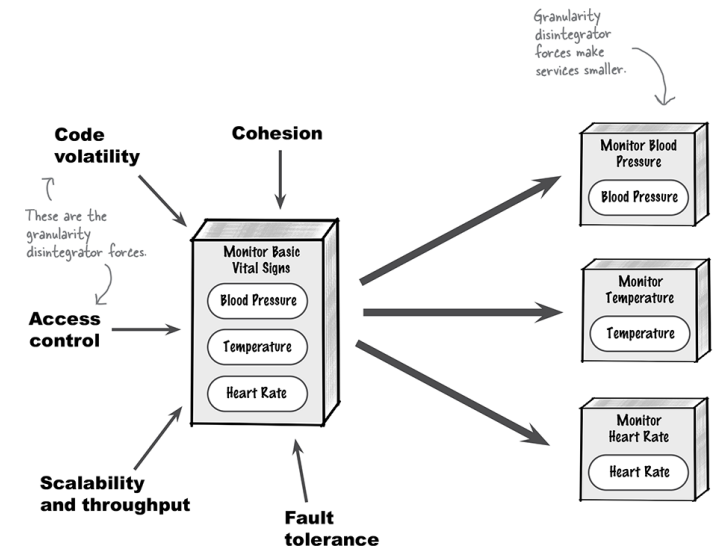
Disintegrators force services to break apart.



Granularity Integrators

When should you consider making your services bigger, with more functionality?

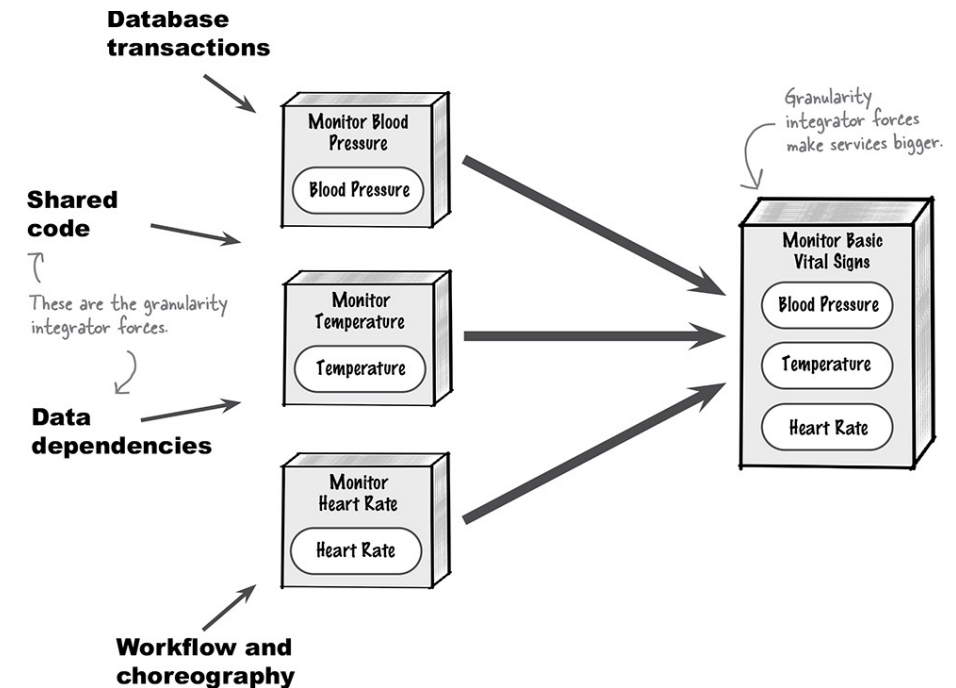
Integrators force services to come together.



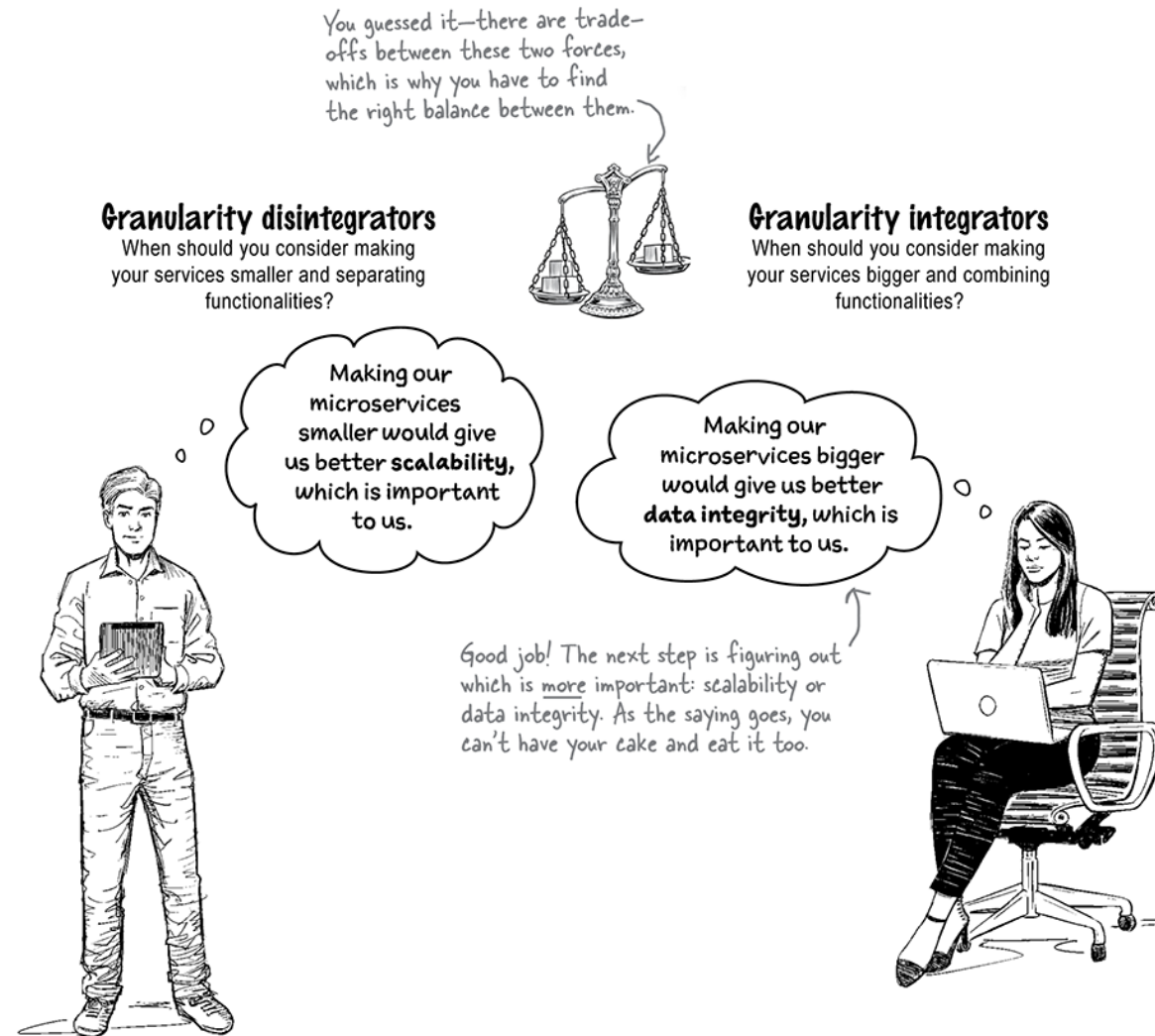
Granularity Integrators

(Reasons to Make Services Larger)

- ❖ **Database Transactions:** Easier to manage single commit/rollback operations.
 - Order creation and inventory deduction in one service.
- ❖ **Data Dependencies:** Maintain tightly coupled data together.
 - User profiles and preferences managed together.
- ❖ **Workflow Efficiency:** Reduce excessive inter-service communication.
 - Checkout service combining cart, pricing, and payment functionalities.

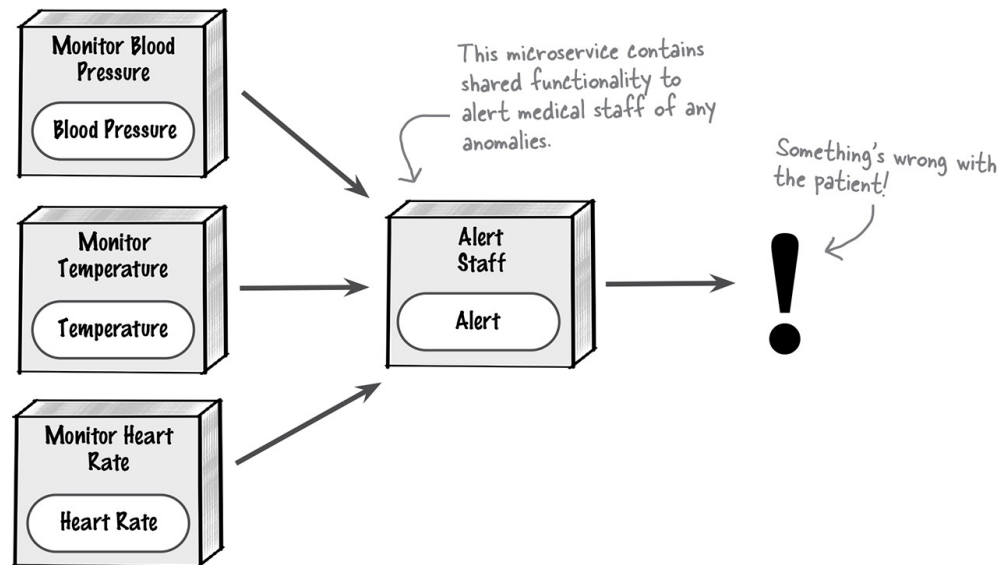


It's about a right balance!



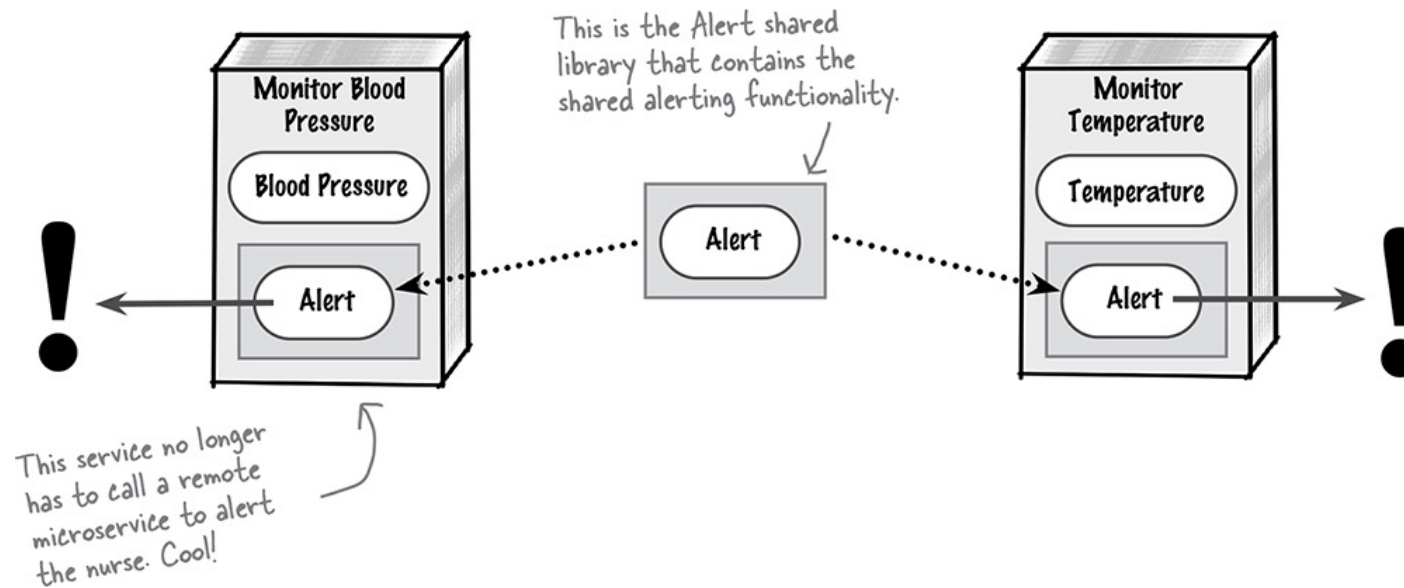
Sharing Functionality

- ❖ **Shared Services:** Standalone microservices accessed remotely.
 - **Authentication** service used by multiple microservices.
 - Shared alert functionality in *MonitorMe* medical alerts



Sharing Functionality

- ❖ **Shared Libraries:** Embedded at compile-time, deployed with each service.
 - Logging and error handling libraries.



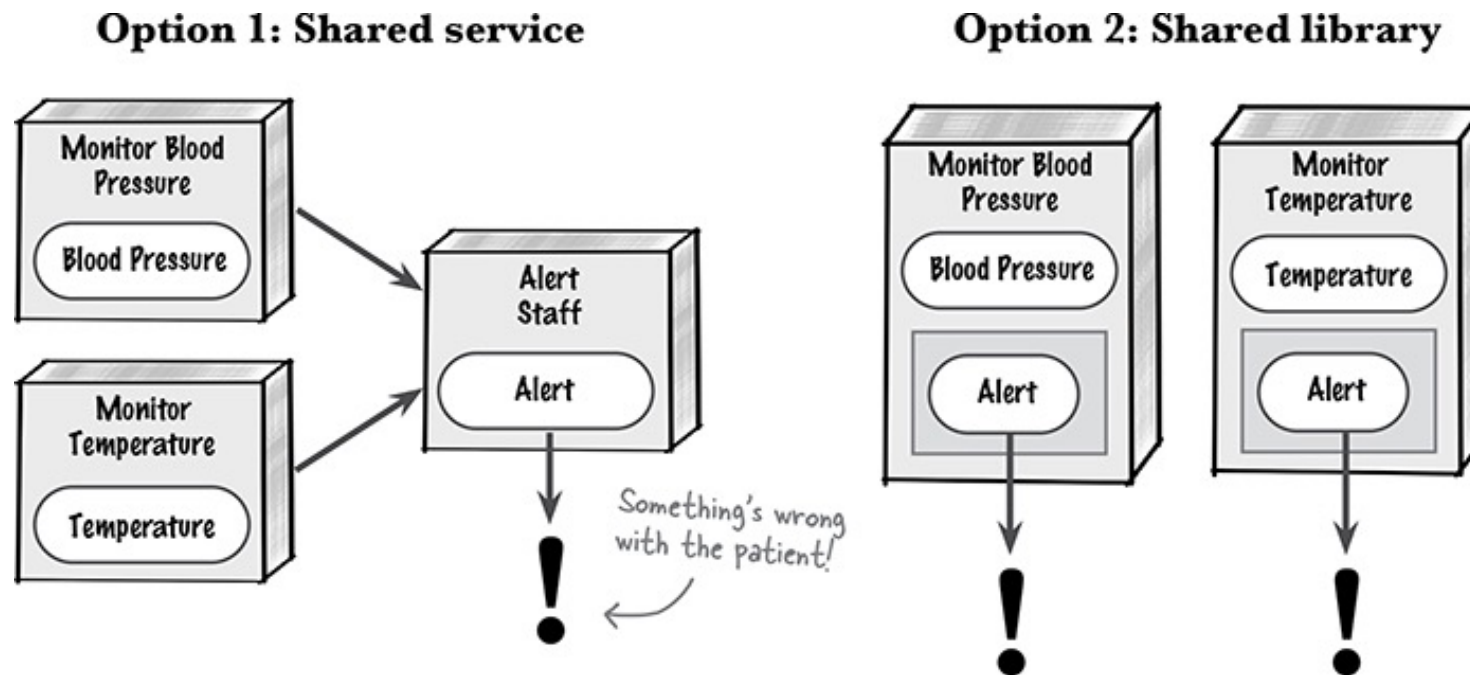
Shared Services *vs.* Shared Libraries

- ❖ **Services:** Agile, suitable for diverse environments, slower, less fault-tolerant.
 - Central user authentication service.
- ❖ **Libraries:** Faster, scalable, robust, but challenging dependency management.
 - JSON parsing libraries used across multiple microservices.

Exercise

Should the alert functionality in *MonitorMe* be a [shared library](#) or a [shared service](#)?

➤ [Justify](#) your decision.



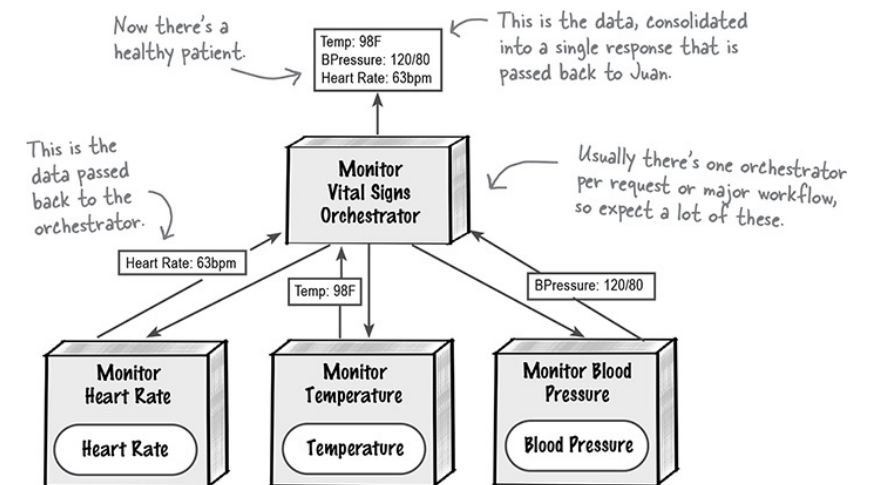
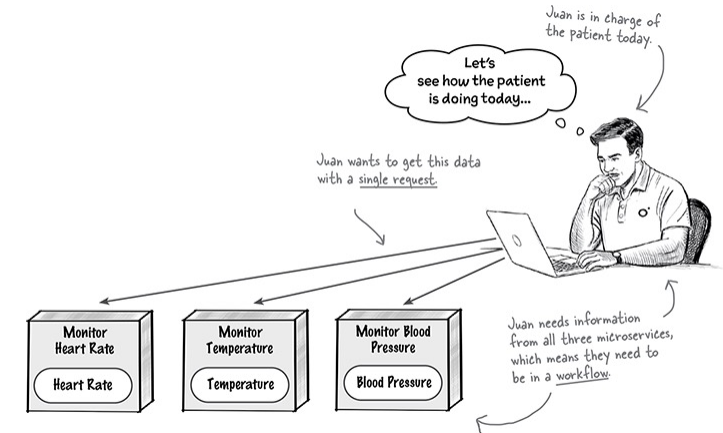
Workflow Management: Orchestration

❖ **Central orchestration** manages workflow, akin to a symphony conductor.

- **Pros:** Centralized management, clear state/error handling.
- **Cons:** Bottlenecks, high coupling, performance concerns.

❖ **Example:**

- Centralised order processing orchestrating payment, inventory, and shipment services.



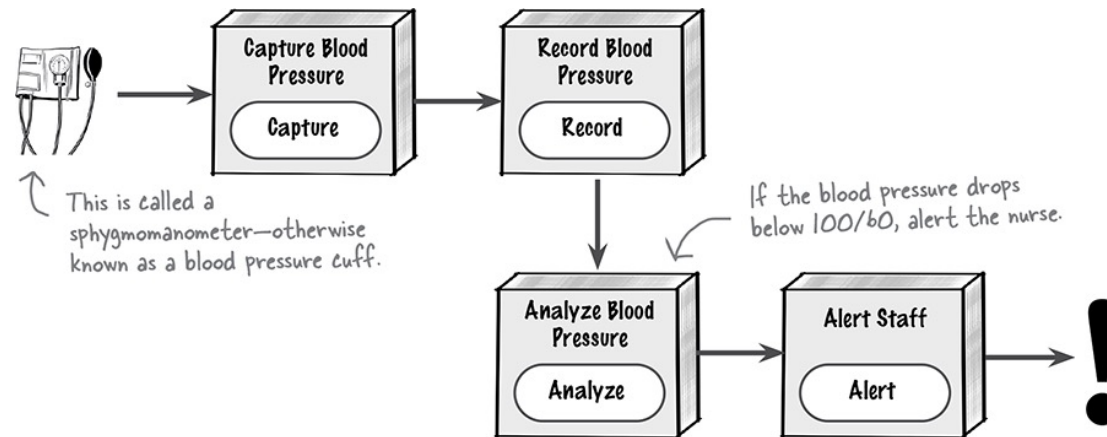
Workflow Management: Choreography

❖ **Peer-to-peer** service communication, like coordinated dance.

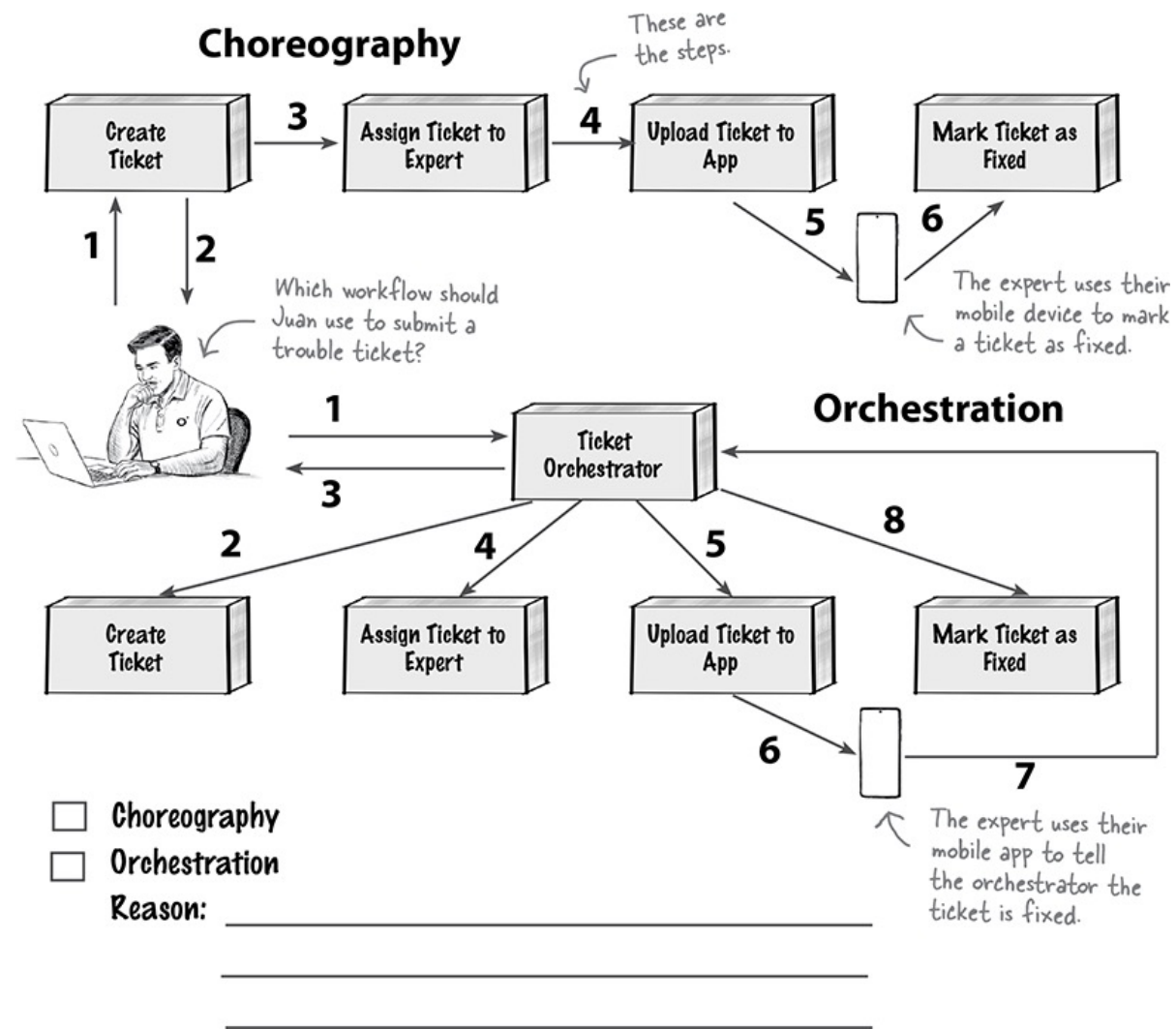
- **Pros:** Scalable, loosely coupled, high responsiveness.
- **Cons:** Complex error and state management.

❖ **Example:**

- Event-driven updates between cart, inventory, and shipping services in an e-commerce site.



Exercise



Advantages of Microservices

❖ [Maintainability](#), Testability, [Deployability](#), [Evolvability](#).

❖ Exceptional [scalability](#) and [fault tolerance](#).

❖ Examples:

- Continuous deployment at [Spotify](#)
- Scalable services at [Netflix](#)

Limitations of Microservices

- ❖ **Complexity**, especially in workflow management.
- ❖ **Performance issues** due to inter-service communications.
- ❖ **Example:**
 - Increased latency in highly interactive systems like gaming or real-time analytics platforms.

Balancing Microservices Architecture

❖ Decision criteria:

- Business agility
- Complexity handling
- Team structure

❖ Optimal balance between granular control and practical maintainability.

❖ Example:

- Amazon's product catalog services balancing granularity and maintainability.

Case Study - StayHealthy MonitorMe

- ❖ **Successful** real-world implementation of microservices.
 - ❖ *Insights*: Balance granularity, effectively manage shared resources.
 - ❖ Continuous focus on **agility** and operational **stability**.
-
- ❖ Example:
 - **Reliable** and **scalable** health monitoring system for critical patient data.

Microservices Star Ratings

Architectural Characteristic	Star Rating
Maintainability	★ ★ ★ ★ ★
Testability	★ ★ ★ ★ ★
Deployability	★ ★ ★ ★ ★
Simplicity	★
Evolvability	★ ★ ★ ★ ★
Performance	★ ★
Scalability	★ ★ ★ ★ ★
Elasticity	★ ★ ★ ★
Fault Tolerance	★ ★ ★ ★ ★
Overall Cost	\$ \$ \$ \$ \$

These characteristics contribute to agility—the ability to respond quickly to change.

We can scale microservices at a function level.

Microservices are HARD.

Too much communication betw microservices slows down requests.

Exercise

Which of the following systems might be **well suited** for the **microservices** architectural style, and why?

An online auction system where users can bid on items

Why? High scalability and elasticity needs; high concurrency; independent functions

- ☒ Well suited for microservices
- ☐ Might be a fit for microservices
- ☐ Not well suited for microservices

A large backend financial system for processing and settling international wire transfers overnight

Why? Microservices' superpowers aren't needed in this kind of complex system

- ☐ Well suited for microservices
- ☐ Might be a fit for microservices
- ☒ Not well suited for microservices

A company entering a new line of business that expects constant changes to its system

Why? High agility and evolvability mean microservices could fit, but we need more info

- ☐ Well suited for microservices
- ☒ Might be a fit for microservices
- ☐ Not well suited for microservices

A small bakery that wants to start taking online orders

Why? The high cost and complexity of microservices would be too much for a small bakery

- ☐ Well suited for microservices
- ☐ Might be a fit for microservices
- ☒ Not well suited for microservices

A trouble ticket system for electronics purchased with a support plan, in which field technicians come to customers to fix problems

Why? Independent functions; good scalability and elasticity; simple workflows

- ☒ Well suited for microservices
- ☐ Might be a fit for microservices
- ☐ Not well suited for microservices

Summary

- ❖ Microservices offer high **flexibility** but involve significant **complexity**.
 - ❖ **Requires crucial** granularity and communication decisions.
 - ❖ Evaluate and **manage trade-offs** carefully.
-
- ❖ **Example:**
 - Transitioning from monoliths to microservices at Uber.

Event Driven Architecture

COMP2511, CSE, UNSW

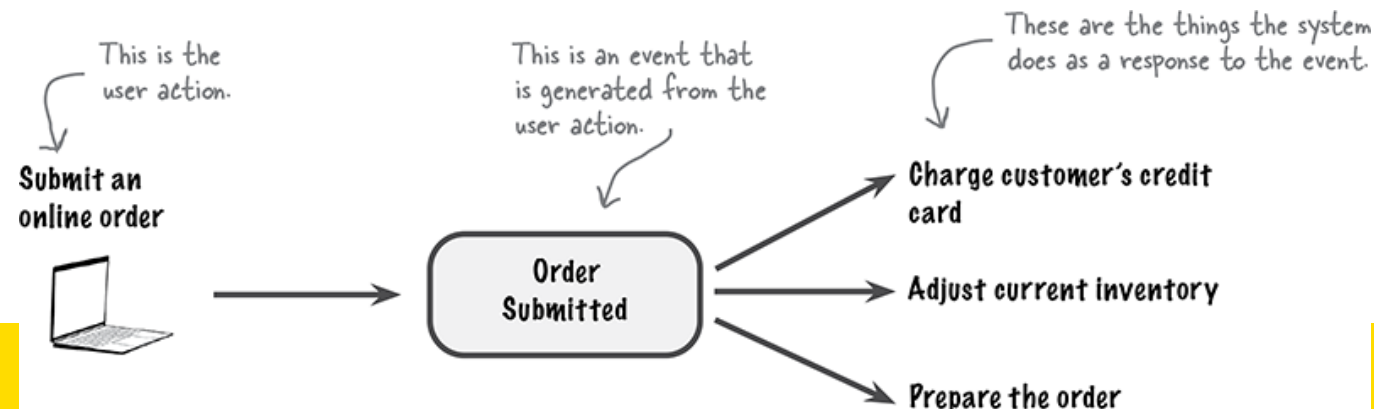


UNSW
SYDNEY

These lecture slides are from the book “*Head First Software Architecture*”,
by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

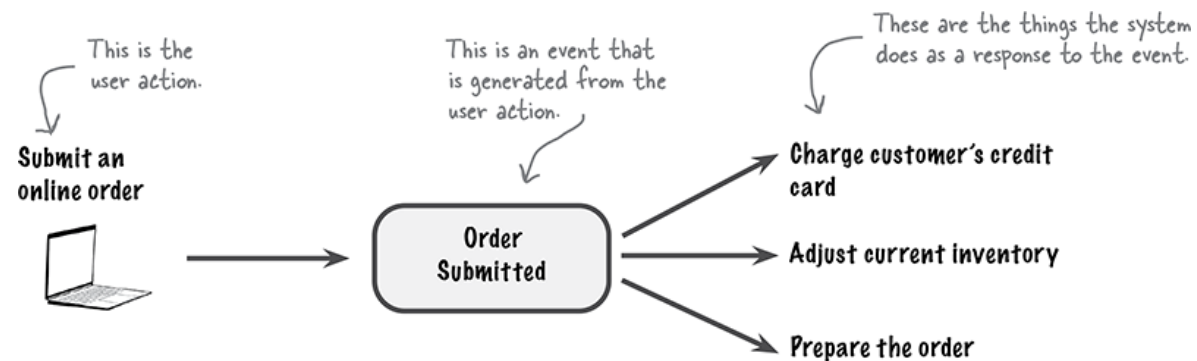
Introduction to Event-Driven Architecture

- ❖ Event-Driven Architecture (EDA) structures systems to **respond to events**, which are significant changes in system state.
- ❖ Unlike request-driven systems, EDA components don't directly call each other.
- ❖ **Example:**
 - An e-commerce system where placing an order triggers inventory updates, payment processing, and shipping—all asynchronously.



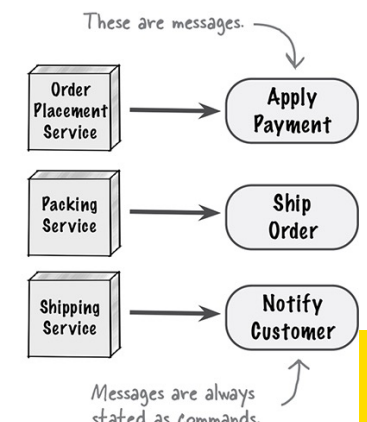
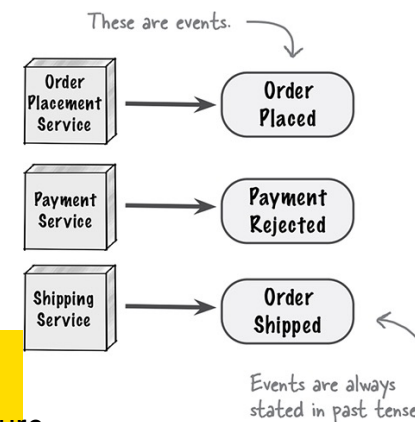
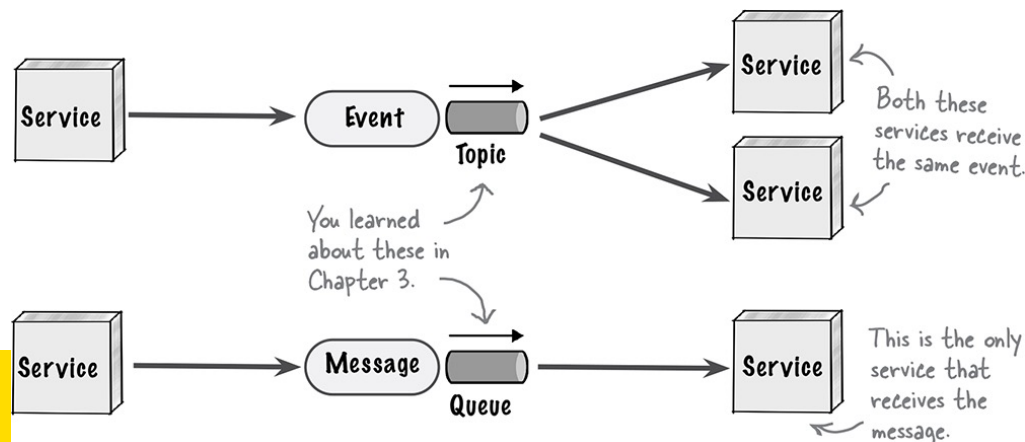
What is an Event

- ❖ An **event represents** something that has already occurred and carries data about it.
- ❖ Events are immutable and often used as **triggers**.
- ❖ **Example:**
 - "User Registered" event might include the user ID, name, and email.



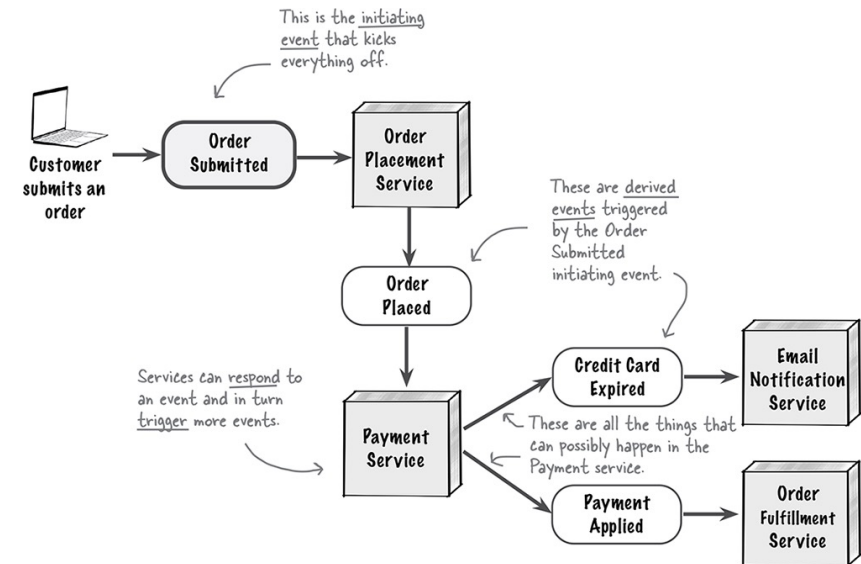
Even vs. Messages

- ❖ **Events** broadcast that something happened, with **no expectation** of response.
- ❖ **Messages** are more targeted, often **demanding action**.
- ❖ **Example:**
 - *Event*: "Item Added to Cart" (anyone can listen).
 - *Message*: "Process Payment" sent directly to payment service.



Initiating and Derived Events

- ❖ **Initiating** events are triggered by users or external systems.
- ❖ **Derived** events are consequences of those events and triggered by services
- ❖ **Example:**
 - *Initiating:* "Order Placed"
 - *Derived:* "Payment Authorized", "Inventory Deducted", "Shipping Scheduled"



Why Publish Events Others May Not Care About?

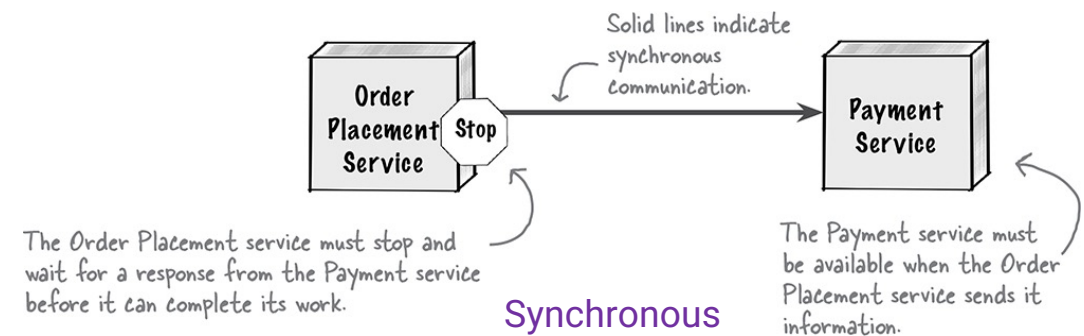
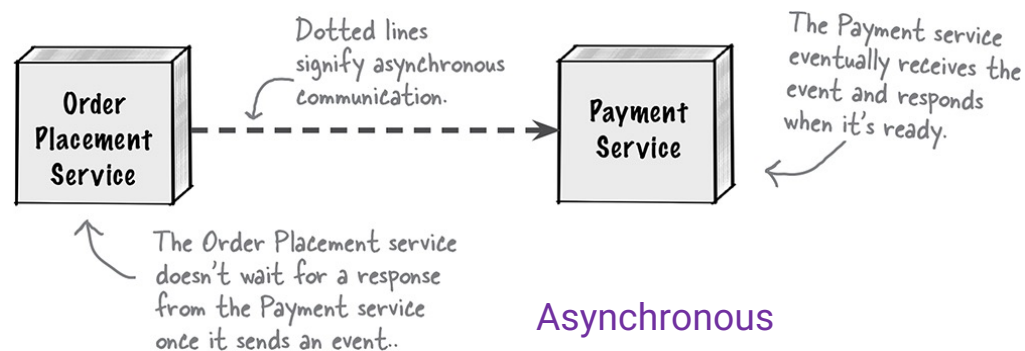
- ❖ **Broadcasting** all events allows new services to listen without modifying existing systems.
- ❖ **Example:**
 - Initially, only billing listens to "Order Placed".
 - Later, analytics can subscribe to the same event to track order trends.

Asynchronous vs. Synchronous Communication

- ❖ In **synchronous** calls, the sender waits for a response.
- ❖ In **asynchronous** communication, it continues immediately after sending.

❖ Example:

- *Synchronous*: REST API call to get shipping quote
- *Asynchronous*: Order service emits "Order Placed" and moves on



Benefits of Asynchronous Communication

- ❖ **Loose coupling** allows services to operate and scale independently, increasing speed and fault tolerance.

Example:

- With async processing, an online store confirms an order in 600ms; with sync processing, it takes 1800ms.

Trade-Offs of Async Communication

❖ Asynchronous systems **complicate debugging** and tracing since there's no immediate response.

❖ **Example:**

- When inventory **update fails**, the order service might not know.
- You **need to monitor** event failures separately.

Database Topologies in EDA

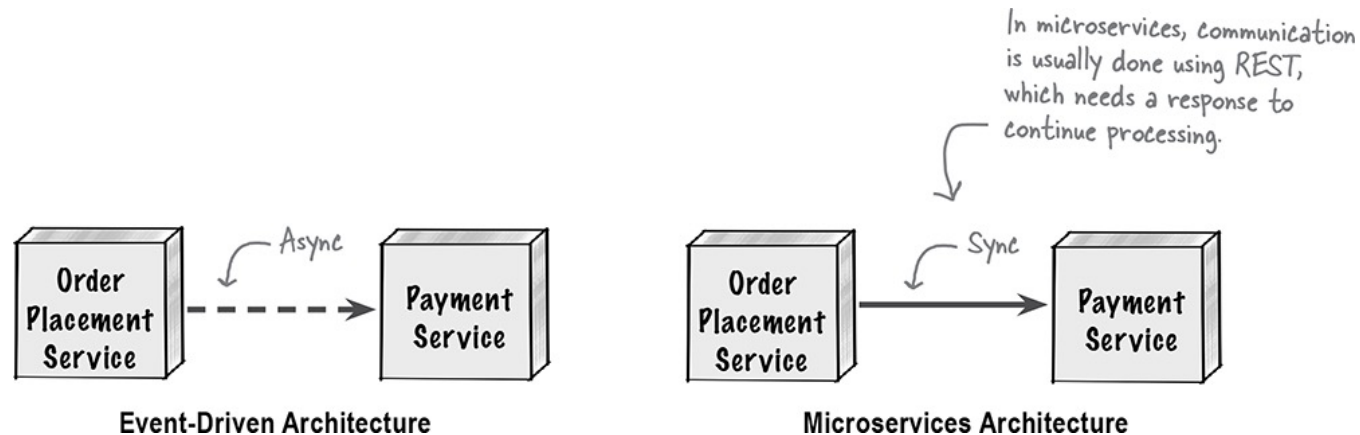
❖ How services access and manage data **affects modularity** and **scaling**.

❖ **Examples:**

- Monolithic DB: All services write to one database (fast, **tightly** coupled).
- Domain-Partitioned: Related services share DB (**moderately** coupled).
- DB-per-Service: Each microservice owns its DB (**fully decoupled**, but **complex** joins require events).

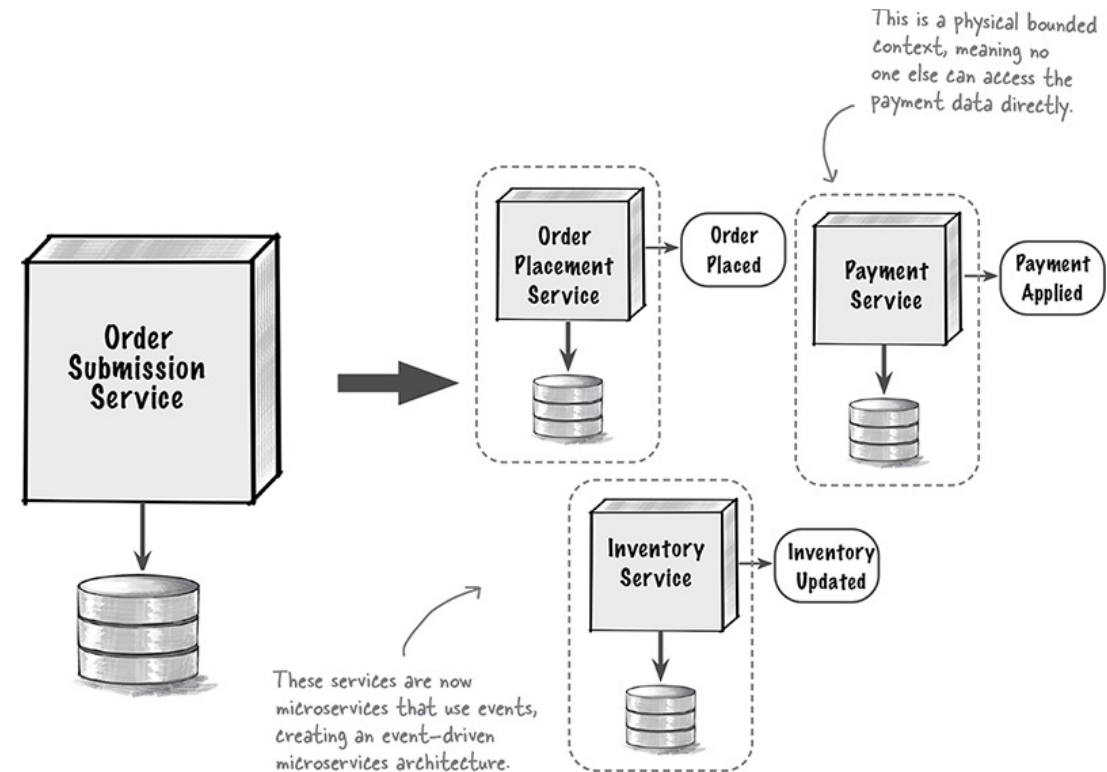
EDA vs. Microservices

- ❖ Microservices focus on small, **self-contained services** communicating via HTTP or RPC.
- ❖ EDA emphasizes **event-based** coordination.
- ❖ **Example:**
 - **Microservice** model uses REST to trigger payment.
 - **EDA** emits "Order Placed", and payment service listens and reacts.



Event-Driven Microservices

- ❖ This **hybrid model** combines independent services with event communication, boosting flexibility.
- ❖ **Example:**
 - Inventory, Shipping, and Billing each own their DB and listen to "Order Placed" to act asynchronously.



EDA Challenges

- ❖ EDA introduces **challenges with observability** and **testing** due to distributed asynchronous operations.
- ❖ **Examples:**
 - Event debugging is complex—errors are not immediate.
 - Testing sequences requires simulating full event flows.

EDA Advantages

- ❖ EDA shines in environments requiring **responsiveness**, **scalability**, and **autonomy**.
- ❖ Examples:
 - **Maintainability**: Add new features without changing existing services.
 - **Performance**: Events processed in parallel.
 - **Scalability**: Individual components scale independently.

Key Concepts

- ❖ EDA **emphasizes** responsiveness and extensibility but requires **thoughtful design** to manage complexity.
- ❖ Key Points:
 - **Events** = Immutable notifications of state change
 - **Asynchronous** = Fire-and-forget
 - **Combined with microservices** for modern architectures

Event-driven Architecture Star Ratings

Architectural Characteristic	Star Rating
Maintainability	★ ★ ★ ★
Testability	★ ★
Deployability	★ ★ ★
Simplicity	★
Evolvability	★ ★ ★ ★ ★
Performance	★ ★ ★ ★ ★
Scalability	★ ★ ★ ★ ★
Elasticity	★ ★ ★ ★
Fault Tolerance	★ ★ ★ ★ ★
Overall Cost	\$ \$ \$

While it's easy to find where to change code, testing and deployment are risky and hard.

Things like error handling and asynchronous communication make EDA complex.

Less service coupling means better scalability and elasticity.

Finally, an architectural style that performs well!

Because most things are asynchronous and decoupled, fault tolerance is really high.

Exercise

Which of the following systems might be **well suited** for the **event-driven architectural** style, and why?

An online auction system where users can bid on items

Why? _____

- ☐ Well suited for event-driven architecture
- ☐ Might be a fit for event-driven architecture
- ☐ Not well suited for event-driven architecture

A large backend financial system for processing and settling international wire transfers overnight

Why? _____

- ☐ Well suited for event-driven architecture
- ☐ Might be a fit for event-driven architecture
- ☐ Not well suited for event-driven architecture

A company entering a new line of business that expects constant changes to its system

Why? _____

- ☐ Well suited for event-driven architecture
- ☐ Might be a fit for event-driven architecture
- ☐ Not well suited for event-driven architecture

A small bakery that wants to start taking online orders

Why? _____

- ☐ Well suited for event-driven architecture
- ☐ Might be a fit for event-driven architecture
- ☐ Not well suited for event-driven architecture

A social media site where users can post and respond to comments

Why? _____

- ☐ Well suited for event-driven architecture
- ☐ Might be a fit for event-driven architecture
- ☐ Not well suited for event-driven architecture

Serverless Architecture

COMP2511, CSE, UNSW



UNSW
SYDNEY

Introduction to Serverless Architecture

- ❖ Serverless computing allows developers to build and run applications **without managing infrastructure**.
- ❖ Developers **focus** on deploying individual **functions** without managing servers.
- ❖ Cloud provider **dynamically manages** server allocation.
- ❖ Function is executed in **response to events**.
- ❖ Also known as **Function-as-a-Service (FaaS)**.

- ❖ **Example Platforms:**
 - **AWS Lambda**: Most popular serverless platform, integrated with the entire AWS ecosystem
 - **Azure Functions**: Serverless platform for Microsoft Azure users
 - **Google Cloud Functions**: Lightweight solution for Google Cloud services
 - **IBM Cloud Functions**: Based on Apache OpenWhisk

How Serverless Works

- ❖ User **sends request** (e.g., API call)
- ❖ API Gateway receives and **triggers** a Lambda/**Function**
- ❖ **Function processes** data and interacts with services (DB, storage, web services)
- ❖ **Result** returned to user

- ❖ **Example:**

1. An S3 bucket (cloud storage) uploads an image
2. The event triggers a Lambda function to resize the image
3. The function stores resized image in another S3 location (cloud storage)

Example: AWS Lambda

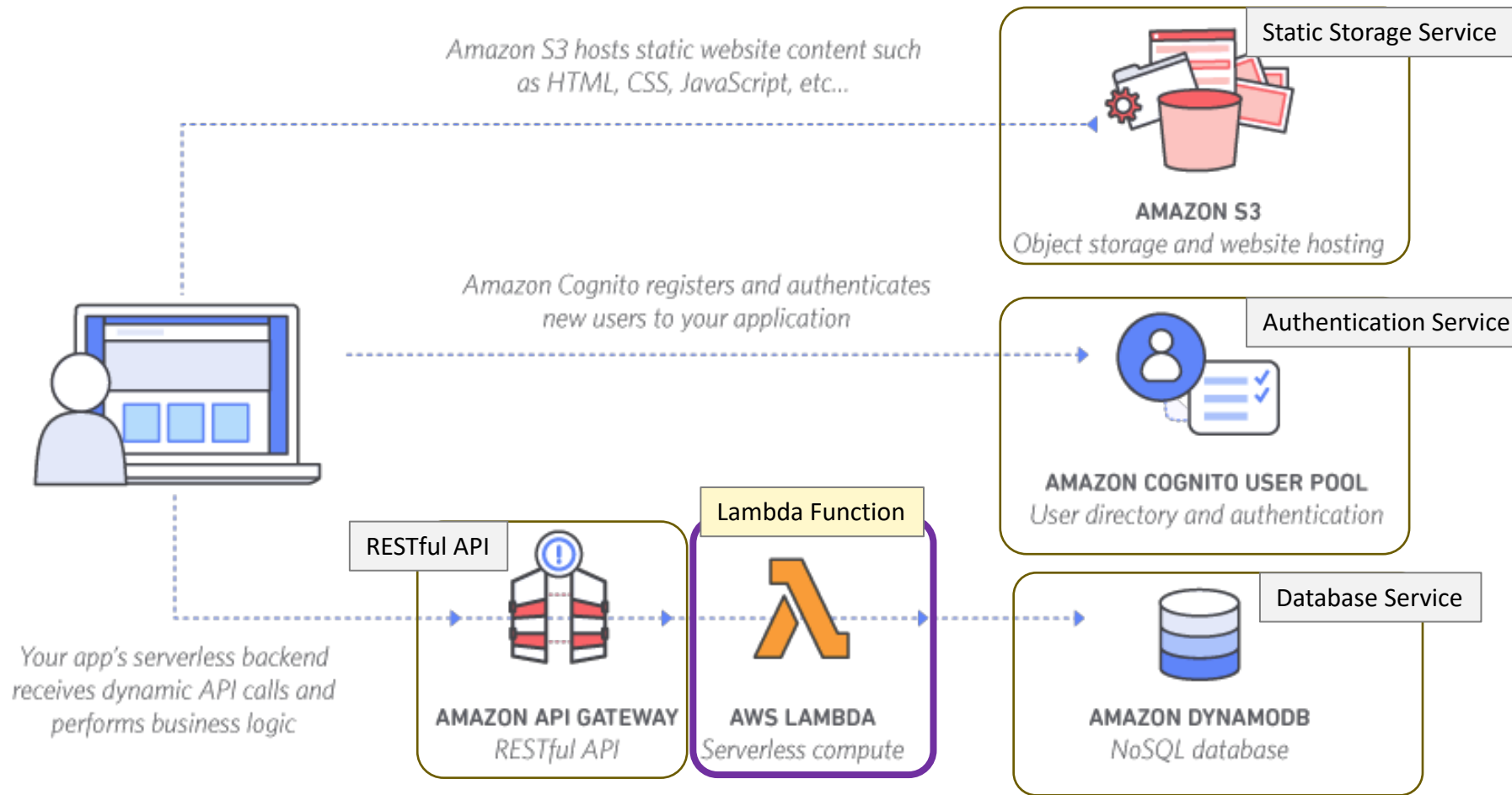


Diagram from <https://aws.amazon.com/lambda/web-apps/>

Key Characteristics

- ❖ **Auto-scaling**: Instantly handles thousands of concurrent executions
- ❖ **Faster time-to-market**: Developers focus on business logic, not infrastructure
- ❖ **High availability**: Functions are distributed across multiple availability zones
- ❖ **Event-driven**: Executes on triggers like HTTP requests, file uploads, or database changes.
- ❖ **Micro-billing**: You pay only for execution time, usage-based cost.
- ❖ **Short-lived functions**: Ideal for tasks that complete quickly.

Serverless Use Cases

- ❖ Form **submission triggers** a Lambda to store data in DynamoDB.
- ❖ Google Cloud Functions **reacts to** Firebase **database changes** and sends real-time notifications to users.
- ❖ Lambda **automatically resizes** images uploaded to S3 for use in different display formats.
- ❖ An e-commerce website uses Azure Functions to handle **inventory updates** on-demand.
- ❖ AWS Lambda processes incoming JSON health data from IoT devices, **generates alerts** if required, and stores data in Amazon DynamoDB for further analysis.

Serverless Design Principles

- ❖ **Stateless:** Don't rely on local memory; use shared storage (e.g., S3, DynamoDB)
- ❖ **Event-driven:** Design workflows around events, not request-response chains
- ❖ **Minimal and composable functions:** Keep single-responsibility per function
- ❖ **Use queues/pubs/subs:** Decouple flows using queues or Publish-subscribe messaging services

Limitations and Challenges

Cold starts:

- ❖ Latency when functions are idle for a while (especially for JVM/.NET)
- ❖ Mitigation: Use warm-up plugins or provisioned concurrency

Vendor lock-in:

- ❖ Tied to provider's ecosystem (e.g., AWS SDKs, IAM policies)

Observability:

- ❖ Harder to trace request flows across functions
- ❖ Solution: Use distributed tracing (e.g., AWS X-Ray, OpenTelemetry)

Resource limits:

- ❖ Timeout (after a few mins on AWS Lambda)
- ❖ Memory and ephemeral storage constraints

Comparison: Serverless vs. Microservices

Feature	Microservices (Containers)	Serverless (Functions)
Deployment Unit	Container	Function
Management	DevOps / CI/CD pipeline	Fully managed by provider
Cost Model	Fixed per compute unit	Per request, per execution time
Scaling	Container autoscaling	Scales with invocations
Startup Time	Low latency (warm)	Cold starts may delay execution
Monitoring	Full stack observability	Requires custom integration

Summary

- ❖ Serverless abstracts server management and **reduces operational burden**
- ❖ Works best for **stateless**, **event-driven**, and **high-concurrency** use cases
- ❖ **Challenges** include observability, cold starts, and vendor-specific tooling
- ❖ **Ideal** as a lightweight, cost-effective architecture for modern cloud-native apps