Introduction for using UML

Mikael Åkerholm, Ivica Crnković, Goran Mustapić

Abstract

The purpose with this document is to provide a brief introduction to the Unified Modeling Language (UML) for programmers and software architects. The focus is on using UML in general, not in combination with a particular tool, problem domain or programming technique.

UML has intentionally been developed as a langue for modeling object-oriented systems. Its use has however widely been spread out. Today UML is used for system specifications. In different domains UML (for example in distribution of electrical power) is used for specification and standardization of different systems or parts of the systems. This standardization makes it possible that different vendors produce products that comply with the standard specification. From UML it is possible to automatically generate different types of descriptions (for example specifications in XML), or even automatic creation of software code. UML is becoming a standard tool for software and system engineers.

TABLE OF CONTENT

1. INTRODUCTION

UML is a result of the evolution of object-oriented modeling languages. It was developed by Rational Software Company by unifying some of the leading object-oriented modeling methods,

- Booch by Grady Booch,
- OMT (Object Modeling Technique), by Jim Raumbaugh and
- OOSE (Object-Oriented Software Engineering), by Ivar Jacobson.

The authors of these languages are sometimes called the three amigos of software engineering. They were participating in the around twenty people strong group which was formed in '94 and submitted UML 1.0 to the Object Management Group (OMG) in '97. The current version of UML is 1.4 (published in Sep 2001) and there is ongoing work within OMG on a new major version 2.0, planned to be released during late 2003 or early 2004.

UML is used for modeling software systems; such modeling includes analysis and design. By an analysis the system is first described by a set of requirements, and then by identification of system parts on a high level. The design phase is tightly connected to the analysis phase. It starts from the identified system parts and continues with detailed specification of these parts and their interaction. For the early phases of software projects UML provide support for identifying and specifying requirements as use cases. Class diagrams or component diagrams can be used for identification of system parts on a high level. During the design phase class diagrams, interaction diagrams, component diagrams and state chart diagrams can be used for comprehensive descriptions of the different parts in the system.

We will start by giving the basic building blocks of UML, and then introduce the most essential UML diagrams one by one, each with a small example. Finally an example with a couple of diagrams is given to illustrate how the diagrams can be combined to describe the design of a small software system.

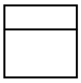
2. BASIC BUILDING BLOCKS OF UML

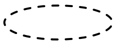
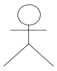
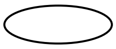
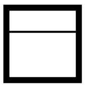
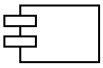
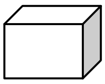
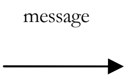
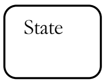
The basic building blocks in UML are things and relationships; these are combined in different ways following different rules to create different types of diagrams. In UML there are nine types of diagrams, below is a list and brief description of them. The more in depth descriptions in the document, will focus on the first five diagrams in the list, which can be seen as the most general, sometimes also referred to as the UML core diagrams.

1. Use case diagrams; shows a set of use cases, and how actors can use them
2. Class diagrams; describes the structure of the system, divided in classes with different connections and relationships
3. Sequence diagrams; shows the interaction between a set of objects, through the messages that may be dispatched between them
4. State chart diagrams; state machines, consisting of states, transitions, events and activities
5. Activity diagrams; shows the flow through a program from an defined start point to an end point
6. Object diagrams; a set of objects and their relationships, this is a snapshot of instances of the things found in the class diagrams
7. Collaboration diagrams; collaboration diagram emphasize structural ordering of objects that send and receive messages.
8. Component diagrams; shows organizations and dependencies among a set of components. These diagrams address static implementation view of the system.
9. Deployment diagrams; show the configuration of run-time processing nodes and components that live on them.

2.1 Things

Things are used to describe different parts of a system; existing types of things in UML are presented in table 1.

Types of Things			
Name	**Symbol**	**Description**	**Variations/other related elements**
Class		Description of a set of objects that share the same: attributes, operations, relationships and semantics.	- actors - signals - utilities

Interface		A collection of operations that specify a service of a class or component.	
Collaboration		An interaction and a society or roles and other elements that work together to provide some cooperative behavior that is bigger than the sum of all the elements. Represent implementation of patterns that make up the system.	
Actor		The outside entity that communicates with a system, typically a person playing a role or an external device	
Use Case		A description of set of sequence of actions that a system perform that produces an observable result of value to a particular actor. Used to structure behavioral things in the model.	
Active class		A class whose objects own a process or execution thread and therefore can initiate a control activity on their own.	- processes - threads
Component		A component is a physical and replacable part that conforms to and provides the realisation of a set of interfaces.	
Node		A physical resource that exists in run time and represents a computational resource.	
Interaction		Set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.	- messages - action sequences - links
State machine		A behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.	- states - transitions - events - activities

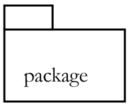
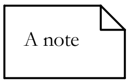
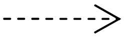
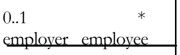
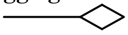
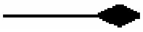
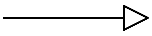
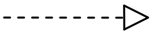
Packages		General purpose mechanism of organizing elements into groups.	- frameworks - models - subsystems
Note		A symbol for rendering notes and constraints attached to an element or a collection of elements.	

Table 1, UML things

2.2 Relationships

The types of UML relationships are shown in the table 2, relationships are used to connect things into well defined models (UML diagrams).

Types of *Relationships*			
Name	**Symbol**	**Description**	**Specialization**
Dependency		A semantic relationship between two things in which a change to one thing may affect the semantics of the dependent thing.	
Association		Structural relationship that describes a set of links, where a link is a connection between objects. Aggregation and composition are "has-a" relationship. Aggregation (white diamond) is an association indicating that one object is temporarily subordinate or the other, while the composition (black diamond) indicates that an object is a subordinate of another through its lifetime.	**Aggregation**  **Composition** 
Generalization		Specialization/generalization relationship in which objects of the specialized element are substitutable for objects of the generalized element.	
Realization		Semantic relationship between two classifiers, where one or them specifies a contract and the other guaranties to carry out the contract. They are used between:	

		- interfaces and classes or components - use cases and collaborations that realize them	

Table 2, UML relations

3 USE CASE DIAGRAMS

Use case diagrams are done in an early phase of a software development project. They express how it should be possible to use the final system. It is important to focus on specifying how an external user interacts with the system, not trying to specify how the system shall solve the tasks. The granularity of a use case is typically larger than a single operation, but smaller than systems. Use cases are a good way to express the functional requirements of a software system, they are intuitive and easy to understand so they can be used in negotiations with non-programmers.

The participants in a UML use case diagram are use cases, one or several actors and relations, associations and generalizations between them. Some small examples are given in figures 1 to 3. Figure 1 shows how a cash dispenser system can be used, an actor customer are associated to the use cases Withdraw Money and Get Account Balance. In general use case diagrams should be as simple as in figure 1.

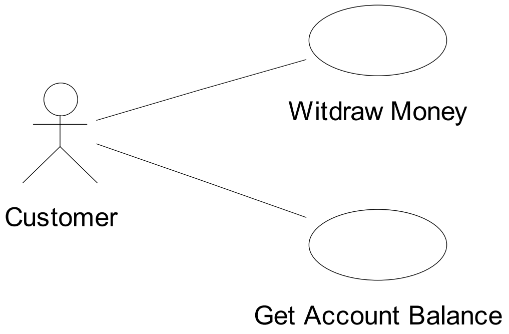


Figure 1, use cases of a cash dispenser system

In figure 2, some use cases of a web based fruit shop is shown, in this example the generalization symbols are used. There are generalizations between the actors Shop Assistant and User and Customer and User, meaning that both a Shop Assistant and a Customer is a user and shall both be able to browse fruits. But it is only a Shop Assistant that can use the system through the use case Add Fruits, likewise it is only a Customer that is associated with the use case Buy Fruits.

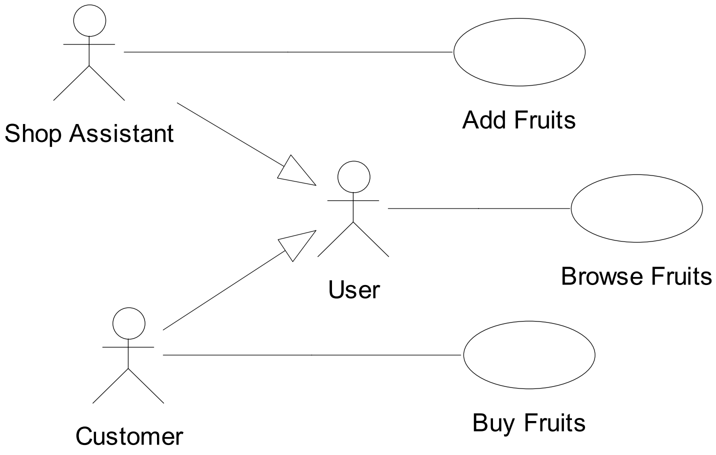


Figure 2, Use case diagram for a web shop

Figure 3, illustrates that use cases can include each other, in this example it shows that using a web surfing station includes a login. The user which is the actor can use the system to access the internet through the Access Web use case, but that includes the Login use case.

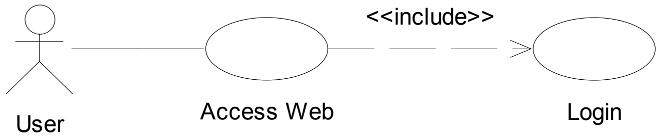


Figure 3, Use case for a web surfing station

4 CLASS DIAGRAMS

Class diagrams can profitably be used both in the early phases of a project and during detailed design activities. This is possible mainly because of the ability to draw class diagrams as conceptual for high level activities, as well as detailed later in the project. Class diagrams express the static structures of a system divided into different parts called classes and also which relations the classes have to each others.

A class diagram consists of the parts classes, associations and generalizations, and can exist in several different levels. Below is an identification of three different useful levels, starting with the least detailed.

- Conceptual class diagrams (conceptual model), represent concepts of the problem domain
- High level class diagrams (type model), describe static views of a solution to a problem, through a precise model of the information that is relevant for the software system
- Detailed class diagrams (class model), include data types, operations and possibly advanced relations between classes

The following example illustrates a design of the software system in a vending machine for soft drinks, during different phases in the development. Figures 4, is a conceptual class diagram done early in the project, while figure 5 is a high level design, and finally figure 6 is the detailed design.

In figure 4, the conceptual class diagram of the problem domain of vending machines is showed. The diagram shows that the problem domain is concerned with Coins, Vending Machines, Soda Cans, and Customers; it also shows how they are related to each others through un-directed associations. For instance, it shows that the Vending Machine is associated to all other conceptual classes in the diagram, whereas Coins are only associated to the Vending Machine class and the Customer class.

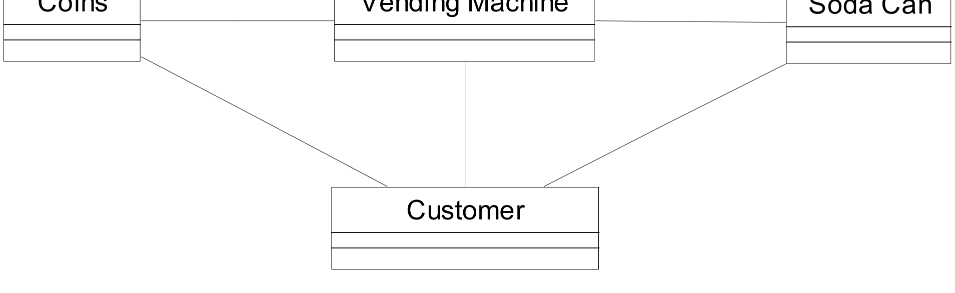


Figure 4, conceptual class diagram of a vending machine

Figure 5, shows the high level class diagram for the same vending machine software project. Some names of the classes in figure 4 and 5 may be the same, but they do not represent the same thing; in the former conceptual case the classes show different physical concepts of the problem domain and how they relate, this diagram could have been specified by a non-programmer. The high level class diagram in figure 5 shows precisely what information the software system must handle and represents a solution to the problem, in form of attributes in some of the classes. The solution add two classes, one class Coin Handler for handling the current amount inserted and dealing with the coins, and another class Stock for handling the soda cans.

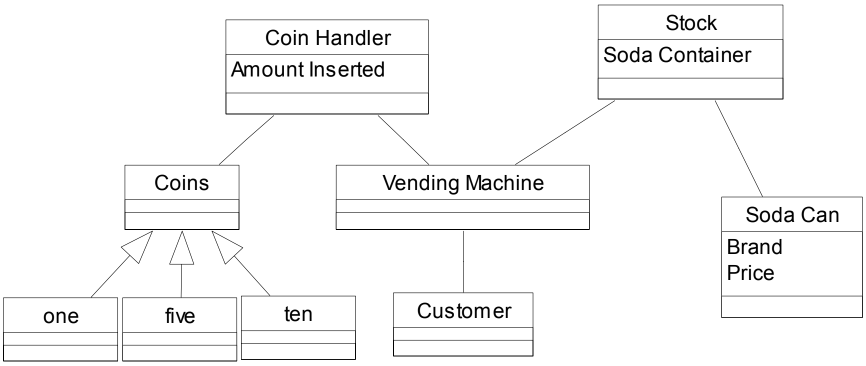


Figure 5, high level class diagram

The customer is no longer related to the Coins class or the Soda Can class, this is a point where there is an obvious difference between the physical relations in the conceptual diagram (figure 4). The customer class could for instance have been related to the Coins class, but they are not related in this particular problem solution. Furthermore, the Coins class has three specializations in the solution, representing coins with different values.

Figure 6 is a detailed class diagram showing the concrete data type of attributes, and operations provided by the different classes for the same example. A detailed class diagram also refines the relations between different classes, often through aggregations or compositions with multiplicity defined. The difference between aggregation and composition, (filled or not filled diamond) is a little bit vague so a suggestion is to make practice to use one of them consequently.

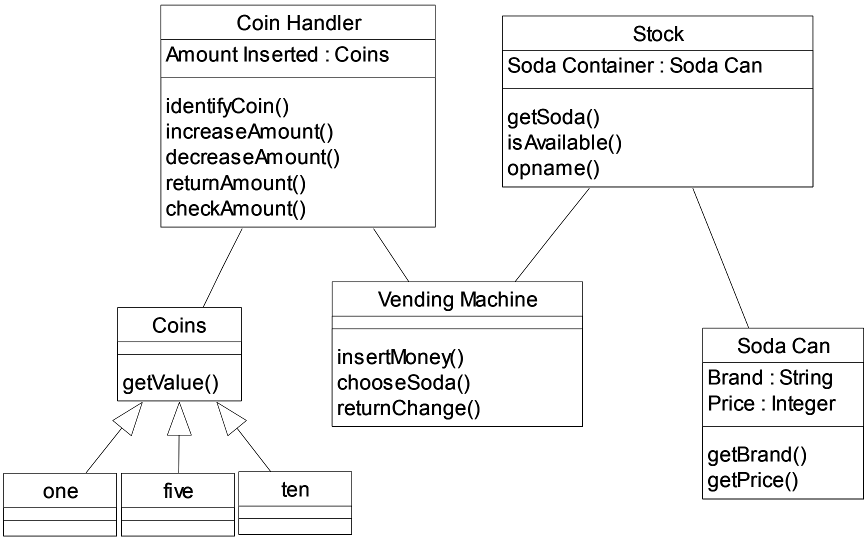


Figure 6, a detailed class diagram of the vending machine

Starting from the Customer class, we can see that it is related to the Vending Machine class, which provide three operations. The Vending Machine class also has one instance of the Stock class, and one of the Coin Handler class. If we focus on the aggregation diamond between the Coin Handler class and the Coins class, it gives the information that one Coin Handler can have zero to many Coins. The relations between the Stock and the Soda Can classes are refined in a similar way, i.e., with aggregation and multiplicity.

5 SEQUENCE DIAGRAMS

UML sequence diagrams are used to model the flow of control between objects. It can be hard to understand the overall flow in a complex system without modeling it. Sequence diagrams model the interactions through messages between objects; it is common to focus the model on scenarios specified by use-cases. It is also often useful input to the detailed class diagram to try to model the specified use cases with sequence diagrams, necessary forgotten operations and relations are usually found.

The diagrams consists of interacting objects and actors, with messages in-between them. Figure 7, shows an example of the use case when a customer successfully buys a soft drink from the vending machine modeled by class diagrams in the class diagram chapter.

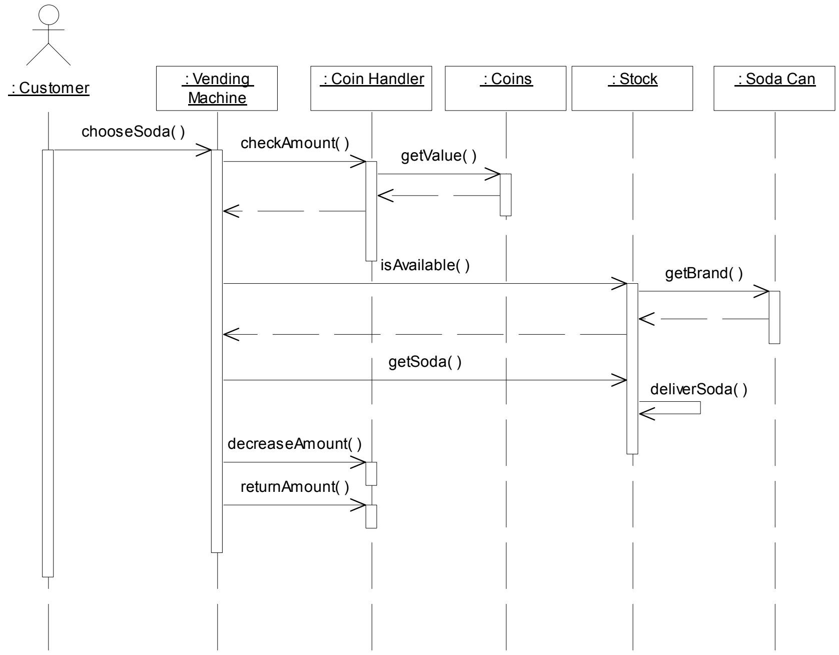


Figure 7, sequence diagram showing messages between objects

First of all the vertical line denotes time, and the rectangles on the lines appearance of the objects. In the diagram, the actor initiates the activity by the message chooseSoda() to the Vending Machine. The Vending Machine in turn triggers the operation checkAmount() which is implemented by the CoinHandler class. The CoinHandler class has several instances of the Coin class (one for each coin), and for every one of them the CoinHandler class calls the getValue() operation. In this sequence diagram there is no distinguished notation for iterative calls, such as the mentioned case when every instance is called. However, sometimes it might be useful to express iterations such as this, and UML has support for this, but most often it is better to keep down the complexity of the figures. The hatched arrow from Coins to CoinHandler is a return arrow, indicating that some kind of return flow are taking place, it is the same type of return arrow from CoinHandler to VendingMachine. Return arrows can be used if desired, but they are often not necessary, it is again often best to keep it as simple as possible. Moving to the Stock class and looking at the communication it generates, we can see that there is one arrow, the one marked with deliverSoda() that is going to the class itself. That is an internal method call, which is generated by the evoked getSoda() method.

6 STATE DIAGRAMS

UML State charts are most often used for low level design, like modeling the internal behavior of a complicated class. But they are also useful on a higher level on modeling different states of a whole system; this can be compared to the usage of class diagrams on several levels.

The basic elements in a state chart are states and transitions, in figure 8 the basic elements and the notation is shown. To the left in the figure there is a state with an arrow symbolizing a state transition going up and then back to itself again, so in this case it is not a transition to another state. The notation of a state transition is showed in the text on top of the arrow. Event is the event that triggers the transition. Inside the braces it stands Guard Condition, which is an additional condition that needs to be fulfilled for the transition to be taken. Eventually, the Action is the action that will take place during the transition. Rightmost in the figure we have two special symbols indicating the start and the stop state respectively.

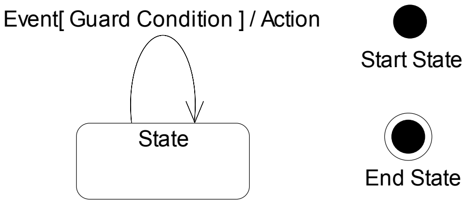


Figure 8, basic symbols in UML state diagrams

In figure 9 is a high level state chart diagram, showing state transitions on the system level in an airplane. This diagram is completely without events, guards and actions, but if they are possible to identify and give relevant information on this level they shall be used. The start state is located to the left in the figure, the initial state is On Ground, followed by Take Off, to Flying, to Landing and finally back to On Ground again.

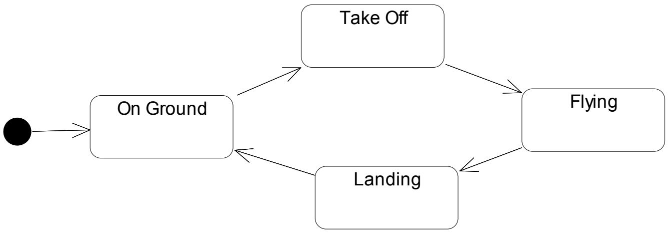


Figure 9, system level state diagram of an airplane

An example of a lower level state diagram is the one in figure 10. It is a candy machine and the start state is again located to the left in the figure. From the start state there is an unconditional transition to the IDLE state, during the transition it is an action Clear Amount taking place. From the idle state it is two transitions, one leading to the IDLE state again which is triggered by the Candy Choice event, and another which is triggered by the Money Insertion event leading to the READY TO SERVE state with the Increase Amount action. The transition to and back to the READY TO SERVE state is triggered by two different events. One eve is the Candy Choice event again with a guard Amount To Small, this shall be treated as if the condition is true the transition is taken. Notice that there is a transition from the READY TO SERVE STATE to the AMOUNT ENOUGH state, which is triggered by the same event Candy Choice but has another guard condition Amount Enough. This illustrates the usage of guard conditions. If the current state is READY TO SERVE and an incoming event is Candy Choice, the machine takes the transition to the AMOUNT ENOUGH state if the condition Amount Enough is true, otherwise the Amount To Small condition must be true and the machine takes the transition back to the READY TO SERVE state again.

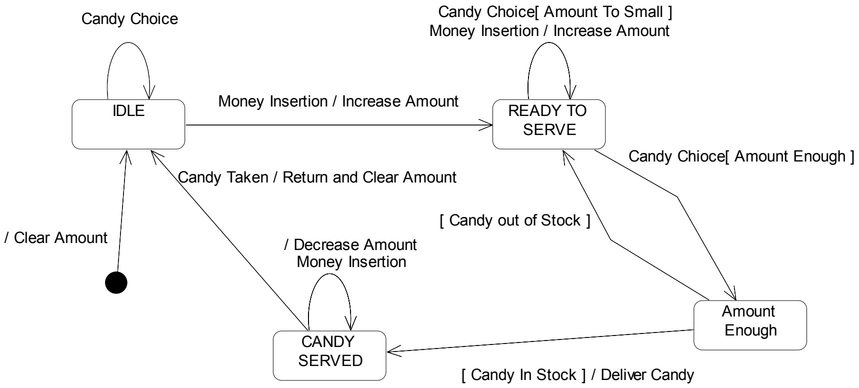

Candy Choice
Candy Choice[Amount To Small]
Money Insertion / Increase Amount
IDLE
Money Insertion / Increase Amount
READY TO SERVE
Candy Chioce[Amount Enough]
Candy Taken / Return and Clear Amount
[Candy out of Stock]
/ Clear Amount
/ Decrease Amount
Money Insertion
Amount Enough
CANDY SERVED
[Candy In Stock] / Deliver Candy

Figure 10, state diagram of a candy machine

7 ACTIVITY DIAGRAMS

Activity diagrams can be in many places in the design process; sometimes even before use case diagrams for understanding the workflow of a process. But they can also be used for defining how use cases interact or even for detailed design.

Basic elements in activity diagrams are activities, branches (conditions or selections), transitions, forks and joins. An example of the workflow of building a house is visualized in figure 11. The diagram starts in black dot in the top, which is the start state. The first activity in the house building process is to select a site, followed by making a bid plan. After the bid plan there is a branch in the workflow either the bid plan is accepted, or in the case when it is not accepted the flow is leading back to the bid plan activity again which means that the bid plan has to be refined. Continuing down to the fork in the case we have accepted the bid plan, it is two flows out from it meaning that it is parallel activities. Trade work and site work are taking place in parallel and ends in the join symbol below them, with only one single workflow out of it leading to the construction activity. Which is the last activity before the flow reaches the end symbol.

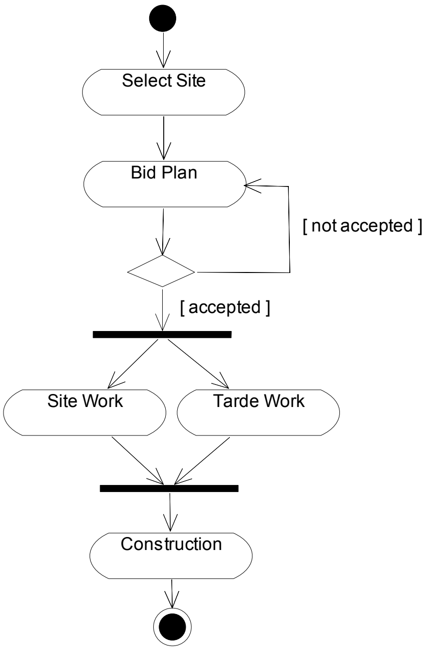


Figure 11, the workflow of building a house

Figure 12, is an example of an on-line record store. In this diagram, there is a feature called swim lanes represented. Swim lanes are useful when a designer wants to express how related activities can be grouped. They can for instance be grouped since they are in the same use case, or in the same organizational unit, different nodes in a distributed system or whatever. Here the modeler has chosen to have the swim lanes Customer, Shop System and Shop Assistant. The flow starts in the swim lane for Customer related activities with a login, if the Shop System accepts the login it lists the records. The customer picks some records, and has to confirm the choice after a listing from the shop system. Upon a confirmation from the customer the Shop System calculates the price for the records and prints the invoice. The invoice is later picked up by a Shop Assistant who actuates the order.

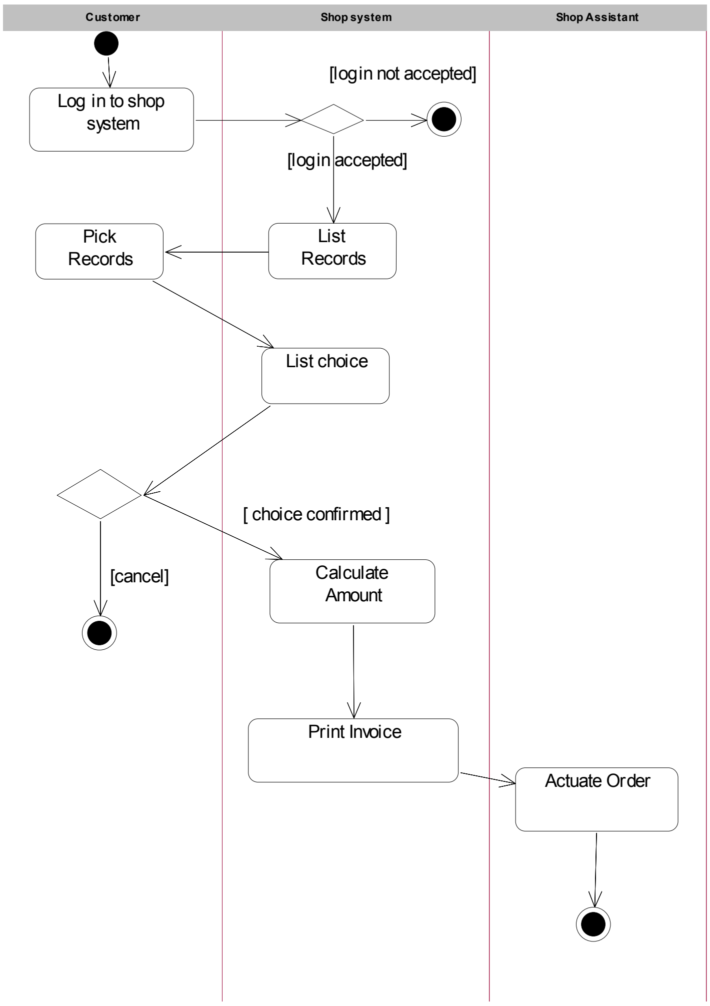


Figure 12, activity diagram of the flow in a web based record store

8 A SMALL SYSTEM DESIGN

Imagine the software system at a library. The main task should be keeping track of all books and the status of each book (out of lone, in stock etc). In this example we are modeling and explaining the system with a sequence of UML diagrams. The first step is the system analysis, and the input to the analysis is the specification of the requirements. In an object oriented and UML approach the requirements are identified with help of identifying of cases of use of the system. This is done by UML use case diagrams. The main goal of this part is to identify the most characteristic use cases, and the actors (i.e. people or other types of "users" of the system). In figure 13, a UML use case diagram shows examples of how the system is intended to be used. To the left in the figure we have identified an actor Librarian, which is a librarian. The librarian can use the system in four ways. Firstly it is possible to add a new book to the system visualized by the AddNewBook use case. Secondly, if a book is out of loan, it is possible for the librarian to make a reservation for a customer; this is showed by the ReserveBook use case. It should also be possible to loan books, i.e. the LoanBook use case. Finally the ReturnBook use case shows that a librarian should also be able to return a book to the system when a customer hands a previously borrowed book in.

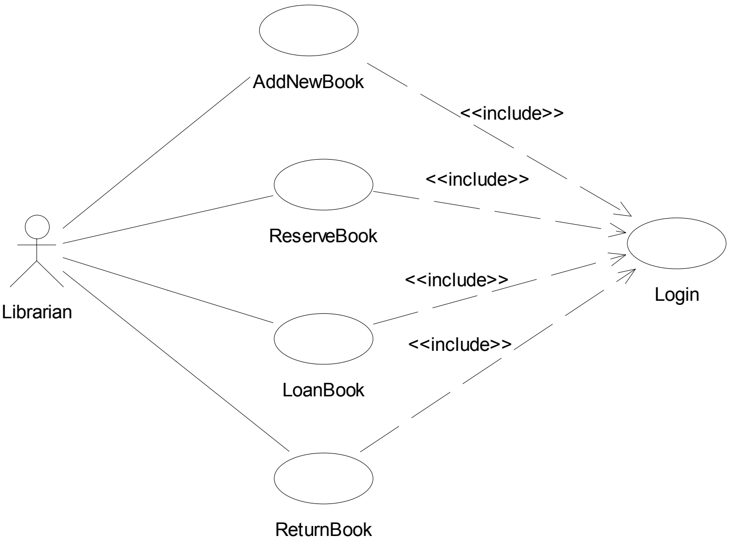


Figure 13, a UML use-case diagram, visualizing how a librarian can use a library booking system.

The next step in the analysis and design process is to identify the objects the system deal with. Form the problem description and use case diagrams it is easy to identify the objects involved in the system: The library, the books, and the librarian. In addition to this we also have a librarian system itself which we can specify as a set of services. We specify the objects by specifying the classes with their attributes and services (methods) they provide. In UML this is done by a class diagram. This diagram also includes specifications of relations between the objects. In Figure 14, a UML class diagram shows the four classes Book, Library, LoginService and Librarian and how they are related to each others. Internal attributes and methods are also shown in the figure.

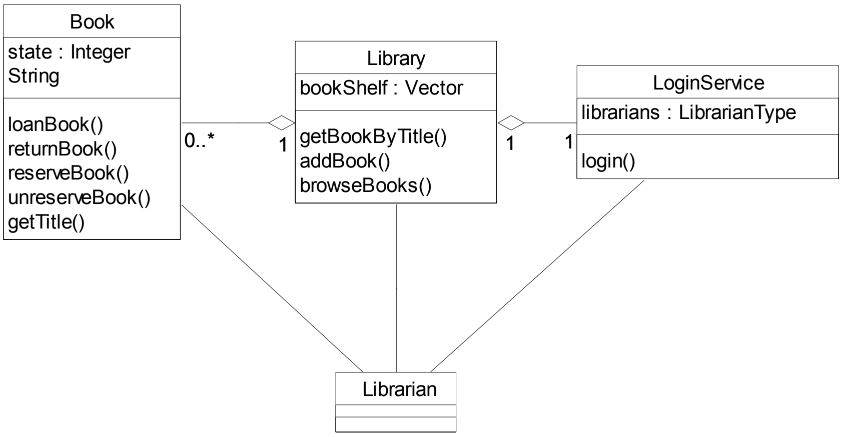


Figure 14, UML class diagram

The UML class diagram shows the static characteristics (i.e. the structure) of the system. The dynamic behavior of the system can be described by state chart diagrams and interaction diagrams. State chart diagrams are used to express the states of the systems, or internal inside classes, and its transition from a state to a state triggered by a particular event. State chart diagrams are variations of finite-state machines, a standard method used in software design and programming.

Figure 15 shows UML state chart diagram showing the internal state of the class Book. The diagram shows that a book can be in one of the four states In stock, Out of loan, Reserved and out of loan and Reserved and in stock. The starting point for a new book is marked with the black dot to the left in the figure. When a new book is entered in the system it enters the In stock state, when the method loanBook is executed the book enters the Out of loan state etc. This diagram has no guards or actions specified, and all events are actually methods in the book class. This might not be the most common usage of state chart diagrams, but it fit quite nicely in this design.

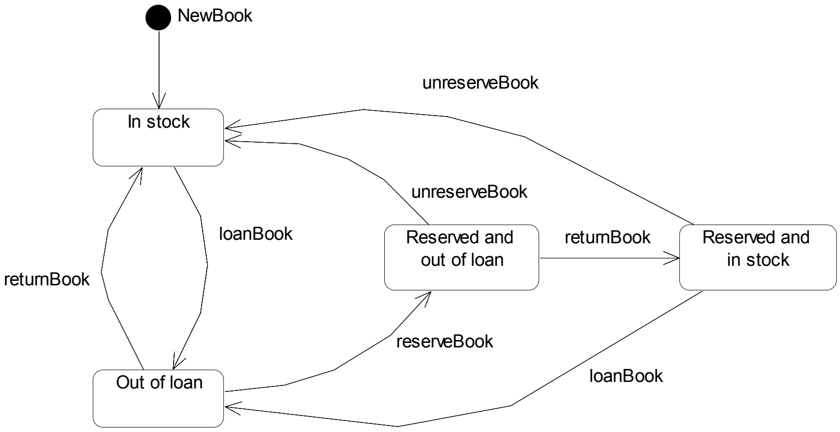


Figure 15, UML state-chart diagram for the class book

UML sequence diagrams are used to show interaction between different objects in a time sequence. The vertical line denotes time, the rectangles appearance of the objects, and the arrows invocation of services of particular objects, or interaction between the objects. In Figure 16, the LoanBook use-case is further developed with a UML interaction diagram, showing the sequence of interactions required to solve the use-case. Firstly the Librarian has to use the login() method provided by the LoginService class. Then a reference to the book is required, and the librarian has to utilize the getBookByTitle() method provided by the Library class. Finally the loaning process can be accomplished by the loanBook method provided by the class representing a book.

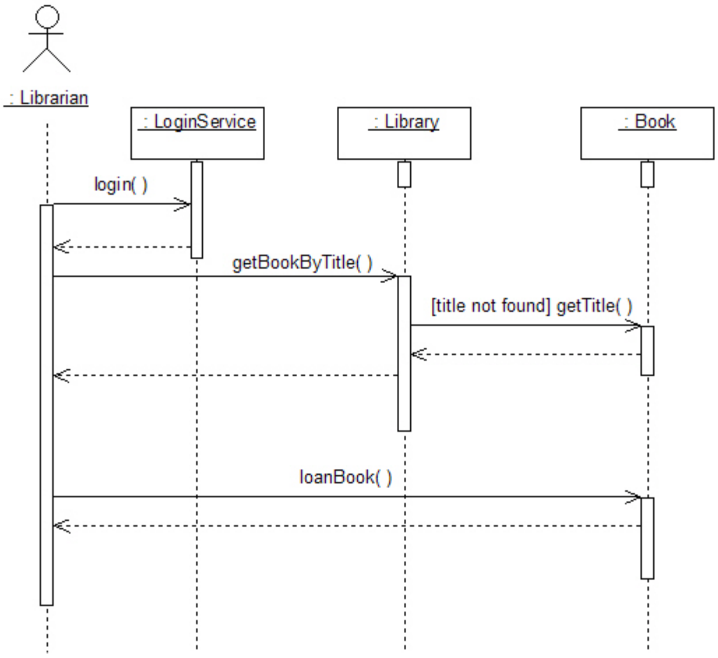


Figure 16, a UML sequence diagram, solving the loan book use-case

9 REFERENCES

Suggestions for further reading!

Grady Booch, James Rumbaugh, and Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley. ISBN 0-201-57168-4

Perdita Stevens, and Ron Pooley, Using UML, Addison-Wesley, ISBN 0-201-648601-1