2.0. Requirements Capture and Analysis with UML

The Unified Modeling Language 2.0 (www.uml.org)

- Class/Object diagrams (static)
- Sequence diagrams (dynamic)
- Statecharts (dynamic)
- OCL (static)

Standard developed by OMG adopting earlier ideas (Booch, Rumbaugh, Jacobson)
2.1. Requirements Models

Models express requirements, architecture & detailed design. (Video Game examples)

**Use Case Model:** *In this situation do the following ...*

e.g. how to engage a foreign character
Object / Class Model: with objects of these classes ...

Object provides a set of services, e.g. engagement class.

State Model: by reacting to these events ...
state/event → reaction

e.g. if foreign character shoots while we are friendly ... run away
2.2. O-O Analysis & Design

The basic **OOA&D approach** (iterative)

1. State the main **use cases**
2. Convert the use cases to **sequence diagrams**
3. Select the resulting **domain classes**
4. If not finished goto 1.
2.3. UML Use Case Diagrams

UML gives us a graphical language to organize all our use case scenarios into one overview:

**Use Case Diagrams**

These display organisation of:

- actors
- use cases
- associations,
- dependencies
Precondition: app has been activated
1. Clerk clicks “check out”
2. Clerk swipes bar code
3. ...

Postcondition: video is registered to customer

UML Comment

Activation

Check in

Check out

Add customer

Add video

Clerk

Stocker
Actor with **role name**, **use case dependency** and **multiplicity**

Clerk

Use case with **name** and **dependent actors** and **multiplicities**

1 – one and only one

* - zero or more

1 ... * - one or more
**Precondition**: app has been activated
1. Clerk clicks “check out”
2. Clerk swipes bar code
3. ...

**Postcondition**: video is registered to customer

Pre and postcondition describing black-box effect of executing a use case on the system

Expressed as a comment only
A requires B - B must complete before A starts
A equivalent B – identical activities and flow, but user sees them as different
A extends B – all activities of B are performed but A adds activities or slightly modifies them.
Can use to **refactor** or **simplify** use cases
2.4. Use Case Modeling

In O-O based IT projects, we can begin the requirements analysis phase with a **Use Case Analysis**

A *use case* is a *common* or representative situation where one or more external *users* and/or systems interact with the system to *solve* an instance of a common *problem*
Notice the underlined word “common”

Software seems to obey the statistical **Pareto Law**

90% of time is spent executing just 10% of the code

Therefore care in designing just this 10% of code is “cheap” and will give a good product 90% of the time (at least in theory!)
Note: we want to decouple specific users (e.g. John Smith) From their roles, e.g. student, teacher, systems administrator, etc.

A role will be called an actor.

A specific proper noun (object) e.g. Karl Meinke, may play several roles: teacher, researcher, sys_admin ... etc.
2.5.2. Central O-O Dogma

- **Common Nouns** e.g. *warehouse, truck*, correspond to classes and attributes
- **Proper Nouns** e.g. *Karl, Fido, KTH*, correspond to objects
- **Verbs** e.g. *unload*, correspond to methods
- **Relational Phrases** e.g. *responsible for*, correspond with system structure
Not every noun will be an actor ... 

... but every actor will be a noun, so:

\[
\text{Actors} \subseteq \text{Nouns}
\]
Step 2 requires us to look at each actor in turn and ask:

- What does this actor want to do (all verbs)?
- What does the actor need from the system to accomplish each activity (each verb)?

Let’s look at a logistics system with an actor “foreman” and try to identify goals ...
(a) A foreman has to be able to move items between warehouses with or without a customer order. This use case scenario is called “manual redistribution between warehouses”.

(b) A foreman also wants to check how far a customer order has been processed. We call this use case scenario “check status of a customer order”.
Let’s try to informally define the 1\textsuperscript{st} use case scenario: “manual redistribution between warehouses”.

This could be broken down into 4 steps

(1) \textbf{Initialisation}: when a \textbf{foreman} gives a request to do a redistribution.
(2) **Planning**: when the system plans how to co-ordinate the various transports, and issues transport requests.

(3) **Loading**: when a truck fetches the items from a source warehouse.

(4) **Unloading**: when a truck delivers items to a destination warehouse.
Is there anything wrong with this use case?

(1) It is very short!
   - but we can add detail later!

(2) It doesn’t say what happens if things go wrong.
   - We need to add exception/error cases.
   - The more of these we add, the more robust our system will be.
2.6. UML Sequence Diagrams

A use case scenario describes a sequence of interactions or messages between a collection of actors and the system in order to carry out some task.
Often we would like to formally model a use case scenario, to study:

- timing aspects (relative or absolute)
- communication patterns
- system states
- exception behavior
- alternative paths or behaviors
2.6.1. Basic Concepts

A basic sequence diagram depicts a set of **objects** or **processes** each of which has its own **timeline**.

Conventionally, **objects go across** the diagram horizontally, while **time goes down** the diagram vertically. Down the timelines, between objects, we show **messages**.
Here’s a sequence diagram with the basic elements:

**object/process**
- C: Computer
- P: Print_server
- D: Device

**message**
- `>lpr file`
- `print(file, device)`
- `print(file, size)`
- `done`

**timeline**
- `>done`
- `done`
Notice that:

• This basic SD has 3 objects and 6 messages
• 1 message comes from the environment:
  >lpr file (from the keyboard??)
• 1 message goes to the environment
  >done(time) (to the screen??)

• The environment is an implicit actor

• Each timeline has a terminating block at the bottom, which does not mean that the object dissappears/ gets deleted, life goes on!!
2.6.2. Events in Time

An SD must show dynamic behavior over time as possible (permissible) sequences of events. Down each timeline we see a set of local events.

A basic event may either be:

- **sending** of a message (arrow tail)
- **receiving** of a message (arrow head)
Each timeline shows the **local relative order** between events, but **not** the absolute time interval between events, **nor** the **global order**.

Thus the basic *model of time* is **relative**

We are interested in a **precise meaning** for UML diagrams

- Precise UML (aka pUML) [www.cs.york.ac.uk/puml](http://www.cs.york.ac.uk/puml)
Partial Ordering Rules

**Local Rules** allow us to infer the **global partial ordering** of events

**Rule 1.** For every message $m$: $\text{send}(m)$ occurs before $\text{receive}(m)$

**Rule 2.** Events are constrained globally by their local order of occurrence down each timeline
This means that, e.g.

```lisp
receive(\textgreater lpr file)
```
occurs before

```lisp
send( print(file, device) )
```
before

```lisp
receive(done)
```
before

```lisp
send(\textgreater done(time) )
```

and `send(done)` occurs before `receive(done)`
We don’t know the absolute time between sending and receiving a message, i.e. message delay. e.g these two diagrams express the same scenario.