# LS 4.5 – Develop Activity Diagram

The *activity diagram* is one of the UML diagrams most suited to capturing the flow of a process or activity over time as an actor interacts with the system.

An activity diagram:

* Illustrates the data flow of the system and shows the flow of control from activity to activity within the system.
* Represents the decision flows and parallel behavior of the processing performed by the system.
* Important to develop activity diagrams for any complicated processing to be performed by the system.
* Help ensure the flow of control and processing is captured correctly.

### When to Use An Activity Diagram

* Used to model dynamic aspects of the system.
* Can be used to model the system as a whole, a subsystem or an operation.
* Typically used to model a workflow or model an operation.
* Modeling the workflow will help the development team understand and document the business processes that are involved in the system.
* When modeling workflow the activity diagram focuses on activities as viewed by the actors that interact with the system.
* Can also be used to model details of complex computations. When modeling an operation the activity diagram focuses on the parameters of the operation and its objects.
* Will often combine many features/use case into a single activity diagram.

Activity diagrams consist of seven major components

**Start point**

* The entry point into the diagram, from which the flow of control originates. Usually, but not always, in response to some user interaction.
* Represented by a black filled circle 

**End point**

* The point at which the process the diagram is representing ends. Every diagram should have an end point.
* Represented by a black outlined circle 

**Activity**

* Some action being taken by the system or another actor. An activity or business process/operation
* Represented by a rounded rectangle with text describing the activity or use case inside 

**Decision**

* A logical test concerning the results or implications of an activity being performed. Not itself labeled, but the transitions emanating from it are. The preceding activity and the transition labels indicate what the decision was based on.
* Represented by a diamond 

**Transition**

* A directional line emanating from an activity, decision or other non-transition portion of the diagram, moving towards some other non-transition part.
* Transitions mark the flow of control or operation through the diagram.
* Transitions may be marked by guards, which restrict the conditions under which a transition may be followed.
* Represented by an arrow pointing from the originating object to the receiving object
* Watch out for over-guarding: you can end up stuck inside of an activity 

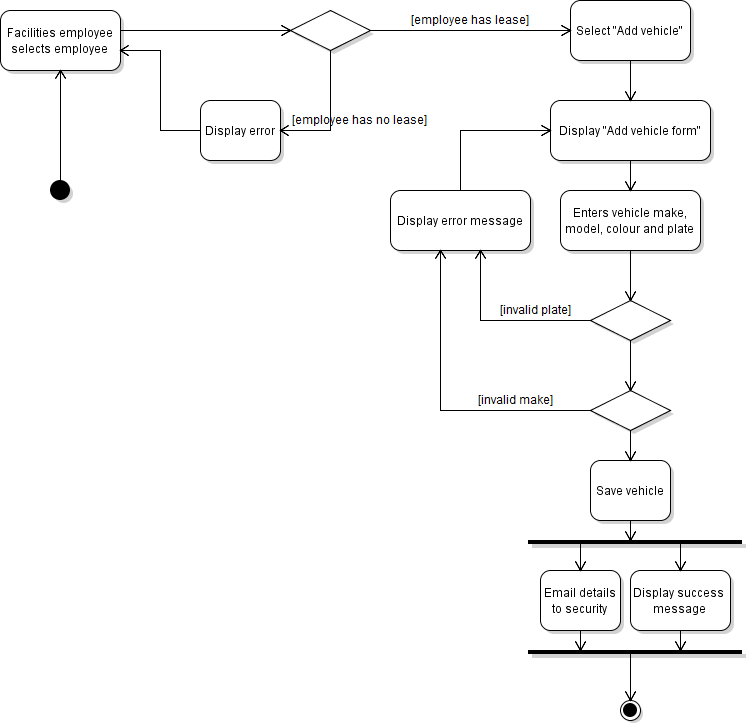
**Split**

* Represents a change from linear to parallel activities. If some decision point results in several operations being undertaken simultaneously, then we would use a split to represent those concurrent lines of control.
* Represented by a solid bar into which one transition enters, but multiple transitions leave 

**Join**

* The counterpart of a split. Once all of the activities spawned by a split have completed, a join converges the lines of control into a single exiting transition.
* Represented by a solid bar into which multiple transitions enter, but only a single transition leaves 

In the parking application described earlier, we could easily model the workflow around adding a vehicle to an employee



# LS 4.6 - Describe Acceptance Testing

Some software errors can be found here:

https://www.bbvaopenmind.com/en/the-5-most-infamous-software-bugs-in-history/

These software bugs highlight the importance of ensuring that the software is error free. Software testing consists of two basic operations:

* **Validation** – ensure the software performs the functions it was intended to perform
* **Verification** – ensure that the software is built correctly.

There are two major categories of software tests:

* **Black Box Testing.** (Acceptance tests) Black box tests are where the tester has no knowledge of how the software was designed or implemented. The goal of black box testing is to test the functionality of the system. Black box testing performs validation testing.
* **White Box Testing.** (Unit tests)are tests where the tester has knowledge of how the software was designed. Used to test that there are no errors in the software. White box testing performs verification testing.

### Acceptance tests

Acceptance tests are

* Tests that demonstrate to the customer the system performs all functionality that is required.
* Sometimes referred to as functional tests.
* Used to test the functionality or suitability of the system BUT are not concerned with how the software is designed or implemented. Generally written based on the requirements of the system.
* Usually created, early in the software lifecycle
* Ideally created by the client.

Key points about acceptance tests:

* **Owned by the customer.** It is ultimately up to the customer as to what they will accept as a functioning system. The acceptance tests are specifying the criteria for the customer to accept the system. These tests must be completed as evidence for the customer to be willing to accept the system as correct and complete.
* Typically, the **developer, customer and tester** will create the acceptance test procedures together. It is not a good idea to have the developer solely responsible for the development of the test procedures as often the developers tend to test specific technical aspects as opposed to functions of the system. It is very easy for a developer to become blind to simple malfunctions in the system due to their constant interaction with it.
* **Focus on the “what”** and not the how. The details of how the software was implemented are not important. Acceptance tests are all about what the customer is getting from the system.
* **Clear and consise.** Each acceptance test should validate a single aspect of the system. The test should be easy to understand and clearly identify the steps required to perform the test.

## Acceptance Tests in an Agile Environment

The acceptance tests for a user story should be determined before the code is written. The tests will drive the creation of the software itself, as ideally the developer will create implement only enough features to satisfy the tests. This, in conjunction with constantly refined unit-test-then-code cycle, is referred to as **Test Driven Development(TDD)**. The tests will identify the required functionality of the system and the required functionality will determine what software needs to be developed. The tests are then driving the development of the software. The software is complete when it can provide the functionality identified by the tests.

There are two levels of acceptance tests that we write:

* **Business level acceptance tests**: a short sentence on the back of a story card, describing the requirement or constraint on the feature being created.
* These are sometimes called acceptance criteria.
* In an agile environment clients will help identify what the user story is supposed to do by specifying what is needed to confirm that the user story is complete.
* **Acceptance test procedures**: detailed, step-by-step description of how to setup, perform and check the result of a test. The tests need to be detailed enough that someone else can repeat the test – this is important when you run across a bug and want to be able to verify that the bug is fixed.

# LS 4.7 - Design Acceptance Tests From Use Case Diagram and User Stories

When designing acceptance tests there are a number of testing areas to consider for behavioral requirements:

* **Normal Cases.** (success cases) Typical functioning of the system.  
  Ex.: Employee logs in using valid credentials.
* **Exception Cases.** Error conditions that may occur. Need to ensure the system is robust enough to handle errors without behaving abnormally.   
  Ex.: Employee logs in using valid credentials when the authentication server is unavailable.
* **Boundary Cases.** It is important to test values right at the edge of being valid. Quite a few software errors occur at the boundary of valid/not valid  
  Ex.: Employee attempts to log in with an empty password, or a password which is longer than the size of the password storage field.

The following types of tests are commonly used to test non-behavioural requirements, though they may be difficult to encode in the testing formats we will discuss.

* **Load tests.** Also known as stress tests. A test that puts the system under extreme conditions to ensure it can operate in those conditions. Could the system handle all SaskPolyTech campus employees logging in and updating their license places at the same time.
* **Soak Tests.** A test that puts the system under an average load for a long period of time. Ex.: Simulate an average of 50-70 employees logging in and performing a standard task, such as viewing or updating their vehicle's license plate. These tests can be helpful in locating errors such as memory leaks or concurrency issues.

### Designing Business level acceptance tests (Acceptance Criteria)

Business level acceptance tests are intended to simply define when a story or feature can be considered "done".

They're a set of simple statements, written in domain (customer) language, which describe in clear "yes/no", "pass/fail" language the conditions under which the functionality implemented as part of the story will be considered complete and satisfactory to the customer. They are not complete, step by step instructions for testing a feature.

Good acceptance criteria will:

* Express intent  
  EXAMPLE: "Employees are presented with an error message when entering invalid credentials" is better than "Employees will see a UITextView with "Invalid Username/Password" when entering invalid credentials"
* Reference other project documentation, rather than repeating it.
* Be high level
* Refer to both functional (behavior) and non-functional requirements.

A commonly used place to start acceptance criteria is to simply ask the customer under what conditions they would consider a given feature complete.

## Example of business level acceptance tests

|  |  |
| --- | --- |
| Employee adds a vehicle to their spot |  |
| As an employee of CST Ventures, I can add a vehicle to my own spot so that I don't get a ticket when I forget my parking pass at home  Prio: 18 Est: 4h | - Employees can only add vehicles to their own spots, not the spots of other employees  - Only employees with an active lease on a spot can add vehicles  - Vehicles registered to an employee are visible in all of their leased spots  - Employees can register a maximum of 3 vehicles  - Vehicle registrations include Make, Model, Colour and License plate |

**FRONT OF CARD BACK OF CARD**

While the requirement itself may have been discovered through conversations with end-users, the requirements do not necessarily come from the same stakeholder. Often, features requested by end-users may not be considered a high priority by primary stakeholders, or may be technically infeasible. When an end user requirement is included in the project, all relevant stakeholders should be consulted in order to discover a comprehensive set of acceptance tests.

The following acceptance tests were discovered in consultation with management-level employees of the Facilities department:

* Employees can only add vehicles to their own spots, not the spots of other employees
* Only employees with an active lease on a spot can add vehicles
* Employees can register a maximum of 3 vehicles

While the Manager of Facilities is unlikely to be the actor concerned with the ability to add vehicles to their parking spot, they still have valuable input to give when creating acceptance tests and limiting the scope of a story. In this case, all three of these acceptance tests apply business rules around security and business process that an end-user would be unlikely to be aware of; it is very important to include multiple stakeholders in conversations around acceptance criteria

The final listed test was discovered in consultation with security personnel, another set of stakeholders in the project.

* Vehicle registrations include Make, Model, Colour and License plate

When an end-user states that they want to avoid receiving a parking ticket, they are not likely to know exactly how their requirement will be implemented, or what steps or other groups are involved in making their request a reality. In this case, neither the end-user nor management level employees knew exactly what information was required in order to prove that a vehicle was in the correct spot; only security personnel, who were actually responsible for checking vehicles and issuing tickets, knew exactly what information was needed in this situation. As you can see, consulting with as wide a variety of stakeholder groups as possible can be essential for discovering business level acceptance tests.

### Designing Acceptance Test Procedures

An acceptance test procedure document is a formal document intended to facilitate black-box testing by someone unfamiliar with the software. They are more frequently created as part of traditional software projects, as agile projects tend to lean more towards business level/story acceptance tests. Acceptance test procedures can be valuable since it gives the team the ability to re-check their previous work at a later date.

Each acceptance test procedure identifies:

* What preconditions must exist (setup)
  + Be very specific. If the user database must contain three specific users before the test begins, describe those users and the process for creating them in detail.
* What steps must be performed (actions)
  + Again, be very specific. Detail each activity which is part of the test in a repeatable, numbered sequence. Because of this format, formal acceptance test procedure documents will often resemble very detailed user scenarios.
* Expected outcome (Inspect and validate)
  + Describe, in detail, how to check the effects or outcome, as well as what should and should not be true if the test is to be considered "passed".
* An indication of the success of the test.
  + This is often misunderstood as meaning that once the test has passed, it does not need to be checked again. Instead, this is simply an indication that the test passed **this time**, or on this test run; it should be run again as part of regression testing at a later time.

It is very important to be specific about the steps to be performed and what the expected outcome is, in order to make it possible to perform the exact execution of the same test over and over again: to make the test repeatable. This is necessary since if the test fails the software must be fixed and retested using the same scenario to ensure the software fix was successful. Also, teams should be re-running previous tests as frequently as is possible to ensure that their recent changes or additions to the project have not broken previous functionality. This process is known as [regression testing](http://en.wikipedia.org/wiki/Regression_testing). Many teams will re-run all of their previous and current release acceptance test procedures as part of the process of building a releasable software package for their customer.

Individual tests are usually documented in some sort of tabular form to make it easier to review for correctness and completion. For example, this is the format used in the CST COSA195 Systems Project class

***Program Name:***

***Test Purpose:***

***Test Author:***

***Date Test Run:***

***Test Executed by:***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Title** | **Program Input** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| **Title of test and any pre-conditions as numbered steps.** | **What you do to the program to test it** | **What you expect to see** | **What the program actually does (filled in when test is run)** | **Filled in when test is run** |

# LS 4.8 - Discuss Traceability of Requirements to Testing

## Traceability Matrix

The **traceability matrix** helps ensure all requirements have associated acceptance tests. Traceability matrixes should be developed while the acceptance tests are being written.

Each requirement is identified in the matrix, and the acceptance test that will verify that requirement will be indicated. This helps find any gaps where an acceptance test hasn’t been defined for a requirement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Requirement Identifier** | **Requirement Description** | **User Story** | **Verification Method** | **Acceptance**  **Test**[TL1] |
| 1.1a) | The system shall allow up to a maximum of 50 characters for name. |  | Demonstration | S1.2 |
| 1.1b) | The system shall ensure age is entered as a value between 0 and 120, inclusive. |  | Demonstration | S1.2 |
| 3.12a) | The system shall use the Java programming language. |  | Inspection |  |
| 3.12b) | The system shall be written in code that follows the coding standard outlined in document S1234.doc. |  | Inspection |  |

**Verification method**

Verification method in a traceability matrix can be:

* **Demonstration** – verify by operating/using the system. (Acceptance tests)
* **Analysis** – verify through analysis. For example: used to determine if the system will meet certain performance requirements or up-time constraints.
* **Inspection** – verify by inspecting the software. For example: requirements for code to be written in a certain language or follow certain coding standards can be verified by inspecting the system.

## Further Reading:

<http://martinfowler.com/bliki/UserStory.html>

<http://martinfowler.com/bliki/UseCase.html>

<http://martinfowler.com/bliki/UseCasesAndStories.html>

<http://martinfowler.com/tags/requirements%20analysis.html>

<http://www.seguetech.com/blog/2013/03/25/characteristics-good-agile-acceptance-criteria>