Software and Programming I

Object-Oriented Design

Roman Kontchakov / Carsten Fuhs

Birkbeck, University of London

Outline

- Discovering classes and methods
- Relationships between classes
- An object-oriented process
  for software development
  - Sections 12.1–12.3

http://higheredbcs.wiley.com/legacy/college/horstmann/1118063317/web_chapters/ch12.pdf

- slides are available at
  www.dcs.bbk.ac.uk/~roman/sp1

Problem Solving: the Story So Far

- **objects** are first-class citizens that exchange messages: `object.method(parameter values)`
- similar objects are organised into **classes**
- organising classes for related concepts into **inheritance hierarchies** (e.g., `Employee` extends `Person`)
  - **polymorphism**: talk to different objects in the same way, but they may process the request differently
  - allows code reuse when they do use the same solution

But how do we know which classes (and methods) to have, and how to come up with the inheritance hierarchies?

Discovering Classes

Starting point: **requirements specification** in natural language

Candidates for classes: **nouns** (from the problem domain)

Examples: `Person`, `Student`, `Employee`, `BankAccount`, `CashRegister`, . . .

Suitable classes may already exist in Java standard libraries / earlier programs; or maybe we can extend an existing class

Class name tells us what its objects are supposed to do

Don't go too far: e.g., address as class `Address`, or just a `String`?

depends what we need from addresses for the task . . .

NB: code may later need classes outside the problem domain for "technical" purposes, e.g., user interface, database access, basic data structures like `ArrayList`, . . .

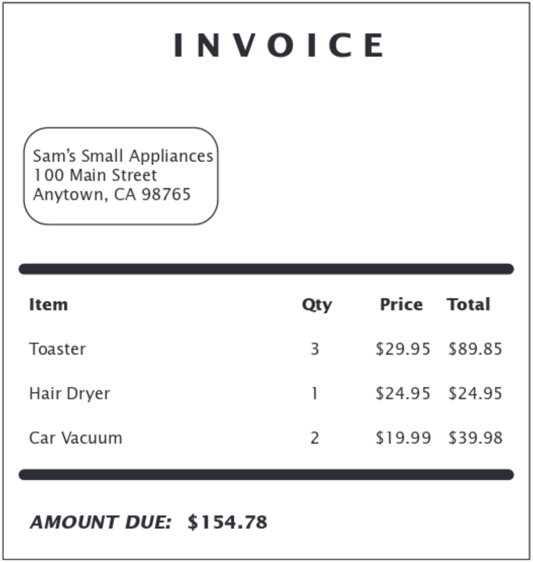
$\Rightarrow$ requirements give us the "**domain model**"

Discovering Classes: Example

Program for invoices that list each item with its price and quantity, the overall total due amount, as well as the address of the customer

Possible classes:

```
Invoice
LineItem
Customer
```



INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

***AMOUNT DUE:* $154.78**

C. Horstmann, Java for Everyone, 2013, p. W551

Discovering Methods: CRC Cards (1)

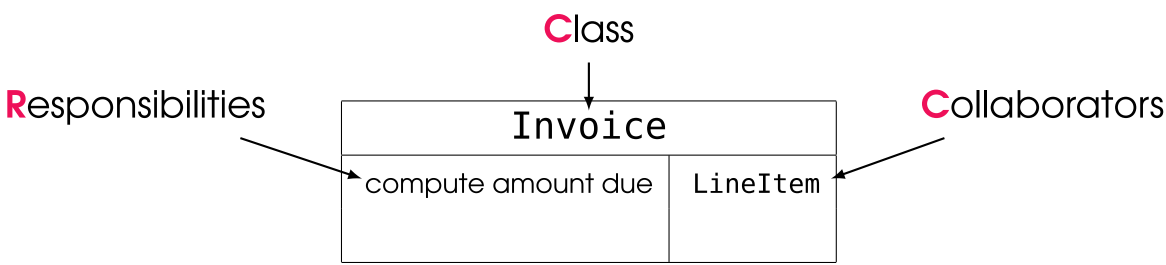
Candidates for methods: **verbs** in the task description

Invoice example: *computing the overall total amount due*

But which of the classes should take the method,

`Invoice`, `LineItem` or `Customer`?

Approach: use **CRC Cards** (index cards)



Discovering Methods: CRC Cards (2)

Responsibility ≈ method, but can be higher level

(Java implementation may need several methods)

Listing collaborators may reveal their own responsibilities

(e.g., `LineItem` must tell its own total)

CRC cards can be rearranged on table,
handy for **discussions**

A single CRC card should not have too many responsibilities

keep design simple

Later: find out how classes are **related**

- Can we move common responsibilities to a superclass?
- Are there independent clusters?

Cohesion (1)

Public interface of a class should be **cohesive**:
everything should be closely related to **single** concept represented by the class

```
public class CashRegister {
    public static final double NICKEL_VALUE  = 0.05;
    public static final double DIME_VALUE    = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    . . .
    public void enterPayment(int dollars, int
       quarters, int dimes, int nickels, int pennies)
        { . . . }
    . . .
}
```

Q: What is wrong here?

Cohesion (2)

Two separate concepts: cash register and values of coins

⇒ Rather have a dedicated `Coin` class
with `Coins` responsible for knowing their own value

Allows us to simplify the `CashRegister` ...

```
public class CashRegister {
    . . .
    public void enterPayment(Coin[] coins) {
        . . .
    }
    . . .
}
```

... responsibilities of cash register and coins are separated

Relationships Between Classes

Good cases

- Can we move some common responsibilities to a superclass?

  → less implementation effort, cleaner design

- Are there (groups of) classes that are completely independent from each other?

  → can assign different programmers to implement them, no worries about one waiting for the other

Dependency

Dependency relationship between classes

aka: "knows about"

- Example: `CashRegister` knows about `Coin` objects
- but `Coin` does not know about `CashRegister`

Notation for dependencies in **UML Class Diagrams**: dashed arrow, "normal" arrow tip

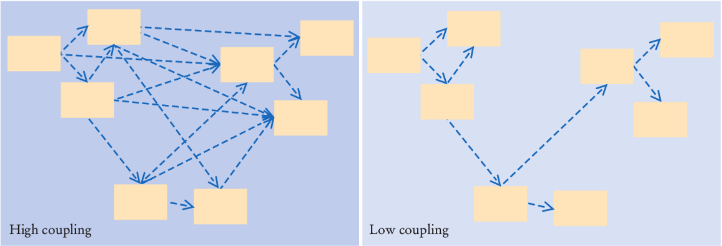
In Java:

`CashRegister` needs `Coin` to **compile**, but not vice versa

$\Rightarrow$ `CashRegister` **depends** on `Coin`, but not vice versa

Coupling

Coupling is the degree of dependency between classes



C. Horstmann
Java for Everyone
2013, p. W555

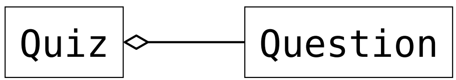
Aim for **low coupling**:

- If, say, `Coin` changes in the future, all classes that depend on `Coin` may need to be changed as well
- Want to be able to use `Coin` in another program without dragging in a lot of **dependencies**

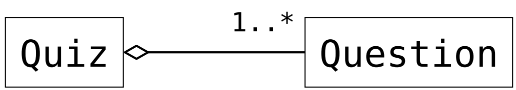
Aggregation (1)

Aggregation relationship between classes   aka: "has-a"

- Example: a `Quiz` contains (1 or more) `Question` objects, so class `Quiz` **aggregates** class `Question`
- UML notation: solid arrow with diamond-shaped tip at the aggregating class



- Can also keep track of **multiplicities** (how many do I have?)



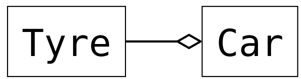
- Later on implementation level:

```
private Question[] questions;
```

as instance variable of `Quiz` (multiple `Questions` in a `Quiz`)

Aggregation (2)

- Another example: `Car` aggregates `Tyre` objects



- Aggregation is a stronger form of dependency:
  - if you **have** something, you certainly **know** about it
  - `Quiz` also depends on `Scanner` (to read input), but `Quiz` does not aggregate `Scanner`
- Generally: need aggregation (→ instance variable) in your class if you need to remember an object **between** calls to the methods of your class

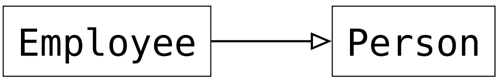
Inheritance

Inheritance relationship between classes

aka: "is-a"

- Example: an `Employee` is a `Person`,
  class `Employee` **inherits from** class `Person`
- Inheritance is also a stronger form of dependency

UML notation: solid arrow with triangle-shaped tip at the superclass



In Java:

```java
public class Employee extends Person {
    . . .
}
```

Inheritance: A Pitfall

Should `Tyre` be a subclass of `Circle`?

Could inherit methods for computing radius, centre point, . . .

No! A `Tyre` is a car part, not a geometric object like `Circle`

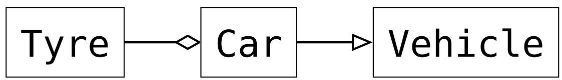
Use **aggregation** instead of inheritance for “code reuse”:

```java
public class Tyre {
    private Circle boundary;
    . . .
    public double radius() {
        // delegate method calls to aggregated object
        return this.boundary.radius();
    }
}
```

Inheritance: Another Example

`Car`: Every `Car` **is a** `Vehicle`, every `Car` **has** `Tyres`

⇒ inherit from `Vehicle` and aggregate `Tyre`



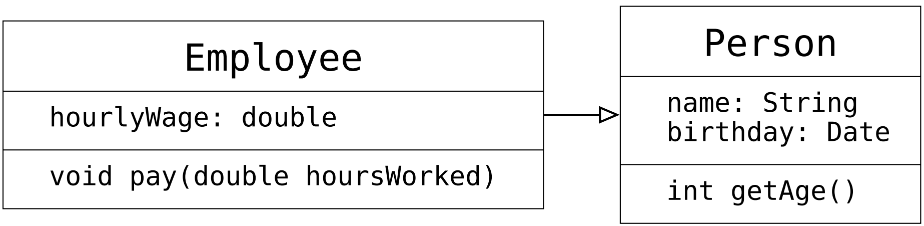
In Java:

```java
public class Car extends Vehicle {
    private Tyre[] tyres;
    . . .
}
```

Attributes & Operations in UML

Often want more details than just class names in nodes

- Attributes (= instance variables) and operations (= methods)



- Use (conceptually) primitive classes as attribute types
- Do **not** represent aggregated classes from the diagram as attributes (redundant information)

Example: Modelling Vehicles

- Every vehicle has an owner.
- A bicycle is a vehicle with a tyre diameter in inches.
- A rickshaw is a special bicycle that can transport a passenger for a fare. Here, the maximum additional weight in kg is a relevant property.
- A car is a vehicle with four tyres and with a power in kW.
- A police car is a car that can toggle a siren and that has a specific number of blue lights.
- A taxi is a car that can transport passengers for a fare.
- Vehicles designed to transport passengers can tell how many passengers are currently in the passenger area of the vehicle.

Example: Modelling Vehicles, v1

Some parts of the spec are ambiguous (and others are missing):

- Don't all vehicles have at least one passenger (the driver)?
- Even if we don't count the driver, can't all vehicles transport also a passenger, even a bicycle?

While we are building the domain model, we may find shortcomings in the results of an earlier activity, the requirements analysis.

Similarly, while coding, you may find that some parts of the design don't make sense as such.

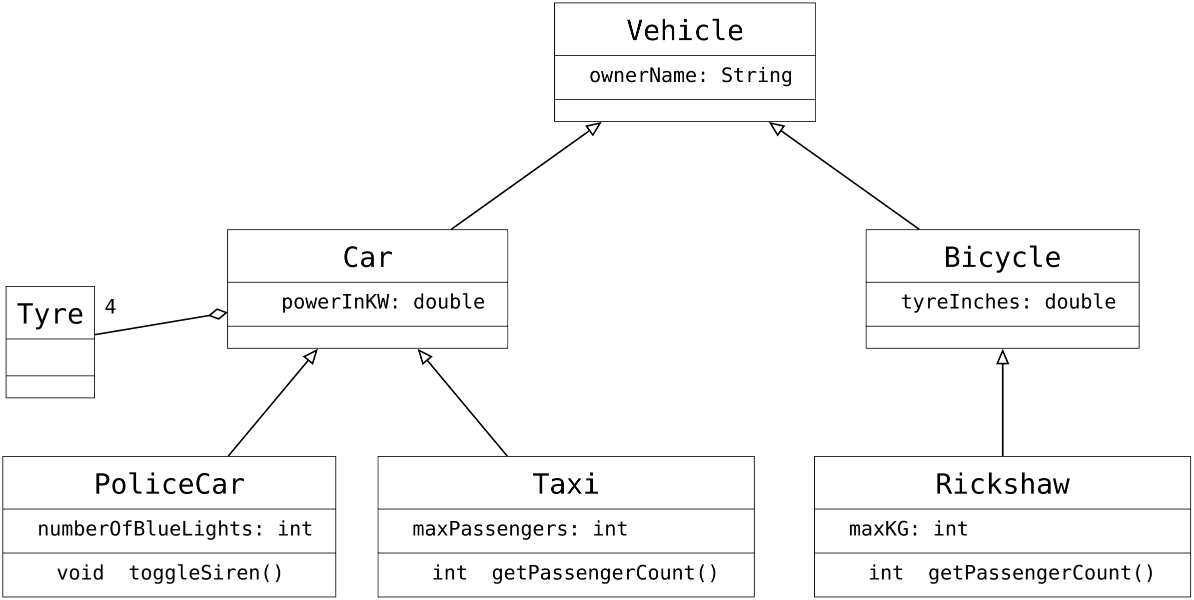
Revisiting (and fixing) the results of earlier activities is actually quite common in software development processes.

Due to the flexibility of software, this is not as costly as in other engineering disciplines (and lets us upgrade the software later with new features). (Lab example: new `cancelLast()` feature in `CashRegister`)

Example: Modelling Vehicles, v2

- Every vehicle has an owner.
- A bicycle is a vehicle with a tyre diameter in inches.
- A rickshaw is a special bicycle such that the driver can transport a passenger for a fare. Here the maximum additional weight in kg is a relevant property.
- A car is a vehicle that has four tyres and that has a power in kW.
- A police car is a car that can toggle a siren and that has a specific number of blue lights.
- A taxi is a car that can transport passengers for a fare. The number of passengers is limited by the taxi's passenger capacity.
- Vehicles whose primary purpose is to transport passengers in addition to the driver can be queried how many passengers are currently in the passenger area of the vehicle.

Example: Modelling Vehicles v2, class diagram



Example: Modelling Vehicles v2, remarks

- Still further variations possible:
  aggregate wheels in `Vehicle`? (but: ships are vehicles too!)
- We do not mention the getters and setters for attributes in the domain model (they are artefacts of the implementation).
- Note the common

  ```
  int getPassengerCount()
  ```

  operation in `Taxi` and `Rickshaw`

  (**Java interfaces** can let us talk to different objects in a uniform way)

Example: Modelling Vehicles v2, implementation (1)

From class diagram to Java:

replace aggregation by instance variable

(single object or array of objects),

get Java code:

```
public class Vehicle {
    private String ownerName;
}
```

```
public class Bicycle extends Vehicle {
    private double tyreInches;
}
```

Example: Modelling Vehicles v2, implementation (2)

```java
public class Rickshaw extends Bicycle {
    private int maxKG;

    public int getPassengerCount() {
        return 0; // TODO Auto-generated method stub
    }
}
```

```java
public class Tyre {
}
```

```java
public class Car extends Vehicle {
    private Tyre[] tyres;
    private double powerInKW;
}
```

Example: Modelling Vehicles v2, implementation (3)

```java
public class PoliceCar extends Car {
    private int numberOfBlueLights;

    public void toggleSiren() {
        // TODO Auto-generated method stub
    }
}
```

```java
public class Taxi extends Car {
    private int maxPassengers;

    public int getPassengerCount() {
        return 0; // TODO Auto-generated method stub
    }
}
```

Architecture (1)

So far: focus on the **domain model**.

In larger projects you may also need

- a dedicated **user interface** (GUI, web app, command line, . . .)
- and a **persistence layer**
  (store data not only in memory, but also on permanent storage, e.g., an SQL database)

Those are not represented in the domain model;
a **separate design model** includes such classes

Architecture (2)

Can often use a **layered** architecture with 3 layers:

Layer 1 User interface (JavaFX, web, command line, . . .)

Layer 2 Business logic

(implementation of the domain-specific aspects goes here, e.g., Invoice, LineItem, . . .)

Layer 3 Persistence layer (databases and other technical services, like logging)

Lower layers **cannot see** subsystems on higher layers

- Flexibility
- Reusability
- Today's web app may become tomorrow's mobile app, but the "business logic" may not have to change

An Object-Oriented Software Development Process

1. Gather requirements specification
   (talk to customer, domain experts, . . .)
2. Use CRC cards to find
   classes, responsibilities, collaborators
3. Use UML class diagram to record classes
   and their relationships in domain model
4. Refine domain model to design model
5. Write classes with corresponding method stubs in Java
6. Use comments to document the desired behaviour
7. Write the implementation in Java
8. Test your implementation

Take Home Messages

Goal: from requirements (in natural language) to Java code

- discovering classes and methods
- representing relationships between classes in a UML class diagram
- translating the UML class diagram to Java code
- a software development process