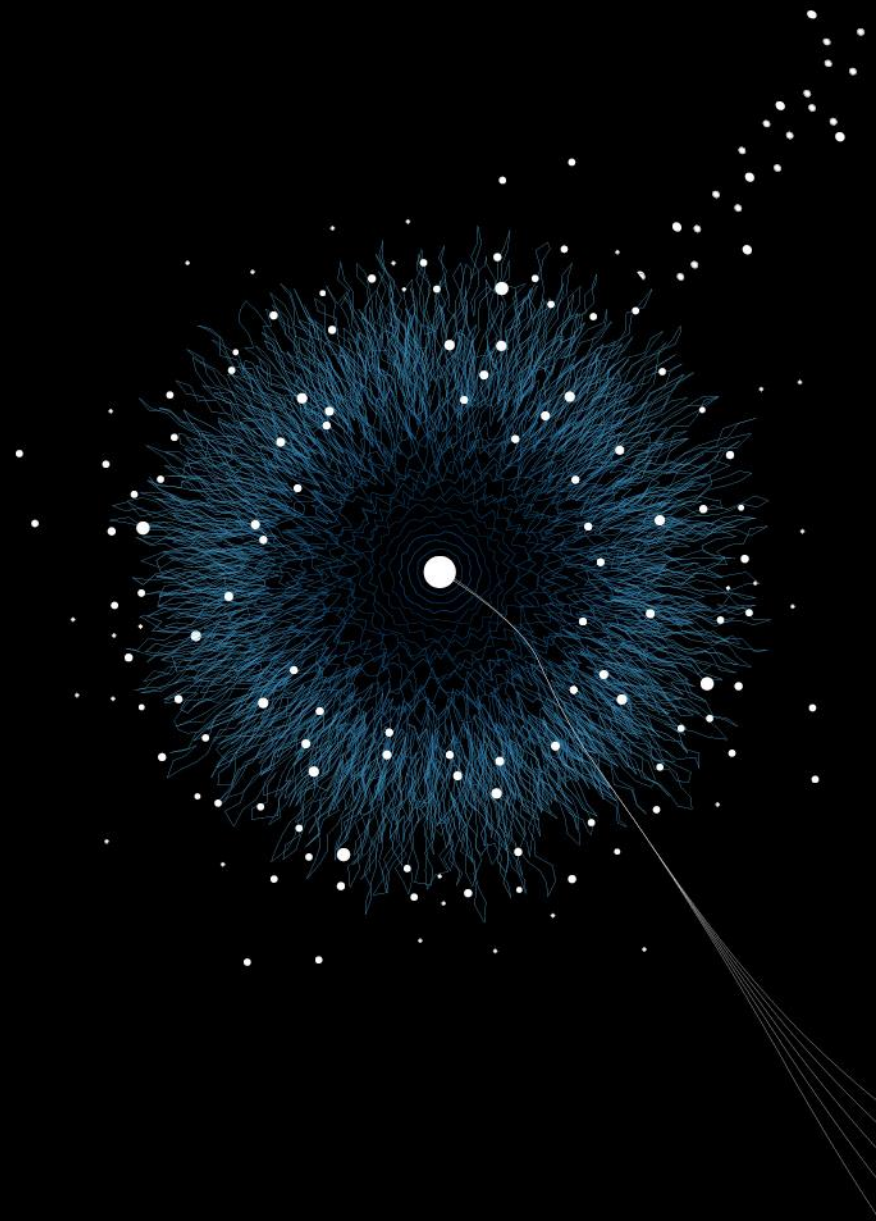
DESIGN PATTERNS

M2 PROGRAMMING LECTURE 5.1

BY TOM VAN DIJK



UNIVERSITY
OF TWENTE.

TOPICS OF THIS LECTURE

- Summary and questions about Week 4
- Design patterns (from another perspective)

PROGRAMMING OVERVIEW

Week 1 (IP) Procedural programming Arrays Debugging	**Week 2 (IP)** Classes and Objects Documenting and Testing Exceptions (AP)	**Week 3 (IP)** Interfaces, Inheritance, Polymorphism Subtyping Streams and Files (AP)
Week 4 (AP) Collections: Set, List, Map Specification with JML	**Week 5 (AP)** Design Patterns User Interfaces	**Week 6 (AP)** Basic Concurrency Project kick-off
Week 7 (AP) Basic Networking Defensive programming Security	**Week 8/9 (Project)** Project	**Week 10 (Project)** Project Exam

PROGRAMMING OVERVIEW

Java Notes book:

- Week 1: Chapter 1 – 4
- Week 2: Chapter 5.1 – 5.4
- Week 3: Chapter 5.5 – 5.8
- Week 4: Chapter 7.1 – 7.3, 8.1 – 8.3, 9.1 – 9.2, 10.1 – 10.4, 11.1 – 11.2
- Week 6: Chapter 12.1 – 12.3
- Week 7: Chapter 11.4

SO FAR

Good programs are

- Functional
- Reliable, testable
- Understandable, documented
- Formally specified / verified
- Designed for **change**

SO FAR

- (**W1**) Procedural programming (variables, types, procedures, control statements)
- (**W2**) Basics of object-oriented programming (classes, packages, static, final, public, private, constructors, standard methods)
- (**W2**) Program design principles (encapsulation, single responsibility principle, cohesion, coupling, command-query separation)
- (**W2**) Testing and code readability
- (**W2**) Exceptions
- (**W3**) Inheritance, polymorphism, typing, abstract classes, inheritance
- (**W3**) Streams and files
- (**W4**) Java Collections (List, Set, Map, generics)
- (**W4**) Specifications with JML

Q&A

Introduction to Programming:

- procedural programming
- computational thinking
- object-oriented programming 1 and 2
- testing and code readability / documentation
- the practice exam

Advanced Programming:

- exceptions
- streams and files
- java collections, list, set, map, generics
- specifications with JML

STRUCTURE

- (**W1**) Variables and types **structure** memory
- (**W1**) Procedures **structure** code
- (**W2**) Classes and packages **structure** variables and procedures
- (**W3**) Inheritance **structures** related classes
- (**W4**) Data**structures** and algorithms are core techniques to organise and process data
- (**W5**) Design patterns are **reusable structures** for solving common design problems

DESIGN PATTERNS

Design patterns are common ways to structure a software system (also: **computational thinking**!)

- (Low-level) design patterns
  - **Creational** patterns — Control how objects are created (e.g., Factory, Builder).
  - **Behavioral** patterns — Organise communication and responsibility between objects (e.g., Strategy, Listener).
  - **Structural** patterns — Organise relationships between classes and objects (e.g., Adapter, Decorator).
- (High-level) architecture patterns
  - **Layered** architecture
  - **Event-driven** architecture
  - **Client-server** architecture
  - Others: Broker, Pipe and Filter, MVC/MVVM, Hexagonal, Microservices, Microkernel
- See: https://refactoring.guru/design-patterns

WHY DESIGN PATTERNS?

- These are best practices to improve the structure of software
- Design for easier change / maintenance / growth
- Simplify reasoning about software and correctness
- Easier to test

CREATIONAL PATTERNS

- Problem: how / where to create objects?

- Idea: create objects in one place only

- **Constructors** solve the **how** problem of creating objects
- **Factory** solves the problem of selecting which class to construct
- **Builder** solves the problem of constructing complex objects

- Only the Factory / Builder class knows the constructor (i.e., the class)
- Other classes (users) only need to know the interface

- Use it to hide complexity of creating objects
- Avoid unnecessary (unstable) coupling

FACTORY / BUILDER PATTERN

- A Factory class knows how to create objects sharing an interface

```
Shape shape = ShapeFactory.getShape(Shapes.CIRCLE);
Element el = UIFactory.createButton("Submit");
Game game = TicTacToeGameFactory.createGame();
```

- Example:
  - `Game` interface
  - `SimpleTicTacToeGame` ***internal*** implementation of `Game`
  - `TicTacToeGameFactory` constructs `SimpleTicTacToeGame` and returns as `Game`
  - Other classes only know `TicTacToeGameFactory` and `Game`, not the actual implementation

  - Can replace the actual implementation with a change in `TicTacToeGameFactory`
  - Package-private `SimpleTicTacToeGame` constructor

FACTORY / BUILDER PATTERN

Builder goes further: step by step construction via a chain of methods.
Solves the problem of constructing a complex object (instead of having many different possible constructors)

```
House house = new House.HouseBuilder()
        .setWalls("Brick Walls")
        .setRoof("Concrete Roof")
        .setPool("Swimming Pool")
        .build();
```

BEHAVIORAL PATTERNS

How objects collaborate, divide responsibilities, etc.

- **Strategy**
  - Enables a class to select different algorithms (behaviors) at runtime.
- **Observer** / **Listener** / **Publisher-Subscriber**
  - Avoids hardcoding responses to state changes or events.
- **Chain of Responsibility**
  - Passes requests through a sequence of potential handlers until one can deal with it.
- **State**
  - Eliminates messy conditional logic by letting an object change its behavior when its internal state changes

STRATEGY PATTERN

Part of the behavior is delegated to an implementation of an interface and can be changed by selecting a different implementation.

Examples

- Different AI strategies for winning Tic-Tac-Toe: random move, greedy heuristic, minimax, Monte Carlo sampling, etc.
- Different algorithms to compute the best route between two points
- Different payment methods in an online shop
- Different encoding/compression/encryption techniques
- Different file export formats

Why?

- Replace inheritance by composition
- Easily allow selecting an algorithm without changing much code

LISTENER / PUBLISHER-SUBSCRIBER

- One or more objects should react to certain events in the system
- Want to avoid “hardcoding” behavior where the event occurs
- Want to add / change behavior later, without changing code where the event occurs

- Example:
  - Graphical user interface (button clicked, mouse moved, window resized)
  - Game (a game starts, a game is won, a move is played)
  - Chat application (message is sent or received, contact comes online)
  - Sensor monitoring (change of temperature, position, low battery, object detection)

- Either place all handling code right there, or call listeners
- Often the business logic should not know about UI, logging, sound, notifications, etc.

APPLYING LISTENER / PUBLISHER-SUBSCRIBER

- Create an interface `???Listener` with one (or more) methods for the event(s)
  - Example: `GameListener` with `gameMovePlayed(Game game, Player player, Move move);`
  - Example: `ConnectionListener` with `lineReceived(Connection connection, String line);`

- Create a field in the **publisher class**
  - `private Set<???Listener> listeners;`
  - `private List<???Listener> listeners;`

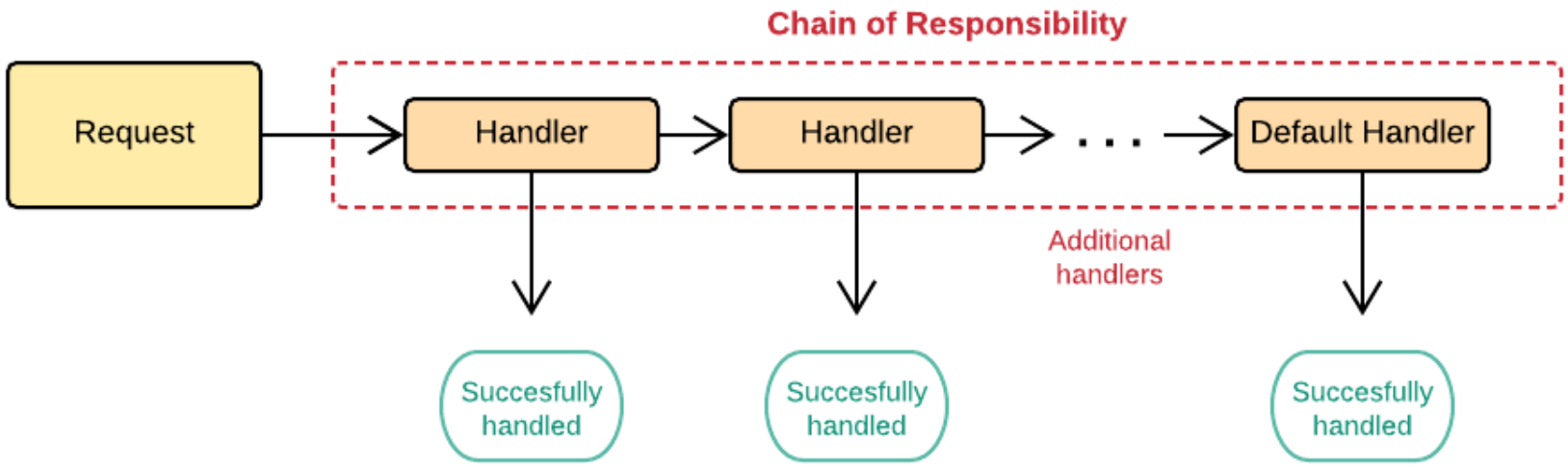
- Create methods in the **publisher class**
  - `addListener(???Listener listener)`
  - `removeListener(???Listener listener)`

- Implement the interface to create **subscriber classes** (listeners)

- Use a `for` loop to invoke all listeners by calling the suitable method of the interface

CHAIN OF RESPONSIBILITY

- Pass a request along a chain of handlers
- Each handler either processes the request or passes it to the next handler
- For example, handling different commands in a server



CHAIN OF RESPONSIBILITY

- First define a `XXXXHandler` interface!

```
boolean processRequest(...);
```

- Each handler implements this interface

- Then create the chain:
  - Define a List (order!) of handlers
  - Consider a default handler at the end of the chain
  - When handling the request, follow the chain until a handler processes it

- Advantage: cleaner code (separation of concerns) resulting in fewer bugs
- Easier to change by adding/removing a handler.

CHAIN OF RESPONSIBILITY

Example of receiving a request from a client in Client-Server architecture:

- Handle logging in?
- Handle authentication and encryption?
- Handle a chat message?
- Handle a game move?
- Handle a list-of-online-players command?
- Handle an unknown request… (default handler)

STATE

- The object changes behavior depending on internal state.

```
public class Thing {
    interface State {
        State doThis(int parameter);
        State doThat(String text, int number);
    }

    private State state = new InitialState();
    public void doThis(int parameter) { state = state.doThis(parameter); }
    public void doThat(String text, int number) { state = state.doThat(tekst, number); }
}
```

- Different than.a **Strategy**: a State is internal and changes internally.
- Often associated with state machines: depending on the state, the object reacts differently to actions or events.
- Advantage: cleaner code, separation of concerns, different states don't accidentally interact

STRUCTURAL PATTERNS

Patterns related to composition of objects

- **Facade**
  - Provides a simplified interface to a complex subsystem.
- **Adapter**
  - Transforms an interface to make classes compatible.
  - Code written for one API can now work with another API via the Adapter
- **Decorator**
  - Dynamically adds functionality to objects by wrapping them.
  - Examples: The InputStream/OutputStream classes; or adding encryption to a connection.

MODEL-VIEW-CONTROLLER

- Separate the Model from the User Interface (View+Controller)

- Any **display things code** goes to the **View** classes
- Any **responding to input code** goes to the **Controller** classes
- The **rest** is in the **Model**

- Or: Model-View-ViewModel (MVVM)
- Or: Model-View-Presenter (MVP)
- Or: etc etc
- MVVM, MVP, etc, offer different design choices and (dis)advantages

MODEL-VIEW-CONTROLLER

Design patterns in the Model-View-Controller architecture

- You can use **Listeners** to separate UI from model (change UI without any changes in the model)

- Further design patterns to structure the UI component
  - **Strategy** and **Listener** within **Controller** (what does a button do?)
  - **Composite** within **View** (complex hierarchical layouts)

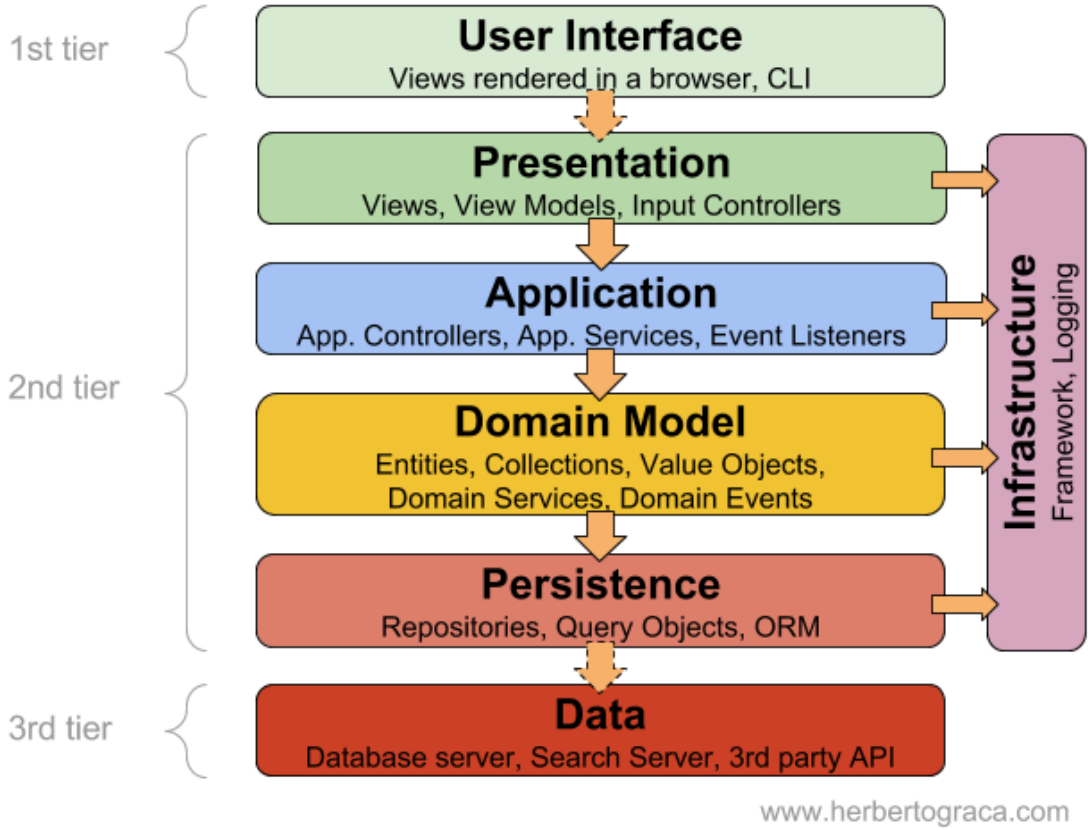
DESIGN PATTERNS

Design patterns are common ways to structure a software system (also: computational thinking!)

- (Low-level) design patterns
  - **Creational** patterns — Control how objects are created (e.g., Factory, Builder).
  - **Behavioral** patterns — Organise communication and responsibility between objects (e.g., Strategy, Listener).
  - **Structural** patterns — Organise relationships between classes and objects (e.g., Adapter, Decorator).
- (High-level) architecture patterns
  - **Layered** architecture
  - **Event-driven** architecture
  - **Client-server** architecture
  - Others: Broker, Pipe and Filter, MVC/MVVM, Hexagonal, Microservices, Microkernel
- See: https://refactoring.guru/design-patterns

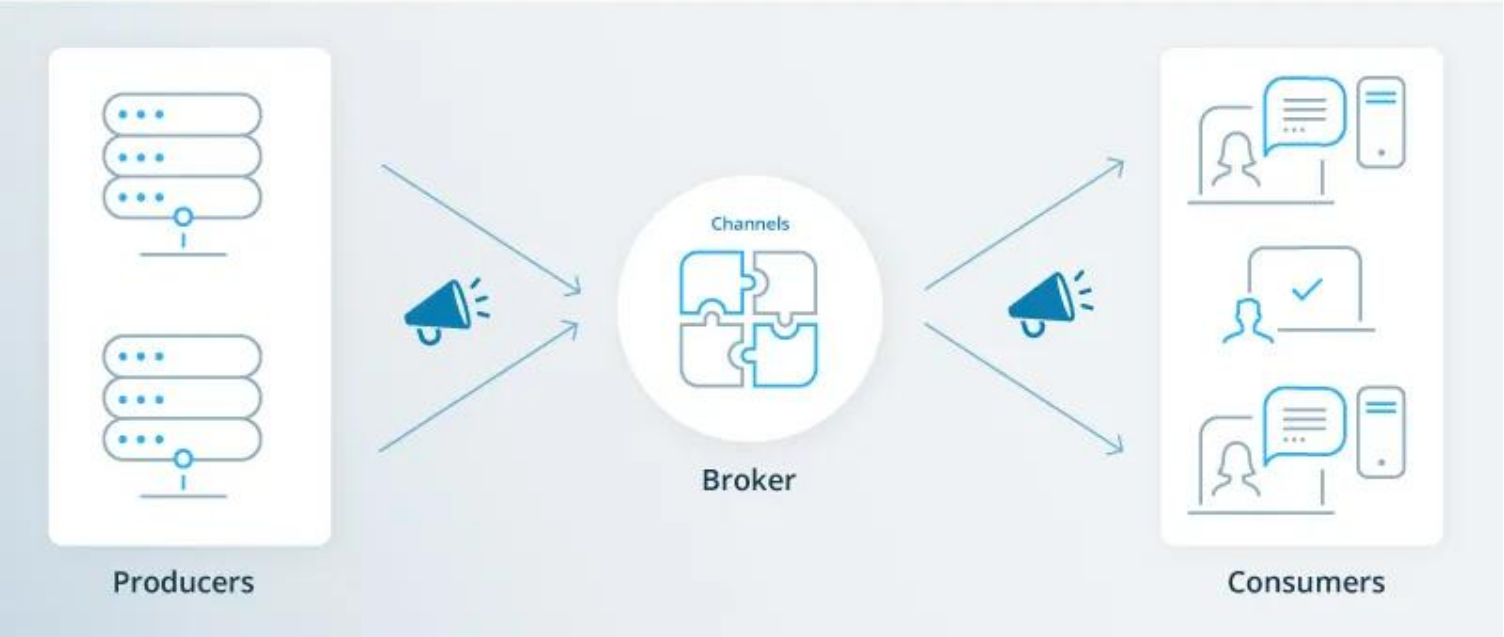
LAYERED ARCHITECTURE

- Layers are **independent**
  Changes do not affect other layers
- Layers are **closed**
  A layer only depends on the layer directly below (coupling)
- Lower layers **must not reference** higher layers
  Information flows upward via return values / listeners
- **Advantages**
  - Separation of concerns, loose coupling
  - Easier to understand and reason about
  - Easier testing and maintainability, security
  - More reusable
  - Each layer can be developed/maintained independently



EVENT-DRIVEN ARCHITECTURE

- Based around an **event bus** or **event broker** or **event router**
- **Producers** generate events when a change or action happens
- **Consumers** react to events by processing data or performing an action



EVENT-DRIVEN ARCHITECTURE

- Examples of **webshop**
  - New order from the website
  - New items in stock
  - Question to customer relations
  - Update on delivery

- Examples of **game**
  - A client connects to the server
  - A user queues for a game
  - A move is played in a game
  - A game is finished
  - A chat message is received

EVENT-DRIVEN ARCHITECTURE

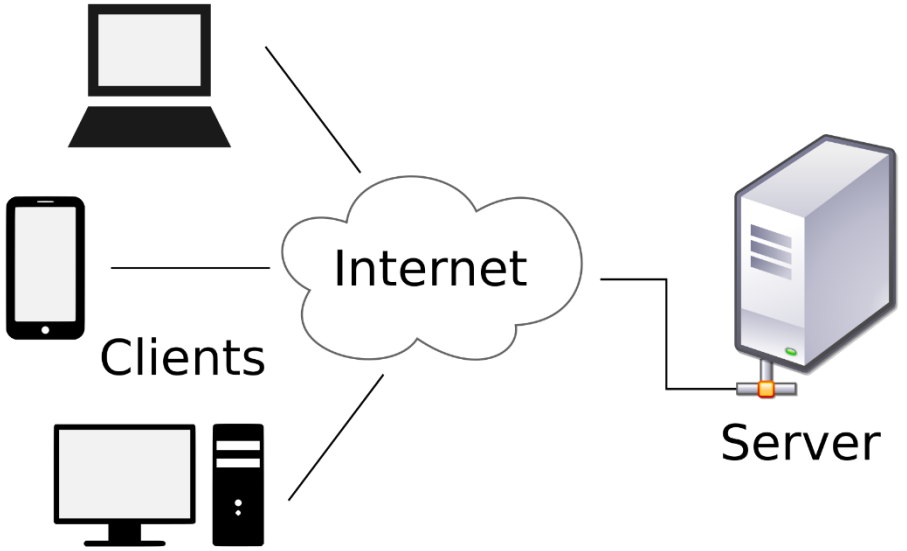
- **Advantages**
  - Loose coupling: components communicate through events, they do not know about each other
  - Easy to modify: can add or change producers and consumers without changing other producers or consumers
  - More responsive system
- **Pitfalls**
  - Debugging can be tricky
- **Implementation**
  - For every event: an event class (or interface) and a listener interface (for subscribers to implement)
  - Either publishers/subscribers register centrally in an event bus, or subscribers register with publishers

CLIENT-SERVER ARCHITECTURE

- Central **server**, many **clients**
- Communication between parts via network
- Communication **protocols**, for example REST, HTTP
- **Thin client** or **fat client**

- \+ Centralized management of resources on the server
- \+ Centralized security
- \- Single point of failure
- \- Relies on network communications

- Examples
  - Almost every website
  - The programming project



ARCHITECTURAL DESIGN PATTERNS

- Architectural patterns are often **combined**
- Layered + Client-Server + Event-Driven
  - Some layers are in the clients, some layers are in the server
  - Clients are often event-driven (events from the UI, events from the server connection)
  - Event-driven can be used inside layers and between layers (but must not break layer structure)

FINAL NOTES

- Common errors:
  - Not actually making an interface (design patterns almost always use interfaces)
  - Defining an interface, but not actually applying the pattern
  - Applying complex patterns when unnecessary
- In the Module 2 project:
  - Strategy
  - Listener
  - MVC
  - Client-Server architecture
  - Chain of Responsibility? State?
  - Layered? Event-driven?

SUMMARY

- Low-level design patterns
  - Creational: Constructors, Factory, Builder
  - Behavioral: Strategy, Listener, Chain of Responsibility, State
  - Structural: Facade, Adapter, Decorator

- High-level architectural patterns
  - Layered
  - Event-Driven
  - Client-Server

- See also: https://refactoring.guru/design-patterns

Any questions about **Design patterns**?