Understanding Repository – 03

UML Diagrams Research

Eurico Costa

CIT-360 W16

# Research – Use Case Diagrams

Use cases are used to explain and document the interaction that is required between the

user and the system to accomplish the user’s task. Use cases are created to help the development team understand more fully the steps that are involved in accomplishing the user’s goals. Once created, use cases often can be used to derive more detailed functional requirements for the new system. (Dennis, 2014)

Use cases help us understand and clarify the users’ required interactions with the system and can help us more fully understand the functional requirements of the new system. Consequently, use cases are used extensively in the analysis phase when working with the users in interviews and workshop settings as a means of discovering user and functional requirements.

A use case represents how a system interacts with its environment by illustrating the activities that are performed by the users of the system and the system’s responses. The goal is to create a set of use cases that describe all the tasks that users need to perform using the system. Use cases are oft en thought of as an external or functional view of a business process, showing how the users view the process rather than the internal mechanisms by which the process operates. Since use cases describe the system’s activities from the user’s perspective in words, it is essential to involve users in their development. Therefore, creating use cases helps ensure that users’ insights are explicitly incorporated into the new system. (Dennis, 2014)

What is a use case? A use case depicts a set of activities performed to produce some output result. Each use case describes how an event triggers actions performed by the system and the user. With this type of event-driven modeling, everything in the system can be thought of as a response to some trigger event. When there are no events, the system is at rest, patiently waiting for the next event to trigger it. When a trigger event occurs, the system (and the people using it) responds, performs the actions defined in the use case, and then returns to the waiting state.

Use cases specify only what your system is supposed to do, i.e., the system’s functional requirements. They do not specify what the system shall not do, i.e., the system’s nonfunctional requirements. Nonfunctional requirements often include performance targets and programming languages, etc. (Miles, 2006)

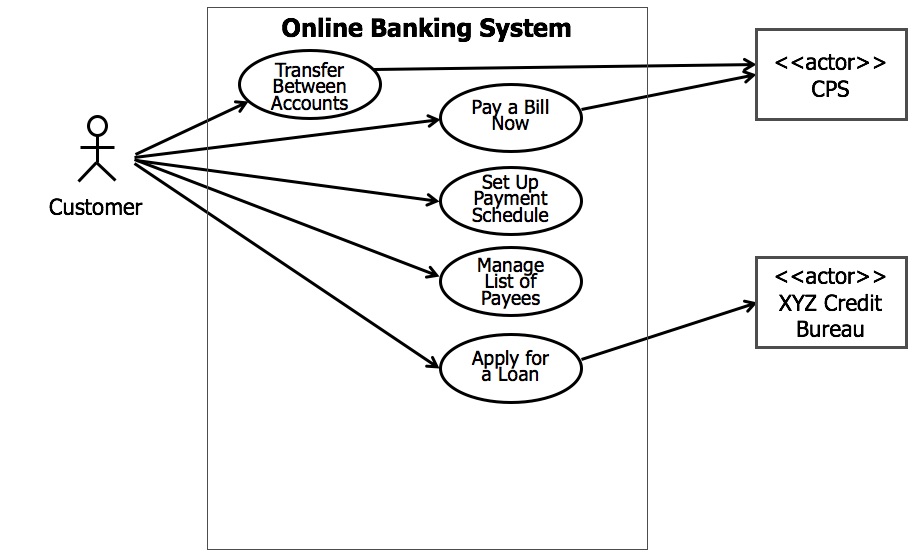
Use case analysis is a technique for eliciting, analyzing, documenting, and communicating the functional requirements of a business or system.

* Natural way to organize functional requirements
* Easier for users to understand than traditional textual requirements

A system Use Case

* Describes a behavior
* Describes how actor use a system to accomplish goals
* Is expressed by consequences of exchanges between that system and actors
* Does not reveal internal structure
* Bundles together a set of related scenarios

Example of a UML use case:



Value of use cases:

* Discover Stakeholder Priorities
* Explain what the system will do
* Focus on requirements, not design
* Provides a context for requirements
* Plan and manage the project
* Provide foundation for test cases

How to find Candidate System Use Cases?

* Start with the business use cases
  + Identify system use cases to be implemented by the system
* For each system actor…
  + Determine what services the system must provide for that actor
* Investigate gaps in the requirements
  + Initial mapping to uses cases, will identify high level gaps
  + Use cases already realized in a product, for COTS solutions, require less elaboration.

Use case diagraming tips:

* Choose a good name
  + Verb-noun phrase
  + Name that represents a behavior that can be completed
  + 4 words or less
* Consider review session with users
* Work with one actor and goal at a time
* Limit each use case to one goal
* Avoid individual CRUD use cases
  + “Update Address”
  + “Read Address”
* Group these into single use cases
  + “Maintain Address”

Online References on Use Cases:

* UML 2 Use Case Diagrams: An Agile Introduction: <http://www.agilemodeling.com/artifacts/useCaseDiagram.htm>
* UML Use Case Diagrams: Guidelines: <https://msdn.microsoft.com/en-us/library/dd409432.aspx>
* UML 2 Use Case Diagram: http://www.sparxsystems.com/resources/uml2\_tutorial/uml2\_usecasediagram.html

# Research - Use Case Documents

A diagram showing your use cases and actors may be a nice starting point, but it does not provide enough detail for your system designers to actually understand exactly how the system’s concerns will be met. How can a system designer understand who the most important actor is from the use case notation alone? What steps are involved in the use case? The best way to express this important information is in the form of a text-based description — every use case should be accompanied by one. There are no hard and fast rules as to what exactly goes into a use case description according to UML, but some one example of the types of information are shown below (Miles, 2006):

|  |  |
| --- | --- |
| Use case description detail | What the detail means and why it is useful |
| Related Requirements | Some indication as to which requirements this use case partially or completely fulfills |
| Goal In Context | The use case’s place within the system and why this use case is important |
| Preconditions | What needs to happen before the use case can be executed |
| Successful End Condition | What the system’s condition should be if the use case executes successfully |
| Failed End Condition | What the system’s condition should be if the use case fails to execute successfully |
| Primary Actors | The main actors that participate in the use case. Often includes the actors that trigger or directly receive information from a use case’s execution |
| Secondary Actors | Actors that participate but are not the main players in a use case’s execution |
| Trigger | The event triggered by an actor that causes the use case to execute |
| Main Flow | The place to describe each of the important steps in a use case’s normal execution |
| Extensions | A description of any alternative steps from the ones described in the Main Flow. |

Miles, Russ; Hamilton, Kim (2006-04-25). Learning UML 2.0 (Kindle Locations 700-723). O'Reilly Media.

Examples of use case documents:

* Use Case Scenario Example and Template: <https://www.lucidchart.com/pages/use-case-scenario-example-and-template-UML>
* Writing and effective Use Case: [https://www.visualparadigm.com/tutorials/writingeffectiveusecase.jsp](https://www.visual-paradigm.com/tutorials/writingeffectiveusecase.jsp)
* Develop a Use Case Document: <http://www.cragsystems.co.uk/development_process/develop_use_case_document.htm>

# Research - State Diagrams

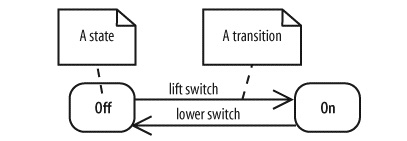
State machine diagrams are heavily used in special niches of software and hardware systems, including the following:

* Real-time/ mission-critical systems, such as heart monitoring software
* Dedicated devices whose behavior is defined in terms of state, such as ATMs
* First-person shooter games, such as Doom or Half-Life

(Miles, 2006)

State machine diagrams are often referred to informally as state diagrams. You may also have seen them referred to as a state chart diagrams in the past, since this diagram has undergone many name changes.

Below is a state diagram example, depicting the states of a light:



A state diagram consists of states, drawn as rounded rectangles, and transitions, drawn as arrows connecting the states.

A transition represents a change of state, or how to get from one state to the next.

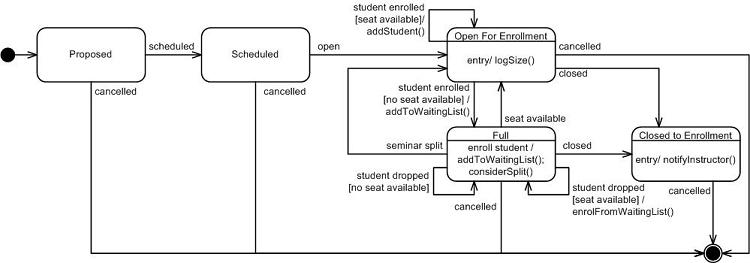
A state is active when entered through a transition, and it becomes inactive when exited through a transition. The event causing the state change, or trigger, is written along the transition arrow. The light in the diagram above has two states: Off and On. It changes state when the lift switch or lower switch triggers occur.

States A state is a condition of being at a certain time. A state can be a passive quality, such as On and Off for the light object. A state can also be an active quality, or something that an object is doing. For example, a car has the state Running during which the engine is running.

The state diagram in the Unified Modeling Language is essentially a Harel statechart with standardized notation which can describe many systems, from computer programs to business processes. In UML 2 the name has been changed to State Machine Diagram. The following are the basic notational elements that can be used to make up a diagram:

* Filled circle, representing to the initial state
* Hollow circle containing a smaller filled circle, indicating the final state (if any)
* Rounded rectangle, denoting a state. Top of the rectangle contains a name of the state. Can contain a horizontal line in the middle, below which the activities that are done in that state are indicated
* Arrow, denoting transition. The name of the event (if any) causing this transition labels the arrow body. A guard expression may be added before a "/" and enclosed in square-brackets ( eventName[guardExpression] ), denoting that this expression must be true for the transition to take place. If an action is performed during this transition, it is added to the label following a "/" ( eventName[guardExpression]/action ).
* Thick horizontal line with either x>1 lines entering and 1 line leaving or 1 line entering and x>1 lines leaving. These denote join/fork, respectively.

The diagram below represents a state machine diagram for a seminar lifecycle:



Additional information on State machine diagrams can be found here: <http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm>

# Research - Sequence Diagrams

Sequence diagrams are an important member of the group known as interaction diagrams. Interaction diagrams model important runtime interactions between the parts that make up your system and form part of the logical view of your model (Miles, 2006)

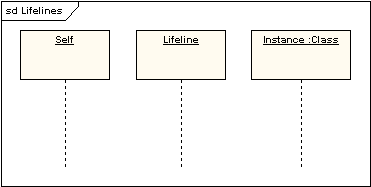
## Tutorial (Sparx Systems)

**Sequence Diagrams**

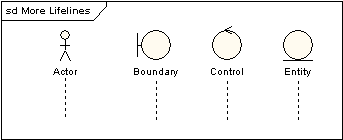
A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which other objects; and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic.

**Lifelines**

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is "self", that indicates that the lifeline represents the classifier which owns the sequence diagram.

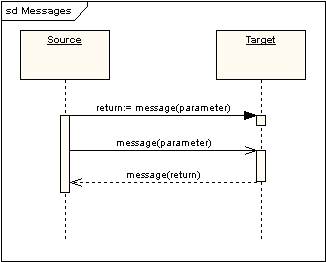
****

Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.

****

**Messages**

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by line arrowhead), and the third is the asynchronous return message (denoted by the dashed line).

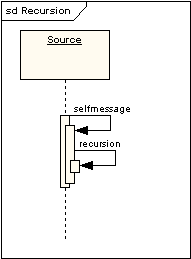
****

**Execution Occurrence**

A thin rectangle running down the lifeline denotes the execution occurrence, or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning a reply; and the third is the target object receiving an asynchronous message and returning a reply.

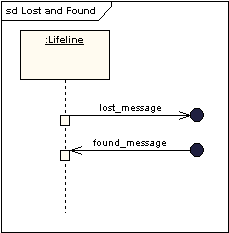
**Self Message**

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline’s execution occurrence.



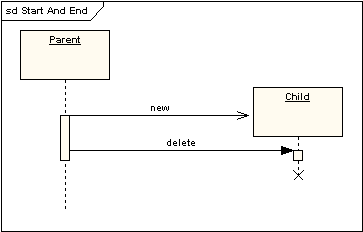
**Lost and Found Messages**

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.



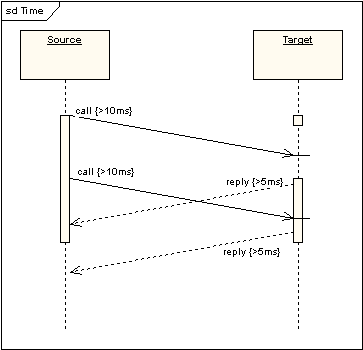
**Lifeline Start and End**

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.



**Duration and Time Constraints**

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modelling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.



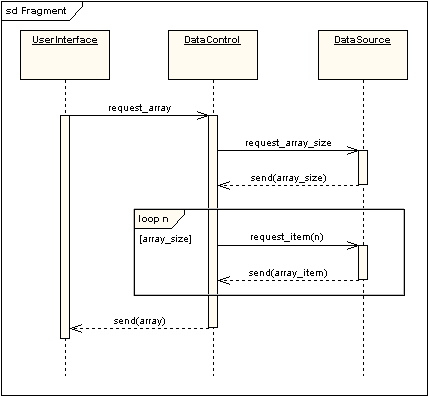
**Combined Fragments**

It was stated earlier that sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances.

The fragments available are:

* Alternative fragment (denoted “alt”) models if…then…else constructs.
* Option fragment (denoted “opt”) models switch constructs.
* Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
* Parallel fragment (denoted “par”) models concurrent processing.
* Weak sequencing fragment (denoted “seq”) encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don’t share a lifeline.
* Strict sequencing fragment (denoted “strict”) encloses a series of messages which must be processed in the given order.
* Negative fragment (denoted “neg”) encloses an invalid series of messages.
* Critical fragment encloses a critical section.
* Ignore fragment declares a message or message to be of no interest if it appears in the current context.
* Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
* Assertion fragment (denoted “assert”) designates that any sequence not shown as an operand of the assertion is invalid.
* Loop fragment encloses a series of messages which are repeated.

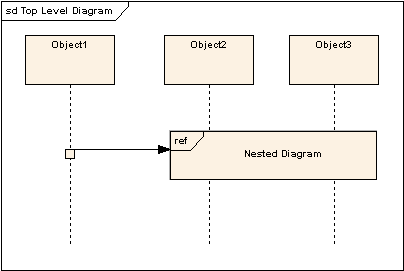
The following diagram shows a loop fragment.

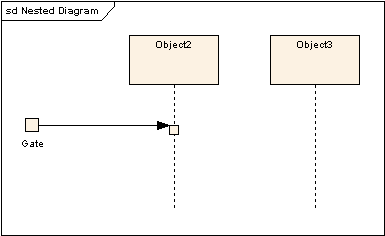


There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

**Gate**

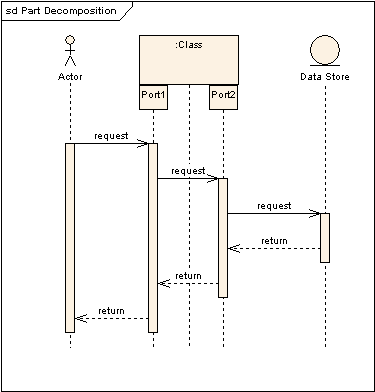
A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame. Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages. The following two diagrams show how they might be used in practice. Note that the gate on the top level diagram is the point at which the message arrowhead touches the reference fragment - there is no need to render it as a box shape.





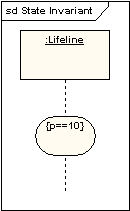
**Part Decomposition**

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.



**State Invariant / Continuations**

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.



A continuation has the same notation as a state invariant, but is used in combined fragments and can stretch across more than one lifeline.

Other sources on Sequence Diagrams:

* Sequence Diagraming Guidelines: <http://agilemodeling.com/style/sequenceDiagram.htm>
* UML 2 Sequence diagrams, an agile introduction: <http://www.agilemodeling.com/artifacts/sequenceDiagram.htm>
* UML Basics – Sequence Diagrams: <http://www.ibm.com/developerworks/rational/library/3101.html>
* A quick introduction to Sequence diagrams: <http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/>

# Research - Class Diagrams

A class in UML is drawn as a rectangle split into up to three sections. The top section contains the name of the class, the middle section contains the attributes or information that the class contains, and the final section contains the operations that represent the behavior that the class exhibits. The attributes and operations sections are optional. If the attributes and operations sections are not shown, it does not necessarily imply that they are empty, just that the diagram is perhaps easier to understand with that information hidden.

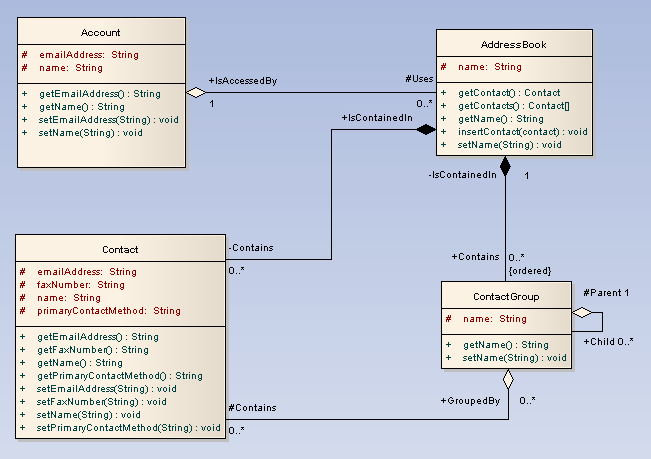
A class’s name establishes a type for the objects that will be instantiated based on it. (Miles, 2006).

**Class Diagrams Tutorial (Sparxsystems)** (‘UML 2 class diagram’, 2000)

The class diagram shows the building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations.

Class diagrams are most useful in illustrating relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively.

The diagram below illustrates aggregation relationships between classes. The lighter aggregation indicates that the class "Account" uses AddressBook, but does not necessarily contain an instance of it. The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes, for example Contact and ContactGroup values will be contained in AddressBook.

****

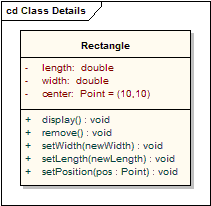
**Classes**

A class is an element that defines the attributes and behaviors that an object is able to generate. The behavior is described by the possible messages the class is able to understand, along with operations that are appropriate for each message. Classes may also have definitions of constraints, tagged values and stereotypes.

**Class Notation**

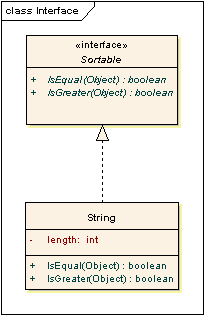
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.  In the diagram below the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values.

The final compartment shows the operations setWidth, setLength and setPosition and their parameters. The notation that precedes the attribute, or operation name, indicates the visibility of the element: if the + symbol is used, the attribute, or operation, has a public level of visibility; if a - symbol is used, the attribute, or operation, is private. In addition the # symbol allows an operation, or attribute, to be defined as protected, while the ~ symbol indicates package visibility.

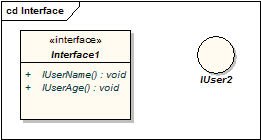


**Interfaces**

An interface is a specification of behavior that implementers agree to meet; it is a contract. By realizing an interface, classes are guaranteed to support a required behavior, which allows the system to treat non-related elements in the same way – that is, through the common interface.

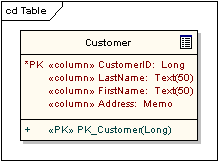


Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



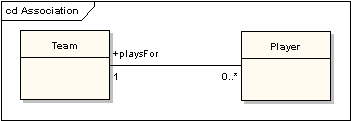
**Tables**

Although not a part of the base UML, a table is an example of what can be done with stereotypes. It is drawn with a small table icon in the upper right corner. Table attributes are stereotyped «column». Most tables will have a primary key, being one or more fields that form a unique combination used to access the table, plus a primary key operation which is stereotyped «PK». Some tables will have one or more foreign keys, being one or more fields that together map onto a primary key in a related table, plus a foreign key operation which is stereotyped «FK».



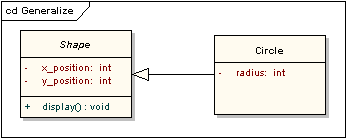
**Associations**

An association implies two model elements have a relationship - usually implemented as an instance variable in one class. This connector may include named roles at each end, cardinality, direction and constraints. Association is the general relationship type between elements. For more than two elements, a diamond representation toolbox element can be used as well. When code is generated for class diagrams, named association ends become instance variables in the target class. So, for the example below, "playsFor" will become an instance variable in the "Player" class.

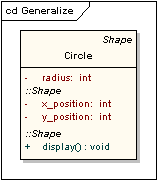


**Generalizations**

A generalization is used to indicate inheritance. Drawn from the specific classifier to a general classifier, the generalize implication is that the source inherits the target's characteristics. The following diagram shows a parent class generalizing a child class. Implicitly, an instantiated object of the Circle class will have attributes x\_position, y\_position and radius and a method display(). Note that the class "Shape" is abstract, shown by the name being italicized.

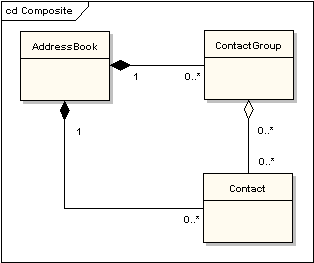


The following diagram shows an equivalent view of the same information.



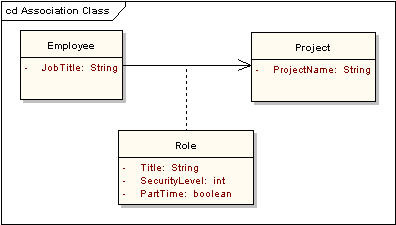
**Aggregations**

Aggregations are used to depict elements which are made up of smaller components. Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class.  A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a part can be individually removed from a composition without having to delete the entire composition. Compositions are transitive, asymmetric relationships and can be recursive.   The following diagram illustrates the difference between weak and strong aggregations. An address book is made up of a multiplicity of contacts and contact groups. A contact group is a virtual grouping of contacts; a contact may be included in more than one contact group. If you delete an address book, all the contacts and contact groups will be deleted too; if you delete a contact group, no contacts will be deleted.



**Association Classes**

An association class is a construct that allows an association connection to have operations and attributes. The following example shows that there is more to allocating an employee to a project than making a simple association link between the two classes: the role the employee takes up on the project is a complex entity in its own right and contains detail that does not belong in the employee or project class. For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



**Dependencies**

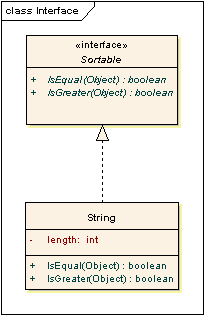
A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two elements, but it is too early to know exactly what the relationship is. Later in the design process, dependencies will be stereotyped (stereotypes available include «instantiate», «trace», «import», and others), or replaced with a more specific type of connector.

**Traces**

The trace relationship is a specialization of a dependency, linking model elements or sets of elements that represent the same idea across models. Traces are often used to track requirements and model changes. As changes can occur in both directions, the order of this dependency is usually ignored. The relationship's properties can specify the trace mapping, but the trace is usually bi-directional, informal and rarely computable.

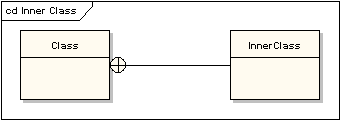
**Realizations**

The source object implements or realizes the destination. Realizations are used to express traceability and completeness in the model - a business process or requirement is realized by one or more use cases, which are in turn realized by some classes, which in turn are realized by a component, etc. Mapping requirements, classes, etc. across the design of your system, up through the levels of modeling abstraction, ensures the big picture of your system remembers and reflects all the little pictures and details that constrain and define it. A realization is shown as a dashed line with a solid arrowhead.



**Nestings**

A nesting is connector that shows the source element is nested within the target element. The following diagram shows the definition of an inner class, although in EA it is more usual to show them by their position in the project view hierarchy.



Other sources on Class Diagrams:

* UML Class Diagrams – Reference: <https://msdn.microsoft.com/en-us/library/dd409437.aspx>
* UML 2 Class diagrams, an agile introduction: <http://www.agilemodeling.com/artifacts/classDiagram.htm>

# Research - System Level and Unit Tests

There are two distinct roles in many software projects that are involved with testing: developers and testers. Should they take the same approach to testing, or are there some principles that apply to only one of the roles? What should they do to coordinate their work? <http://www.agileconnection.com/article/unit-vs-system-testing-its-ok-be-different>

**Unit Testing**

Unit testing is the practice of testing small pieces of code, typically individual functions, alone and isolated. If your test uses some external resource, like the network or a database, it’s not a unit test.

Unit tests should be fairly simple to write. A unit tests should essentially just give the function that’s tested some inputs, and then check what the function outputs is correct. In practice this can vary, because if your code is poorly designed, writing unit tests can be difficult. Because of that, unit testing is the only testing method which also helps you write better code – Code that’s hard to unit test usually has poor design.

In a sense, unit testing is the backbone. You can use unit tests to help design your code and keep it as a safety net when doing changes, and the same methods you use for unit testing are also applicable to the other types of testing. All the other test types are also constructed from similar pieces as unit tests, they are just more complex and less precise.

Unit tests are also great for preventing regressions – bugs that occur repeatedly. Many times there’s been a particularly troublesome piece of code which just keeps breaking no matter how many times I fix it. By adding unit tests to check for those specific bugs, you can easily prevent situations like that. You can also use integration tests or functional tests for regression testing, but unit tests are much more useful because they are very specific, which makes it easy to pinpoint and then fix the problem.

When should you use unit testing? Ideally all the time, by applying test-driven development. A good set of unit tests do not only prevent bugs, but also improve your code design, and make sure you can later refactor your code without everything completely breaking apart.

Popular tools for unit testing include Mocha, JUnit, etc.

**System Level Testing**

System testing is the type of testing to check the behavior of a complete and fully integrated software product based on the software requirements specification (SRS) document. The main focus of this testing is to evaluate Business / Functional / End-user requirements.

This is black box type of testing where external working of the software is evaluated with the help of requirement documents & it is totally based on Users point of view. For this type of testing do not required knowledge of internal design or structure or code.

This testing is to be carried out only after System Integration Testing is completed where both Functional & Non-Functional requirements are verified.

In the integration testing testers are concentrated on finding bugs/defects on integrated modules. But in the Software System Testing testers are concentrated on finding bugs/defects based on software application behavior, software design and expectation of end user.

Why system testing is important:

1. In Software Development Life Cycle the System Testing is perform as the first level of testing where the System is tested as a whole.
2. In this step of testing check if system meets functional requirement or not.
3. System Testing enables you to test, validate and verify both the Application Architecture and Business requirements.
4. The application/System is tested in an environment that particularly resembles the effective production environment where the application/software will be lastly deployed.

Generally, a separate and dedicated team is responsible for system testing. And, System Testing is performed on staging server which is similar to production server. So this means you are testing software application as good as production environment. (Ramakrishna, 2016)

# References

Dennis, Alan; Wixom, Barbara Haley; Roth, Roberta M. (2014-10-29). Systems Analysis and Design, 6th Edition (Page 120). Wiley.

Oracle, Inc., (2011). Oracle Unified Method Level 3 Training. oracle.com.

Miles, R., & Hamilton, K. (2006-04-25). *UML 2.0* (2nd ed.). Sebastopol, CA 95472: O’Reilly.

State diagram (UML) (2015). . In Wikipedia. Retrieved from <https://en.wikipedia.org/wiki/State_diagram_%28UML%29>

Ramakrishna. (2016). *Software testing class*. Retrieved March 14, 2016, from <http://www.softwaretestingclass.com/system-testing-what-why-how/>

UML 2 state machine diagrams: An agile introduction. (1999). Retrieved March 14, 2016, from <http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm>

UML 2 sequence diagram. (2000). Retrieved March 14, 2016, from <http://www.sparxsystems.com/resources/uml2_tutorial/uml2_sequencediagram.html>

UML 2 class diagram. (2000). Retrieved March 14, 2016, from <http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_classdiagram.html>