Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE UNIVERSITÄT DARMSTADT

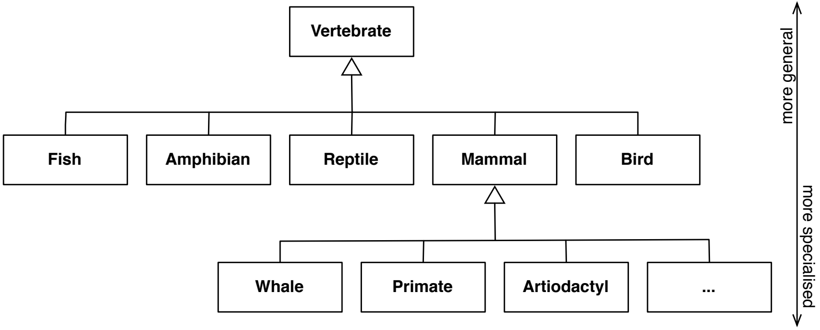
2015-04-21

Inheritance

"A mechanism that allows the data and behavior of one class to be included in or used as the basis for another class"

[Armstrong 2006]

- A generalisation/specialisation relationship between classes
- Instrumental in building class hierarchies
- Class hierarchies can be trees or DAGs
- Inheritance defines *is-a* relationship



Inheritance and Coupling

Recall: High coupling between components reduces maintainability, re-usability (e.g. changes to one component will likely impact the other)

Inheritance causes strong coupling:

- Base class may expose implementation details to subclasses (breaks encapsulation)
- Changes in base class may break subclasses

```java
class A {
    public void foo() {...}
}
class B extends A {
    public void foo() { super.foo(); somethingElse(); }
}
```

Types

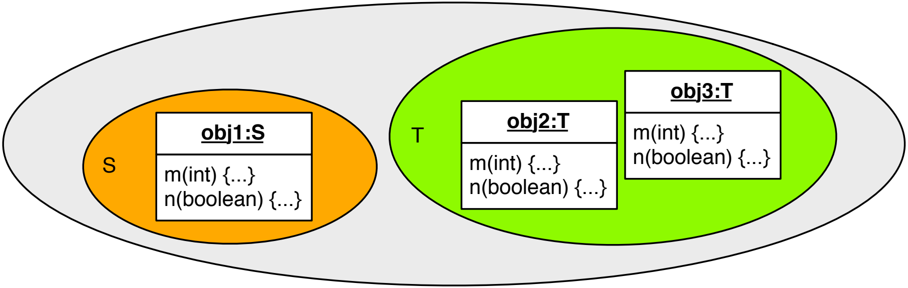
A type is a set of values sharing some properties.
A value v has type T if v is an element of T.

What are the "properties" shared by the values of a type?

- *Nominal* types: type names
  Examples: C++, Eiffel, Java, Scala
- *Structural* types: availability of methods (and fields)
  Examples: Python, Ruby, Smalltalk

Nominal and Structural Types

Type membership



Type equivalence

- S and T are different in nominal type systems
- S and T are equivalent in structural type systems

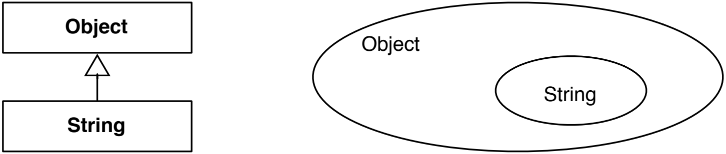
```
interface S {
    m(int);
    n(boolean);
}
```

```
interface T {
    m(int);
    n(boolean);
}
```

Subtyping

Subtype relation

The subtype relation corresponds to the subset relation on the values of a type.



`String` <: `Object`

Substitution principle

Objects of subtypes can be used wherever objects of supertypes are expected.

Classes, Interfaces and Types

Type

A *specification* of behaviour (at some level of abstraction)

Class

An *implementation* of that behaviour

Interface

“A set of messages (methods) that defines the explicit communication to which an object can respond”

$\Rightarrow$ Interfaces describe the behaviour of objects.

Classes, Interfaces and Types: In practice

Types are usually defined using interfaces:

```
interface MusicPlayer { void play(); void pause(); }
```

Classes implement behaviour:

```
class IPod implements MusicPlayer {
    void play() {...}
    void pause() {...}
    void syncWithItunes() {...}
}
```

In some languages (e.g. Java) classes also define types:

```
MusicPlayer m = new IPod();
m.play();
m.syncWithItunes(); // type error
```

```
Ipod p = new IPod();
p.play();
p.syncWithItunes(); // OK
```

`Ipod` <: `MusicPlayer`

Inheritance vs. Subtyping

Inheritance

A mechanism for sharing (re-using) code

Subtyping

A *substitutability* relationship (recall *substitution principle*): allows an object to be used in place of another.

Inheritance vs. Subtyping (2)

In many languages (Java, C++, Scala, ...), inheritance is equivalent to subtyping. Example:

```
class A { void foo() {...} }
class B extends A { }
```

Consequently

1. B inherits implementation of A
2. B is a subtype of A.

This can lead to problems:

- If A and B are unrelated ("B is not an A") but share some behaviour, we want inheritance but no subtyping.
- If "B is an A" but their behaviour is different, we want the subtyping relation but no inheritance (recall that inheritance causes high coupling).

Decoupling inheritance and subtyping in Java

1. B “inherits” (reuses) implementation of A; B is not subtype of A

```
class A { void foo() {...} }
class B {
    A a; // Composition
    void foo() { return a.foo(); }
}
```

2. B is subtype of A; B does not inherit implementation of A

```
interface A { void foo(); }
interface B extends A { }
class AImpl implements A { void foo() {...} }
class BImpl implements B { void foo() {...} }
```

Composition/Aggregation

Composition/aggregation is the principle of building new abstractions from old (or larger abstractions from smaller).

- “Has-a” / “part-of” relationship (“B has an A”; “A is a part of B”)
- Composition implies that the parts cannot exist without the whole
- Aggregation implies that the parts are independent from the whole in their existence
- A mechanism for sharing/reusing code (compare to inheritance)
- Delegation principle: the whole *delegates* work to its parts

Delegation

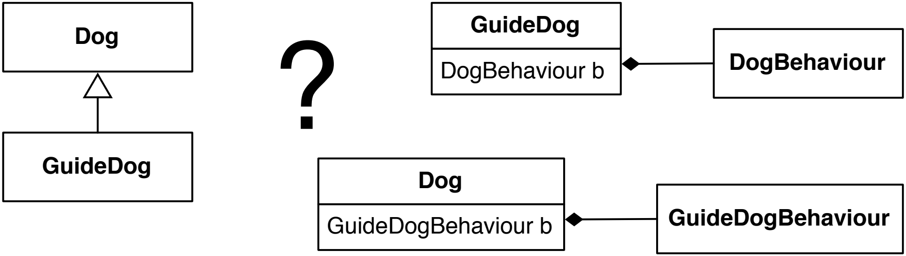
The principle of one object relying upon another to provide some specified functionality

- Classes built by composition/aggregation use the delegation principle to implement behaviour.

```
class Car {
    Engine engine; // composition
    Headlights lights;
    AirconSystem aircon;

    public void start() {
        engine.start(); // delegation
        aircon.on();
        lights.on();
    }
}
```

Inheritance vs. Composition



Which to use?

- What kind of relationship? – “is-a” vs. “has-a”
- Inheritance causes strong coupling between classes
- Inheritance may establish subtyping relationship
- ”Favor object composition over class inheritance.” [Gamma et al. 1994]

Polymorphism

The quality of existing in or assuming different forms
[Merriam-Webster Dictionary]

In the context of programming:

A program part is polymorphic if it can be used for objects of several types

$\Rightarrow$ Polymorphism is a code reuse mechanism.

Some types of polymorphism:

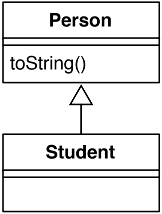
- Subtype polymorphism
- Ad-hoc Polymorphism
- Dynamic dispatch
- Parametric polymorphism

Subtype Polymorphism

Substitution principle: Objects of subtypes can be used wherever objects of supertypes are expected.

Subtype polymorphism is a direct consequence of the substitution principle: program parts working with supertype objects work as well with subtype objects.

Example:
The method `toString` works with both `Person` and `Student` objects.



Ad-hoc Polymorphism

- Ad-hoc polymorphism allows several methods with the same name but different arguments
- Also called *overloading*
- Can be modeled statically (at compile time) by renaming

```
Integer add(Integer i1, Integer i2) {
    return i1 + i2;
}
Integer add(Integer i, String s) {
    return i + Integer.parseInt(s);
}
Integer add(String s1, String s2) {
    return Integer.parseInt(s1) + Integer.parseInt(s2);
}
```

Parametric Polymorphism

- Parametric polymorphism uses type parameters
- One implementation can be used for different types
- Type mismatches can be detected at compile time
- Examples: Java generics, CLOS generic functions

```
class List<G> {
   G[] elems;
   void append( G p ) { ... }
}
```

```
List<String> myList;
myList = new List<String>(); myList.append("Hello");
```

```
myList.append(myList);
```

Dynamic Dispatch

Dynamic Dispatch selects the implementation of a polymorphic method *at runtime*.

- Different from method overloading (where the method implementation is chosen at compile time)

```
class Vehicle {
    void drive() { print("driving a vehicle"); }
}
class Car extends Vehicle {
    void drive() { print("driving a car"); }
}
class Bike extends Vehicle {
    void drive() { print("driving a bike"); }
}

Vehicle v = new Bike();
v.drive();
```

Dynamic Dispatch (2)

How to determine the “most appropriate” method?

Single Dispatch

- The method inside the *receiver*’s class

Multiple Dispatch

- The method whose parameter types most closely match the types of the method call’s arguments

Multi-dimensional dispatch

Based on…

- The method *receiver* (callee)
- The method sender (caller)
- Context[1] of the method invocation

[1] Context is “anything that is computationally accessible” [Hirschfeld et al. 2008]

Example: Single Dispatch in Java

```
abstract class Vehicle {
   void collide(Vehicle v) { print("vehicle collision"); }
}
class Car extends Vehicle {
   void collide (Vehicle v) { print("Car hits vehicle"); }
   void collide (Bike b) { print("Car hits bike"); }
}
class Bike extends Vehicle {
   void collide (Vehicle v) { print("Bike hits vehicle"); }
   void collide (Car c) { print("Bike hits car"); }
}

Vehicle car = new Car();
Vehicle bike = new Bike();
car.collide(bike); // prints "Car hits vehicle"
```

Same code compiled using Groovy – a language with *multiple dispatch* – would print “Car hits bike”.

Example: Simulating Multiple Dispatch in Java

```java
abstract class Vehicle {
    void collide(Vehicle v) { print("vehicle collision"); }
    abstract void collideWithCar(Car car);
    abstract void collideWithBike(Bike bike);
}

class Car extends Vehicle {
    void collide (Vehicle v) { v.collideWithCar(this); }
    void collideWithCar(Car c) { print("Car hits car"); }
    void collideWithBike(Bike b) { print("Bike hits car"); }
}

class Bike extends Vehicle {
    void collide (Vehicle v) { v.collideWithBike(this); }
    void collideWithCar(Car c) { print("Car hits bike"); }
    void collideWithBike(Bike b) { print("Bike hits bike"); }
}
```

The “Double Dispatch” pattern simulating multiple dispatch

This Week's Reading Assignment

- Oscar Nierstrasz, *Ten Things I Hate About Object-Oriented Programming*. Banquet speech given at the European Conference on Object-Oriented Programming (ECOOP), 2010

- Download link:
  `http://blog.jot.fm/2010/08/26/ten-things-i-hate-about-object-oriented-programming/`

References I

Armstrong, Deborah J. (2006). "The Quarks of Object-oriented Development". In: *Communications of the ACM* 49.2, pp. 123–128.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides (1994). *Design Patterns*. Addison-Wesley.

Hirschfeld, Robert, Pascal Costanza, and Oscar Nierstrasz (2008). "Context-oriented Programming". In: *Journal of Object Technology* 7.3, pp. 125–151.

Slides 4–6 are based on the lecture "Concepts of Object-Oriented Programming" by Peter Mueller, ETH Zurich, `http://www.pm.inf.ethz.ch/education/courses/coop`.