UML - a Brief Look

UML grew out of great variety of ways
Design and develop object-oriented models and designs
By mid 1990s
Number of credible approaches reduced to three
Work further developed and refined
By 1997 version 1.1 of UML
Submitted and accepted by Object Management Group - OMG
   OMG body that defines standards in many areas of computer science
Current version is 2.1.2

UML and the Process

UML designed to be independent of any sw development process
Its designers use three views that work best in conjunction with UML
   • Use case driven
   • Architecture centric
   • Iterative and Incremental

UML views development of software system as series of cycles
   The series ends with release of version of system to customers
   These may be inside or outside of the company
   Captures and supports iterative nature of design process

Number of other life cycle models
  • Waterfall
  • V Cycle
  • Spiral Boehm
  • Rapid Prototype
  • Agile

Within the unified process
   Each cycle contains four phases
   • Inception
      Goal - establish viability of proposed system
Define scope of the system
Outline candidate architecture
Identify critical risks
  Determine when and how they will be addressed
Start to make case that project should be done

- Elaboration
  
  Goal - establish ability to build system given all constraints
  Capture majority of remaining functional requirements
  Expand candidate architecture into full architectural baseline
  Finalize business case for project
  Elaborate on plans for next phase

- Construction
  
  Goal - build a system capable of operating successfully in beta customer site
  Build system iteratively and incrementally
  Make certain visibility always evident in executable form

- Transition
  
  Goal - roll out fully functional system to customer
  Correct any defects
  Modify system to correct any previous unidentified problems

Within process
We identify five workflows that cut across all four phases
Each workflow is set of activities project people perform

- Requirements
  
  Establish requirements for system
  These are high-level functional requirements
  For system being modeled
  These give the what of the design not the how
  Allows people to agree on
  Capabilities of system
  Conditions to which it must conform

- Analysis
  
  Build analysis model
Used to refine and structure
Functional requirements captured earlier

- Design
  Build design model
  Describes the physical realization
  From requirements and analysis models

- Implementation
  Build implementation model
  Describes how requirements packaged into software components

- Test
  Build test model
  Describes how system integration and systems test
  Will exercise components from implementation model

**UML Diagrams**
UML uses diagrams and models
As a first step towards expressing
Static and dynamic relationships amongst objects

We’re not working with classes or objects per se
However UML still relevant to procedural programs
Typically found in embedded applications
We can still think in objects
Even if not using an object centered language

While an important part of the standard
Authors do not see such diagrams as the main thrust of the approach
Rather a philosophy of a *Model Driven Architecture (MDA)*
In which UML is used as a programming language is more common.

High level goal is to create an environment in which tool vendors
Can develop models that can work with a wide variety of other MDA tools

On the user side
Designers who work with UML range from
Those who are putting together a ‘back of the envelop’ sketch
To those who utilize it as a formal (high level) design and programming language, the current standard recognizes thirteen different classes of drawings.

As a design evolves, these different perspectives offer a rich set of tools. Whereby we can formulate and analyze potential solutions. Such tools enable one to model several different aspects of a design. It’s rare that all of the types are used in a single design.

UML diagrams and models reflect static and dynamic relationships. Amongst classes and class instances, static relationships will give us the architecture of our design. Dynamic relationships will give us behaviour of our system at runtime.

In many embedded applications, an application built as series of cooperating tasks or processes. These are directly analogous to classes and objects. We see in UML, our tasks have exchange of information – messages, activities, active or inactive times, persistence, actions.

Let’s look at several of the static and dynamic components and relationships. Static diagrams will be a portion of our work. Dynamic diagrams will be more relevant.

To start we will introduce and use the static aspects of UML models.
Use Case Diagrams

The first diagram that we’ll look at is the Use Case

Use cases widely employed

As a mechanism for capturing user requirements

In a form that can be used to drive the rest of the development process

Once agreed to by the customer

Use cases become basis for all further

• Analysis
• Design
• Construction
• Testing
• Deployment

of the software system

At each phase in the process

Results are validated against the requirements

Embodied in the use cases

Use case scenarios form the basis for the functional tests

That verify the software does what it is supposed to do

Use case

Gives outside view of the system

Describes the public interface for the module or system

Answers the questions

What is the behavior that the user sees?

What is the behavior the user expects?

Repeatedly poses the question

What? until the external view of the system has been satisfactorily captured

The use case diagram

Intended to present the main components of the system

How the user interacts with those components

Like many of the diagrams we’ll work with

Use case diagram can be hierarchical in nature

From top level drawing, one can expand each use case

Into sub use cases as necessary
Components

Diagram comprises three components

The **system**

The **actor(s)**

The **use case(s)**

**System**

Meaning of system is self evident

It’s expressed in the diagram as a box

We’ll often leave this off the diagram

**Actors**

An actor represents

“A coherent set of roles users of use cases play when interacting with these use cases.”

Booch 1999, pp. 221

Represent any one or any thing that might be using the system

- **Human**
- **Hardware device**
- **Another system**

Drawn as simple stick figures

Viewed as being outside of the system

**Use Cases – Graphical View**

Use cases represented as a solid oval

Identify the various behaviors of the system or ways it might be used

They encapsulate the events or actions

That must occur to implement the intended behavior of the system

Are stated or expressed from the point of view of the user

Accompanying each use case

Is a textual component fully describing it

Use case diagrams can be a very powerful tool

During the early stages of a project

When trying to identify, define, and capture the requirements for system

As we construct the diagram
We place the actor that executes the use case on the left hand side.
Supporting actors appear on the right hand side.
Not restricted to human users.
An actor can be a computer or other system as well.
Set of use cases appears in the center of the drawing.
With arrows indicating the actors involved in the use case.

A generic use case diagram given as

System comprises three use cases.
Actor 0 is using the system.
Appears on the left hand side.
Actor 1 is supporting UseCase 2.
Placed on the right hand side.

It’s important to remember to keep things simple.
When putting the use case diagram together.
If system being designed shows twenty five to fifty use cases.
On the top level drawing.
Time to rethink the design.

Use Cases – Textual View

Use case diagram.
Captures a graphical representation of the public interface.
To the module or system.
Useful to be able to visualize the relationships between use cases.
Show the static relationship between use cases.
Equally important to analyze what a use case means.
In terms of the functionality the system must deliver.
Associated with each use case is a textual description.
Called the use case specification.
Such a description can be decomposed into two pieces.
Normal activity of the use case
How exceptional conditions are to be handled

Use case specification describes
What actions the actor is to perform
How the system is expected to respond
Gives set of sequences of actions, including variants
System performs
That yields an observable result
Of value to an actor

When OO analyst-designers talk about the use cases related to a system
Mean the combination
UML use case diagrams
Use case specifications

Class Diagram – Objects and Tasks
Class diagram presents the various kinds of objects in the system
Permits capturing the relationships amongst them
Called associations.

Notation for a class is a rectangle
Simple version with just name
Often used during exploratory phases of modeling
When primary concern is
Structural relationships between classes
Rather than with their attributes and operations

Later when more detail needed
Rectangle subdivided into three areas
• Top area gives the name of the class or object
• Middle section identifies all of the properties of the object
  Will generally be declared inside the module implementation
  Thereby hidden from the casual user
• Third pane identifies the operations object is intended to perform
These establish the external behavior of the object
Provide the public interface to the object.

For us embedded systems typically implemented as collection of tasks
Task is collection of activities
    Having some purpose
Like classes - tasks have
    Names
    Attributes
    Operations

Object diagram or class diagram
    Reflects exactly the information we need
        To express a task - we have
            Function which implements the task
            Data which task utilizes
            Can then include task and intertask relationships

Intertask Relationships
    We can define number of different relationships
        Among tasks

    Such relationships can be
        Static
        Dynamic
        Both

Static relationships
    Will give us the architecture of our design
Dynamic relationships
    Will give us behavior of our system
        At runtime

Static Relationships
    We’ll start with static relationships
    Relationships
Containment

*Containment* conveys the idea

One object is made up of several others
Implements a whole – part relationship

Under UML we can express two different forms of containment
- Aggregation
- Composition.

Aggregation

*Aggregation* which expresses a *whole – part relationship*

In which one object or module
Contains another module

Key characteristic of an aggregation

One or more smaller functions are parts of whole
More complex function decomposed
Into number of smaller functions or modules

Owned module(s) may be *shared* with other modules
Outside of the aggregation
Linked list represents good example

Under such conditions
Rules must be established
To ensure proper management of the shared module

Diagram illustrates design in which

Graphics display implemented as
Aggregate of windows
Windows can exist
Outside of display

UML diagram for the aggregation relationship
Presents both the whole and its parts
Connected via a solid line
Originates at an open diamond on the end associated with the whole
Terminates on the end associated with the part

Composition
The composition relationship is similar to aggregation
Notion of ownership of the parts by the whole is much stronger
Elements of the composition
Cannot be part of another object
Exist outside of the whole object
Idea is loosely analogous to local variables in a function
Once one leaves the scope of the function
Local variables disappear

Consider a schedule
Made up of a number of intervals
Without the schedule
Intervals have no meaning
We express such a relationship as given
The schedule is composed of 1 to n intervals
Diagram is similar to that for the aggregation
The connecting line now originates in a solid rather than open diamond
We annotate the relationship as a 1 to n composition

Dynamic Relationships
Dynamic relationships
Provide information about behaviour of system
While performing intended task
Provide information about interaction among tasks
While performing task

As we discussed earlier important considerations include
- Active
  Using CPU

- Interaction
  Other tasks
  Outside world

- Concurrence
  Simultaneous activity

- Persistence
  Taking up memory

Interaction Diagrams

For our work
  Understanding and modeling
  Dynamic behaviour of our system is essential

Dynamic behaviour
  Gives us information about the lifetime of a task
  Tells us when that task is active
    When it’s using the CPU
  Models interactions amongst tasks
    Such interaction takes form of messages

We've seen message is communication
  Between two or more tasks
  Can take several forms
    Event
    Rendezvous
    Message – bad choice of words

Generally message results in
  One or more actions
    Such actions are executable functions within the task
      Result in change in values of one or more attributes

Begin with only two tasks
UML explicitly supports five kinds of actions

- **Call and Return**
  Call action invokes method on object
  Return returns value in response to call

- **Create and Destroy**
  Create action creates object
  Destroy does opposite

- **Send**
  Sends signal to object

These actions shown in following diagrams

The dashed line emanating from each object or class
Called lifeline

**Call and Return**

Express call action
Solid arrow from calling object to receiving object
Express return action
Dashed arrow from receiving object to calling object

**Create and Destroy**

Express create action
Solid arrow from creating object to created class instance
Express destroy action
Solid arrow from destroying object to destroyed class instance

Send
Express send action
Solid arrow with half arrow head
From sending task to receiving task
Sender does not expect response

Sequence Diagrams
Purpose of sequence diagram
Express time ordering of message exchange
Between objects
Build from components of interaction diagram

Have 4 key elements
- Objects
  Appear along top margin
  For our implementation these will be the tasks
- Lifeline
  Described earlier

- Focus of control
  Thin rectangular box
  Straddles task’s lifeline
  Indicates time during which object
  In control of flow
  Executing method or
  Creating another task

- Messages
  Show actions objects perform
  Each other
  Self

Diagram gives sequence diagram
Logging into system

Observe that the loginTask has spawned
_loginSubtask_

As the sequence proceeds
The _validateSubtask_ spawned and
Confirms login parameters

Fork and Join

When we work in multitasking system
Like most embedded applications
Common sequence
Parent process or task to start
Spawn several child tasks
These do the real work
Child tasks complete
Child tasks terminate
Parent class terminates

The process of splitting flow of control into two or more flows
Called fork
Each flow operates independently of the others

Synchronization of multiple flows into one
Called join
We model control flow behaviour of processes and tasks
Using Fork and Join diagram

Such diagram reflected as follows

Forks and joins represented by thick black rectangle
Called synchronization bar
Fork occurs after first activity or action completes
Following action
Task spawns subtasks
Suspends itself until subtasks complete
FTP application is good example

Once all subtasks have completed
Join occurs
Original task resumes its activities
Branch and Merge

Another form of flow of control is *branch*. The thread of execution is determined by value of some control variable. Such a structure permits one to model alternate threads of execution. A *merge* brings the flow back together again. Each is represented by the diamond symbol that is commonly found in the familiar flow chart.

Sequential flow

Shown by a solid arrow.

Individual tasks or activities

Shown using a rounded rectangle.

Simple diagram with two alternate paths of execution

For a portion of the overall task, given in adjacent diagram.

Following completion of activities in right hand path, flow of control merges back to a single path.

At each branch point, one can associate a guard condition to stipulate under what conditions the branch is to be taken.

The guard condition shown in square brackets on the transition arrow.

Activity Diagram

An *activity diagram* permits the capture of all of the procedural actions or flows of control within a task. Such actions may be:

- Branch and merge
• Fork and join
• Simple transition from state to state

The initial node in the diagram
Given by a solid black circle
The final node
Solid black circle surrounded by a second circle

Accompanying diagram shows how we might combine
Earlier activities into a larger task
Conversely, one can show how a larger task
Decomposed into its components

Events State Machines and State Chart Diagrams

Events
Any embedded application must interact
With world around it
System will accept inputs and produce outputs
Inputs generally result in some associated action
Actions may or may not lead to an output

Such inputs outputs and actions
Referred to under various names

Under UML umbrella
Inputs and outputs collected under name *events*

Event is any occurrence of interest to the system
Generally to one of the tasks in the system

UML supports 4 kinds of events
• Signal
  A synchronous exchange between tasks
• Call event
  Synchronous communication involving
    Sending message to another task
Sending a message to self

- Time event
  Event occurs after specified time interval elapsed
- Change event
  Event occurs after some condition satisfied

State Machines and State Chart Diagrams

We have studied and used state machines
Model and implement behaviour of system in time

We now apply those concepts to design and implementation of
Embedded applications

UML supports and extends
Traditional notion of state machines

A state is written as rectangle with rounded corners

Transitions between states
Reflect change in system from one state to another
Expressed as an arrow directed from
Source to destination

Transition occurs when
- Event of interest to system occurs
- System has completed some action
  Ready to move to next state
  Called triggerless transition
- We may have an action associated with the transition
- We may have a transition to self

We see all four types of transition in
Accompanying figure
Guard Conditions

A guard condition

Boolean expression that must evaluate to true
Before transition can fire

We show a guard condition in square brackets
Near transition arrow

- If guard condition associated with event
  \[ EventName \ [guardCondition] \]
  If condition evaluates to false
  Transition not taken

- If event, guard condition, action
  \[ EventName \ [guardCondition] / Action \]
  If condition evaluates to false
  Action not executed and transition not taken

- Guard condition by itself
  \[guardCondition\]
  Repeated self transition until met

State Machines and State Chart Diagrams

\textit{State machine} term that describes
- States an system can enter during life time
- Events to which system can respond
- Possible responses system can make to an event
- Transitions between possible states

\textit{State chart diagram}
Nothing more than the state diagram we've been using
There are some extensions / modifications under UML

In following diagram
Solid black circle
  Represents initial state
Solid circle with surrounding open circle
  Represents final state

We also make the following definitions

- **Entry action**
  Action system always performs
  Immediately on entering state

  Appears as *entry/actionName* within state symbol

- **Exit action**
  Action system always performs
  Immediately before leaving state

  Appears as *exit/actionName* within state symbol

- **Deferred event**
  Event that is of interest to system
  Handling deferred until system reaches another state

  Appears as *eventName/defer* within state symbol

  Such events get put into queue
  When system changes state
Composite States

States we've looked at so far
   Called simple states
UML extends notion of simple state to include
   Multiple nested states - called \textit{composite states}

These come in several varieties

\textit{Sequential States}

If the system exists in
   A composite state and
   Only one of the state's substates at a time
   Substates called \textit{sequential substates}
We can have transitions between such substates
   As we've seen for full states
Using sequential substates
   We can decompose behaviour of state into smaller pieces

\textit{History States}

When system makes transition into composite state
   Assumed that flow of control
   Starts in initial substate
However
   Can use \textit{history substate} to remember
   Last state system in before leaving composite state
We see such a state useful when modeling interrupt behaviour
Under interrupt we leave present state
   Return to same state following interrupt
Concurrent Substates

A system may be in a composite state
Also in more than one of the substates

Such is a situation in which we may have
Two or more sets of substates
Representing parallel flows of control

When system enters composite state with concurrent substates
Enters into initial state of both flows
We resynchronize by showing a final state for each flow

Have only touched on some of capabilities
UML diagrams
This will be sufficient for what we'll be doing
Vast amount of literature available
For those who are interested
Let's now try to put what we've learned
to work