Génie logiciel
Software Engineering
C. Petitpierre

Thèmes du cours
• UML (Unified modeling language)
• Processus de développement
• Design Patterns
• Java Server Faces, EJB3
• JET, use of templates
• Chargement de databases en XML
  SAX, FSM, JavaCC
• Langage permettant la validation:
  CCS et threads

Personnages et sites
• Martin Fowler
  http://martinfowler.com/
• Scott Ambler
  http://www.ambysoft.com/

Projet de génie logiciel
• Gestion des commandes des clients dans un restaurant:
  Prendre une commande
  L'afficher au comptoir et à la cuisine
  Indiquer quand elle est prête
  Faire les factures
  ...
• Plan de travail disponible sur le Web:
  http://ltiwww.epfl.ch/~petitp/GenieLogiciel

UML
Unified Modeling Language
(OMG object management group)
http://www.uml.org/
http://www.omg.org/

UML usage
• UML is particularly useful to sketch IT applications (pencil and eraser or PowerPoint)
• It can be used to document applications (reverse engineering consists of recreating the diagrams from the code, which is a cheap way of making a “cheap” documentation)
• It is used by ArcStyler and OptimalJ (among others) to create applications (it is difficult to enter the application details in these systems)
UML / RUP: history

• Rational founded in 1981 around a processor and development environments for Ada and C++
• In 1990, Grady Booch develops Rose for Rational, a tool to specify models
• In 1995, Rational hires James Rumbaugh, buys Ivar Jacobson’ company, and the three “amigos” define UML (unified modeling language)
• Same year, Philippe Kruchten defines RUP (Rational unified process)
• In 2002, IBM buys the company for ~ $2B (beside UML, they are consultants for many customers)

UML Rules of Thumb
(features not bugs !)

• Nearly everything in UML is optional
• UML models are rarely complete
• UML is designed to be open to interpretation
• UML is intended to be extended

UML 2.0 in a nutshell, Dan Pilone, O'Reilly, 2005

UML

• Use cases
• Class diagrams
• Sequence diagrams
• Collaboration diagrams
• State diagrams / Activity diagrams

Use case diagram for an ATM
(automate de banque)

UML: Use case (scenario)

1. A **client** inserts a card into the ATM.
2. The **system** reads and validates the card information.
3. The **system** prompts for a PIN.
4. The **client** enters the PIN. *(client, system = actors)*
5. The **system** validates the PIN
6. The **client** selects “Withdraw Money”
7. The **client** enters the requested amount.
8. The **system** requests the amount from the client's account.
9. The **system** asks the client to remove the card.
10. The **system** dispenses the requested amount from the banknote buffer *(only done if the requested amount has been granted in 8).*

UML: Use case (extension)

1. A **client** inserts a card into the ATM.
2. The **system** reads and validates the card information.
3. The **system** prompts for a PIN.
4. The **client** enters the PIN. *(client, system = actors)*
5. The **system** validates the PIN
6. The **system** validates the PIN
7. The **client** selects “Withdraw Money”
8. The **system** dispenses the requested amount from the banknote buffer *(only done if the requested amount has been granted in 8).*

Extensions
5a. If the PIN is invalid, the system rejects the card
6. The **client** selects “Withdraw Money”
7. The **client** enters the requested amount
8. …
If you work in a new domain (bank, assurances, production…) you will need to create a small dictionary.

**Class diagram with visibility**

- **public**: visible from everywhere
- **private**: local to the class
- **protected**: visible only from the derived class
- **package**: visible from the package (Java)

(Actually this is too far into the details for a sketch)

**Class diagram: inheritance**

Superclass

SomeClass

AnotherClass

**Class diagram: association**

SomeClass

AnotherClass

Weakest form of relationship

Can be used, for example, to indicate that a method may call a method in another class.

**Class diagram: aggregation**

Bill

- `numberOfItems`: int
- `addItem(item:Item)`
- `getFirstItem()`: Item
- `getNextItem()`: Item

+ **public**: visible from everywhere
- **private**: local to the class
- **protected**: visible only from the derived class
- **package**: visible from the package (Java)

(Omondo on Eclipse)

Used to show that an object instantiated from that class contains another object (in an attribute) or a collection of objects (collection in an attribute)
Class diagram: composition

Strong relationship. If the main class is deleted, the subclass has no meaning any more and is destroyed. Not often used.

Collaboration diagram (Omondo)

Collaboration diagram (context of the application)

State diagram / state charts

State diagram / state charts
Relation between a state chart and a use case

Activity diagrams

- Inspired from the Petri nets
- May fork and join threads
- The use of forks and joins is not a very good idea!! It makes it very difficult to follow the state of a program

Activity diagram: an example

<< Stereotypes >>

<< Stereotypes >>

UML: a technique to start a development

Identifying classes:

- **Borrowing**  The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to twelve items at one time. Only members of staff may borrow journals.

  - library: out of scope
  - item: vague
  - short term loan: is an event
  - time: out of scope
  - week: a measure of time
  - system: is not part of the domain

  We keep:


Use of a sequence diagram

The example explains how the Java threads work
**Corresponding code: 1st solution**

```java
public class Main {
    static ActiveObject a1;
    static public void main (String [] args) {
        a1 = new ActiveObject (); // create an active object
        a1.start(); // start the active object
    }
}
class ActiveObject extends Thread {
    public void foo() { }
    public void run () {
        for (;;) {
            System.out.println("ActiveObject running");
            try { Thread.sleep(3000); } catch (Exception e) {} // blocking
        }
    }
}
```

**Corresponding code: 2nd solution**

```java
public class Main {
    static AnObject a2;
    static public void main (String [] args) {
        a2 = new AnObject (); // create an active object
        new Thread(a2).start(); // start the active object
    }
}
class AnObject implements Runnable {
    public void foo() { }
    public void run () {
        for (;;) {
            System.out.println("AnObject running");
            try { Thread.sleep(2000); } catch (Exception e) {} // non-blocking
        }
    }
}
```

**Use of sequence diagrams**

Explain how the wait-notify commands work: see code next slide

** Creation of the previous threads**

```java
sem = new Sem();
new Thread(new T1(sem)).start();
new Thread(new T2(sem)).start();
```

**Code corresponding to the sequence diagram**

```java
class T1 implements Runnable {
    Sem tm;
    public T1 (Sem t) { tm = t; }
    public void run () {
        for (;;) {
            tm.stop();
        }
    }
}
class T2 implements Runnable {
    Sem tm;
    public T2 (Sem t) { tm = t; }
    public void run () {
        for (;;) {
            tm.kick();
            Thread.sleep(1000);
        }
    }
}
class Sem {
    public void kick () {
        synchronized (this) {
            notify();
        }
    }
    public synchronized void stop () {
        synchronized (this) {
            try { wait(); } catch (InterruptedException ie) {} // non-blocking
            catch (InterruptedException ie) {}
        }
    }
}
```

There are two different ways of using synchronized.

** Yet another way of using synchronize**

```java
class Sem {
    String s = "" // empty string, or any object
    public void kick () {
        synchronized (s) {
            notify();
        }
    }
    public synchronized void stop () {
        synchronized (s) {
            try { wait(); } catch (InterruptedException ie) {} // non-blocking
        }
    }
}
```

Every Java object has provision for thread synchronization.