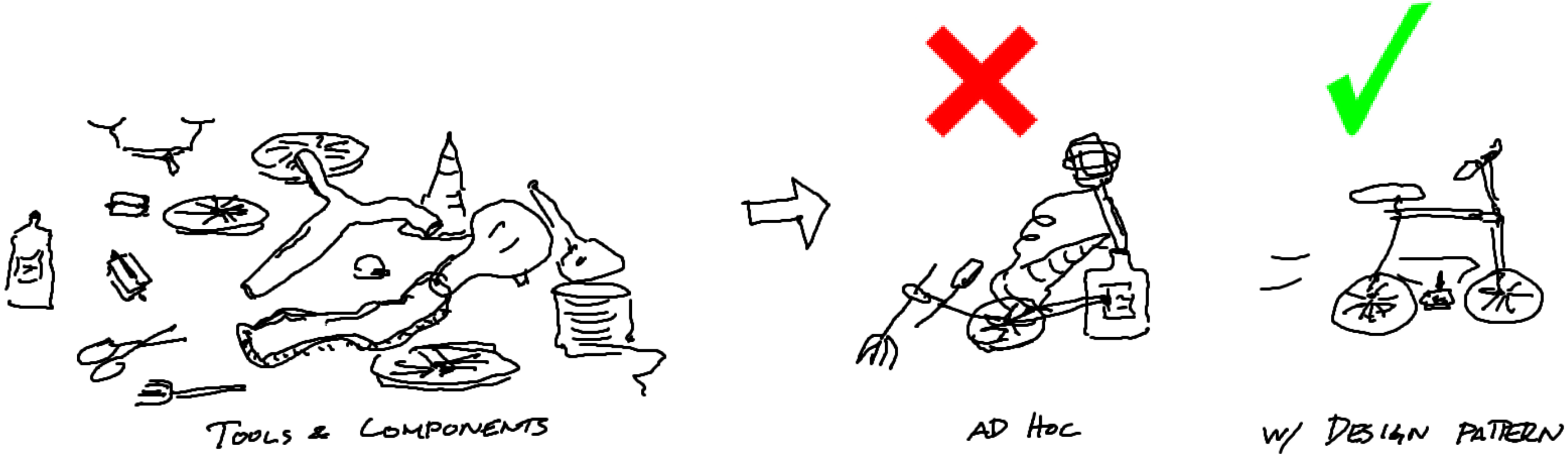
Design Patterns

Introduction

Learning goals

1. Understand the concept of design pattern
2. Understand the benefits of using and learning design patterns.
3. Get an overview of GoF design patterns.

Design patterns



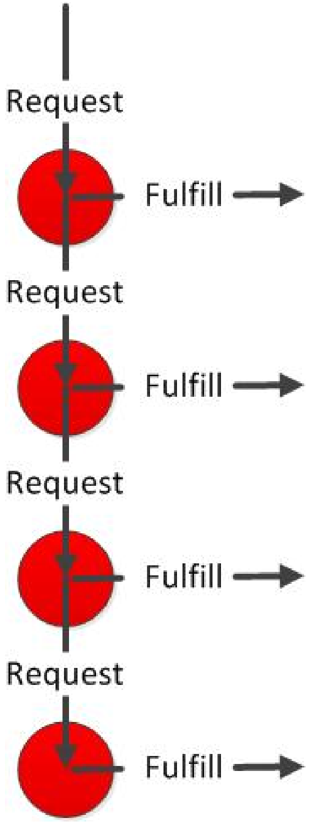
- Design patterns are **typical solutions to commonly occurring problems** in object-oriented programming.

Design patterns

- Design patterns are high-level principles and practices that are known to work well for certain types of problems.
- They are **shared knowledge and expertise** among software developers.
- Design patterns are **tools of thinking** for object-oriented programming. They are language-independent *per se*.
  - Naturally, each DP can be implemented in a programming language.
  - The implementations differ slightly due to language details.

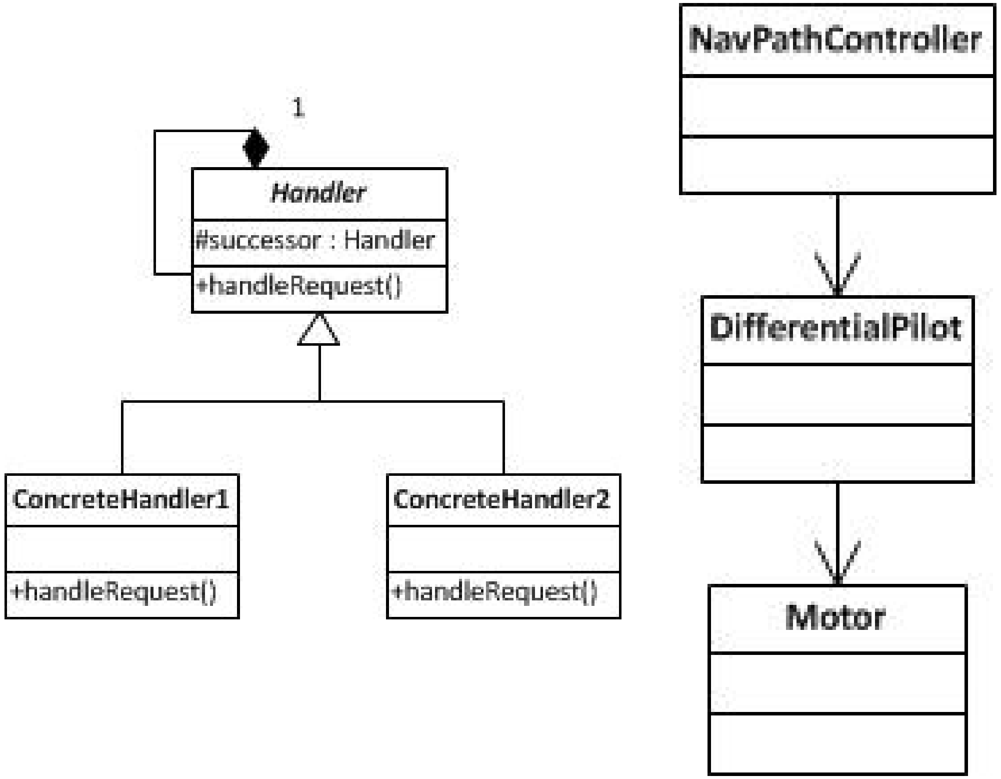
Example of a DP: Chain of Responsibility

- Problem: You are programming a robot.
- The robot needs to execute several commands such as:
  - Go to final location (23.1241, 69.1212)
    - requires route planning
  - Move directly to waypoint (22.5340, 68.2873)
  - Set speed to 12 kmph
  - Verify whether we have reached the destination
  - Get the current speed of the robot.
  - …
- The logic may become really complicated.
- Solution: apply the Chain of Responsibility design pattern to design the classes and the flow of service requests:
  - The **Navigator** object takes care of finding the route to the destination
  - The **Pilot** object is able to move directly to the next waypoint
  - The **Motor** object is able to turn the robot and set the speed of the robot
- Establish a well-defined chain of responsibility:
  - Every object has certain requests it can handle.
  - If the object is unable to handle the request, it passes the request to its successor.



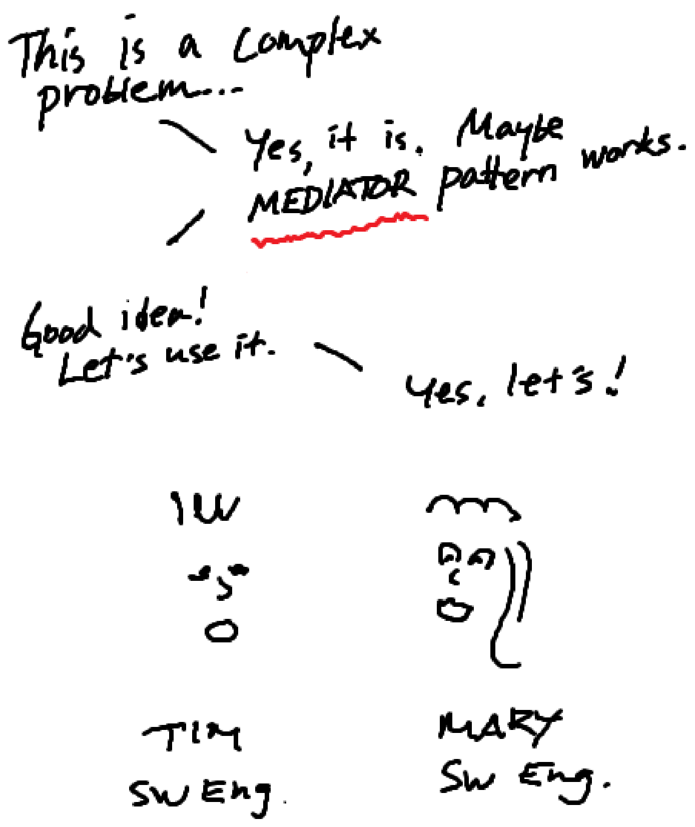
Example of a DP: Chain of Responsibility

- The movement is initiated by commanding the navigator.
- The navigator fulfills the request if it can.
- Navigator passes some commands to lower levels.
  - **`goto(x,y)`** is handled by the pilot.
  - **`isMoving()`** is eventually forwarded to the motors.



DP as means of communication

- Design patterns **provide a way to discuss** approaches and ideas among developers.
- This requires us to have a commons set of DPs that ’everybody’ in the field knows.
- Also, naming is important, as the DPs form a part of a shared vocabulary.
- DPs provide a way to think and communicate at the higher level, with bigger ’building blocks’.


This is a complex problem...
Yes, it is. Maybe MEDIATOR pattern works.
Good idea! Let's use it.
Yes, let's!
TIM
SW Eng.
MARY
SW Eng.

Core objectives

- **Loose Coupling**: Minimize dependencies between components.
  - “Each part of the code knows as few other parts as possible.”
- **High Cohesion**: Group similar parts together and separate different parts.
  - ”Each part of the code does just one thing – and other parts do other things”.

Tight and loose coupling

✗

```
// Class directly dependent on a specific notifier
implementation
public class UserManager {
   private EmailNotifier emailNotifier; // Specific
implementation usage

   public UserManager() {
      this.emailNotifier = new EmailNotifier(); // Direct creation
of a specific implementation
   }

   public void userRegistered(String username) {
      // When a user registers, send a notification
      emailNotifier.notifyUser("User " + username + "
registered successfully!");
   }
}
```

✓

```
// Interface defining the behavior of a notifier
public interface Notifier {
   void notifyUser(String message);
}

// Class implementing the Notifier interface
public class EmailNotifier implements Notifier {
   @Override
   public void notifyUser(String message) {
      // Implementation of sending email notification
      System.out.println("Sending email notification: " + message);
   }
}

// Class using the notifier loosely
public class UserManager {
   private Notifier notifier;

   public UserManager(Notifier notifier) {
      this.notifier = notifier;
   }

   public void userRegistered(String username) {
      // When a user registers, send a notification
      notifier.notifyUser("User " + username + " registered
successfully!");
   }
}
```

Low and high cohesion

✗

```
// Multi-purpose class with low cohesion
public class GeneralProcessor {
   private DatabaseConnector dbConnector;
   private FileParser fileParser;

   public GeneralProcessor() {
      this.dbConnector = new DatabaseConnector();
      this.fileParser = new FileParser();
   }

   public void processData(String data) {
      // Process data using both database connection and file parsing
      dbConnector.connect();
      // Additional database processing logic
      fileParser.parse(data);
      // Additional file processing logic
      dbConnector.disconnect();
   }
}
```

✓

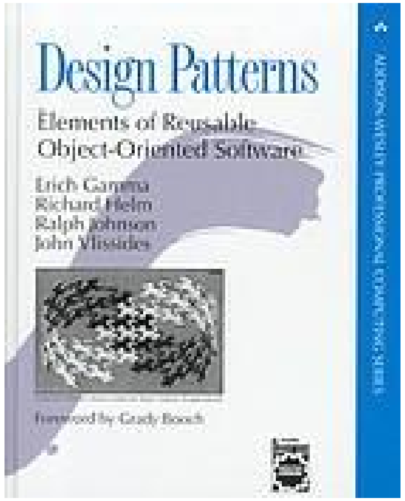
```
// Specified classes with clear tasks
public class DatabaseProcessor {
   private DatabaseConnector dbConnector;

   public DatabaseProcessor() {
      this.dbConnector = new DatabaseConnector();
   }

   public void processData(String data) {
      // Process data using only the database connection
      dbConnector.connect();
      // Additional database processing logic
      dbConnector.disconnect();
   }
}

public class FileProcessor {
   private FileParser fileParser;

   public FileProcessor() {
      this.fileParser = new FileParser();
   }

   public void processData(String data) {
      // Process data using only file parsing
      fileParser.parse(data);
      // Additional file processing logic
   }
}
```

SOLID design principles

- **Single Responsibility Principle (SRP):** A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types without altering the correctness of the program.
- **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules; both should depend on abstractions.

World of design patterns

- Technically there are many design patterns (as one is free to invent their own).
- On this course, we focus on the 23 well-known, established design patterns.
- These are called **GoF design patterns**.
  - GoF = Gang of Four = Gamma, Helm, Johnson, Vlissides
  - Sometimes they are referred to as Gamma patterns.
  - They first appeared in Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995)



The 23 GoF patterns

Creational

- Factory Method
- Abstract Factory
- Singleton
- Builder
- Prototype

Structural

- Composite
- Decorate
- Facade
- Flyweight
- Adapter
- Bridge
- Proxy

Behavioral

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Momento
- Observer
- State
- Strategy
- Template Method
- Visitor

Creational patterns

- Deal with **object creation and instance management.**
- They provide a flexible and maintainable way of creating objects.
- Creational patterns usually help with:
  1. **Object creation**: How does the program create and initialize objects in a way that is flexible and maintainable?
  2. **Object type selection**: How can the program dynamically or based on configuration choose the right type of object?
  3. **Instance management**: How can it ensure that there is only one instance of a certain class?
  4. **Handling object construction in complex systems**: How to deal with complex object creation processes in a way that keeps the code modular and maintainable?

Structural patterns

- Deal with **organizing and composing classes and objects** to form larger structures.
- They provide clear and efficient ways to compose classes and objects to create flexible and scalable software architectures.
- Structural patterns help with:
  1. **Class and object composition**: How can classes and objects be combined to create larger, more complex structures while maintaining flexibility?
  2. **Interface definition**: How can interfaces be defined to facilitate interaction between different parts of a system?
  3. **Code reusability**: How can patterns enable the reuse of existing code components in various contexts?
  4. **Encapsulation of implementation details**: How can the system's internal components be organized to hide implementation details and promote a clean separation of concerns?
  5. **Adaptation and compatibility**: How can structures be adapted to work together seamlessly, especially when dealing with incompatible interfaces?

Behavioral patterns

- Address the **interactions and responsibilities between objects** and define efficient ways for them to communicate and collaborate.
- Improve the flexibility and extensibility of a software system by providing solutions to common communication challenges between objects.
- Behavioral patterns may help with:
  1. **Object communication**: How can objects effectively communicate and collaborate to accomplish a specific task?
  2. **Responsibility assignment**: How can the distribution of responsibilities among objects be organized to ensure a clear and maintainable code structure?
  3. **Algorithms and behavior**: How can algorithms and behavior be encapsulated and exchanged, allowing for interchangeable components?
  4. **Encapsulation of behavior**: How can the behavior of an object be encapsulated and modified without altering its structure?
  5. **Event handling**: How can objects react to events and changes in a system in a flexible and scalable manner?

Design vs. architectural patterns

- At this point, you may be thinking about MVC.
  - Why is it not one of the GoF patterns?
- MVC (Model-View-Controller) is often considered an **architectural pattern** for organizing an application.
- Architectural patterns often describe the high-level structure (such as layers) of the software.
- Design patterns focus on how to solve various recurring problems and challenges.
  - Focus on technical design

Why do design patterns help?

- Design patterns promote:
  - reusability
  - scalability
  - extensibility
- Use of DPs increase the structural quality of a software product.
- Design patterns can help in various tasks:
  1. Determining what classes and objects there should be
  2. Designing classes of proper size and quantity
  3. Specifying the internal and external structure of classes (what properties and methods? which signatures?)
  4. Designing the communication between the objects
  5. Designing the class hierarchy and interfaces
  6. Writing classes with low coupling and high cohesion