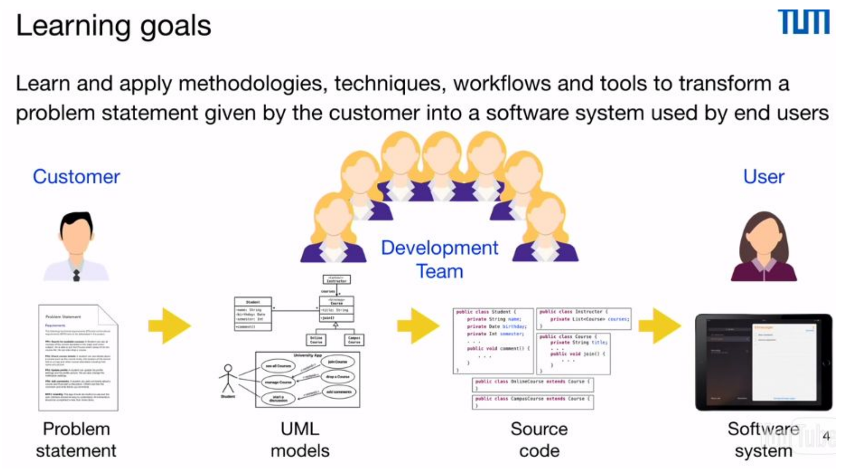

Learning goals

Learn and apply methodologies, techniques, workflows and tools to transform a problem statement given by the customer into a software system used by end users

1. Use Case Definition

1.1. **Fuente.** Object-Oriented Software Engineer (Copia Textual - Literature Review).

1.1.1. **2.3.1 Systems, Models, and Views**

1.1.2. **2.4.1 Use Case Diagrams. Use cases and actors**

Definition of Systems, Models and Views

A system is an organized set of communicating parts. We focus here on engineered systems, which are designed for a specific purpose, as opposed to natural systems, such as a planetary system, whose ultimate purpose we may not know. A car, composed of four wheels, a chassis, a body, and an engine, is designed to transport people. A watch, composed of a battery, a circuit, wheels, and hands, is designed to measure time. A payroll system, composed of a mainframe computer, printers, disks, software, and the payroll staff, is designed to issue salary checks for employees of a company. Parts of a system can in turn be considered as simpler systems called subsystems. The engine of a car, composed of cylinders, pistons, an injection module, and many other parts, is a subsystem of the car. Similarly, the integrated circuit of a watch and the mainframe computer of the payroll system are subsystems. ***This subsystem decomposition can be recursively applied to subsystems. Objects represent the end of this recursion, when each piece is simple enough that we can fully comprehend it without further decomposition.***

Many systems are made of numerous subsystems interconnected in complicated ways, often so complex that no single developer can manage its entirety. ***Modeling is a means for dealing with this complexity.*** Complex systems are generally described by more than one model, each focusing on a different aspect or level of accuracy. Modeling means constructing an abstraction of a system that focuses on interesting aspects and ignores irrelevant details... Modeling allows us to deal with complexity through a divide-and-conquer approach: For each type of problem we want to solve (e.g., testing aerodynamic properties, training pilots), ***we build a model that only focuses on the issues relevant to the problem***... Modeling also helps us deal with complexity by enabling us to incrementally refine simple models into more detailed ones that are closer to reality. In software engineering, as in all engineering disciplines, the model usually precedes the system. During analysis, we first build a model of the environment and of the common functionality that the system must provide, at a level that is understandable by the client. Then we refine this model, adding more details about the forms that the system should display, the layout of the user interface, and the response of the system to exceptional cases. The set of all models built during development is called the system model. If we did not use models, but instead started coding the system right away, we would have to specify all the details of the user interface before the client could provide us with feedback. Thus we would lose much time and resources when the client then introduces changes.

Use Case Function

Use cases are used during requirements elicitation and analysis to represent the functionality of the system**.** Use cases focus on the behavior of the system from an external point of view. A use case describes a function provided by the system that yields a visible result for an actor. An actor describes any entity that interacts with the system (e.g., a user, another system, the system's physical environment). The identification of actors and use cases results in the definition of the boundary of the system, that is, in differentiating the tasks accomplished by the system and the tasks accomplished by its environment. The actors are outside the boundary of the system, whereas the use cases are inside the boundary of the system.

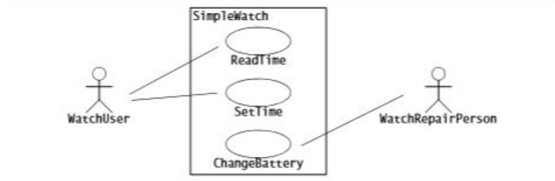


Figure 2-1 A UML use case diagram describing the functionality of a simple watch. The WatchUser actor may either consult the time on her watch (with the ReadTime use case) or set the time (with the SetTime use case). However, only the WatchRepairPerson actor can change the battery of the watch (with the ChangeBattery use case). Actors are represented with stick figures, use cases with ovals, and the boundary of the system with a box enclosing the use cases.

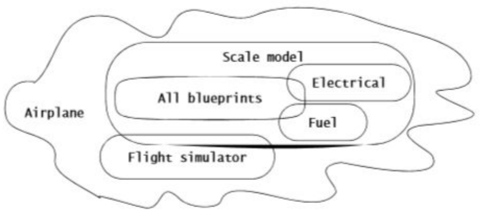


Figure 2-6 A model is an abstraction describing a subset of a system. A view depicts selected aspects of a model. Views and models of a single system may overlap each other.

Use cases and actors

Actors are external entities that interact with the system. Examples of actors include a user role (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a central database, a fabrication line). Actors have unique names and descriptions. Use cases describe the behavior of the system as seen from an actor's point of view. Behavior described by use cases is also called external behavior. A use case describes a function provided by the system as a set of events that yields a visible result for the actors. Actors initiate a use case to access system functionality. The use case can then initiate other use cases and gather more information from the actors. When actors and use cases exchange information, they are said to communicate. We will see later that we represent these exchanges with communication relationships.

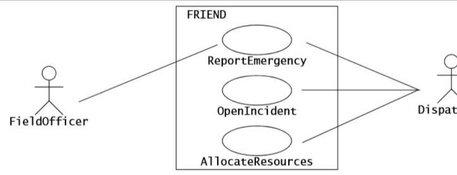
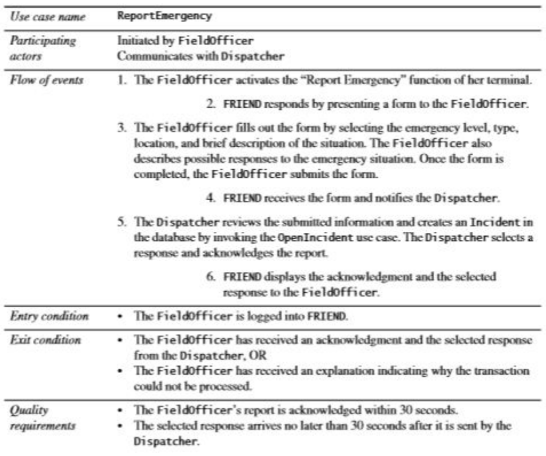


Figure 2-13 An example of a UML use case diagram for First Responder Interactive Emergency Navigational Database (FRIEND), an accident management system. Associations between actors and use cases denote information flows. These associations are bidirectional: they can represent the actor initiating a use case (FieldOfficer initiates ReportEmergency) or a use case providing information to an actor (ReportEmergency notifies Dispatcher). The box around the use cases represents the system boundary.



Use case name	ReportEmergency
Participating actors	Initiated by FieldOfficer Communicates with Dispatcher
Flow of events	1. The FieldOfficer activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the FieldOfficer. 3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form. 4. FRIEND receives the form and notifies the Dispatcher. 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report. 6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
Entry condition	• The FieldOfficer is logged into FRIEND.
Exit condition	• The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR • The FieldOfficer has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The FieldOfficer's report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

For this example, in an **Accident Management System**, field officers (such as a police officer or a firefighter) have access to a wireless computer that enables them to interact with a dispatcher. The dispatcher in turn can visualize the current status of all its resources, such as police cars or trucks, on a computer screen and dispatch a resource by issuing commands from a workstation. In this example, field officer and dispatcher can be modeled as actors. Figure 2-13 depicts the actor FieldOfficer who invokes the use case ReportEmergency to notify the actor Dispatcher of a new emergency. As a response, the Dispatcher invokes the OpenIncident use case to create an incident report and initiate the incident handling. The Dispatcher enters preliminary information from the FieldOfficer in the incident database (FRIEND) and orders additional units to the scene with the AllocateResources use case. For **the textual description of a use case**, we use a template composed of six fields (see Figure 2-14) adapted from [Constantine & Lockwood, 2001]:

• The name of the use case is unique across the system so that developers can unambiguously refer to the use case.
• Participating actors are actors interacting with the use case.
• Entry conditions describe the conditions that need to be satisfied before the use case is initiated.
We organize the steps in the **flow of events** in two columns, the left column representing steps accomplished by the actor, the right column representing steps accomplished by the system. Each pair of actor–system steps represents an interaction.

Use cases are written in natural language. This enables developers to use them for communicating with the client and the users, who generally do not have an extensive knowledge of software engineering notations. The use of natural language also enables participants from other disciplines to understand the requirements of the system. The use of the natural language allows developers to capture things, in particular special requirements, that cannot easily be captured in diagrams.

Use case diagrams can include four types of relationships: communication, inclusion, extension, and inheritance. We describe these relationships in detail next. **Communication relationships:** Actors and use cases communicate when information is exchanged between them. Communication relationships are depicted by a solid line between the actor and use case symbol... Communication relationships between actors and use cases can be used to denote access to functionality. In the case of our example, a FieldOfficer and a Dispatcher are provided with different interfaces to the system and have access to different functionality. **Include relationships:** When describing a complex system, its use case model can become quite complex and can contain redundancy. We reduce the complexity of the model by identifying commonalities in different use cases. For example, assume that the Dispatcher can press at any time a key to access a street map. This can be modeled by a use case ViewMap that is included by the use cases OpenIncident and AllocateResources. **Extend relationships:** are an alternate means for reducing complexity in the use case model. A use case can extend another use case by adding events. An **inheritance relationship** is a third mechanism for reducing the complexity of a model. One use case can specialize another more general one by adding more detail.

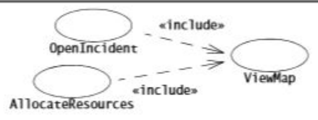


Figure 2-15 An example of an «include» relationship (UML use case diagram).

Use case name	AllocateResources
Participating actor	Initiated by Dispatcher
Flow of events	. . .
Entry condition	• The Dispatcher opens an Incident.
Exit condition	• Additional Resources are assigned to the Incident. • Resources receives notice about their new assignment. • FieldOfficer in charge of the Incident receives notice about the new Resources.
Quality requirements	At any point during the flow of events, this use case can **include** the ViewMap use case. The ViewMap use case is initiated when the Dispatcher invokes the map function. When invoked within this use case, the system scrolls the map so that location of the current Incident is visible to the Dispatcher.

Figure 2-16 Textual representation of include relationships of Figure 2-15. "Include" in **bold** for clarity.

We represent include relationships in the textual description of the use case with one of two ways. If the included use case can be included at any point in the flow of events (e.g., the ViewMap use case), we indicate the inclusion in the *Quality requirements* section of the use case (Figure 2-16). If the included use case is invoked during an event, we indicate the inclusion in the flow of events.

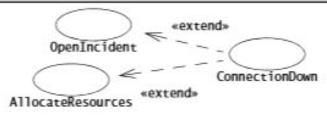


Figure 2-17 An example of an «extend» relationship (UML use case diagram).

Use case name	ConnectionDown
Participating actor	Communicates with FieldOfficer and Dispatcher.
Flow of events	. . .
Entry condition	This use case **extends** the OpenIncident and the AllocateResources use cases. It is initiated by the system whenever the network connection between the FieldOfficer and Dispatcher is lost.
Exit condition	. . .

Figure 2-18 Textual representation of extend relationships of Figure 2-17. "Extends" in **bold** for clarity.

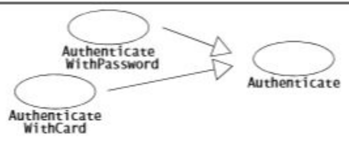
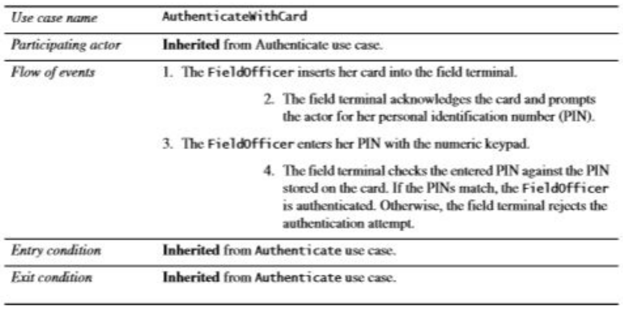


Figure 2-19 An example of an inheritance relationship (UML use case diagram). The `Authenticate` use case is a high-level use case describing, in general terms, the process of authentication. `AuthenticateWithPassword` and `AuthenticateWithCard` are two specializations of `Authenticate`.



Use case name	`AuthenticateWithCard`
Participating actor	**Inherited** from Authenticate use case.
Flow of events	1. The `FieldOfficer` inserts her card into the field terminal. 2. The field terminal acknowledges the card and prompts the actor for her personal identification number (PIN). 3. The `FieldOfficer` enters her PIN with the numeric keypad. 4. The field terminal checks the entered PIN against the PIN stored on the card. If the PINs match, the `FieldOfficer` is authenticated. Otherwise, the field terminal rejects the authentication attempt.
Entry condition	**Inherited** from `Authenticate` use case.
Exit condition	**Inherited** from `Authenticate` use case.

Figure 2-20 Textual representation of inheritance relationships of Figure 2-19.

Los libros de donde extraje imágenes, ecuaciones e información, que recomiendo para iniciar este estudio, son los siguientes:

Object Oriented Software Engineering, Bernd Bruegge & Allen H. Dutoit

Larry Francis Obando – TSU

Escuela de Ingeniería Eléctrica de la Universidad Central de Venezuela, Caracas.

Escuela de Ingeniería Electrónica de la Universidad Simón Bolívar, Valle de Sartenejas.

Escuela de Turismo de la Universidad Simón Bolívar, Núcleo Litoral.

Contact:

WhatsApp: 00593984950376

email: dademuchconnection@gmail.com

Copywriting, Content Marketing**.**