Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE
UNIVERSITÄT
DARMSTADT

2015-04-14

Course Infrastructure

- Lecturer:
  Radu Muschevici
  Software Engineering Group
  Building S2/02, Office A205
  Email: `radu@cs.tu-darmstadt.de`

- Office hours: Tuesdays 16.00–17.00
  (or by email appointment)

- Lecture: Tuesdays 11.40–13.20, Room S202/C110

- For every lecture I will recommend the reading of one scientific paper/article related to the topics discussed in the lecture.

Course Infrastructure (cont.)

- Lecture website and forum:
  `https://moodle.informatik.tu-darmstadt.de/course/view.php?id=438`

  Please register to receive important course announcements!

- Lecture slides will be made available before each lecture.

- Exam: ~~Tuesday, 21 July 2015, 16:30~~
  Wednesday, 15 July 2015, 19.00 (90 minutes),
  50% required to pass

Course Topics Overview

1. Object-Oriented Programming
2. Prototype-based Programming
3. Traits & Mixins
4. Aspect-Oriented Programming
5. Subject-Oriented Programming
6. Context-Oriented Programming
7. Feature-Oriented Programming
8. Delta-Oriented Programming

Core Concepts of (Object-Oriented) Programming (Languages) – A Recap

Code Reuse (a.k.a. Software Reuse)

Taking advantage of code that already exists, thus saving resources (time, costs, memory etc.)

How to…

- Organise code to avoid repetition?
- Maximise opportunities for reusing code fragments?

Examples of Code reuse

- Copy & Paste (duplication) – difficult to maintain, error-prone
- Software library
- Generative programming
- Design patterns
- Polymorphism
- Inheritance
- …

Abstraction – Example

Media player controls:

- *Abstract* the concepts of playing, pausing, fast-forwarding, etc.
- Implementation of these concepts varies, but is *hidden*
- When a new implementation of a media player arrives, users can continue to use it just as they did before.

Abstraction

Reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose

Why?

To cope with complexity.

What to abstract in a program

- Data
- Control (behaviour)

Examples of abstraction mechanisms

- Subroutine, function
- Abstract data type, (abstract) class, object …

Information Hiding

Hide information from the participants of an interaction, if that information is irrelevant for the interaction. $\Rightarrow$ Leads to concepts of *encapsulation* and *abstraction*

What information to hide?

- Representation of data
- Properties of a device, other than required properties

Why? – To facilitate future changes

- Hide system details likely to change independently
- Separate parts that are likely to have a different rate of change
- Expose (in interfaces) only assumptions unlikely to change

Personal Mastery: If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.

[Ingalls 1981]

Encapsulation

- Define the boundaries of objects (or modules, classes): What is public (can be used freely), what is private (off-limits)?
- Object access only through its public *interface*

Encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of an object that do not contribute to its essential characteristics.

[Booch et al. 2007]

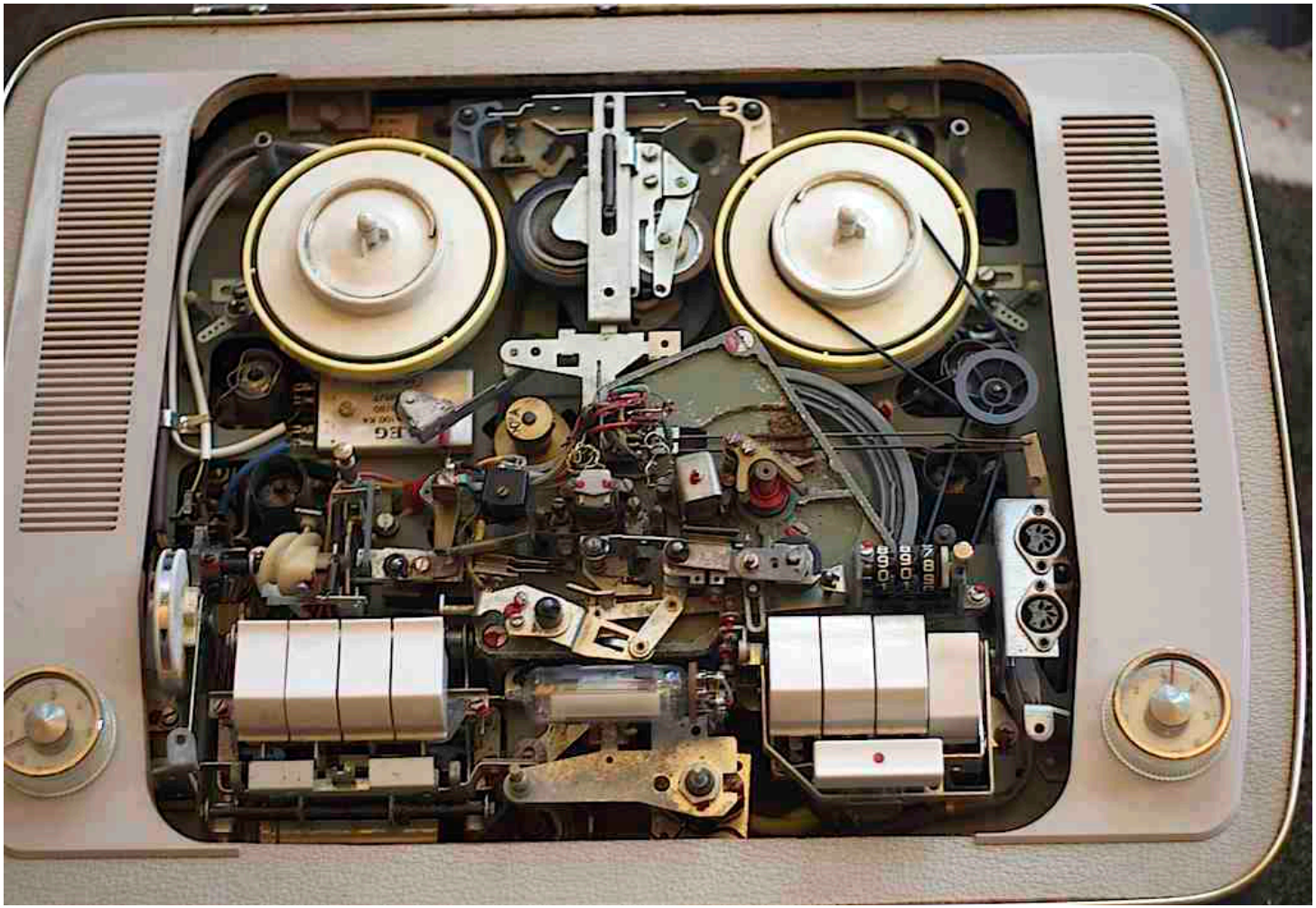
Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.

[Booch et al. 2007]

Encapsulation – Example

- Access only via public interface
- Inner workings hidden away (off limits)

Encapsulation – Example



- Access only via public interface
- Inner workings hidden away (off limits)

Modularity

> No component in a complex system should depend on the internal details of any other component.
>
> [Ingalls 1981]

Idea

- Decompose a system into modules (as opposed to a monolithic design)
- Hide the module's implementation details (encapsulation).
- Define the relationships among the modules of a system.

Code Modules

Properties of modules

- *Separation of concerns* into independent modules: each program task forms a separate module
- Module provides well-defined *interface* for the outside world (i.e. other modules): requirements and provisions
- Possible to reason about each module independently and in parallel (develop, compile, test)
- A module is an autonomous unit of deployment

Example: Java 9 (Project Jigsaw)

```
module jdk.jndi @ 8-ea {
  requires local jdk.auth.internal@8-ea;
  requires local jdk.base.internal@8-ea;
  requires optional jdk.desktop@8-ea;
  requires jdk.rmi@8-ea;
  exports com.sun.security.auth.*;
  exports javax.naming.*;
  provides service
    sun.net.spi.nameservice.NameServiceDescriptor
    with sun.net.spi.nameservice.dns.DNSNameServiceDescriptor;
  ...
}
```

- **requires**: other modules (with their versions)
- **exports**: public types available to other modules
- **provides**: services (set of interfaces) **with** implementation

Interfaces

Encapsulating code into modules raises the need for *interfaces* that define how separate modules relate and interact.

An interface is a *contract* that defines…

- what clients of the module can depend upon (*provided* interface)
- what the module itself depends upon (*required* interface)

Types of interfaces

- Syntactic interfaces: How to invoke operations (signatures)
- Semantic interfaces: What the operations do (pre and post conditions, performance & reliability guarantees, etc.)

Object/Class Interfaces

A set of messages that defines the explicit communication to which an object can respond

- Interfaces describe the behaviour of objects
- Object's implementation is hidden (encapsulated) behind its interface

Responsibility-Driven Design[1]

Design the object's interface according to its responsibilities:

- What actions is this object responsible for?
- What information does this object share?

Conversely:

- Limit access to object state: private fields
- Limit access to behaviour unless part of the interface: private methods

[1] Wirfs-Brock et al. 1989.

Separation of Concerns

The principle of dividing a program into distinct features with as little overlap in functionality as possible

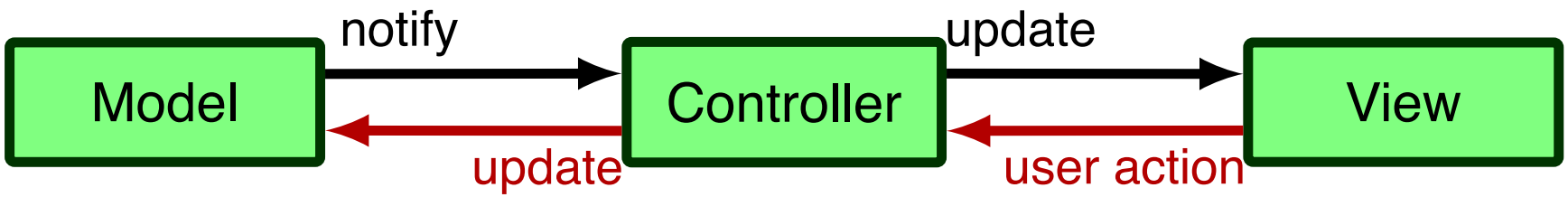
- Single Responsibility: each component should be responsible for only one specific feature or functionality
- Frequent problem: Some concerns are *cross-cutting* (e.g. security, logging, caching) so they cannot be easily separated and encapsulated

Separation of concerns leads to…

- Modularity
- Low coupling, high cohesion

SoC Example: Model-View-Controller

MVC is a standard software architectural pattern for implementing user interfaces. Three responsibilities – three separate objects:



1. *Model* manages internal state and behaviour (data, logic) of the application
2. *View* presents information to the user
3. *Controller* updates the view; accepts user input and notifies the model accordingly

Coupling and Cohesion

Coupling

- A measure of how closely interconnected *separate* software components are.
- High coupling between components reduces maintainability, re-usability (e.g. changes to one component will likely impact the other)

Cohesion

- A measure of how closely related the elements of *one* module are.
- Lack of cohesion leads to poor readability and high complexity (e.g. hard to navigate and understand code)

Coupling and Cohesion (2)

- Cohesion and coupling are interrelated. The greater the cohesion of individual modules in the system, the lower the coupling between modules.
- Low coupling & high cohesion decrease complexity and increases code readability, re-usability and maintainability.

How to achieve loose coupling and high cohesion?

Generally, a class or module should be…

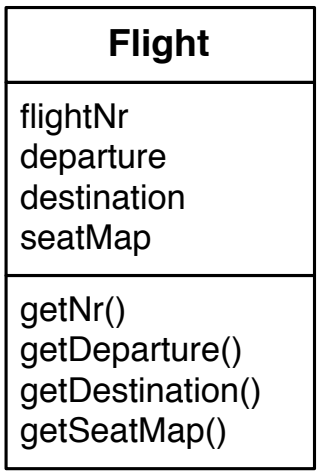
- *sufficient*: capture enough characteristics of the abstraction to permit meaningful & efficient interaction
- *complete*: captures all meaningful characteristics of the abstraction
- *primitive*: provide only primitive operations.

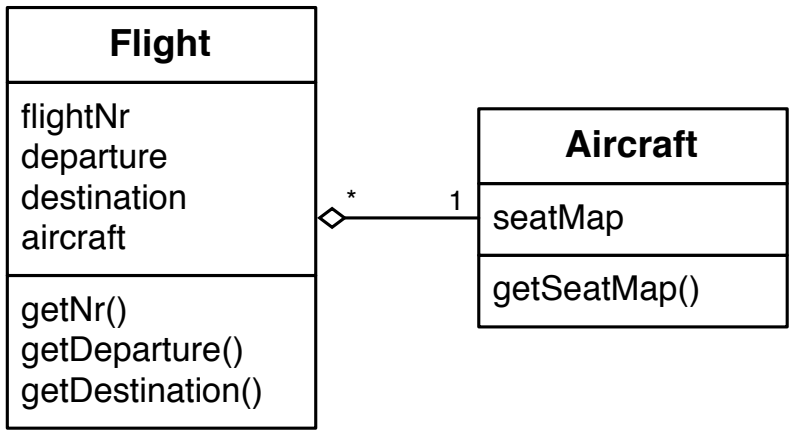
Coupling: Java Example

```
1 class Storage {
2     public void m() {
3         Package myPackage = new Package();
4         int v = myPackage.volume;
5     }
6 }
7 class Package { public int volume; }
```

```
1 class Storage {
2     public void m() {
3         Box myPackage = new Package();
4         int v = myPackage.getVolume();
5     }
6 }
7 interface Box { int getVolume(); }
8 class Package implements Box { int getVolume() { ... }; }
```

Cohesion: Example





Increasing cohesion by delegating responsibilities

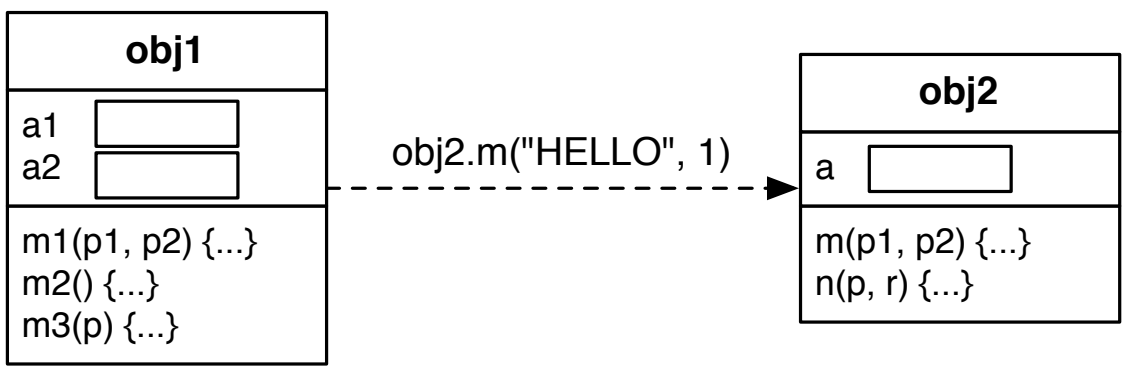
Object Model – The Philosophy

The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.

[Lehrmann Madsen et al. 1993]

The Object Model

- A software system is a set of cooperating objects
- Objects have state and behaviour
- Objects exchange messages



Objects

> An object is a value exporting a procedural interface to data or behavior.
>
> [Cook 2009]

An object has

- state
- behaviour
- identity

Objects lead to a characteristic

- program structure: class hierarchy
- execution model: communicating objects

Objects: Example

```
class Integer {
    private int value; // state

    public Integer add(Integer other) { // behaviour
        value += other.getValue();
        return value;
    }
    public boolean equals(Integer other) {
        return value == other.getValue();
    }
    ...
}
Integer a = new Integer(19);
Integer b = new Integer(19);
a.equals(b); // test equality => true
a == b; // test identity => false
```

Methods & Message Passing

Methods

Methods are operations that an object offers to its clients.

- object construction
- object destruction
- read and/or modify the state of the object

Message Passing

- Sending a message to an object $==$ calling a method of an object
- Send information from object to object (“cooperating objects”)
- The callee object is known as the message/method *receiver*
- Choosing the receiver of a message is generally a runtime activity (due to dynamic dispatch)

Classification & Classes

Classification

A general technique to hierarchically structure knowledge about concepts, items, and their properties.

Classes in OO languages

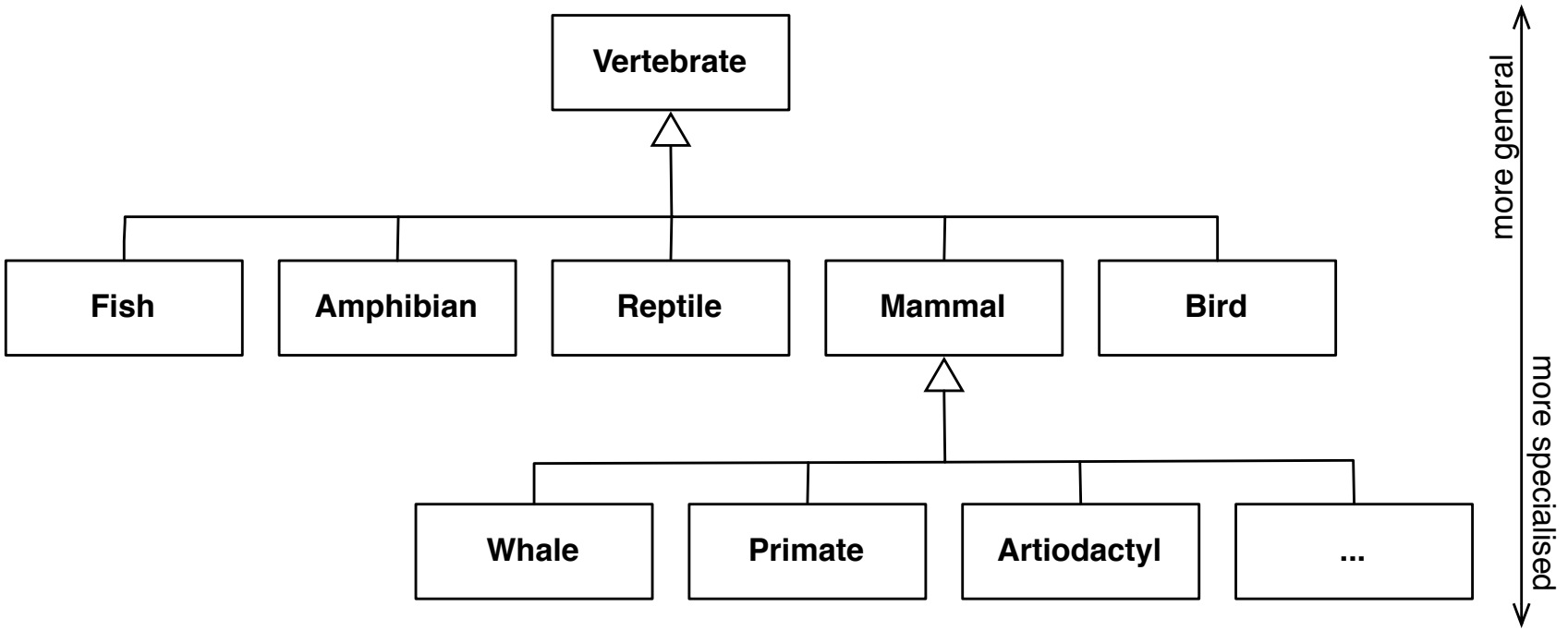
- A means to classify similar objects
- Describe the structure and behaviour of objects
- Source (template) for creating new objects
- The chief mechanism for extension in (most) OO languages

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

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

This Week's Reading Assignment

- Aldrich, J. *The power of interoperability: Why objects are inevitable*. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (New York, NY, USA, 2013), Onward! 2013, ACM Press, pp. 101–116.

- Download link:
  `http://dl.acm.org/citation.cfm?id=2509578.2514738`
- Freely accessible from within the TUD campus network

References I

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

Booch, Grady, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston (2007). *Object-Oriented Analysis and Design with Applications*. 3rd. Addison-Wesley.

Cook, William R. (2009). “On Understanding Data Abstraction, Revisited”. In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’09. ACM Press, pp. 557–572.

Ingalls, Daniel H. H. (1981). “Design Principles Behind Smalltalk”. In: *BYTE Magazine*.

Lehrmann Madsen, Ole, Birger Møller-Pedersen, and Kristen Nygaard (1993). *Object-oriented Programming in the BETA Programming Language*. New York, NY, USA: Addison-Wesley.

References II

Wirfs-Brock, Rebecca and Brian Wilkerson (1989). “Object-oriented Design: A Responsibility-driven Approach”. In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’89. ACM Press, pp. 71–75.

Slides 23, 24 and 25 are based on the lecture “Concepts of Object-Oriented Programming” by Peter Mueller, ETH Zurich, `http://www.pm.inf.ethz.ch/education/courses/coop`.