Secure OOP with Java

Lecture Unit - 11

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Design Patterns

Someone has already solved your problems...

Definition

The design patterns [...] are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

— Design Patterns - Elements of Reusable Object-Oriented Software

What Design Patterns are

- Experience
- Tried and tested solutions
- Shared vocabulary

What Design Patterns are NOT

- Algorithms
- Complete solutions
- Code snippets of code templates
- Libraries or frameworks
- Best practices
- A solution for all problems
- APIs


THEY'RE MORE WHAT YOU'D CALL
GUIDELINES
THAN ACTUAL RULES
quickmeme.com

Object-oriented Design Principles

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

SOLID Principles

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

Interface-based Programming

Code against interfaces, not implementations.

- Decoupling
- Flexibility
- Testing
- Maintainability

Gang of Four

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Elements of Design Patterns

- Pattern name
- Problem description
- Solution
- Consequences

GoF Catalog

- Creational Design Patterns
- Structural Design Patterns
- Behavioral Design Patterns

Creational Design Patterns

Characteristics

- Abstract the instantiation process
- Make a system independent of how the objects are created, composed, and represented
- Object creators determine
  - the concrete data type of the object
  - how to create the object
  - who to involve in the object creation
  - and when to create the object

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

Singleton

Intention

- Ensure a class has only one instance at runtime
- Provide a global point of access to it

Consequences

- Controlled access
- Reduced name space
- Permits a variable number of instances
- More flexible than class operations

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    public static Singleton get() {
        return INSTANCE;
    }

    private Singleton() {
        // noop
    }
}
```

Builder

Intention

- Separate the construction of complex objects from its representation
- The same construction process can create different representations

Consequences

- Lets you vary the internal representation
- Isolates construction and representation
- Gives fine control over the construction process

```java
public class Person {
    private final String firstname;
    private final String lastname;

    public Person() {
        this(null, null);
    }

    public Person(final String lastname) {
        this(null, lastname);
    }

    public Person(final String firstname, final String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getFirstname() {
        return firstname;
    }

    public String getLastname() {
        return lastname;
    }
}
```

```
public class PersonBuilder {
    private String firstname;
    private String lastname;

    public PersonBuilder addFirstname(final String firstname) {
        this.firstname = firstname;
        return this;
    }

    public PersonBuilder addLastname(final String lastname) {
        this.lastname = lastname;
        return this;
    }

    public Person build() {
        return new Person(firstname, lastname);
    }
}
```

```
Person person1 = new PersonBuilder().addLastname("Mustermann").build();
Person person2 = new PersonBuilder().addFirstname("Martina")
    .addLastname("Musterfrau").build();
Person person3 = new PersonBuilder().build();
```

java.util.Calendar.Builder

```
Calendar calendar = new Calendar.Builder()
    .set(Calendar.DAY_OF_MONTH, 24)
    .set(Calendar.MONTH, Calendar.MAY)
    .set(Calendar.YEAR, 2022)
    .set(Calender.HOUR, 12)
    .set(Calendar.MINUTE, 05)
    .set(Calendar.SECOND, 30)
    .build();
```

Factory Method

Intention

- Define an interface for creating an object
- But let the subclasses decide which class to instantiate

Consequences

- Provides hooks for subclasses
- Connects parallel class hierarchies

java.text.NumberFormat

```
NumberFormat defaultFormatter = NumberFormat.getInstance();
NumberFormat localizedFormatter = NumberFormat.getInstance(Locale.GERMAN);
```

Abstract Factory

Intention

- Provide an interface for creating families of related or dependent objects without specifying their concrete class

Consequences

- Isolates concrete classes
- Facilitate the exchange of implementations

Prototype

Intention

- Specify the kinds of objects to create using prototypical instance
- Create new objects by copying this prototype

Consequences

- Adding and removing prototypes at runtime
- Specifying new objects by varying values
- Promotes consistency
- Supporting new data types is difficult

clone()

```
Calendar timestamp = Calendar.getInstance();

Calendar sametime = timestamp.clone();
```

Structural Design Patterns

Characteristics

- Objects may combine other objects in many ways (→ composition)
- Creates higher-level abstractions (beyond inheritance and interface implementations)

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter

aka Wrapper

Intention

- Convert the interface of a class into another interface a client expects
- "Glue code" for incompatible interfaces

Consequences

- Determine the scope of adaption
- Pluggable adapters
- Two-way adapters


TRAVEL ADAPTER PLUG
WITH SAFETY SHUTTER

```
FileInputStream inputStream = new FileInputStream(String path);
InputStreamReader reader = new InputStreamReader(file, Charset.forName("UTF8"));
```

```
FileOutputStream outputStream = new FileOutputStream(String path);
OutputStreamWriter writer = new OutputStreamWriter(file, Charset.forName("UTF8"));
```

Bridge

Intention

- Decouple an abstraction from its implementation so that the two can vary independently

Consequences

- Decoupling of interface and implementation
- Improved extensibility
- Hiding implementation details

Bridge Examples

- `java.sql.Driver` and `java.sql.DriverManager`
- Generic GUI definition and platform-specific implementations

Composite

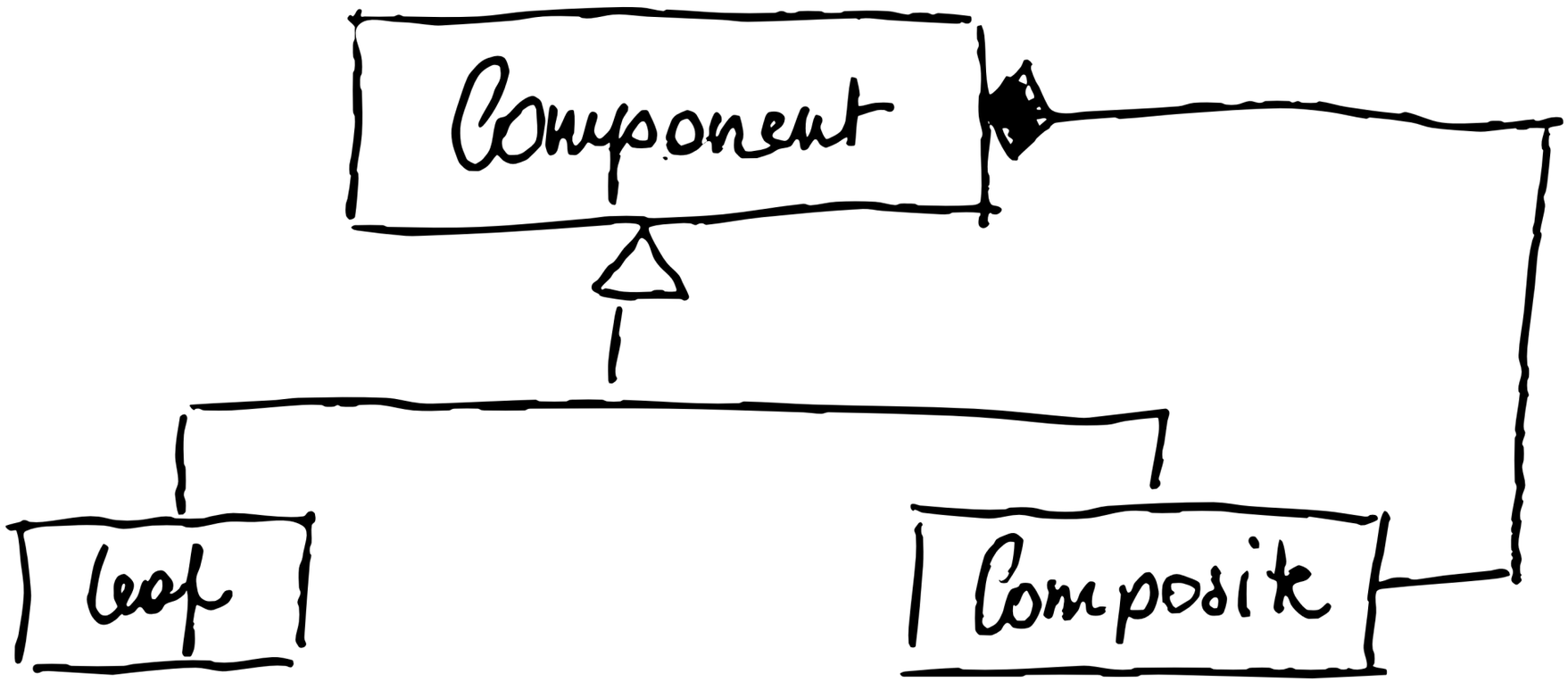
Intention

- Build objects in tree-like structures to represent part-whole hierarchies
- Lets clients treat individual objects and compositions uniformly

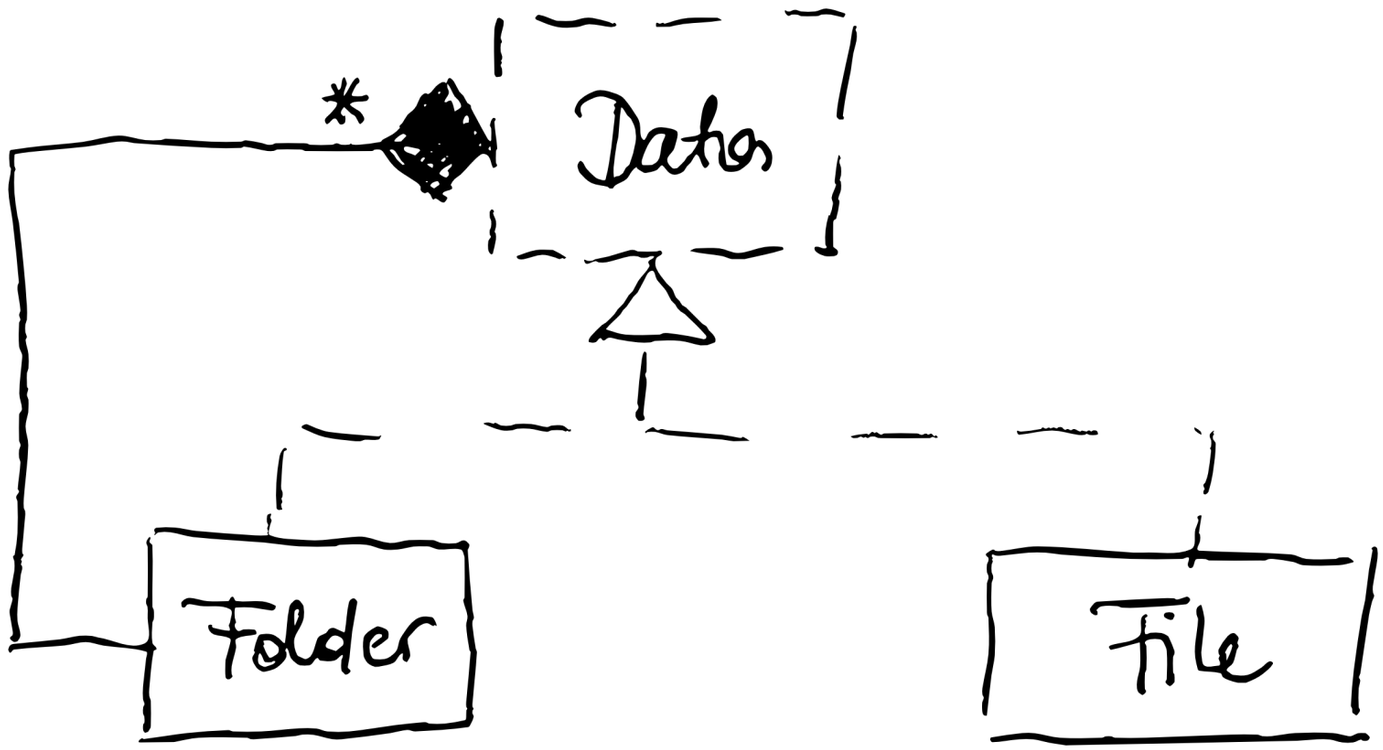
Consequences

- Defines class hierarchies of primitive and composite objects
- Makes clients simple
- Facilitates the addition of new components
- Makes design overly general

Composite Structure



Composite Example



Decorator

Intention

- Attach additional responsibilities to an object dynamically
- Decorators provide a flexible alternative to subclassing for extending functionality

Consequences

- More flexibility than static inheritance
- Avoid feature-laden classes high up in the hierarchy
- A decorator and its component are not identical
- May produce lots of little objects

```
Reader input = new LineNumberReader(new FileReader(file));
```

Facade

Intention

- Provide a unified interface to a set of interfaces in a subsystem
- Facades define a higher-level interface for easier use

Consequences

- Shields clients from subsystem components
- Promotes weak coupling

```
public final InputStream openStream() throws java.io.IOException {
    return openConnection().getInputStream();
}

public final Object getContent() throws java.io.IOException {
    return openConnection().getContent();
}
```

Flyweight

aka Cache

Intention

- A flyweight object is a shared object that can be used in multiple contexts simultaneously

Consequences

- May introduce run-time costs associated with transferring, finding and/or computing extrinsic state
- Costs are offset by space savings

Flyweight Examples

- Java API wrapper classes for primitive data types
- java.lang.String

Proxy

Intention

- Provide a surrogate or placeholder for another object to control access to it

Consequences

- Introduces a level of indirection when accessing an object

Behavioral Design Patterns

Characteristics

- Concerned with algorithms and the assignment of responsibilities between objects
- Patterns of communication
- Complex control flow at run-time

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

Template Method

Intention

- Define a skeleton of an algorithm in an operation, deferring some steps to subclasses
- Subclasses may redefine certain steps of an algorithm without changing the whole algorithm's structure

Consequences

- Fundamental method for code reuse
- "Don't call us, we'll call you"

Strategy

aka Policy

Intention

- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Lets the algorithm vary independently of the clients that use it

Consequences

- Families of related algorithms
- An alternative to subclassing
- Strategies eliminate conditional statements
- Clients must be aware of different strategies
- Increased number of objects

State

Intention

- Allow an object to alter its behavior when its internal state changes
- The object will appear to change its class

Consequences

- Localizes state-specific behavior
- Partitions behavior for different states
- Makes state transitions explicit
- State objects can be shared

Observer

aka Publish-Subscribe

Intention

- Define a one-to-many dependency between objects so that when one object changes state, all dependents are notified and updated automatically

Consequences

- Abstract coupling between subject and observer
- Support for broadcast communication
- Unexpected updates

Chain of Responsibility

Intention

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Chain the receiving objects and pass the request along the chain until an object handles it.

Consequences

- Reduced coupling
- Added flexibility in assigning responsibilities to objects
- Receipt is not guaranteed

Command

aka Action

Intention

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues, or log requests
- Supports undoable operations

Consequences

- Decouples object that invokes an operation from the one that knows how to perform it
- Commands are first-class objects
- Multiple command can be assembled to a composite command
- Easy to add new commands

Interpreter

Intention

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Consequences

- It is easy to change and extend the grammar
- Implementing the grammar is easy, too
- Complex grammars are hard to maintain
- Adding new ways to interpret expressions

Iterator

aka Cursor

Intention

- Provide a way to access the element of an aggregate object sequentially without exposing its underlying representation

Consequences

- Supports variations in the traversal of an aggregate
- Simplify the aggregate interface
- More than one traversal can be pending on an aggregate

Mediator

Intention

- Define an object that encapsulates how a set of objects interact
- Promotes loose coupling by keeping objects from referring each other explicitely
- Lets you vary their interaction independently

Consequences

- Limits subclassing
- Decouples colleagues
- Simplifies object protocols
- Abstracts object cooperation
- Centralizes control

Memento

aka Snapshot

Intention

- Capture and externalize an object's internal state without violation encapsulation
- Enable the restoration of an object's state at a later time

Consequences

- Preserves encapsulation boundaries
- Using mementos might be expensive
- Hidden costs in caring for mementos

Visitor

Intention

- Represent an operation to be performed on the elements of an object structure
- Lets you define a new operation without changing the classes

Consequences

- Visitor makes adding new operations easy
- Gathers related operations and separates unrelated ones
- Makes it hard t add new classes
- Visiting across class hierarchies
- Accumulate state

How to Use Design Patterns

How Design Patterns Solve Design Problems

- Finding appropriate objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementations
- Putting reuse mechanisms to work
- Relating run-time and compile-time structures
- Designing for change

How to Select a Pattern

- Consider how design patterns solve design problems
- Scan intent sections
- Study how patterns interrelate
- Study patterns of like purpose
- Examine a cause of redesign
- Consider what should be variable in the design

How to Use a Design Pattern

- Read the pattern once trough for an overview
- Go back and study the structure, participants and collaborations
- Look at the sample code
- Choose names for pattern participants that are meaningful in the application context
- Define classes
- Define application-specific names for operations in the pattern
- Implement the operations to carry out the responsibilities and collaborations in the pattern

The overuse of design patterns can lead to code that is downright over-engineered.

- Go with the simplest solution that fulfills the requirement
- Implement patterns where and when the need emerges

Anti-Patterns

Examples for Anti-Patterns

- Spaghetti Code
- God Class/God Object
- Fear of Adding Classes
- Magic Numbers and Strings
- Big Ball of Mud
- and many more...

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at