

## **Summary**

- OO Principles
- OO Design Principles
- OO Design Patterns: Past, Present and Future



## **OOP Principles**

### Encapsulation

- Methods and data are combined in classes
- Not unique to OOP

### Information Hiding

- Implementation of a class can be changed without affecting clients
- Not unique to OOP (Ada / Modula provided information hiding)

### Polymorphism

- Invocation based on the dynamic type of a reference
- Substitution rule: Subtypes must fulfill the contract of its base type
  Liskov Substitution Principle

#### Inheritance

Classes can be adapted through inheritance (=> fragile)



- Open-Closed Principle: Classes should be open for extension but closed for modification
  - Classes should be designed so that they never need to be changed
  - To extend the behavior of the system, new code is added. Old code should not be modified
  - Provide extension points through which classes can be extended
  - Pattern Applications

<ul><li>Strate</li></ul>	egy no	switch	statement	in subject,	would d	change upo	n
--------------------------	--------	--------	-----------	-------------	---------	------------	---

extension with new strategies

Decorator provide new behavior without changing an existing class

Template final template method is closed but open for extension

through hook methods

Dependency Injection configuration of the system without code change



### Encapsulate what varies

- Minimize the impact what varies / what changes (i.e. the implementation)
- By identifying what varies and hiding it (its implementation) behind an interface, then you can change its implementation without violating its contract

#### Pattern Applications

Strategy varying strategies

State varying behavior depending on a state

Observer different behavior in separate classes upon a state change

Factory the creation of instances is encapsulated

Dependency Injection the configuration of the system is encapsulated



### Program to an interface, not to an implementation

- Avoid referencing concrete classes, declare interfaces only
- Allows to exchange the implementation class, e.g. by a mock class (for testing purposes) or by a proxy (remoting).

#### Pattern Applications

	Strategy	Strategy interface implemented by concrete Strategies				
•		ect keeps track of objects implementing the Observer face (not concrete observer implementations)				
	Factory	ory use factory interface instead of referencing concrete classe				
•	Decorator	use of interfaces allows to replace concrete class by a decorated one				
•	Proxy	may be used instead of original object				
	Dependency Inje	ction allows to inject different implementations				



### Favor Composition over Inheritance

- Method of reuse by composing new functionality
- Black-box reuse, i.e. no dependency on implementation details
- More flexibility through composition, i.e. configuration at runtime
- Potentially more complex class diagrams

#### Pattern Applications

Strategy
 Strategies are connected through composition (in contrast

to specializing the context)

Decorator Avoids combinatorial explosion of classes through

composition



- Don't call us, we call you
  - Base classes "run the show" and call subclasses when needed
  - Supports planned reuse / planned extension
  - Pattern Applications

Template Template Method is called when needed

Callback Separation of layers, no cyclic dependencies

Strategy
 Same principle, but extension is bound by composition

Factory Same as strategy

# **OO Design Principles: SOLID**

### Single Responsibility Principle

A class should have a single purpose and only one reason to change

#### Open/Closed Principle

 Software entities should be open for extension, but closed for modification

### Liskov substitution principle

Subtypes must be substitutable for their base types

### Interface segregation principle

 Make fine grained interfaces that are client specific instead of general-purpose interfaces

### Dependency inversion principle

- Depend on abstractions, not on concretions => dependency injection
- Prerequisite for modular / component-oriented programming and framework design => software architecture [swa] module

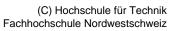














## **Design Patterns – Past, Present and Future**

- Dangerous Patterns [http://fose.ethz.ch/slides/gamma.pdf]
  - Mediator
  - Singleton
- Pattern voting [OOPSLA workshop 2004]
  - Voted off patterns (people felt they were sufficiently uncommon)
    - Factory Method confusion with Factory
    - Bridge
    - Flyweight
    - Interpreter
  - Split decisions
    - Singleton
    - Chain of Responsibility

## **Gamma: A new categorization**

#### Core

- Composite
- Strategy
- State
- Command
- Iterator
- Proxy
- Template Method
- Facade
- Null Object

the patterns the students should learn

#### **Peripheral**

- Abstract Factory
- Memento
- Chain of Responsibility
- Bridge
- Visitor
- Type Object
- Decorator
- Mediator
- Singleton
- Extension Objects

#### Creational

- Factory Method
- Prototype
- Builder
- Dependency Injection

#### Other (Compound)

- Interpreter
- Flyweight

learn on demand



### Gamma: What hasn't changed

- Object-oriented design principles
- Most of the patterns
  - With a focus on the core patterns
- The importance of decoupling and cohesion
  - Supported by the module system of Java 9