Introduction to UML in Requirements Analysis

Ananda M. S. Amatya
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, U.K.
Email: Ananda.Amatya@dcs.warwick.ac.uk

October 24, 2000

Abstract

First UML is introduced from the point of view of Modelling in Object oriented Software Development. This is followed by the use of UML models for Requirement Analysis. Actors provide the external environment that communicate with the system by passing messages. Inside the system there are use cases that capture the functional requirements of the system. Main and supplementary use case diagrams provide all the models for Requirement Analysis. Communications between use cases are considered in order to capture all the scenarios. This involves use of extensions using stereotyped relationships (`<<include>>`, `<<extend>>`). Distinction is made between main and secondary actors, use cases and use case diagrams to deal with main behavioural features followed by alternative and error situations.
Key-words: Actors, Modelling, Requirement Analysis, Scenarios, System Boundary, UML, Use case, Use case diagrams, Stereotyped Relationships.

1 Topics covered

In this lecture we will give an *Introduction To UML* and its role in *Requirement Analysis*. Our objective is to explain *what* UML is and *how* Requirements Analysis is expressed in UML. Some *Recommended Books* are listed with brief descriptions at the end of the lecture. This is followed by end of lecture exercises.

2 Introduction To UML

We start with a definition of UML, and an explanation of the role of modelling in Object Oriented Software Development.

2.1 What is UML?

UML(Unified Modelling Language) is a modelling language for documenting specification, analysis, design and implementation of OOSD (Object-Oriented System Development) process. It is a *graphical* language that uses diagrams and texts with proper syntax and semantics in its models. UML has *rules and modelling elements* to model fully the whole software life-cycle. It is the *OMG* (Object Management Group) standard for describing *ALL object-oriented* software. The current standard is *UML 1.3*, but is undergoing much revision and updating in order to serve the needs of software in all application areas. Version 1.4, and subsequently version 2.0 are expected to follow soon. Also special versions to deal with specific applications such as real time systems have been released. Many software *tools* that support the use of UML in OOSD already exist or are being developed.

2.2 What is Modelling in OOSD?

In Software Engineering, modelling is an activity that is used to simplify a software system throughout its life-cycle. Its purpose is to focus on the relevant issues of the system for a given point of view by ignoring unnecessary details. Object oriented approach to software uses abstraction to simplify a complex problem. Different views correspond to the architecture of the system. Each view will correspond to a particular model of the system. For example, use case models represent use case views, static and dynamic models represent logical (analysis & design) views, process and thread models represent process views, component models represent implementation views, deployment models represent distribution of software on hardware nodes and communication between them giving deployment views.

3 UML in Requirements Analysis

UML uses **Actors**, **Use Cases** and **Use Case Diagrams** for modelling *Requirement Analysis* in Software Development. Such models are referred to as **Use Case Models**. In Use Case Models, **Actors** represent external users (environment) of the system, **Use Cases** represent the system operations (*functional requirements*). Use cases visualise, capture and describe all possible “snapshots” of the system behaviour characterised by all the different sequences of actions that are performed in response to external user requests. The emphasis is on **What** happens and NOT **How** it happens. A **Use Case Model** delimits the *system boundary* separating the system operations (**Use Cases**) from the external environment (**Actors**) specifying what is required and what is not. An **Actor** views a system as a black box. **Use Case Diagrams** specify *what* the system and its parts do in response to requests by its **actors** but *not how* the response is realised. **Use case Diagrams** are the **Use Case Models** that capture the communicative interaction between the system and its users. They form the basis for the iterative consultative process between system developers and clients for specifying the system. In Forward Engineering Use Cases model functional requirements of software being developed. In Reverse Engineering, they describe the system functionality of the existing software.

3.1 Actor

UML uses **stick-man** icon with a **name** label for an **actor** (**Figure 1**). An accompanying textual **documentation** *specifies* it and may be accessed by a *UML tool*. **Actors** are **classes** with instances (**objects**) that interact with the system. They represent users external to the system. They are **stereotyped** as `<<actor>>` in their class names, and their specific **roles** are to use one or more **use cases** in the system (**Figure 2**). **Actors** form the domain specific part of the environment of the system. They are *External* to the system, and do *NOT* form part of the system being built. An **Actor** may be a human *user* or a *software system* or a *hardware* seeking some service from the system. **Actors** capture the **roles** of specific uses of the system. **Actors** exchange information (**communicates**) with the system. They perform non-deterministic actions on the system and have **communication relationships** with the use cases in the system. The interface for this communication must be compatible with the interface of the communicated use case in order for the messages to be sent or received. **Primary actors** are the users of the main functionalities of the system. **Secondary actors** use the system to support the primary actors. They perform the secondary functionalities of the system. For example in an Insurance System, Insurance Brokers are the main actors as they are responsible for registering clients and arranging their insurances, whereas Accountants are secondary as they are responsible for managing the accounts which support the primary action of insuring clients. **Actors** initiate system operations. The textual description of the **actor** will show its boundary. The structure of an **actor** is not relevant, it interacts with a system as a single unit. To determine **actors** one needs to find the potential users of the system, i.e., who will benefit from the system and why is the system being built. Some active underlying systems may be **actors** (supporting system) while others like the underlying operating system are NOT **actors**. Identifying **actors** helps determine **use cases**. The same physical object may play different **roles** and become different **actors** that use different **use cases** (e.g., the same person may be the Customer **actor** who orders goods and Salesman **actor** who processes sales). An actor could be a subsystem interacting with another subsystem of the whole system. By interacting with different use cases an actor may use different traits of the same role (e.g., Customer orders goods, pays bill). Actors may be passive (only responding to messages received) or active (initiating actions). Identifying actors for a system helps in determining its use cases.

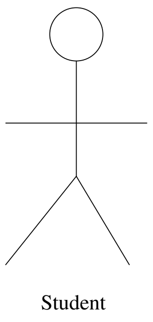

Figure 1: Actor Icon

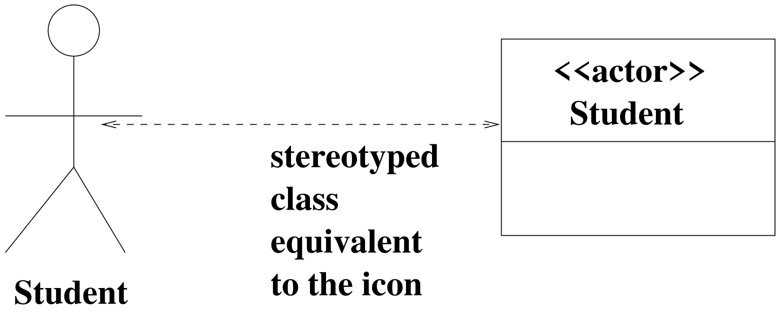

Figure 2: Stereotyped Class

3.1.1 Actor Interactions with a System:

An **actor** may *input information* to the system or *receive information* from the system or *both input and receive* information to and from the system. It has a **communication** (participating) association with one or more **use cases.** Association between actors is not relevant for the use case model (**Figure 3**), except when it represents a subsystem interacting with another subsystem of the whole system. However generalisation relationship between actors are relevant for the use case model.

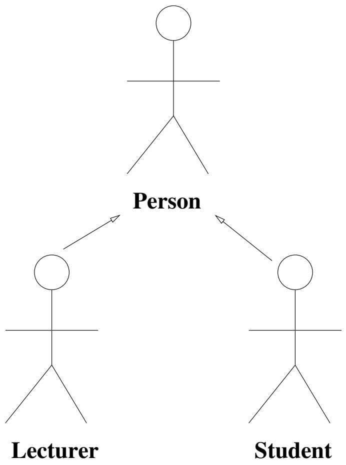

Figure 3: Actor Generalisation

3.1.2 Example: Actors in a Student Registration System

Some actors are: Student (a student interacts with the system *to register for a module*, Lecturer (a lecturer interacts with the system *to select course to teach*), Registrar (the registrar interacts with the system *to create the teaching schedule, to maintain module information*), Billing System (the Billing System interacts with the system *to obtain billing information*).

3.2 Use Case

UML uses an ellipse **icon** and a **name** label inside or below the ellipse icon for a use case (**Figure 4**). Sometimes instead of a **simple name** (e.g., placeOrder), a **path name** may be used (e.g., Order::placeOrder) showing the package (Order) to which the use case (placeOrder) belongs. Textual description is used for its **documentation**. A use case captures the **scenarios**, sequences of flows of events caused by the **actors'** interactions with the system. **Primary scenarios** represent normal flow of events. **Secondary scenarios** represent alternative flows and error situations. A use case is a **class** with **instance** (object) that manifests as a **scenario**, a particular action sequence. A use case encompasses *all possible scenarios* for the same functionality. Several alternative scenarios correspond to a **use case**. A use case captures a *system functionality* (user requirement). A *typical use* of a Computer System by an *external user* (actor) involving one or more sequences of actions in the system on behalf of the user. A system functionality may be available to one or more **actors**. Each **use case** should refer to *one complete system functionality*. Use cases may have generalisation/specialisation relationships to inherit behaviour. The **stereotypes**: `<<include>>` (to include factored out common behaviour) & `<<extend>>` (to extend into variants by adding extra behaviour) show dependency relations between use cases. They allow use cases to be combined to handle variation in functionalities. To extend a **use case**, one needs to determine the extension point (i.e., the point in the flow of events at which extra sequence of events are to be performed before continuing with the next event in the normal flow), the required variants (the sequence of flow to be added), and the options (condition on the extended use case attribute at the point of extension, if any, that must be satisfied for the extension to take place). If condition not satisfied original use case occurs. If two use cases extend at the same extension point the result is non-deterministic. To include a **use case**, determine the point in the flow of events of the original use case at which the flow of events of the included use case is to take place before continuing with the next event of the original use case. **use case** represents *what* the system does, *NOT how* it does it. To determine a **use case** find out which **actor** initiates it, what task is required by which **actor**, what information is needed by which **actor**, what information is to be received by which **actor**, what information is to be read by which **actor**, what are the main and alternative flows of events. Use-cases are used by developers and users for *validating system architecture* and for *verifying* system services. Use-cases are used in *integration testing*. They provide means to *communicate with domain experts* to **refine** the conceptual model of the system being developed. Any 2 use cases of the same system must be different (orthogonal partitioning of requirements).

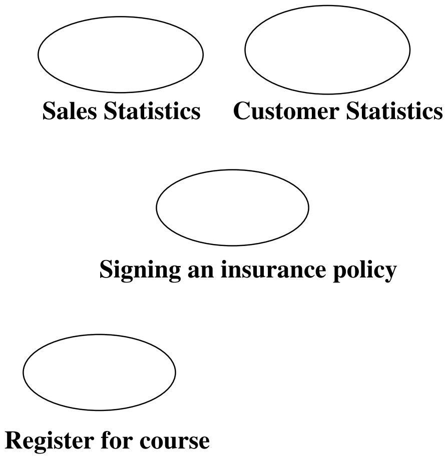

Figure 4: Use Case Icon

3.2.1 Example Use cases of a Student Registration System

A typical use case started by a student actor is *register for a module*. This use case should have the facility of creating, modifying, deleting and reviewing a student schedule.

3.2.2 Documentation Property of Use Case:

To fully specify a use case its documentation must contain: (a) *Purpose*: goal being achieved, functionality provided by it, (b) *User*: **actor** initiating it, (c) *Messages exchanged with* **actor**: events, updates, retrievals, decisions, (d) **Generalisations**, (e) **Extension stereotypes**: `<include>>` & `<<extend>>` (f) **Scenarios as use case instances** to complement **use case** description and *alternative scenarios (exceptions)*, (g) Precise but *informal* for communicating with user, (h) **Termination conditions**, results returned to **actors** (i) **Activity diagrams** to supplement the **use case**, if needed.

3.3 Use Case Relationships

(a) **Communication** - Participation of an **actor** in a **use case**; (b) **Generalisation** - Inheriting parent behaviour, adding and overriding with the child's own behaviour; (c) `<<include>>` **Relationship** - One **use case** including another **use case** to include the latter's behaviour; (d) `<<extend>>` **Relationship** - One **use case** is extended to another **use case** so that the latter has the behaviour of the former in addition to its own; extension can only be done at **extension points** of the **use case** being extended. Use cases may be abstract, meaning that they can only be used to extend other use cases, but cannot on their own perform any functionality of the system (cannot instantiate). Concrete (or real) use cases are those that can perform system functionalities (i.e., they can be instantiated) by an initiating actor.

3.3.1 $<< include >>$ Relationship

Behaviour *common* to many **use cases** is *factored off* into another **use case** *used* by the others. For example, Analyse Deal and Price Deal **use cases** both *use* Valuation **use case**; Signing Car

Insurance **use case** and Signing Life Insurance **use case** both *use* Signing Insurance Policy **use case.** <<include>> **use cases** must be *used* by the **use cases** that *use* them before the latter can be complete – a **Stereotyped** relationship. A dashed line, labelled <<include>>, with an open arrow pointing to the included **use case** is used to show this relationship (**Figure 5**). <<include>> is a dependency relation NOT a generalisation relation. The sequence of actions included are referenced, i.e., included use case performed before resuming the original use case. Generalisation would require these actions to be physically in the included use case.

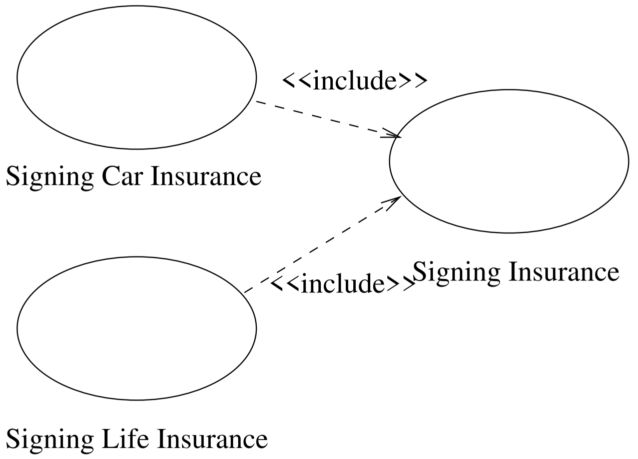

Figure 5: $<< include >>$ Relationship in Insurance

3.3.2 $<< extend >>$ Relationship

A **use case extends** another **use case** to do more than the latter. For example: Capture Deal **use case** (keeps to the specified limits) is **extended** by Exceed Limits **use case** (to include Capture Deal with limits exceeded); Signing Car Purchase Contract **use case** (normally not dealing with insurance) is **extended** by Signing Insurance Policy **use case** (to include the car sale with an insurance for it). <<extend>> *adds* more information to allow the already complete **use case** to **extend** its use – a **Stereotype** Relationship. A dashed line, labelled <<extend>>, with an open arrow pointing to the **use case** being extended is used to show this relationship (**Figure 6**).

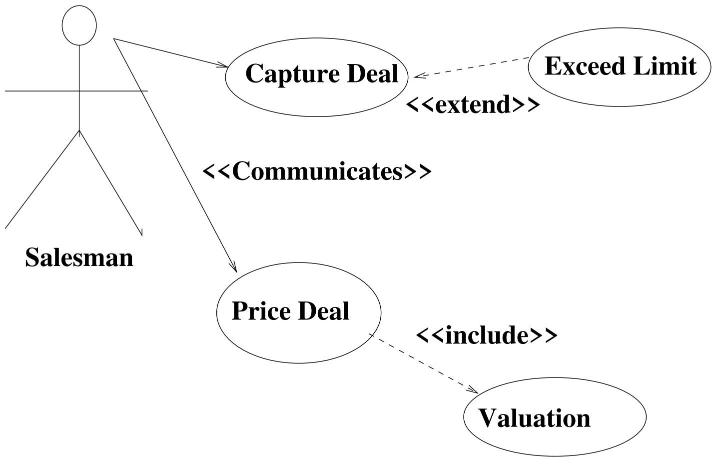

Figure 6: $<<\ extend >>$ Relationship in Car Sales

3.3.3 Extension Point

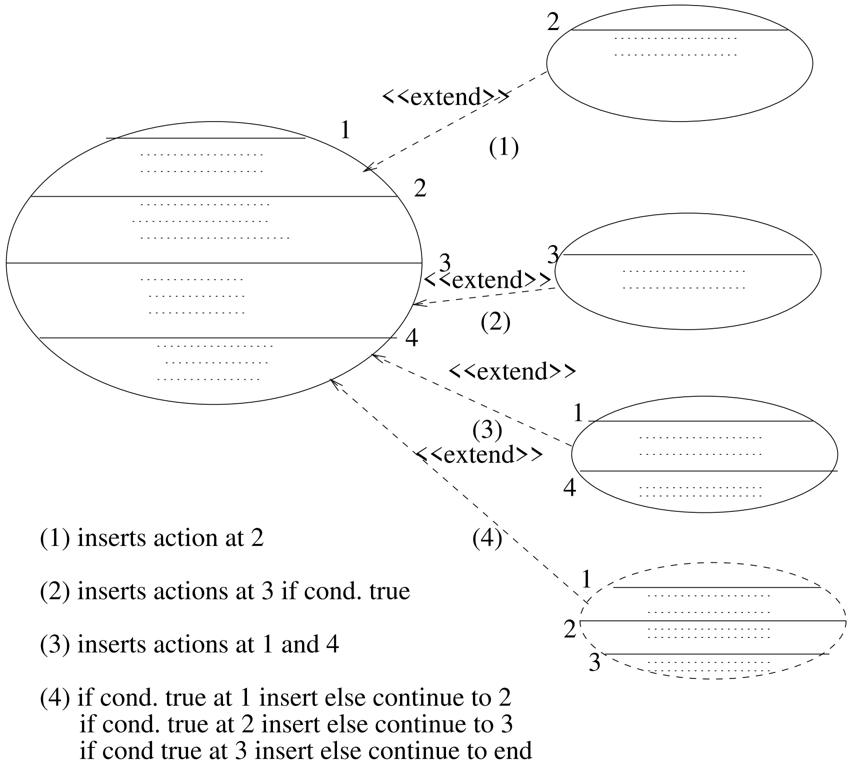

Figure 7: $<<\ extend >>$ Relationship in Car Sales

Conditions:

See **Figure 7**.

1. If no condition, always inserts the sequence of actions from the client use case to the base (server) use case at the extension point.

2. If condition present and satisfied by the attribute of the base (server) use case at the extension point the insert the sequence of actions of the client use case at the extension point of the base (server) use case.

3. If no condition, but more than one sequence of insertion actions in the client use case, they are inserted at the appropriate extension points of the base (server) use case.

4. If conditions present, test each condition on the attribute variable as the extension point on the base (server) use case is reached and if true insert the corresponding sequence of actions of the client use case, otherwise skip to the next condition and continue till all the insertions are handled.

Client use cases are not normally activated on their own but only together with the extended use case. Base (server) use case has extended behaviour only when client use case activates it. If directly activated by an actor base (server) use case has no extended behaviour. We note that the notation for the relationship between **use cases** has changed more than once. The current usage is dashed line with open arrow head. For `<<extend>>` the arrow head is at the **use case** being extended. For `<<include>>` the arrow head is at the included **use case**.

3.4 Use Case Diagrams

Actors are shown external to the box representing the current system. **Use cases** for the system are shown inside the box (**Figure 8**). **Use Case Diagrams** determine *system functionalities* as seen by **actors** communicating to **use cases**. **Actors** may have **Generalisation relationships**. **Use case** are related by **Extend** and **Include** relationship. **Documentation property** specify textual documentation of functionalities. **Sequence, collaboration and activity diagrams** associated with the **use case diagram** describe its *flow of events*. System update and modification will require changes in **use cases** and **actors**. **Use case models** package **use cases** and **actors** with relations among them. **Communication associations** between **actors** and **use cases** describe interaction between users and the system. **Use case diagrams** capture trace-ability through all the phases of the software development in order to (a) *validate* the *system architecture*, (b) *verify* the *system services*, and (c) *test* the *system parts* and its *integrity*. Customer approval and acceptance of the system tested by **use case** provided (validation) User interfaces for the **use cases** determined by simulation of prototypes. **Use case models** are (a) expressed in terms of domain objects, (b) structured by analysis model, (c) realised by design model, (d) implemented by implementation model, and (d) tested by Testing model. They **specify** the outcome of the communication between the system and external human or automated systems **actors** who are its *users, developers and domain experts*. They analyse all the **scenarios** (action sequences) in response to external user requests. They provides support for the documentation and marketing team.

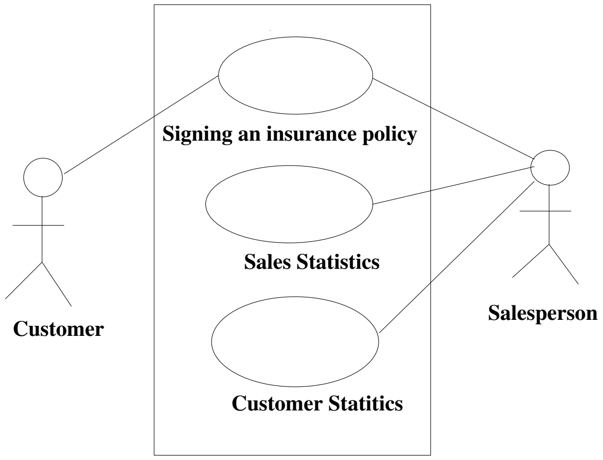

Figure 8: Use Case Diagrams

3.4.1 Communication Relationships:

Unidirectional or bidirectional communication association takes place between an **actor** and a **use case**. Unidirectional association is shown by an arrow head to the destination.

3.4.2 Event Flows in Use Case Diagrams:

Main use case diagrams show (a) System boundary, (b) **actors**, (c) functionality (**use cases**), (d) communication associations between **actors** and **use cases**, and (e) **use case** relationships (if any), e.g., generalisation, `<<include>>` and `<<extend>>` relationships.
Additional use case diagrams show (a) all the **use cases** for a selected **actor**, (b) all the relationships (to **actors** and **use cases**) for a **use case**, and (c) all the interactions for **actors** and **use cases** for a particular iteration of the development process. iteration,

3.4.3 Activity Diagrams for Use Case Event Flows

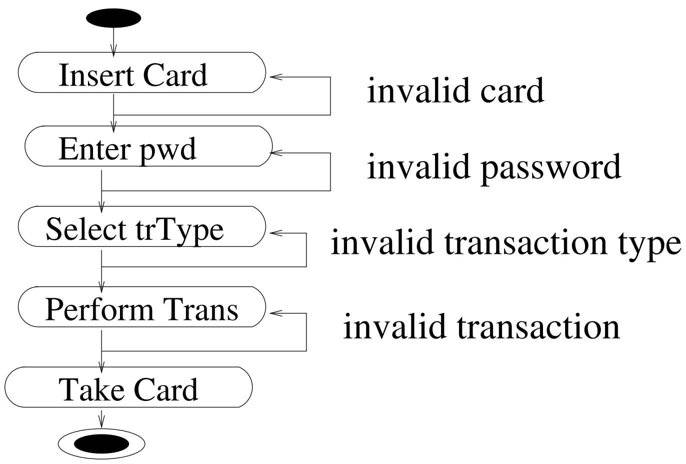

Figure 9: Activity Diagrams for Use Case Event Flows

Activity diagram shows work flow in the system as a flow of control from one activity to another including parallel activities (**Figure 9**). **Activity diagram** for a **use case** shows the main and alternative event flows in it. **Activity diagrams** may also be used to show work-flow through several **use cases.** **Activity diagram** icons consist of a *capsule* for an **activity**, a *directed line* for a **transition**, a *diamond* for a **decision**, q *synchronisation bar* for **concurrent transitions**, a *circular blob* for a **start** and a *circular blob in a circle* for a **stop.**

3.4.4 Testing Use Cases

Validation is done by making sure that a **use case** matches a user requirement, i.e., **use cases** are correctly specified. *Walkthroughs* with user involve ensuring that all the required **use cases** are there. *Verification* involves making sure that the system built has the functionality of the **use case**, i.e., **use cases** are correctly implemented. Every **use case** is implemented to produce the functionality specified by it.

3.4.5 Use Cases, Collaborations & Scenarios

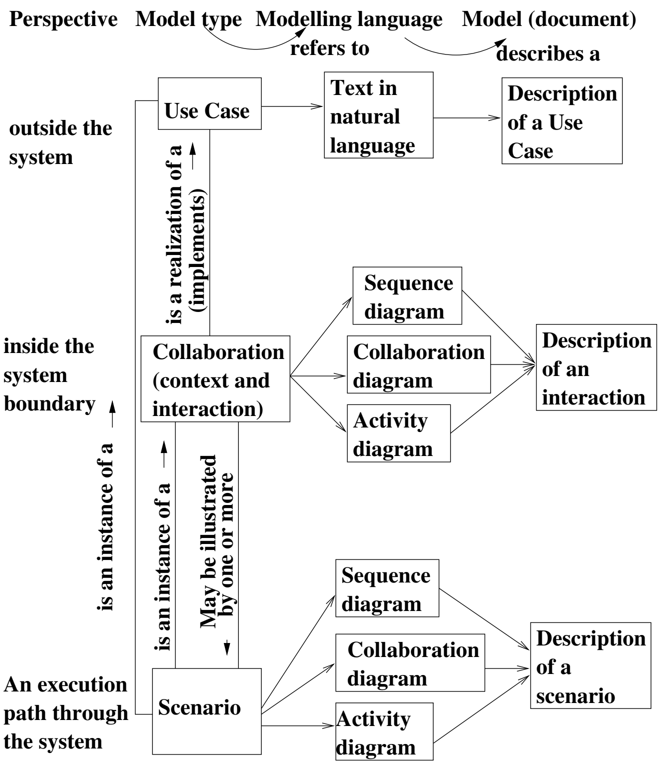

Figure 10: Use Case, Collaboration, Scenario

A **scenario** is an **Instance of a use case**, giving one path in the *flow of events* for the **use case.** **Scenarios** help to **determine objects, classes, and object interactions** for a functionality specified by a **use case.** **Primary scenarios** give the *normal flow*. **Secondary scenarios** give alternative *what if cases*. The two types of **interaction diagrams for scenarios**, **Sequence (time order)** and **collaboration (object linkage order)** are semantically equivalent (**Figure 10**).

4 Recommended Books

1. **Booch**, Grady and Rumbaugh, James and Jacobson, Ivar, *The Unified Modelling Language User Guide*, Addition-Wesley, 1998: the most authoritative and follows the **1.3 standard**; conceptually quite hard at places; some examples rather sketchy, and language bindings (Java or C++) touched upon only briefly. ($\sim$ $\pounds 40.00$).

2. **Eriksson**, H. and Penker, M., *UML Toolkit*, John Wiley, 1998: about **UML 1.1**; well written; comes with a demonstration version of **Rational Rose** (version 4) and has a complete case study. ($\sim$ $\pounds 35.00$).

3. **Fowler**, Martin and Scott, Kendall, *UML Distilled: Applying the Standard Object Modelling Language*, Addison-Wesley, 1998: uses **UML 1.1**; rather **slim** with good examples. The technical content is excellent, though quite hard; recently updated ($\sim$ $\pounds 25.00$).

4. **Pooley**, Rob and Stevens, Perdita, *Using UML*, Addison-Wesley, 1999: a **slim** volume containing much advanced material (components, patterns, frameworks); recently updated. ($\sim$ $\pounds 28.00$).

5. **Priestley**, Mark, *Practical Object-Oriented Design with UML*, McGraw-Hill, 2000: well-written; introduces **UML 1.1** notation and Object-Oriented software development; contains a complete case study (including working Java 1.1 code.) ($\sim$ $\pounds 24.00$).