Class Design I
Class Attributes and Methods

Learning goals:

determine some appropriate **attributes** for a class given a general description of the class

determine some appropriate **methods** for a class given a general description of the class

assess whether a given class description is **cohesive** and **robust**

Reading:
2nd edition
Chapter: 9,
Sections: 9.1-9.4, 9.6-9.9
3rd & 4th edition
Chapter: 8,
Sections: 8.1-8.4, 8.6-8.9

Some ideas come from:

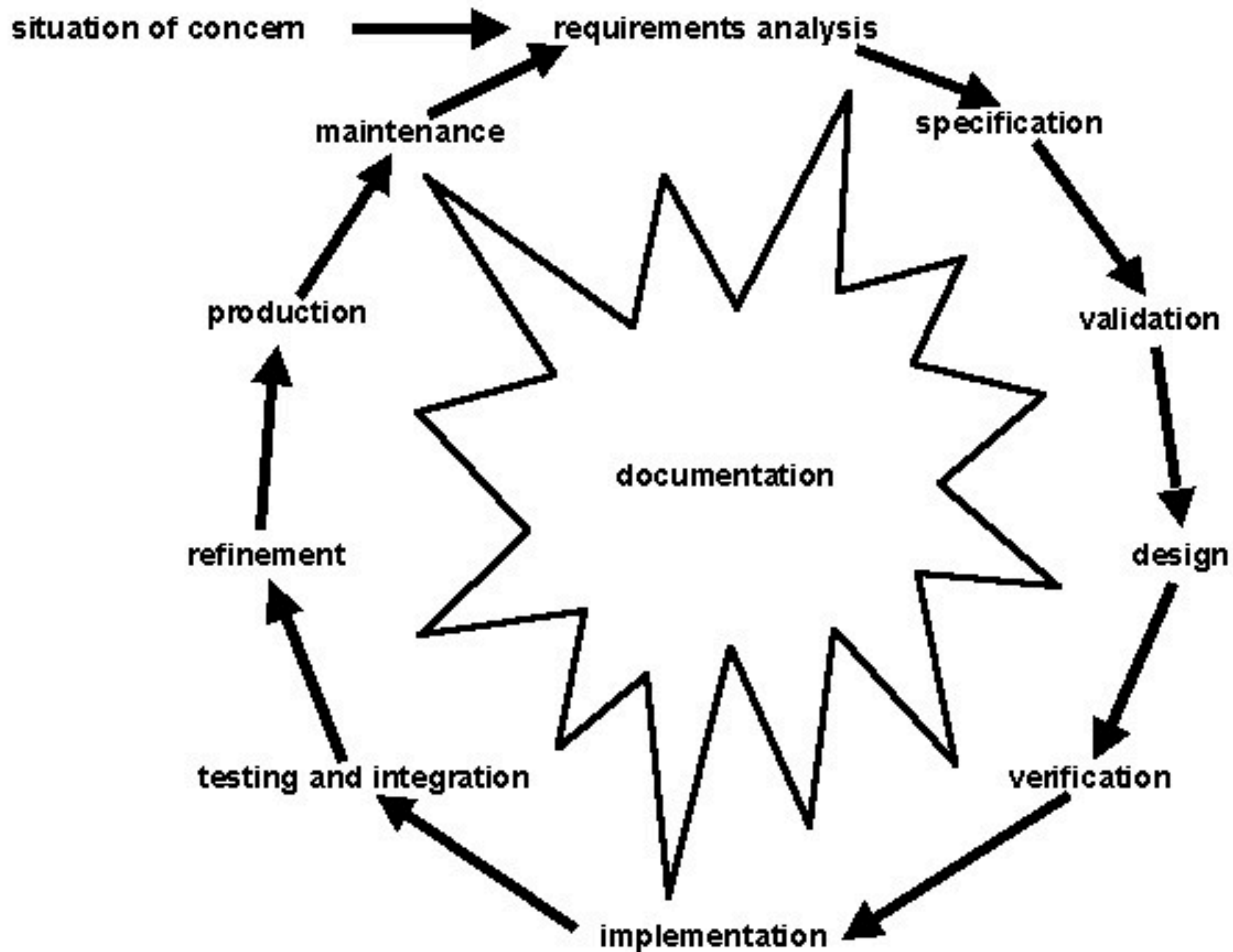
- ***"Practical Object-Oriented Development with UML and Java" R. Lee, W. Tepfenhart, Prentice Hall, 2002.***
- ***"Object-Oriented Software Development Using Java", Xiaoping Jia, Addison Wesley, 2002***

Software Engineering

How do we engineer quality software?

- Program structuring and modularization
- Abstraction
- Efficient data structures
- Efficient algorithms
- *Design*

Design Life Cycle



Design

What is design? What makes something a design problem? It's where you stand with a foot in two worlds --- the world of technology and the world of people and human purposes --- and you try to bring the two together.

- Mitchell Kapor, A Software Design Manifesto (1991)

A concrete example…

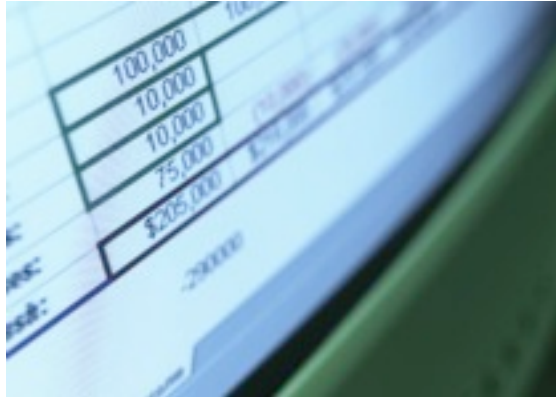
Technology?

Human purpose?

Design

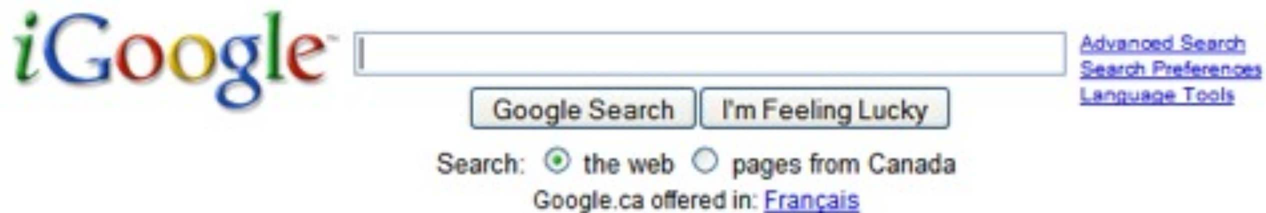
A software example…

Design: Many different aspects



Software program

Architecture



Human-computer interface

Software Design

Software program

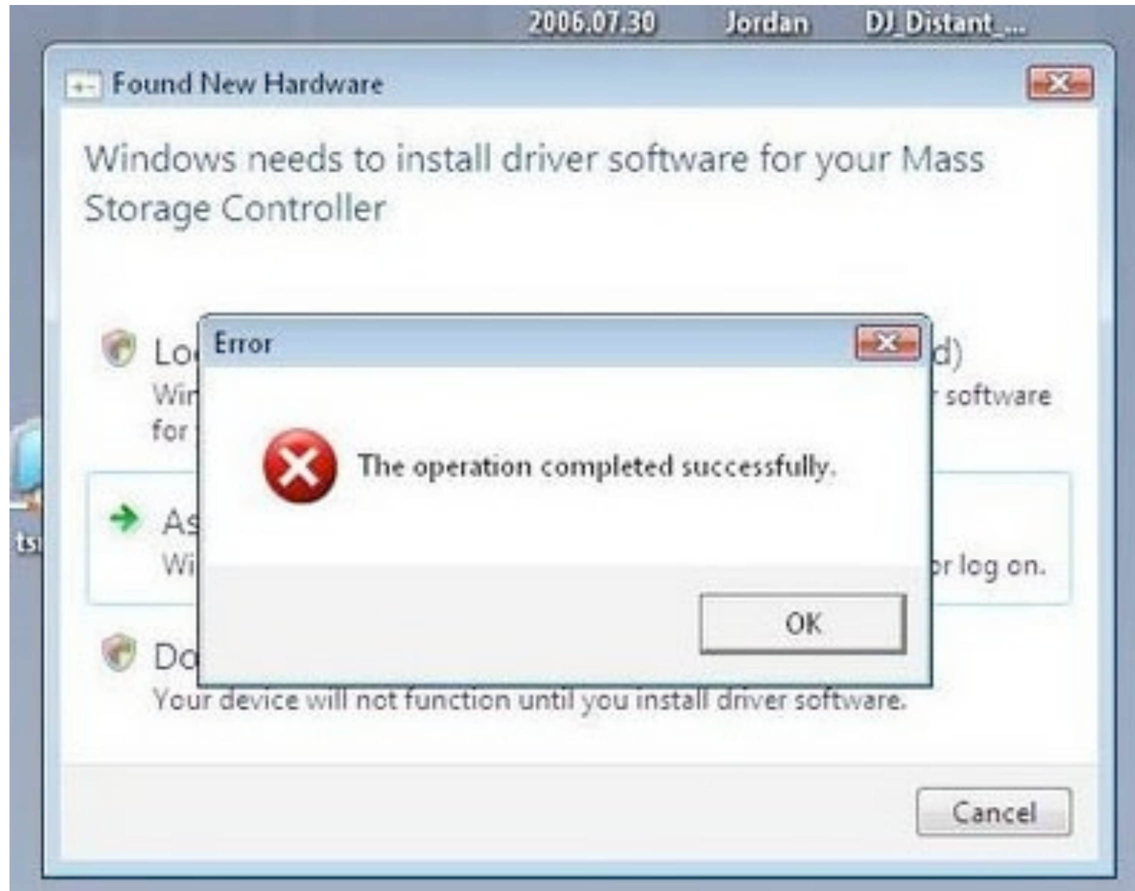
Based on a description of what the system **should** do (requirements), we need to identify and define:

- classes
- attributes of each class
- behaviour of each class
- relations between classes

During design, **focus is on how the system will work, not on implementation (precise) details**

Design is guided by **principles** and **heuristics**, not definitive rules

Software design... we don't always get it right!



Yup, I didn't expect that to happen....

Example: A music system for a phone

- What a music system for a phone should be able to do…

- Let’s identify some classes…

Class Design (aka low-level design)

Our focus now is on how to design a **single class**. We'll assume that we know which class(es) we need; designing classes and their relationships will be a topic later this term

For each class we are designing, we need to define

- *The data (attributes or fields) associated with the class' concept*
- *The behaviour (responsibilities, public services) associated with the class' concept; this includes public methods and the class invariants*

We will ignore for now…

- *Private methods*
- *The data structures used to implement collections of data*

Designing a single class: Identifying attributes

Objective: identify and name all data that a class needs to support the behaviour of objects of that class

Goal: each class should have **high cohesion**

- each class must represent a single concept
- all data and operations must be highly related to each other

Initial heuristic: consult the requirements (problem description), looking for **adjectives** and **possessive phrases** related to objects of the class of interest to discover what information the objects of the class will need

Designing a single class: Identifying attributes

Review: eliminate any false attributes:

- attributes whose value depends on the context

  E.g., Consider a `Person` class. Such a class is unlikely to have an `employee_id` attribute because a person may have zero, one, or more jobs

- attributes that are unrelated to the rest

  Either these attributes do not belong or the class should be split

Designing a single class: Identifying attributes

For each attribute, must distinguish:

1. Kind of attribute:
   - **instance attribute** : value of attribute depends on the object
   - **class attribute**: one value per class
2. Visibility modifiers (e.g. private, protected, package, public)
3. Kind of value (type)
   - primitive values (int, double)
   - references to objects
4. Whether it is a constant attribute
   - in Java will be declared as `final static`

Designing a single class: Identifying attributes

Objective: identify and name all operations a class needs to provide/support

Initial heuristic: Consult the requirements (problem description), look for verbs related to objects of the class of interest to discover the likely responsibilities of the class

Review: check for problem specific methods needed to

- ***maintain*** the state (attributes) of the object
- perform ***calculations*** the class is responsible for
- ***monitor*** what objects of the class are responsible for detecting and responding to
- respond to ***queries*** that return information without modifying an object of the class

It is often helpful to identify and go over some user scenarios to ensure as complete behaviour as possible is designed

Designing a single class: Designing each method

For each method, need to distinguish:

- Type, i.e.
  - **instance methods** are associated with an object
  - **class methods** are applied to a class and are independent of any object-- declared as static and can only access static attributes (not instance attributes)
- Visibility modifiers: *private, protected, package, public*
- Signature (i.e. method name + parameter types)-- *a class cannot have two methods with the same signature*

Notes…

- `final` methods cannot be overridden in any subclass
- overloaded method = method name with more than one signature

Designing a single class: Additional guidelines

Ensure each class has

- a “good”---useful for clients---set of **constructors**
- appropriate **accessors** for certain attributes (getter methods)
- appropriate **mutators** for some attributes (setter methods)
- a **destructor** if necessary (in Java this is done by defining the ***finalize()*** method in the class; we’ll use *very* sparingly, if at all)
- equality method – ***equals()***
- string representation method (good for debugging) – ***toString(***)

May need to define methods for

- cloning : for creating copies – ***clone()*** or ***copy constructor***
- hash code: returns an integer code that “represents” the object - ***hashCode()***

We’ll talk more about cloning, hashCode, etc. later in term. See “Effective Java” book by Joshua Bloch if interested in class design.

Designing a single class: Additional guidelines

A **side effect** of a method is any modification that is observable outside the method

Some side effects are necessary; some are acceptable; others are wrong

Some guidelines:

- Accessor methods should not have any side effects
- Mutator methods should change only the implicit argument
- Avoid designing methods that change their explicit arguments, if it is possible
- Avoid designing methods that change another object i.e. in class `Account`:
  - bad design: method printBalance that prints balance on System.out
  - good design: method getBalance that returns balance

Bank Account Example

Problem Description:

The bank wants a software system to maintain customer accounts. Each account belongs to a single customer and is identified by a unique id assigned by the bank. The owner and the id of an account can never change. A customer is identified by their name and can open an account, deposit and withdraw money into it and check the account balance, which must never be negative.

Bank Account Example

Problem Description:

> The bank wants a software system to maintain customer accounts. Each account belongs to a single customer and is identified by a unique id assigned by the bank. The owner and the id of an account can never change. A customer is identified by their name and can open an account, deposit and withdraw money into it and check the account balance, which must never be negative.

. . .

Suppose we design a class `Account` to represents a single account. What would be the attributes (data components) for the `Account` class?

Would be correct to add the customer address and phone number as components to `Account` class?

Bank Account Example

What should be the operations?

Summary:

Well-designed modules (classes) should be...

Highly cohesive

The elements within should be *related*

Loosely coupled

Few to no dependencies on other modules (e.g. message coupling)

Adequately general

Don’t overdo it!

Representing Class Design: UML

When designing software, we need to focus on **how the design works**, not all of the details of expressing the design in a programming language

Software developers sometimes use UML (Unified Modelling Language) to express a design

UML’s graphical modelling notation lets developers focus on

- *classes and their important attributes and methods*
- *relationships between classes*
- *and to see that information in a condensed form*

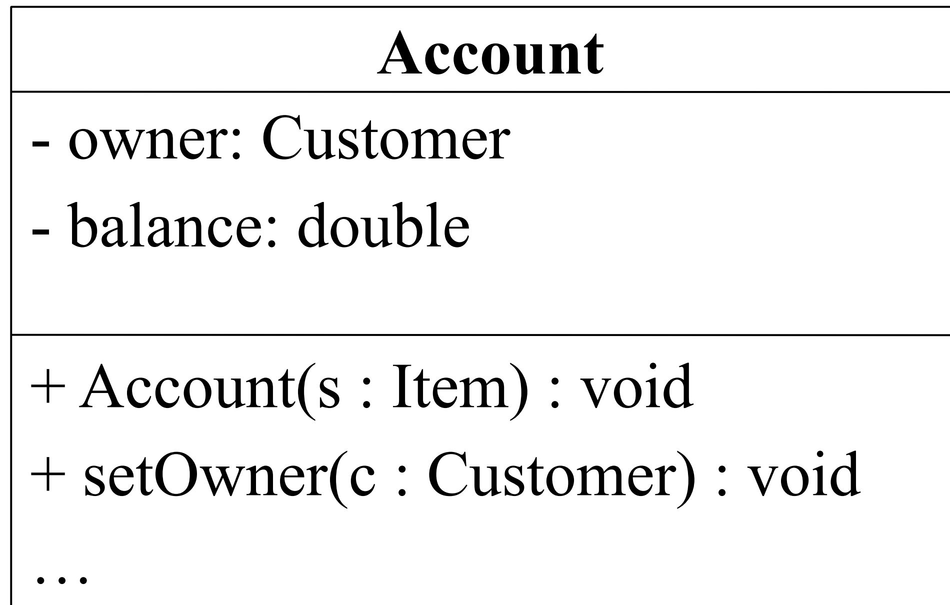
UML has many different diagram types, we’ll consider only class diagrams in this course

UML Class Diagrams

Use a rectangle with 3 compartments showing

1. the class **name**
2. the class **attributes** (data components or data fields)
3. the class **methods**

Example:



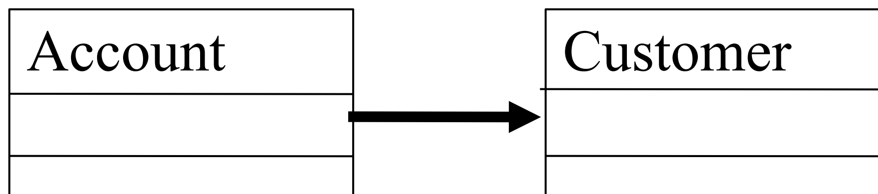
UML Class Relationships

Relationships are shown by arrows

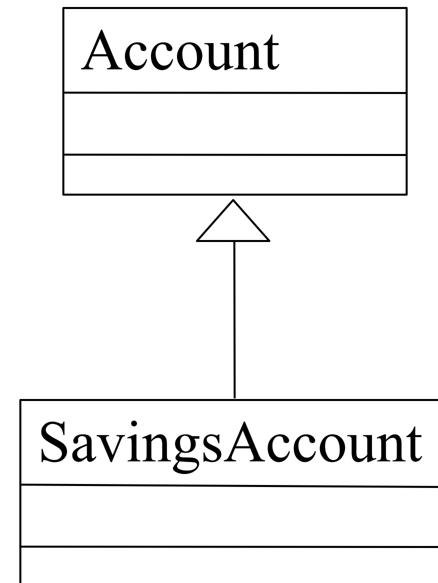
We'll consider just two types of relationship:

Association : one class contains one or more references to another class

Inheritance : one class extends another class



Association Example



Inheritance Example

Is this enough?

We have seen how to

- identify attributes for a class
- identify methods (the behaviour) of a class

We need a way to specify the behaviour of each method

- specification must be **independent of programming language**
- must balance between
  - o the important aspects that **need** to be captured by any implementation
  - o give an implementor the **freedom** to decide on the rest

Next class we'll discuss **class contracts** help specify method behaviour