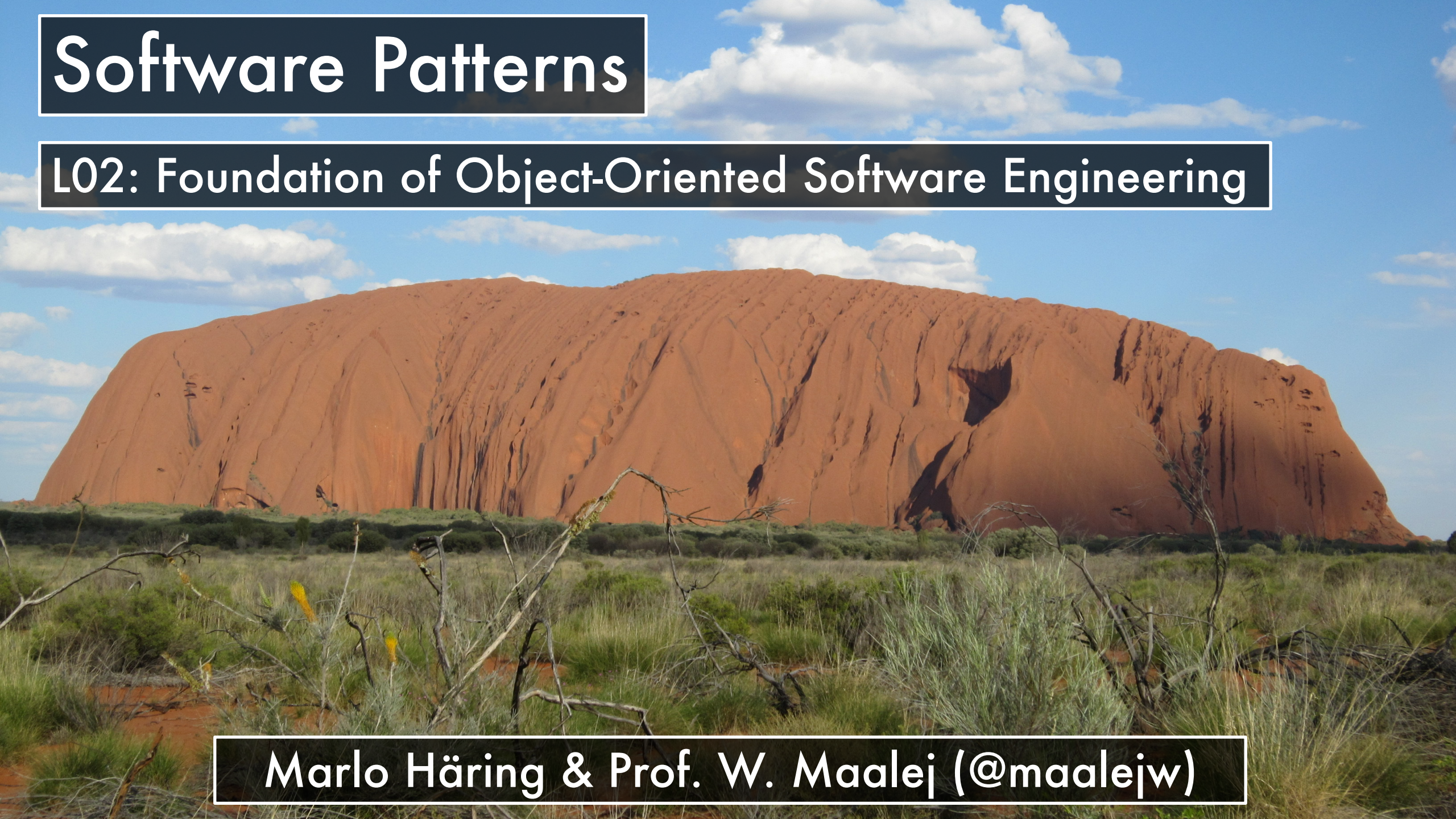
Software Patterns

L02: Foundation of Object-Oriented Software Engineering



Marlo Häring & Prof. W. Maalej (@maalejw)

Many design patterns use a combination of polymorphism and delegation

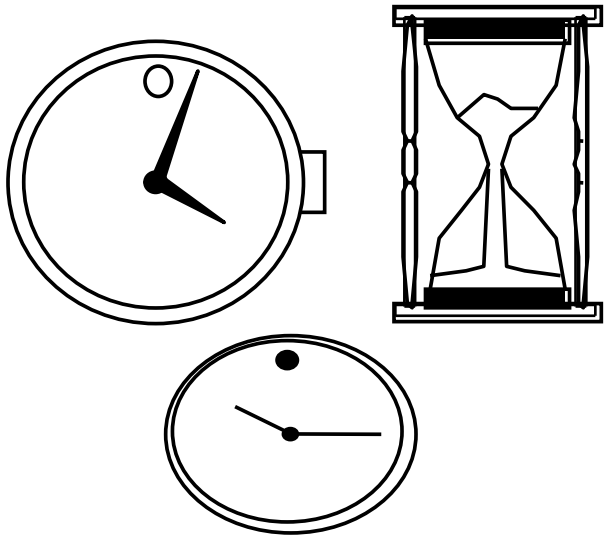
Outline of the talk

1. **Typing**
2. Information Hiding
3. Coupling and Cohesion
4. Polymorphism
5. Binding
6. Delegation

Concepts and phenomena

- **Phenomenon**
  - An object in the world of a domain as you perceive it
    - Examples: This lecture at 10:15 , my golden watch
- **Concept**
  - Describes the common properties of phenomena
    - Example: All lectures on software engineering
    - Example: All golden watches
- **A Concept is a 3-tuple**
  - Name: The name distinguishes the concept from other concepts
  - Purpose: Properties that determine if a phenomenon is a member of a concept
  - Members: The set of phenomena which are part of the concept

Concepts, phenomena, abstraction and modeling

Name	Purpose	Members
Watch	A device that measures time.	

- Definition Abstraction
  - Classification of phenomena into concepts
- Definition Modeling
  - Development of abstractions to answer **specific** questions about a set of phenomena while ignoring **irrelevant** details

Type and instance

- Type:
  - A concept in the context of programming languages
    - Name: boolean
    - Purpose: logical
    - Members: {true, false}
    - Name: int
    - Purpose: integral number
    - Members: {0, -1, 1, -2, 2, ...}
- Instance is a member of a specific type
- The type of a variable represents all possible instances of the variable
- The following relationships are similar:
  Concept <-> Phenomenon
  Type <-> Variable
  Class <-> Object

Type system

- Type system
  - Specifies rules how variables may behave in a program
  - Does not allow behavior outside these rules
  - Ensures that operations expecting a certain kind of value are not used with values for which that operation makes no sense
    - E.g.: 3/"Hello"
- Type safety
  - The ability to detect typing errors at compile time

- Usually two kinds of distinctions regarding typing:
  - Static vs. Dynamic typing
  - Strong vs. Weak typing

Static vs. Dynamic typing

- Static typing: Type checking at compile time
  - The ability to check, based on the source code alone, that no execution will try to apply an operation to an object that is not applicable to that object
  - Examples: C, C++, C#, Java, Haskell
  - Advantages: no runtime type checking, more efficient
  - Disadvantages: conservative, can reject working programs
- Dynamic typing: Type checking at runtime
  - The applicability of operations to their target objects is only checked at run time, prior to executing each operation
  - Examples: JavaScript, Objective-C, PHP, Python, Ruby
  - Advantages: Flexibility
  - Disadvantages: Possibility of runtime type errors

Strong vs. Weak typing

- Languages with a **strong type system**
  - No implicit type conversion ("recasting")
  - Examples: Haskell, Ruby, Java (for classes)
- Languages with a weak type system
  - Allow implicit type conversion
  - Allow methods with different types of arguments (overloading)
  - Examples: C, C++, Java (some primitive types: char→int, int→float)

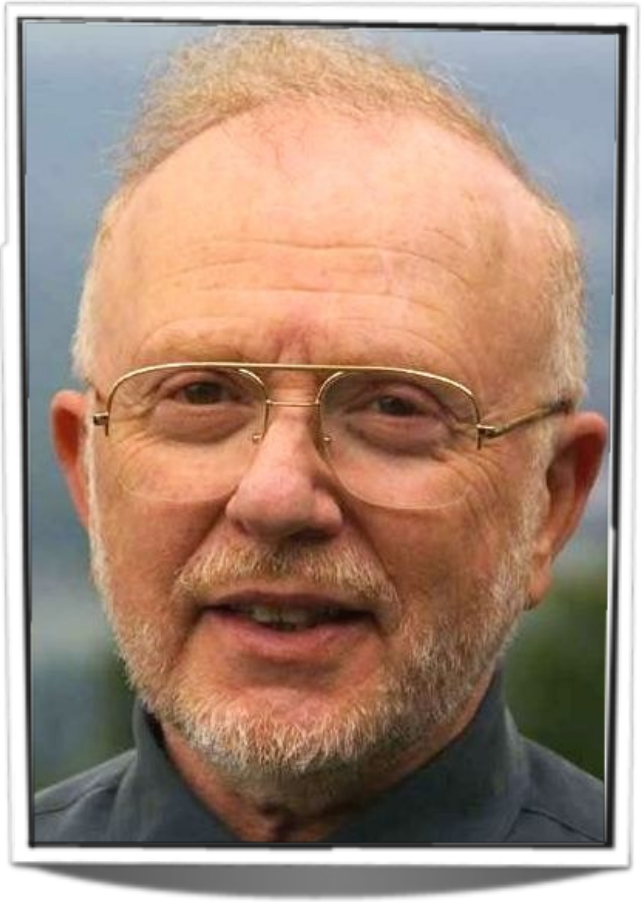
Outline of the talk

1. Typing
2. **Information Hiding**
3. Coupling and Cohesion
4. Polymormism
5. Binding
6. Delegation

Information hiding

- The goal of good system design:
  - Reduce system **complexity** while allowing **change**
- Principle of information hiding (Parnas):
  - A calling module (class, subsystem) does not need to know anything about the internals of the called module
- Good object design heuristic:
  - Use visibility modifiers to minimize the visibility of attributes and operations
  - Make all attributes of a class private
  - Make all the operations of a class private, unless the operations are needed by a class user

Information hiding - history



- David Parnas (*1941)
  - Ph.D. from Carnegie-Mellon University (in electrical engineering)
  - One of the first researchers to apply traditional engineering practices to software engineering

- The Principle of Information Hiding was published in the paper On the criteria to be used in decomposing systems into modules, Communications of the ACM archive, Volume 15 , Issue 12 (December 1972), Pages 1053 - 1058

Information hiding in design patterns (examples)

- Adapter Pattern:
  - Hides legacy code and thus reduces complexity
- Bridge Pattern, Proxy Pattern:
  - Hide the details of the underlying implementations

Outline of the talk

1. Typing
2. Information Hiding
3. **Coupling and Cohesion**
4. Polymorphism
5. Binding
6. Delegation

Coupling and cohesion

- Coupling: Measures the dependencies between subsystems
- Cohesion: Measures the dependencies among classes within a subsystem
- What makes a good design?
  - Low coupling:
    - The subsystems should be as independent of each other as possible
    - A change in one subsystem does not affect any other subsystem
  - High cohesion:
    - Functionalities in a subsystem, accessed through its interface, have much in common

Law of demeter

- To achieve low coupling and high cohesion, “Principle of Least Knowledge”:
- A method M of an object O may only invoke the methods of the following kinds of objects:
    - O itself
    - M's parameters
    - Any objects created within M
    - O's direct component objects
- For Java, this can be formulated as:
    - “Use only one dot”
        - Use only 1 level of method calls: o1.getResult();
    - Never use two or more levels
        - Not good: o1.getResult().doSomething();

Good design

- Good design reduces complexity
- Good design is prepared for change
- Good design provides low coupling
- Good design provides high cohesion

Outline of the talk

1. Typing
2. Information Hiding
3. Coupling and Cohesion
4. **Polymorphism**
5. Binding
6. Delegation

Polymorphism

Polymorphism (General definition)

- The ability of an object to assume different forms or shapes
  - Poly = many, Morph = form (ancient Greek)

Polymorphism (Computer Science):

- The ability of an abstraction to be realized in multiple ways
- The ability of an interface to be realized in multiple ways
- The dynamic treatment of objects based on their type

Polymorphism types

- Parametric Polymorphism
  - Generics
- Inclusion Polymorphism
  - Subtyping
  - Subclassing and Overriding
- Ad-Hoc Polymorphism
  - Overloading
  - Coercion

Parametric polymorphism - example

- Consider the following List implementations:

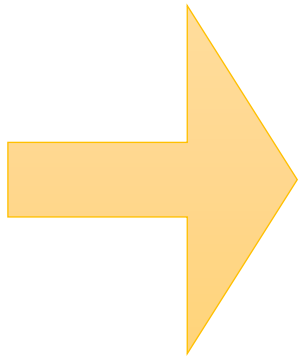
```
class StringList {
  boolean add(String x){};
  String get(int index){};
}
class DateList {
  boolean add(Date x){};
  Date get(int index){};
}
```

- Can we exploit the similarities of these classes?
- Yes, by using parametric polymorphism (i.e. a generic type)

Parametric polymorphism - example

- We define a generic type List!

```
class StringList {
  boolean add(String x){};
  String get(int index){};
}
class DateList {
  boolean add(Date x){};
  Date get(int index){};
}
```



```
Class List<E> {
  boolean add(E x){};
  E get(int index){};
}
```

<E> is a Type Parameter

Generic types and operations

- A type is called a generic type if it has a type parameter
  - For example ArrayList<E> is a generic type
- A type parameter is a placeholder for a specific type
  - The type parameter must be specified at the instantiation of a variable of the generic type
  - <E> is an example of a type parameter
- Operations on generic types are called generic operations
  - add() and get() are generic operations

Instantiating a generic type

- The generic type List<E> allows us to use lists with objects of arbitrary type
- Example: Instantiating a List of Strings:

```
String s = "test";
List<String> a = new ArrayList<String>();
a.add(s);
String t = a.get(0);
```

- But it doesn't allow us to construct lists with objects of mixed types!

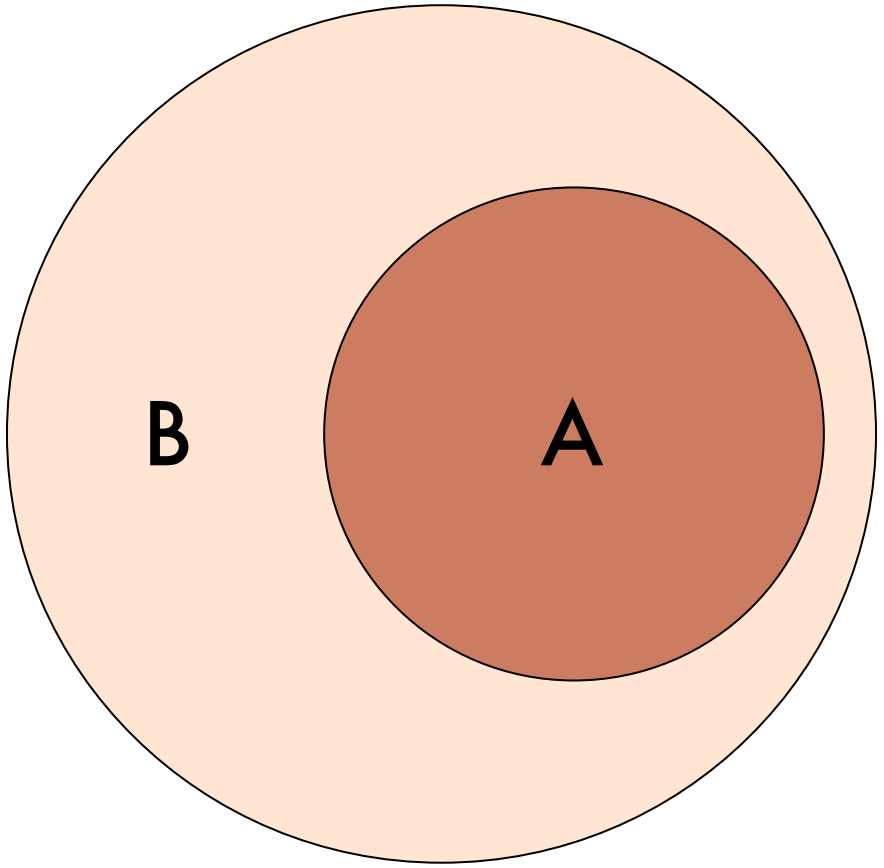
```
a.add(new Date()); // ERROR
```

Polymorphism types

- ✓Parametric Polymorphism
  - ✓Generics
- Inclusion Polymorphism
  - Subtyping
  - Subclassing and Overriding
- Ad-Hoc Polymorphism
  - Overloading
  - Coercion

Subtyping

Type A is a subtype of another type B,
when all members of A are also members of B

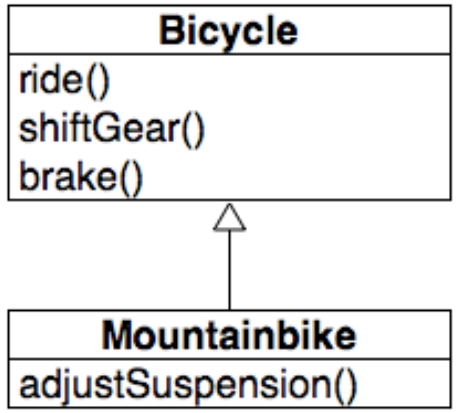


Example:
Week = {Mon,Tue,Wed,Thu,Fri,Sat,Sun}
WeekEnd = {Sat,Sun}

WeekEnd is subset of Week:
WeekEnd ⊆ Week.

Example: mountainbike and bicycle

- Is every Mountainbike a Bicycle?
  - Yes: a Mountainbike is a Bicycle with added suspension fork



Mountainbike ⊆ Bicycle

- Whenever a Bicycle is used in a program, a Mountainbike can be substituted without problems

This is called the Liskov's substitution principle

Liskov's substitution principle

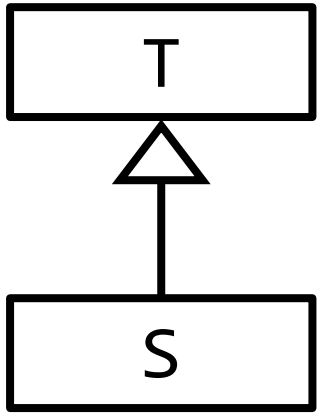
S is a subtype of T, if an object of type S can substitute an object of type T without changing the behavior of a program.

Barbara Liskov, MIT

Turing Award 2008

Example:

- We have old code, that uses only T
- Then, we extend the class by S
- S has a few additional methods
- We can replace T by S in the old code, if we don't use the new methods!



Subtyping

- A subtype must satisfy the Liskov's substitution principle
- The following rules apply to subtypes:
  - Operations:
    - All supertype operations must have corresponding subtype operations
    - Each subtype operation has both:
      - A weaker precondition: must require less (or the same) than the superclass operation
      - A stronger postcondition: guarantees more (or the same)
  - Invariants (Called properties by Liskov):
    - Any invariant of the supertype (e.g. value constraints) must be guaranteed by the subtype as well

Subtyping in Java

- Java does not provide invariants as a language construct
- Therefore it is easy to construct subtypes in Java that do not follow the Liskov substitution principle
  - Example: The supertype invariant requires elements to be sorted in ascending order. The subtype sorts the elements in descending order.
- The responsibility for maintaining the Liskov substitution principle lies on the shoulders of the programmer!

Ideal situation for subtyping

- In an ideal language, the compiler should tell us if the substitution principle is fulfilled

```
S oS;
T oT;
oT = oS;
```

S is a subtype of T ⇒
Compiler shows no Error

S is not a subtype of T ⇒
Compiler shows an Error

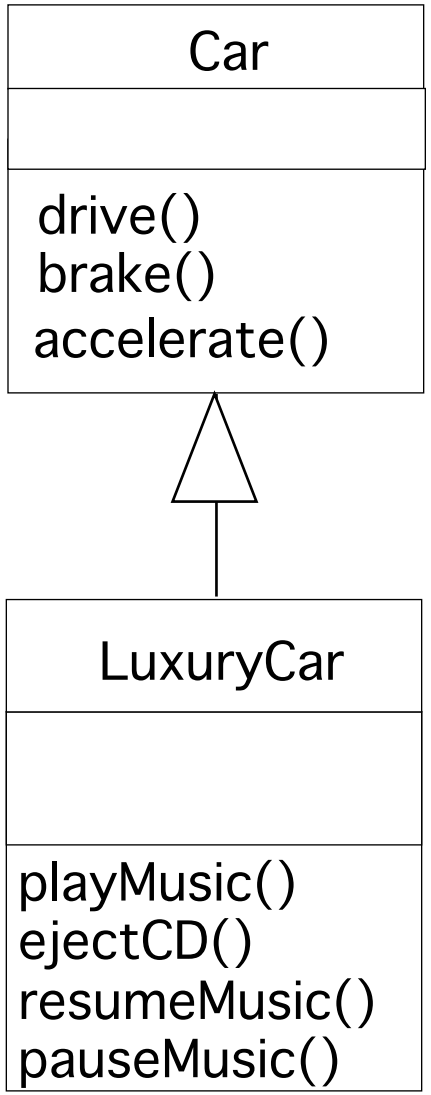
Polymorphism types

- ✓Parametric Polymorphism
  - ✓Generics
- Inclusion Polymorphism
  - ✓Subtyping
  - Subclassing and Overriding
- Ad-Hoc Polymorphism
  - Overloading
  - Coercion

Subclassing

- Subclassing simply denotes the usage of an inheritance association
- Subclassing is mainly used…
  - In Analysis:
    - To formulate taxonomies (class hierarchies)
  - In Object Design:
    - To reuse implementations

Example for subclassing in a taxonomy



Superclass:

```
public class Car {
        public void drive() {…}
        public void brake() {…}
        public void accelerate() {…}
    }
```

Subclass:

```
public class LuxuryCar extends Car {
        public void playMusic() {…}
        public void ejectCD() {…}
        public void resumeMusic() {…}
        public void pauseMusic() {…}
}
```

Overriding

- Overriding is a special case of subclassing where a method implemented in the superclass is reimplemented in the subclass
  - This concept is at the heart of OO programming
  - It means, there are several methods, all with the same signature, but with different implementations
  - Which of these methods should be invoked?
    - The selection depends on the type of the object at runtime

    - The selection of which method to execute at runtime allows to ***change the behavior*** of an object without extensive case distinctions.
    - Example for bad code:

```
if (ItIsNight) P = IncredibleHulk else if
    (ItIsDay) P = BruceBanner)
```

Modeling bicycles by overriding

- Bicycle:
  - Several gears, hand brake
- OldBicycle:
  - One gear, backpedaling brake
- Is an OldBicycle a subtype of Bicycle?
  - No
- Is an OldBicycle a subclass of Bicycle?
  - Yes
- Does it follow the Liskov Substitution Principle?
  - No, because any client who expects to accelerate by upshift() or decelerate by downshift() will experience unexpected behavior, namely nothing

A really bad use of overriding

- One can override the operations of a superclass with completely new meanings

- Example:

```
public class SuperClass {
        public int add (int a, int b) { return a+b; }
        public int subtract (int a, int b) { return a-b; }
}
public class SubClass extends SuperClass {
        public int add (int a, int b) { return a-b; }
        public int subtract (int a, int b) { return a+b; }
}
```

- We have redefined addition as subtraction and subtraction as addition!!

Polymorphism types

- ✓Parametric Polymorphism
  - ✓Generics
- ✓Inclusion Polymorphism
  - ✓Subtyping
  - ✓Subclassing
- Ad-Hoc Polymorphism
  - Overloading
  - Coercion

Overloading – example

- Assume we want to determine the maximum of two values?
- What happens if you execute these statements in Java?
  - `Math.max(1, 2);`
  - `Math.max(1L, 2L);`
  - `Math.max(1.0f, 2.0f);`
  - `Math.max(1.0d, 2.0d);`
- How is `max()` implemented?

Overloading – example

- There are four signatures for max() in the java.lang.Math class:

```
public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)
```

- As a developer, you can just invoke max(), e.g.:
  - Math.max(1, 2);
- The compiler then selects the appropriate overloaded method

Definition of overloading

- Overloading: The same name is used to denote different operations
- The signature is used to decide which of the possible operations is meant by a particular instance of that name
  - Signature: Parameter types and return result type of a method
- The compiler makes the decision

Overloading puzzle

- What does the following program print?

```
public class Foo {
    private Foo (Object o) {
        System.out.println("Object");
    }

    private Foo (double[] dArray) {
        System.out.println("double array");
    }

    public static void main(String[] args) {
        new Foo (null);
    }
}
```

Possible Answers:

a) "Object"

b) "double array"

c) Error: NullPointerException

Adapted from: Java Puzzlers, #46

The case of the confusing constructor

Overloading - extreme

- Can I invoke `max(1.0f, 2)`?
- This leads us to the next topic: Coercion

Polymorphism types

- ✓Parametric Polymorphism
  - ✓Generics
- ✓Inclusion Polymorphism
  - ✓Subtyping
  - ✓Subclassing
- Ad-Hoc Polymorphism
  - ✓Overloading
  - Coercion

Coercion

- Coercion: Converts an argument to the expected type in a situation that would otherwise result in a type error
- Also called type promotion or type conversion
- Coercion is a semantic operation

Coercion example

- Consider the sqrt method in java.lang.Math:

```
public static double sqrt(double a)
```

- This call will work fine: Math.sqrt(16.0d);
- How about this call: Math.sqrt(16); ?
- Works fine too, because the integer 16 is coerced to the double 16.0d by the Java Virtual Machine (JVM).

Coercion puzzle

- What does the following program print?

```
public class LastLaugh {
    public static void main(String args[]) {
        System.out.print("H" + "a");
        System.out.print('H' + 'a');
    }
}
```

Possible Answers:

a) “HaHa”

b) “Ha169”

c) “169Ha”

d) “169169”

Java Puzzlers, #11

The last laugh

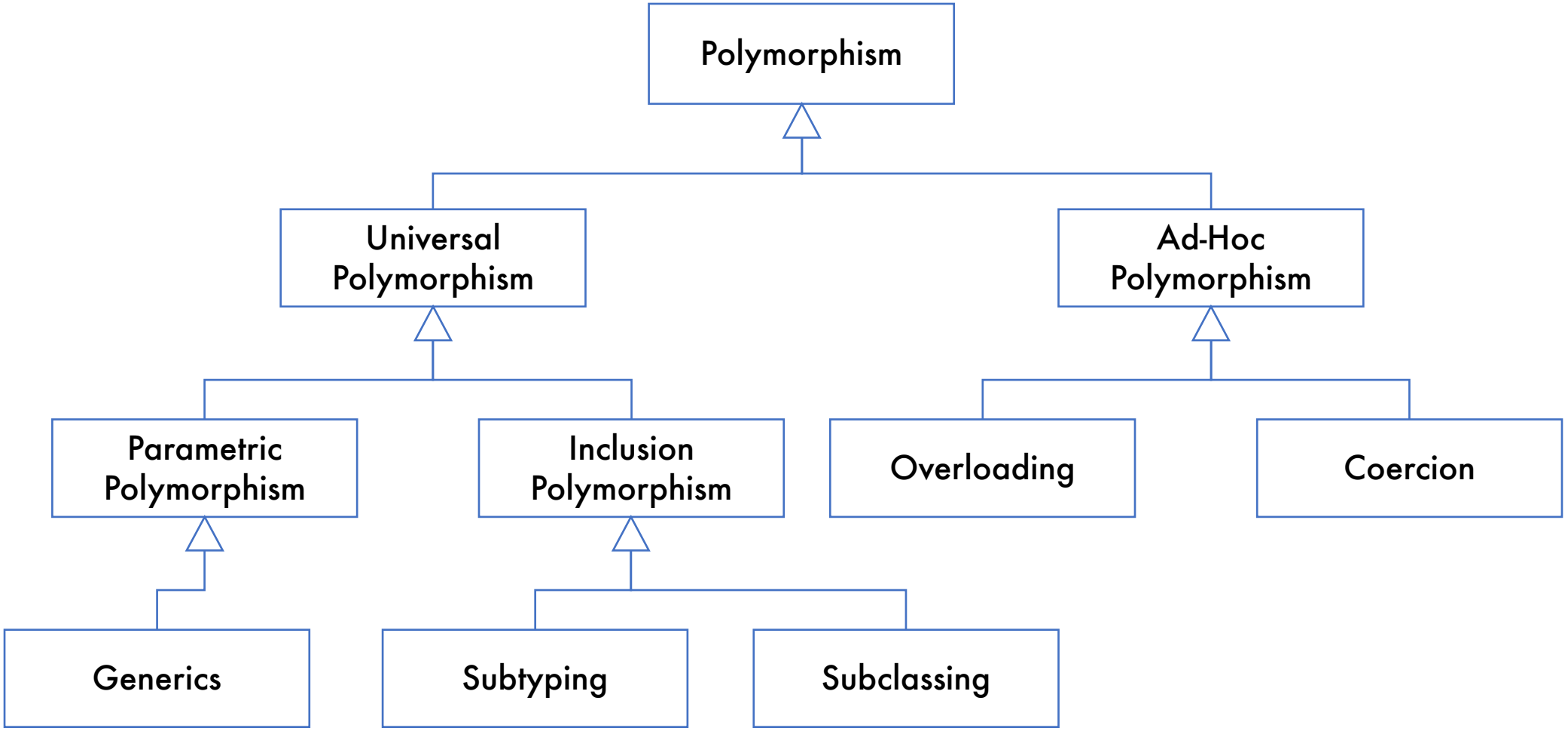
Solution

- It prints „Ha169“
- Why does it do that?
  - The first call uses the “+” operator to concatenate the two strings and outputs “Ha” as expected
  - In the second call, the arguments to “+” are not strings, but two char literals
    - Since “+” is not defined for chars, the compiler first uses coercion (promotion) to convert the chars into int (‘H’ → 72, ‘a’ → 97)
    - Then it uses the overloaded version of “+” to add two integers, and calling the overloaded version of System.out.print(int i) for integer values, resulting in the output of “169”

Polymorphism types

- ✓Parametric Polymorphism
  - ✓Generics
- ✓Inclusion Polymorphism
  - ✓Subtyping
  - ✓Subclassing
- ✓Ad-Hoc Polymorphism
  - ✓Overloading
  - ✓Coercion

Polymorphism - A taxonomy



Adapted from: Cardelli and Wegner, On Understanding Types, Data Abstraction, and Polymorphism. 1985

Outline of the talk

1. Typing
2. Information Hiding
3. Coupling and Cohesion
4. Polymorphism
5. **Binding**
6. Delegation

Binding

- Establishes the mappings between names and data objects and their descriptions

- Early binding (static binding, at compile time)
  - The premature choice of operation variant, resulting in possibly wrong results and (in favorable cases) run-time system crashes

- Late binding (dynamic binding, at run time)
  - The guarantee that every execution of an operation will select the correct version of the operation, based on the type of the operation's target

    - "Typing addresses the existence of at least one operation; binding addresses the choice of the right one among these operations, if there is more than one candidate." [B. Meyer, 97]

Early binding – Java example

- Class methods are selected at compile time:

```
class B {
    public static String foo() { return “foo”; }
}
class A extends B {
    public static String foo() { return “bar”: }
}
public class C {
    public static void main(String[] args) {
        B o1 = new B();
        B o2 = new A();
        A o3 = new A();
        System.out.println(o1.foo());
        System.out.println(o2.foo());
        System.out.println(o3.foo());
    }
}
```

Results in:

“foo”

“foo”

“bar”

Late binding – Java example

- Instance methods are selected at run time:

```
class B {
    public String foo() { return “foo”; }
}
class A extends B {
    public String foo() { return “bar”: }
}
public class C {
    public static void main(String[] args) {
        B o1 = new B();
        B o2 = new A();
        A o3 = new A();
        System.out.println(o1.foo());
        System.out.println(o2.foo());
        System.out.println(o3.foo());
    }
}
```

Results in:
“foo”
“bar”
“bar”

Outline of the talk

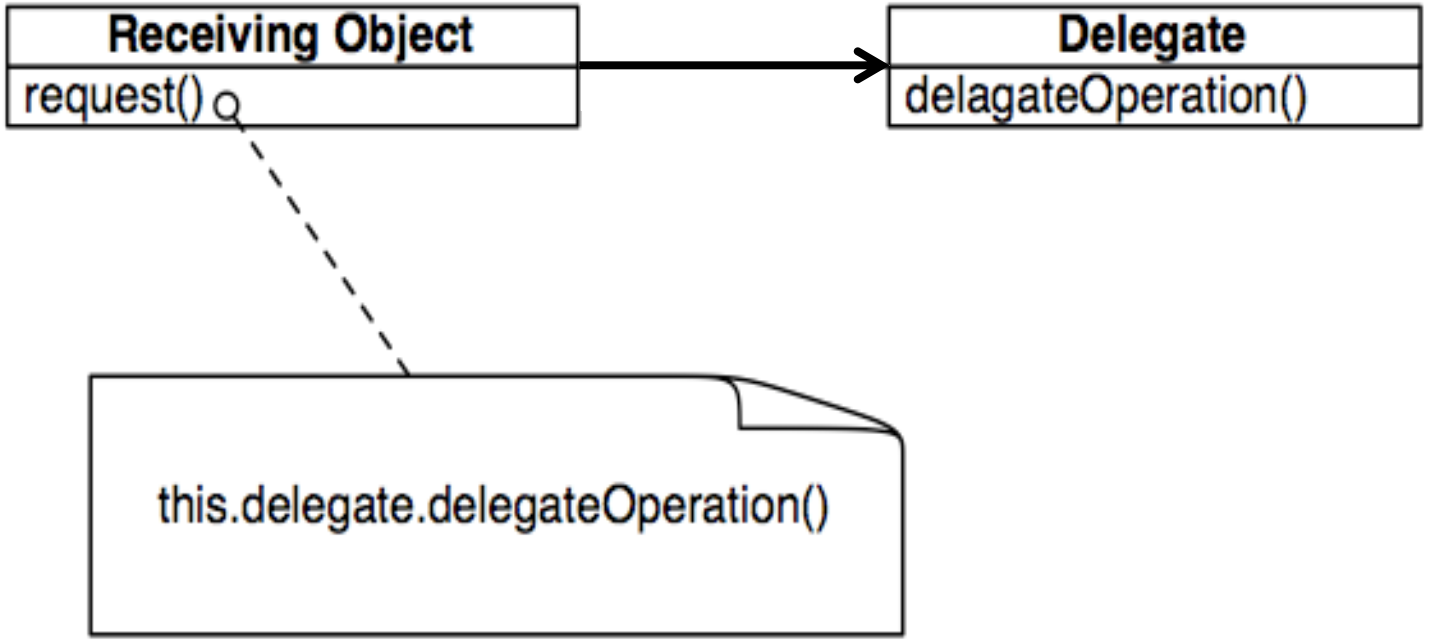
1. Typing
2. Information Hiding
3. Coupling and Cohesion
4. Polymorphism
5. Binding
6. **Delegation**

Delegation - real life examples

- I would like to open the door of the classroom. I ask my assistant to do it for me
  - My assistant is my delegate for opening the door

Delegation in object-oriented programming

- The previous examples gave an intuitive view about delegation
- First, some terminology:
  - The Receiving Object delegates a request to perform an operation to its Delegate

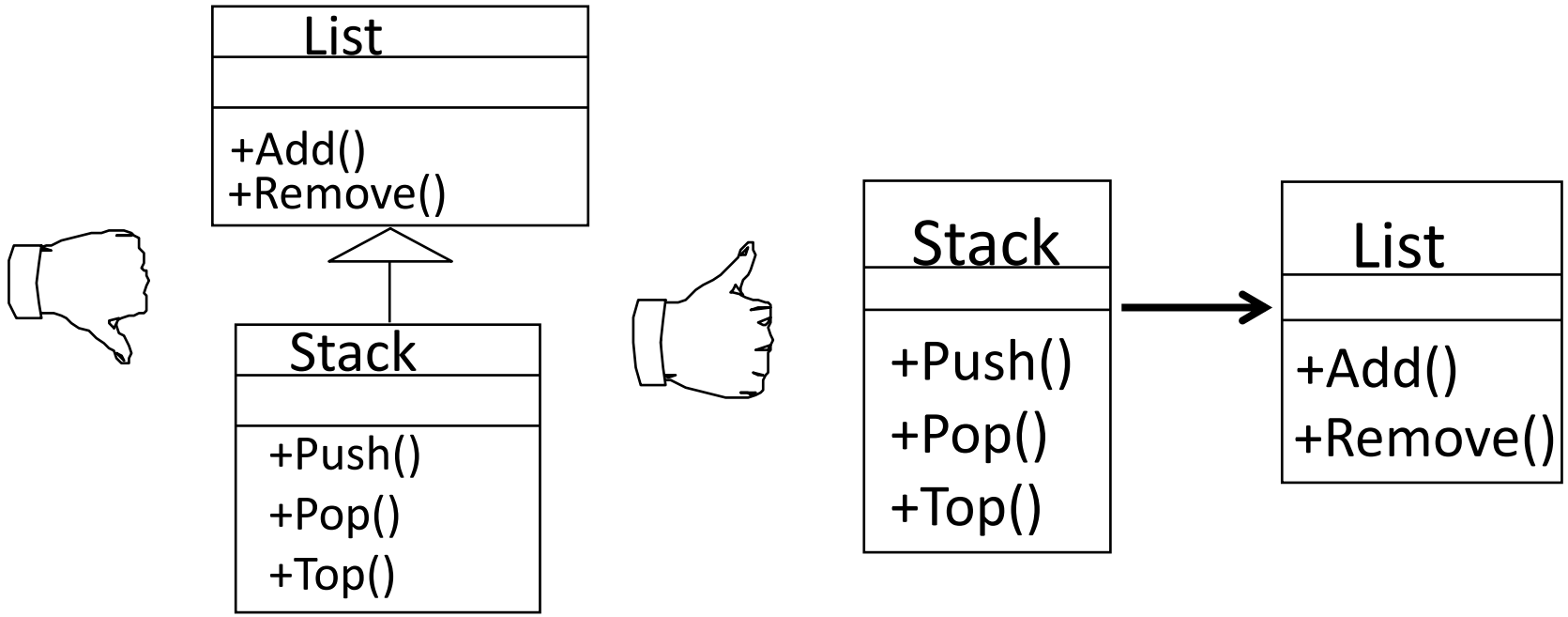


Delegation in object-oriented programming

- Delegation has several aspects
  - Delegation simply involves passing a method call to another object, transforming the input if necessary
  - Delegation extends the behavior of an object
- Delegation: A mechanism for code reuse in which an operation resends a message to another class to accomplish the desired behavior.

Delegation vs Subclassing

- **Delegation:** A class catches an operation and sends it to another class
- **Subclassing:** A subclass extends a super class with new operations or overrides existing operations
- What is better?



Summary

1. For design and architecture patterns, it is important to understand underlying concepts from language design

2. As architect needs to know building materials, you should know how compilers and languages support patterns

3. Polymorphism and delegation are basic building blocks of many design patterns and architectural patterns