

# Software Engineering - Design

Summer Term 2024  
04 Modular Design

Prof. Dr. Robert Lokaiczyk  
Frankfurt University of Applied Sciences  
Faculty of Computer Science and Engineering  
[robert.lokaiczyk@fb2.fra-uas.de](mailto:robert.lokaiczyk@fb2.fra-uas.de)

Die Karrieremesse auf deinem Campus

**meet@frankfurt-university**

22.-23. Mai 24

10-16 Uhr

Nibelungenplatz 1

Foyer, Gebäude 4

# Aussteller

Messtag 1 | 22. Mai 2024

\* Teilnahme nur Online

Arbeitgeber  
Soziale Arbeit

# Aussteller

Messtag 2 | 23. Mai 2024



Arbeitgeber  
Soziale Arbeit



Johanniter-Unfall-Hilfe e. V.



\* Teilnahme nur Online

# Das erwartet dich:



> 50 Unternehmen



Freier Eintritt



Lockere Atmosphäre



Expertenwissen



Keine Anmeldepflicht



Karriere pushen



Barista



iPad gewinnen

# Agenda for today

Motivation Modular Design

Principles:

- Separation of concerns

- Cohesion and coupling

- Encapsulation

- Abstraction

Organize classes into packages

# What is modular design?

## Homogeneous design



Foto: vu anh <https://unsplash.com/de/@naomi365photography>

## Modular design



Foto: Ninoon / iStock.com

# Top 10 most popular Java projects on Github May 24

Name	Contributors	Used by	Classes	Packages	LOC
spring-projects/spring-ai	99		649	150	48.540
apache/hertzbeat	198		545	117	45.347
halo-dev/halo	121	184	973	74	67.017
Stirling-Tools/Stirling-PDF	106		182	26	93.069
alibaba/cola	29	221	393	99	9.402
skylot/jadx	117		1.584	220	124.041
xuxueli/xxl-job	58	7.900	133	51	9.104
alibaba/nacos	363	7.500	2.286	432	160.870
Xiaojieonly/Ehviewer_CN_SXJ	12		407	55	63.203




## SOFTWARE

This article is more than 1 year old

# How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript

172 

Code pulled from NPM – which everyone was using

 [Chris Williams](#), Editor in Chief

Wed 23 Mar 2016 // 01:24 UTC



**UPDATED** Programmers were left staring at broken builds and failed installations on Tuesday after someone toppled the Jenga tower of JavaScript.

A couple of hours ago, Azer Koçulu unpublished more than 250 of his modules from NPM, which is a popular package manager used by JavaScript projects to install dependencies.

Koçulu yanked his source code because, we're told, one of the modules was called Kik and that apparently attracted the attention of lawyers representing the instant-messaging app of the same name.

According to Koçulu, Kik's briefs told him to rename the module, he refused, so the lawyers went to NPM's admins claiming brand infringement. When NPM took Kik away from the developer, he was furious and unpublished *all* of his NPM-managed modules. "This situation made me realize that NPM is someone's private land where corporate is more powerful than the people, and I do open source because Power To The People," Koçulu blogged.

Unfortunately, one of those dependencies was left-pad. The code is below. It pads out the lefthand-side of strings with zeroes or spaces. And thousands of projects including Node and Babel relied on it.

# Modularity

- The basic idea behind modularity is to partition the system such that parts can be designed and revised independently (another application of the “divide and conquer” approach).

*“Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.” - Barbara Liskov*

*“The connections between modules are the assumptions which the modules make about each other.” - David Lorge Parnas*



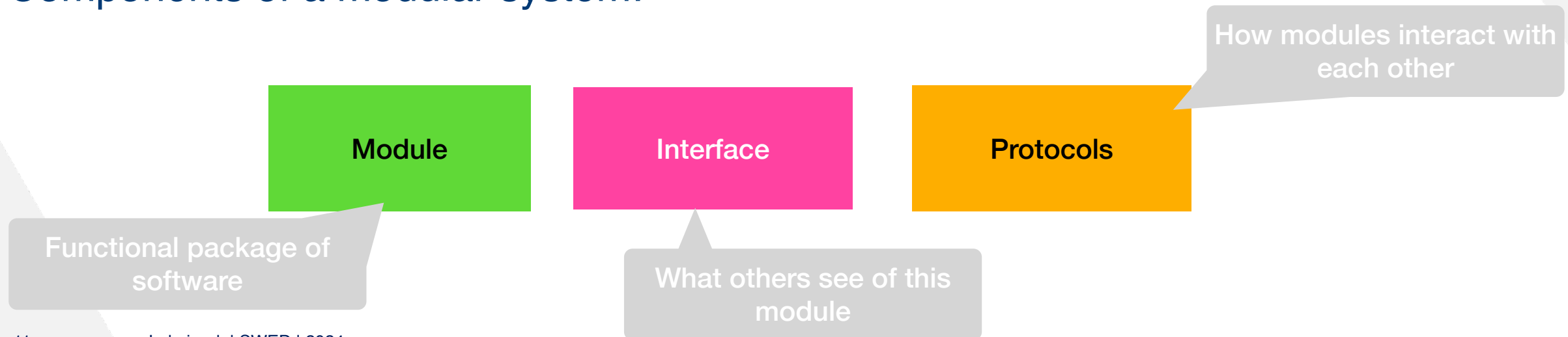
# Definition

## modular decomposition.

(1) The process of breaking a system into components to facilitate design and development; an element of modular programming.

IEEE Standard Glossary of Software Engineering Terminology, 1990

## Components of a modular system:



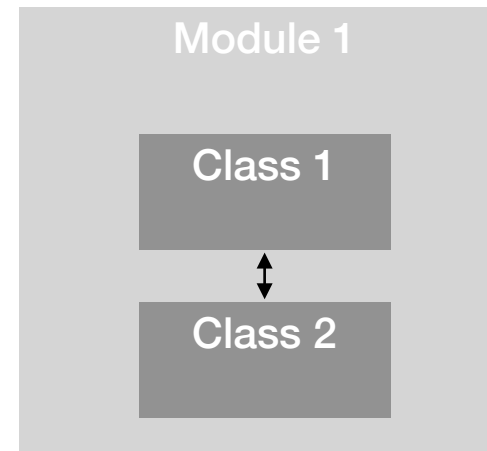
# Separation of concerns

How to unbundle a complex system of software?

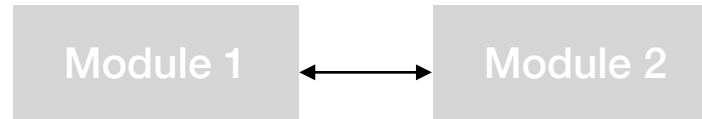
- Separation of Concerns (SoC) is the principle of **breaking a system into distinct, loosely coupled components**, each **responsible for a single** aspect of **functionality**.
- **Impact on Modularity**
  - By separating concerns, **changes to one part** of the system are **less likely to impact other parts**.
  - Modules should be changeable and reusable independent of other modules
  - SoC promotes modularity by reducing complexity and dependencies between components, allowing for **easier development, testing, and maintenance**.

# Cohesion and coupling

- **Cohesion:**
  - Cohesion of a class(es) refers to the degree to which the responsibilities of the class(es) are related and focused towards a single purpose or functionality. High cohesion exhibits strong internal unity, with its methods and attributes closely related and working together to achieve a specific task or objective. Low cohesion indicates that the class(es) have disparate or unrelated responsibilities.



# Cohesion and coupling



- **Coupling:**

- Coupling between classes refers to the level of dependency and interaction between different classes of a system. It measures how closely a set of classes (module) is interconnected and relies on other modules.
- Low coupling indicates that modules are relatively independent and have minimal dependencies on each other.
- High coupling, on the other hand, signifies strong dependencies between modules, where changes in one module can have significant impacts on other modules.

# Cohesion and coupling

- **Forms of coupling in Java:**
  - A has an attribute referring to a B instance or B.
  - A object calls services of a B object.
  - A features a method referencing B, or an instance of B.
  - A is a direct or indirect subclass of B.
  - B is an interface, A implements interface.

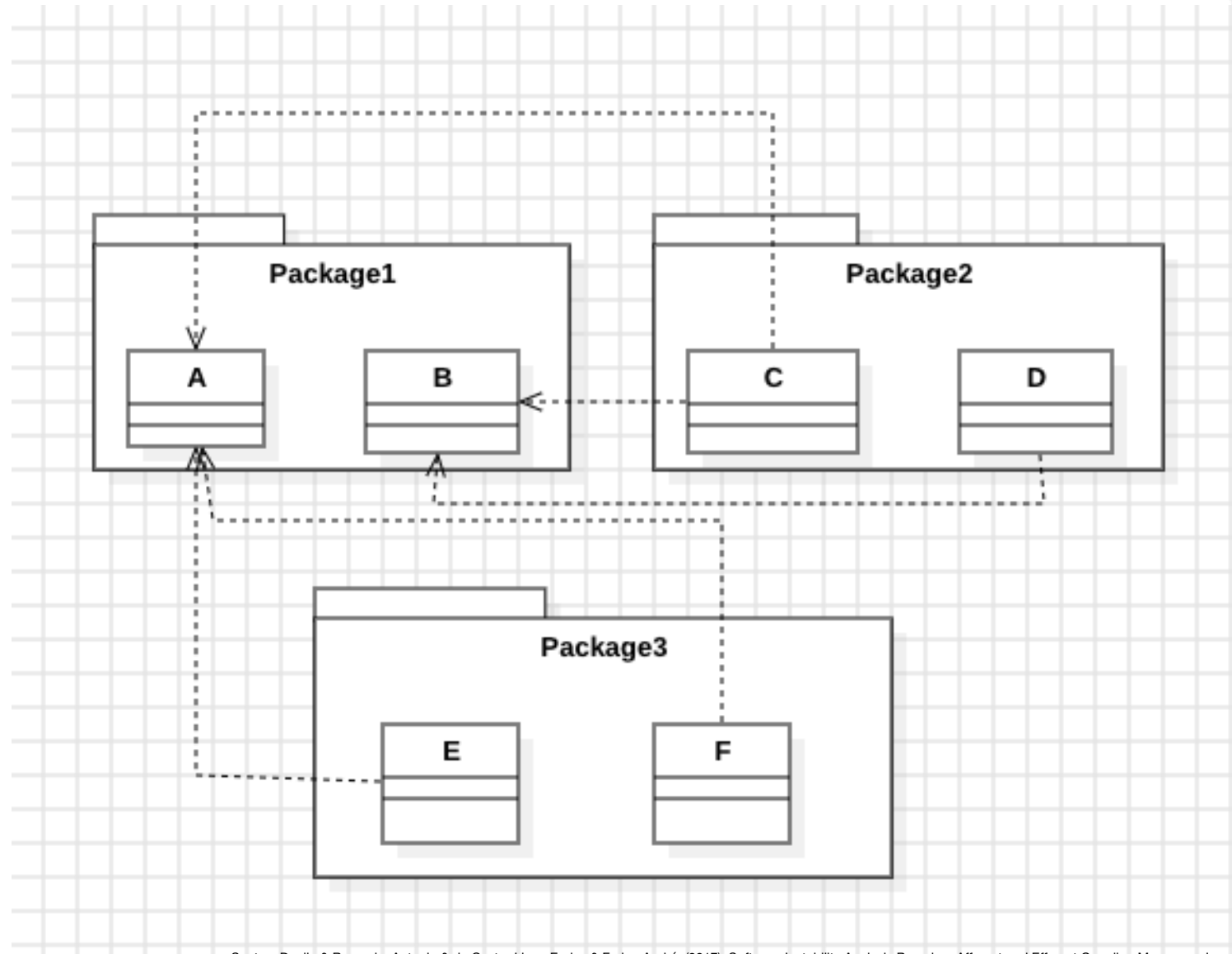
# How to measure?

- **Afferent (incoming) coupling** (AC) is the number of classes outside of a particular module that depend on classes within that module.
- **Efferent (outgoing) coupling** (EC) refers to the number of classes within a particular module that depend on classes outside of that module.
- **Instability** of a module is defined
  - Closer to 1 means unstable
  - Closer to 0 means stable

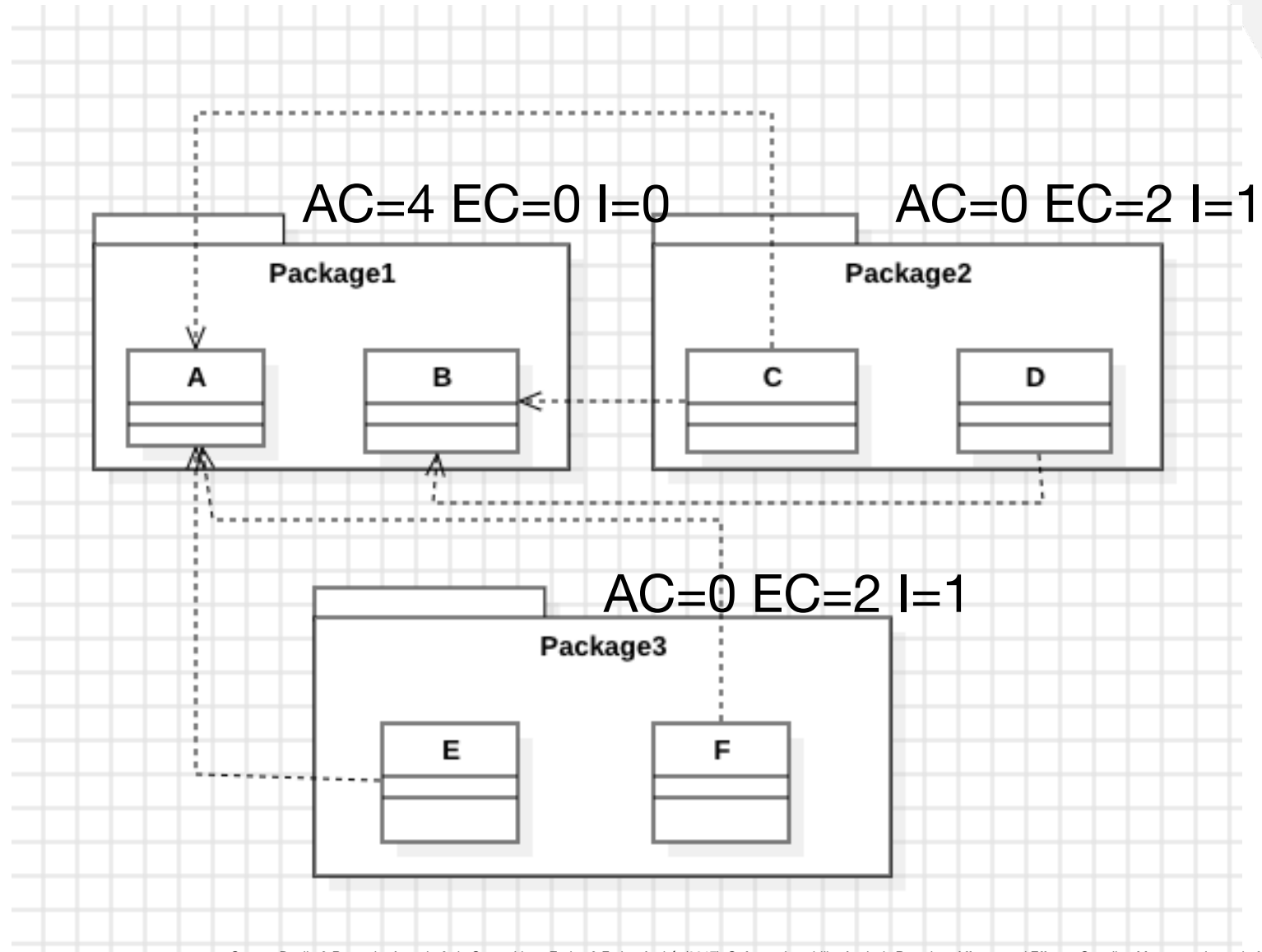
$$I = \frac{EC}{EC+AC}$$



# Example



# Example



# Example

- In order to **calculate the value of AC** for Package1, a software engineer should **count the number of classes out of the package that have dependencies incident on it**. For example, the package Package1 has classes called A and B. The sets that define the classes dependent on **A and B** are **{C, E, F}** and **{C, D}**, respectively. The union of these two sets is the set of classes that depend upon Package1. In this case, the **union set is given by {C, D, E, F}**. Therefore, a software engineer concludes the AC value for Package1 is **equal to 4**, the total elements in this union set {C, D, E, F}. It is noteworthy that class C has more than one dependency relationship focusing on Package1, as seen in the two dependency sets. However, the software engineer **counts all dependencies only once**, even a class that has more than one incident dependency relationship on the same package, like in the case of class C.

# The Law of Demeter

How to reduce coupling?

- *The **Law of Demeter** (or Principle of Least Knowledge), named after the greek goddess of Demeter: one should only talk to friends or (more formally) the Law of Demeter for functions requires that a **method M** of an **object O** may **only invoke the methods of the following objects**:*
  - *O itself*
  - *M's parameters*
  - *any objects created/instantiated within O's direct component objects*
  - *a global variable, accessible by O, in the scope of M*

```
1 // instead of
2
3 address = order.item[item_id].manufacturer.address
4
5 // use
6
7 address = order.getManufacturerAddressFor(item_id)
8
9 address = shop.getAddressFor(item_id)
10
```

# Abstraction

Consider a **car** as an **abstraction**. When you **interact** with a car, you don't need to understand all the many details of how the engine, transmission or braking system work internally. Instead, **you interact with high-level concepts** such as accelerating, braking, and steering. The car **abstracts away the complexities** of its internal components, allowing you to **operate it effectively** without needing to understand the inner workings of each subsystem. This abstraction simplifies the user's mental model and makes the car **easier to use**.



# Example in Java

```
1 // Abstract class representing a shape
2 abstract class Shape {
3     // Abstract method to calculate area
4     abstract double calculateArea();
5 }
6
7 // Concrete subclass representing a circle
8 class Circle extends Shape {
9     private double radius;
10
11     // Constructor
12     public Circle(double radius) {
13         this.radius = radius;
14     }
15
16     // Implementation of abstract method to calculate area
17     @Override
18     double calculateArea() {
19         return Math.PI * radius * radius;
20     }
21 }
```

```
23 // Concrete subclass representing a rectangle
24 class Rectangle extends Shape {
25     private double width;
26     private double height;
27
28     // Constructor
29     public Rectangle(double width, double height) {
30         this.width = width;
31         this.height = height;
32     }
33
34     // Implementation of abstract method to calculate area
35     @Override
36     double calculateArea() {
37         return width * height;
38     }
39 }
40
41 public class AbstractionExample {
42     public static void main(String[] args) {
43         // Create instances of Circle and Rectangle
44         Circle circle = new Circle(5);
45         Rectangle rectangle = new Rectangle(4, 6);
46
47         // Calculate and print areas
48         System.out.println("Area of Circle: " + circle.calculateArea());
49         System.out.println("Area of Rectangle: " + rectangle.calculateArea());
50     }
51 }
```

# Encapsulation principle

**Definition:** Encapsulation is the bundling of data and methods that operate on that data into a single unit, known as a class with the purpose of hiding the internal state of an object and restrict access to it from outside the class.

- no part of a complex system should depend on internal details of another
- internal details (state, structure, behaviour) become the object's secret

# Example in Java

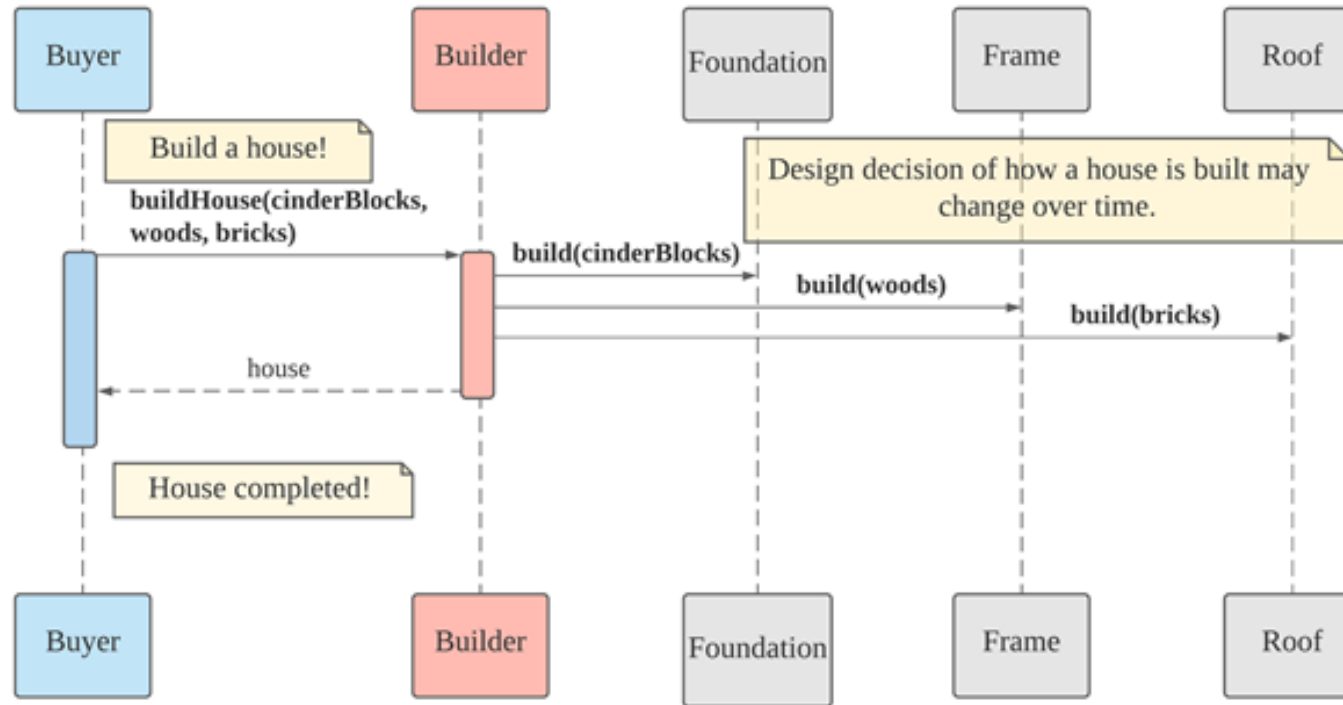
```
1 public class BankAccount {
2     private String accountNumber; // Encapsulated data
3     private double balance; // Encapsulated data
4
5     // Constructor
6     public BankAccount(String accountNumber) {
7         this.accountNumber = accountNumber;
8         this.balance = 0.0; // Initialize balance to zero
9     }
10
11     // Method to deposit money into the account
12     public void deposit(double amount) {
13         if (amount > 0) {
14             balance += amount;
15             System.out.println(amount + " deposited successfully.");
16         } else {
17             System.out.println("Invalid amount for deposit.");
18         }
19     }
20
21     // Method to withdraw money from the account
22     public void withdraw(double amount) {
23         if (amount > 0 && amount <= balance) {
24             balance -= amount;
25             System.out.println(amount + " withdrawn successfully.");
26         } else {
27             System.out.println("Insufficient funds or invalid amount for withdrawal.");
28         }
29     }
30
31     // Method to get the current balance
32     public double getBalance() {
33         return balance;
34     }
35 }
```



# Access modifiers recap

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
private	Yes	No	No	No

# Why information hiding?



- Hide implementation details within modules or classes and expose only necessary interfaces to interact with them

# How to organize classes into packages in Java

- Choose package name that reflects the project's structure, domain and organization
- Common conventions include using domain names in reverse order (com.uas.project) or functional groupings (e.g., com.uas.project.controller, com.uas.project.service)
- Keyword `package` in Java
- Group related classes together based on their functionality, responsibilities, or domain concepts
- Classes that collaborate closely or serve similar purposes should be organised into the same package.
- Aim for **high cohesion** within packages, meaning that classes within the same package should have strong relationships and work together closely and **low coupling** among packages

```
1 // Define a package named "com.example.myapp"
2 package com.example.myapp;
3
4 // Define a Class within the package
5 public class MyClass {
6     // Class implementation
7 }
```

# Advantages of modularity

- **Ease of Maintenance:** Modular systems allow developers to isolate and work on individual components independently, making it easier to understand, maintain, and update the codebase. Changes made to one module have minimal impact on other parts of the system.
- **Code Reusability:** Modular systems allow the creation of reusable components that can be easily integrated into different projects or used across multiple parts of the same project. Consequently we don't need to recreate functionality from scratch.
- **Scalability:** Modular architecture facilitates the scaling of software systems by allowing components to be added, removed or modified without disrupting the entire system.
- **Concurrent Development Speedup:** With a modular system, development teams can work on different modules concurrently, speeding up the development process and enabling parallelisation of tasks - resulting in a faster time-to-market.
- **Testing:** Isolated components can be tested and debugged separately. We can find and fix errors faster, leading to higher software quality and reliability.
- **Flexibility and Adaptability:** New features (as new components) can be added or existing ones modified with minimal disruption to the overall system uptime.

# Final remarks

- **Summary:**
  - You understand the concept of modularity.
  - You can calculate different versions of coupling given classes.
  - You understand how to reduce coupling.
  - You can define, understand and apply the basic principles of modularity.
  - You understand why modularity is helpful in the software engineering process.
- **Next week:** Design Pattern
- **Please read:**