Chapter 7. **Activities**

What is it that people do in their roles?

In this chapter, I explain what is happening in each of the Quality with Agile through Pictures process activities. This is primarily from the perspective of a business analyst. Examples are included with each activity, to demonstrate how the process may be used to develop a product.

I do not describe the activities within a sprint in detail, since these are well documented elsewhere and they are not within my expertise.

Each activity is assigned a responsible role, and supporting roles. Its inputs and outputs are identified and examples are provided.

- **See Scrum.org for more information about sprint activities.**

This chapter is primarily concerned with clearly defining the business analyst and quality assurance involvement in these activities.

The Quality with Agile through Pictures process includes 13 activities, 3 repositories and several component artifacts attached to data flows. Each activity is represented with an activity flow diagram. The activity is described in in terms of its actors and the information that is used by and output from, the activity.

- **The activity flow diagram was created for this book. See appendix A for the activity flow diagram notation.**

Figure 17 shows the complete process and the artifacts flowing between the activities in the process.

- **The artifacts that are shown on data flows between activities are not intended to be a complete set. They represent the artifacts that are described in this book.**

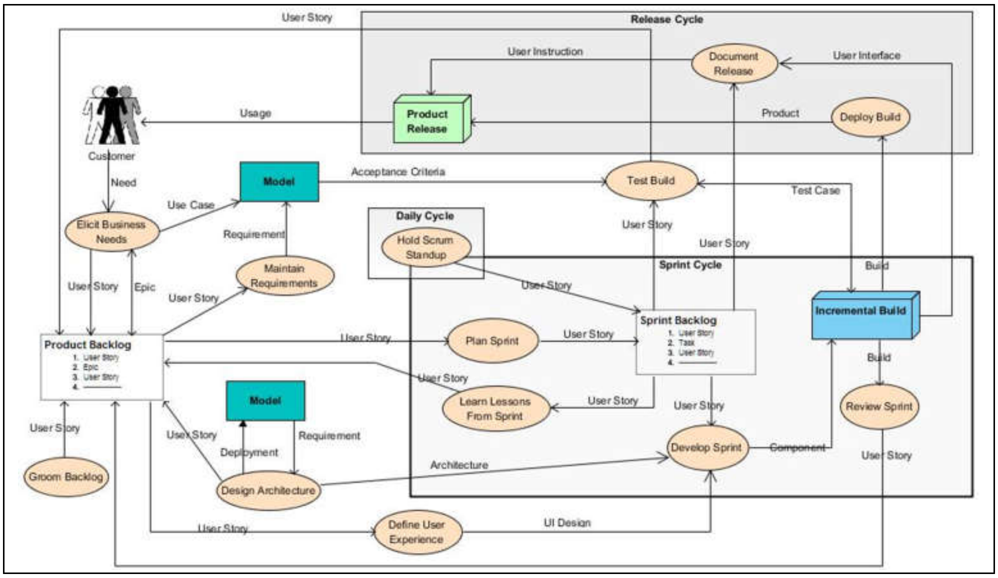


Figure 17 –Quality with Agile through Pictures Process Flow Diagram

The process includes 3 time-boxed containers. These containers encompass activities that occur within the same timeframe.

- Daily – activities occur every working day
- Sprint – activities occur once per sprint
- Release – activities occur once per system release

Activities outside of these time boxes are performed on a continuous basis.

The activities that are added to the Scrum process framework do not necessarily add to the work performed in a regular Scrum development. Much of this work is already performed by the development team, inside a sprint cycle. The Quality with Agile through Pictures process pulls this work out of the sprint cycle and explicitly calls it out in separate and distinct activities. These activities explicitly describe work that adds quality to an agile work effort. This quality may not be obvious when working with the traditional Scrum framework.

7.1 **Scrum Activities**

There follows a brief description of the Scrum ceremonies. This book does not change the Scrum framework (it simply adds to it), so I am not going to attempt to detail existing Scrum activities. For information about Scrum ceremonies (activities) and their responsibilities, see the Scrum.org website.

Five Scrum activities are shown within the box labeled Sprint. The box labeled Daily contains the daily Scrum standup meeting activity. It is shown on the boundary of the sprint, indicating that this activity is part of the sprint and that it includes people from outside of the sprint cycle.

- Plan Sprint – is concerned with populating the sprint backlog
- Develop sprint – is concerned with creating a product build
- Review Sprint – is concerned with demonstrating the results of a sprint
- Learn Lessons From Sprint – is concerned with identifying improvements from sprint activity that may be used for future sprints
- Hold daily scrum – is concerned with product status and planning

Scrum identifies 2 repositories. The product backlog captures user stories and epics that are candidates for development. The sprint backlog captures the user stories (and tasks) being developed during the sprint.

Not shown in the Scrum process diagram Figure 7; Scrum identifies 3 roles; the product owner, the development team and a Scrum master. (The Scrum master role is out of scope for this book.)

♦ **I am rarely fortunate to work with a dedicated Scrum master. As the business analyst, I often perform most of the duties normally assigned to a Scrum master. Typical Scrum master activity is captured below, but it is assigned to one of the roles identified in Chapter 4.**

7.2 **Quality with Agile through Pictures Activities**

Eight activities, 1 repository and 1 repository are included in the extension to the Scrum process. The 8 activities are listed in order of sequence that information is passed between them. However, there is no suggestion that these activities have to be performed in any specific order. When sufficient inputs are available to an activity, that activity may be performed in parallel, with any other activity.

- Elicit Business Needs – is concerned with capturing epics and user stories
- Groom Backlog – is concerned with prioritizing and reviewing the product backlog
- Maintain Requirements – is concerned with maintaining a model of the system
- Design architecture – is concerned with the architectural design of the system
- Define User Experience – is concerned with designing a consistent user interface
- Test Build – is concerned with validating the quality of a product build
- Deploy Build – is concerned with installing a build into a production system
- Document Release – is concerned with the creation of system user instructions

The Quality with Agile through Pictures process introduces a model repository, a release artifact, a release cycle and several data flow components into the process.

Although not shown in the diagram, the deployment manager, independent quality assurance, user experience, solution architect, writer and the business analyst roles have also been added to the process.

7.3 The Example Project

In my previous book ‘Analysis Through Pictures’, I used the example of an automated menu ordering system (AMOS) to demonstrate the process described in that book.

- **Analysis Through Pictures describes a process based on RUP and UML.**

The example in this book builds on the same example system by adding the ability for customers to pay their bill using the same system. In this manner, the following examples do not show building a system from a green field environment. Instead, they build upon a system that is already in production.

7.3.1 As-Is System

The system that is already in production is an Automated Menu Ordering system (AMOS). This system was built from a business need to reduce the work performed by wait staff in a restaurant. Specifically reducing the time spent taking a customer’s order.

A restaurant owner uses AMOS to allow customers to order food from their table. Orders are sent to the kitchen where cooking staff prepare the ordered meals. Wait staff are available to attend to customer requests and a host organizes seating of customers.

A logical view of the AMOS model is represented in Figure 18.

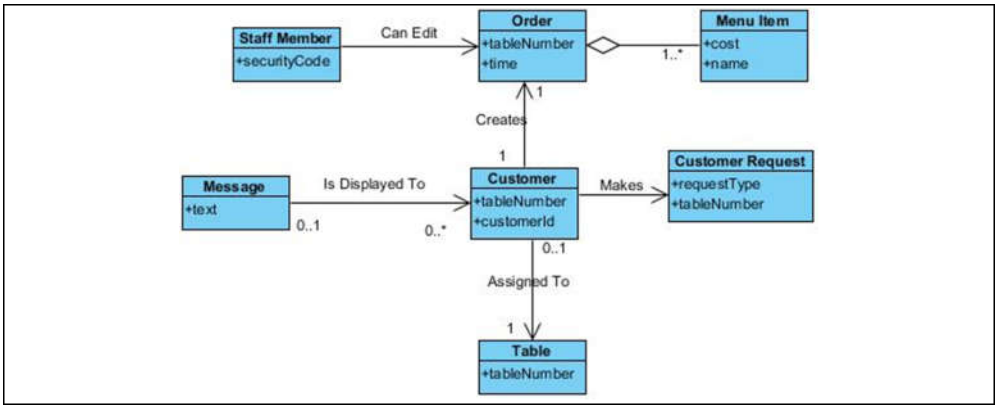


Figure 18 – As-Is Class Diagram

The class diagram shows that:

- each table has a single customer or no customer
- the customer creates an order and selects menu items to place on that order
- staff members can edit the order
- the customer can make a request for assistance
- the customer can be displayed messages from the system

The main functionality of the as-is model can be captured by a system use case that includes the following steps:

Preconditon:

- There is a free table in the restaurant

Steps:

1. A customer is assigned to a table
2. The menu is displayed to the customer
3. The customer creates an order
4. The system displays no items in the order and a running total of zero
5. The customer selects a menu item and adds it to the order
6. The system displays the menu item in the order and updates the running total with the cost of the menu item
7. The customer continues adding menu items to the order until they make a request to submit their order
8. The order is sent to the kitchen

Postcondition:

- Kitchen staff has enough information to prepare the order

This is an example of a system use case main flow in textual form. The activity diagram is show in Figure 19.

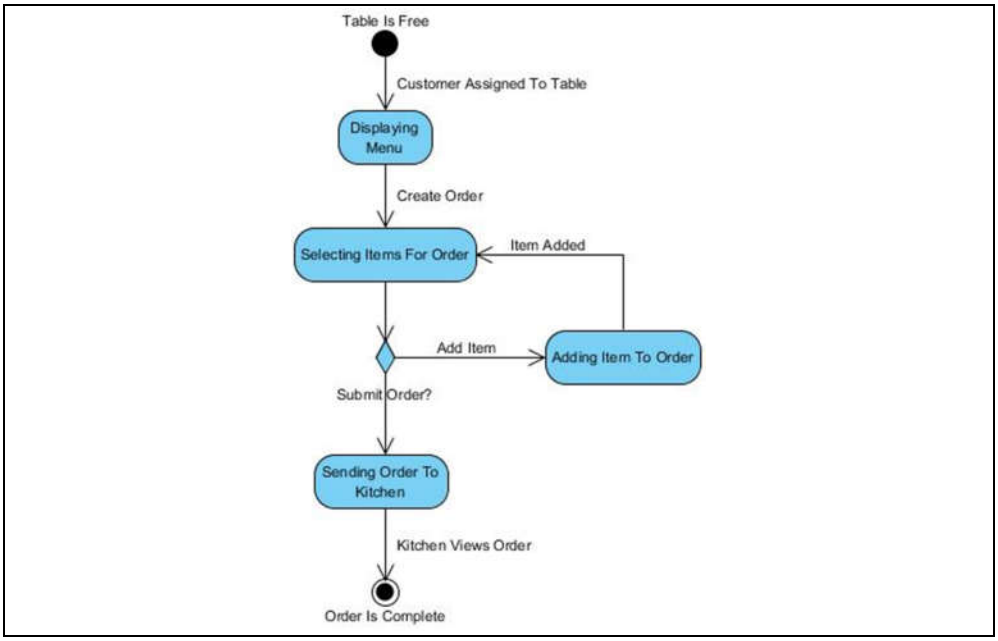


Figure 19 – Order From Menu Main

The alternate paths are shown below. The alternate path steps are not detailed.

♦ **The reader may wish to update the activity diagram to add the alternate paths.**

Alternate Flows:

- prior to submitting an order, menu items may be removed from the order
- after submitting an order, the customer may add more items to their order
- subsequent order submissions only send new items on the order to the kitchen
- the customer may request staff assistance at any time
- the customer may be displayed predefined messages at any time
- a staff member may adjust the menu items on the order at any time

System use cases are detailed section 7.4.1.2.

7.3.2 To-Be System

The restaurant owner has requested that the system be able to take payment from customers while seated at their table. Payments can be made electronically using any major bankcard or popular online payment method. The restaurant will still accept current payment methods, allowing a customer to pay their bill as they do today.

As the business need progresses through each activity in the process, examples are given of the inputs and outputs from that activity.

7.4 **Activity Details**

In the following sections, process activities in are listed by order of initiation. Each activity is described with an activity flow diagram. This diagram includes a use case that represents the activity. The activity is documented in terms of its actors, inputs, outputs and guidelines for conducting the activity. Each activity flow diagram contains:

- a use case representing the activity
- a primary actor who is responsible for the success of the use case
- optional supporting (or secondary) actors who contribute to the activity described by the use case.
- input information from other activities or repositories
- outputs to other activities or repositories

The primary and secondary actors are identified, and a short statement given about their purpose. The data used by the use case is identified and the purpose of the use case is described by its outputs.

The result is an activity flow diagram, actors list, and artifact list and an example of the usage of each activity.

♦ **Since this book is written by a business analyst, you might notice more detail is included in activities that are the responsibility of the business analyst.**

7.4.1 Elicit Business Needs

Figure 20 shows the actors, inputs and outputs of the Elicit Business Needs activity.

Responsible Actor – The business analyst derives product requirements from the needs of the customer.

Contributors – The product owner acts as a stakeholder representing the business.

Inputs – Need is a business process that could be improved with a product feature.

Outputs – Epics and user stories are populated in the product backlog. Use cases are populated in the system model.

The purpose of the Elicit Business Needs activity is to specify business needs that are elicited from business stakeholders. These business stakeholders' needs are captured in the product backlog as epics. Each epic describes a goal of the business in terms of what is needed, why it is needed and who needs it. Epics include any additional information that is useful to analyze the need and create requirements. During elicitation of business needs, user stories are derived from epics and use cases are captured in the model.

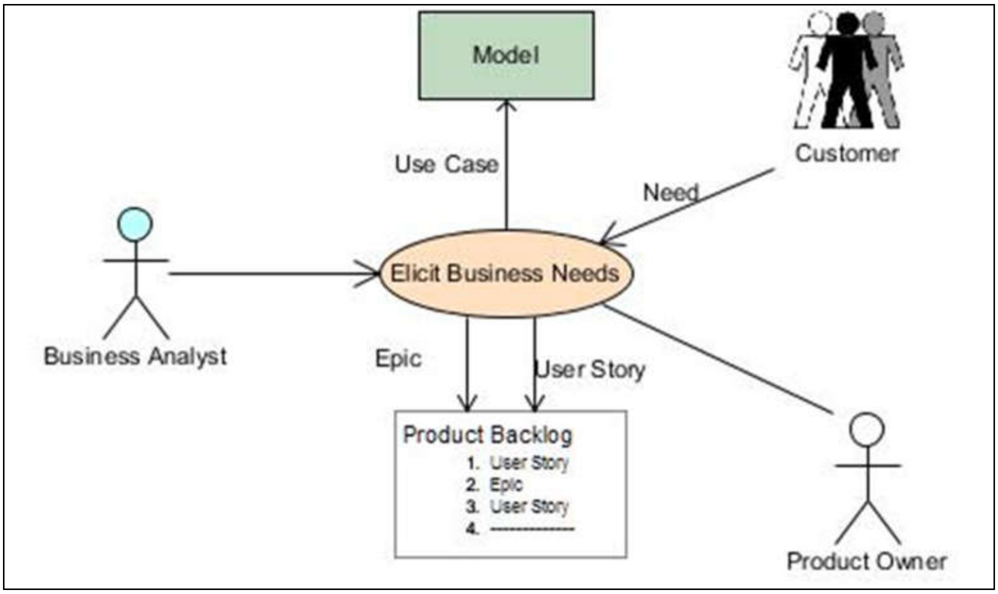


Figure 20 – Elicit Business Needs Use Case

The business analyst is responsible for capturing business needs and deriving requirements for development and testing. The product owner supports business needs elicitation with additional information about the business.

Elicitation of business needs involves many steps for the business analyst. In addition to interviewing stakeholders, the business analyst:

- documents epics and enters them into the product backlog
- creates user stories and populates them in the product backlog
- updates the model repository with business and system use cases

The following tasks are generally performed in sequence; however, that does not mean that an epic needs to be analyzed into its component user stories before the next highest priority epic is worked on. Prioritized epics may be analyzed to an extent that allows the complexity and size of the work to be determined before deciding which child user stories are created for subsequent sprints. The business analyst creates and analyzes use cases, in order to populate the associated user stories in the product backlog for the next sprint. Some epics may be created and never get to be modeled as use cases, because they are removed from the backlog before they are prioritized. Similarly, system use cases may be modeled and never developed because the associated user stories are never assigned to a sprint.

The business analyst optimizes their workload to ensure that every sprint backlog is populated, but also to ensure that there are enough analyzed use cases to replace any user stories that may not make it into the sprint. To this end, I always plan to have enough use cases analyzed and user stories detailed, in order to populate 2 sprint backlogs ahead of time.

7.4.1.1 Elicitation

Elicitation is defined as various data collection techniques used to gather knowledge or information from people. The business analyst uses various collection techniques to gather business information from stakeholders.

Elicitation often begins with written notes that are formulated into an agenda for a meeting with business stakeholder(s).

♦ **Stakeholder time is valuable; I would not start an elicitation conversation with the words 'What would you like us to do?'**

Stakeholders will describe (in great detail, if you encourage them) how they obtain a positive result from the business activity that they are performing. The business analyst captures the steps of this activity, documents an epic, user stories, use cases and any other supporting information. If you do not have experience with elicitation, at this point you may think that elicitation is complete and hand the user story(s) to development.

This is just the start of elicitation. Capturing the steps of an activity performed by the business is just the first stage of eliciting requirements. The main objective is to determine the purpose of the business activity. Find out 'Why are you doing this and what benefit does it bring to your business?'.

♦ **Business analysis is about understanding the business, not just mimicking it.**

Once the business activity is understood, I want to know what the stakeholder dislikes about the process they use to perform this business activity today and how they believe that it could be improved. Try to elicit answers to the following questions:

- What happens when following the normal steps does not produce expected results?
- What are the least efficient parts of the process?
- Which parts of the process do you find most frustrating?

Answers to these questions are needed to fully analyze the business process.

An epic level user story (For .., who wants .., so that.., unlike ..) is created to capture the reason for the current business process. Other relevant information that was discovered during elicitation is also captured in the epic.

The complete business process (including error paths and alternate flows) is modeled with a business activity diagram. Working with the stakeholders the most efficient business process flow is documented and it is this that is turned into user stories.

When interviewing stakeholders the tendency is for them to describe the details before the big picture.

- **This is a natural thought process for human beings. Consider writing an address for example. You write your house number first, the street name next, followed by city, postal code and maybe country of residence.**

I do not discourage eliciting detailed information first, but I also do not take too much care to document the details until I know the big picture. The big picture (or epic user story) allows me to find out if we even want to continue with this business need.

- **Imagine documenting the details of a business process only to discover once the purpose is known, that the system already satisfies that need, or even worse, the system is not intended to satisfy that need.**

When it has been agreed with the stakeholders that the 'to-be' business process has been accurately captured, the business analyst works with the product owner to capture the business steps that will be automated.

- **More about eliciting business needs is documented in the BABOK version 3 under Elicitation and Collaboration.**

The steps to be automated are captured with system use cases. These use cases are broken into steps describing the interaction between the system and its actors. The steps are documented with user stories in the product backlog.

In summary the steps of this activity are:

1. interview stakeholders to identify the business need
2. capture the business need as an epic in the product backlog
3. work with subject matter experts to determine the steps of the business process
4. capture the steps of the to-be business process in a business activity diagram
5. identify opportunities for automation in the business process
6. capture automation steps with system use cases
7. document system use cases in the product backlog as user stories
8. detail acceptance criteria for the user stories

7.4.1.2 Elicit Business Needs Example

The product owner identifies that adding bill payment to the AMOS is a feature that the project team should investigate. With assistance from the product owner, the business analyst interviews stakeholders and SMEs who will be using the delivered system.

Epic Example

The product owner and business analyst work together to create an epic for the bill payment feature. Figure 21 shows an example of the epic for this business use case.

Pay Bill Feature	
Customers should be able to pay their bill at their table so that wait staff do not have to take payment from the customer deliver it to the cashier.	
For	restaurant owner
who wants	customers of my restaurant to be able to automatically pay their bill at their table
so that	wait staff and cashiers do not have to wait on customer s who have finished their meals
unlike	wait staff asking customers when they will be ready to pay their bill, delivering the bill to the customer, taking their payment to the cashier and delivering a receipt to the customer table.
Additional information	
Wait staff	Customers must be encouraged to leave a tip The customer should be able to select a percentage amount for the tip and the system automatically adds it to their bill. Wait staff should be able to adjust the bill in cases where the customer has a legitimate complaint.
Cashiers	The bill, should indicate how much tax was included in the customer bill The customer can make payment using major credit cards, debit cards or with Paypal Customers may elect to pay with cash (or other means) at the cashier station
Restaurant owner	The bill will itemize the prices on the customer order The customer will be able to split the bill by the number of people seated at the table The customers can select which items are included in each itemized bill Each itemized bill may have a different payment method Customer may view their bill at any time after they have made an order.
Cooking staff	Would like to be able to see the customer tips for their shift
Hosting staff	…

Figure 21 - Pay Bill Epic Example

The business analyst identifies system requirements for the Pay Bill Feature by creating a business use case.

Business Use Case Example

Wait staff are the primary actor of Pay Bill business use case. They gain benefit from spending less time with customers after they have already been served. Figure 22 shows a business use case representing this feature.

- ♦ **The notation for the business use case diagram is described in Appendix A - .**

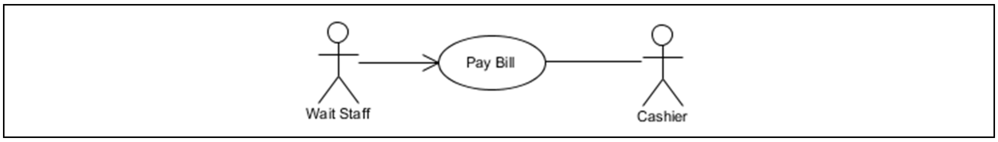


Figure 22 - Pay Bill Business Use Case

The wait staff actor is connected to the business use case with a directed relationship, indicating that wait staff is the primary actor. The cashier actor supports the business use case. The cashier is connected to the business use case using an undirected relationship, which indicates that the cashier is a secondary actor.

Business Use Case Activity Example

When detailing a use case I recommend documenting the expected path first. Each alternate path is then appended to the main path, as a separate path through the use case.

- ♦ **Note that when detailing business use case steps you are capturing what the business workers do. You should not be designing steps for a new system.**

The steps for this business use case are as follows:

Preconditions:

- Customer has ordered a meal

Steps:

1. Customer requests their bill
2. Wait staff asks cashier for the customer bill
3. The cashier produces the customer bill and gives it to the wait staff
4. The wait staff delivers the bill to the customer table
5. The customer indicates that they are ready to pay the bill
6. The wait staff takes the customer payment
7. The wait staff delivers the customer payment to the cashier
8. The cashier processes the customer payment
9. The cashier gives a receipt for the customer payment, to the wait staff

10. The wait staff delivers the customer payment method, receipt and any monetary change to the customer

The use case ends.

Postcondition:

- The customer is ready to leave the table

A.1 Alternative Flow –Wait staff is prepared with the customer bill

11. At step 2, the wait staff already has the customer bill
12. Continue from step 4

A.2 Alternate Flow – The customer has questions about the bill

13. At step 5, the customer indicates that they have a question about the bill
14. The wait staff addresses the customer question
15. Continue from step 6

A.3 Alternate Flow – The customer payment is rejected

16. At step 8, the customer payment cannot be processed
17. The wait staff returns the customer payment method to the customer
18. Continue from step 6

A.4 Alternate Flow – The customer does not pay for their meal

19. At step 14, the wait staff is unable to address the customer question or At step 6, the customer does not have a valid payment method
20. The wait staff gets the restaurant manager

The use case ends.

Postcondition:

- The customer is not ready to leave the restaurant

Notes about the business use case:

- There are 2 terminating post-conditions - One where the customer payment is processed successfully. One where the customer payment has to be addressed by the restaurant manager.
- No systems are assumed in the process – There is no mention of the menu ordering system (AMOS) or other existing systems in the business use case steps.

♦ **Note that UML tools do not distinguish between business and system use cases. The same diagram types and components are used within a business use case view and a system use case view of the model.**

An activity diagram describing these steps is shown in Figure 17.

♦ **The notation for the business use case activity diagram is described in Appendix A - .**

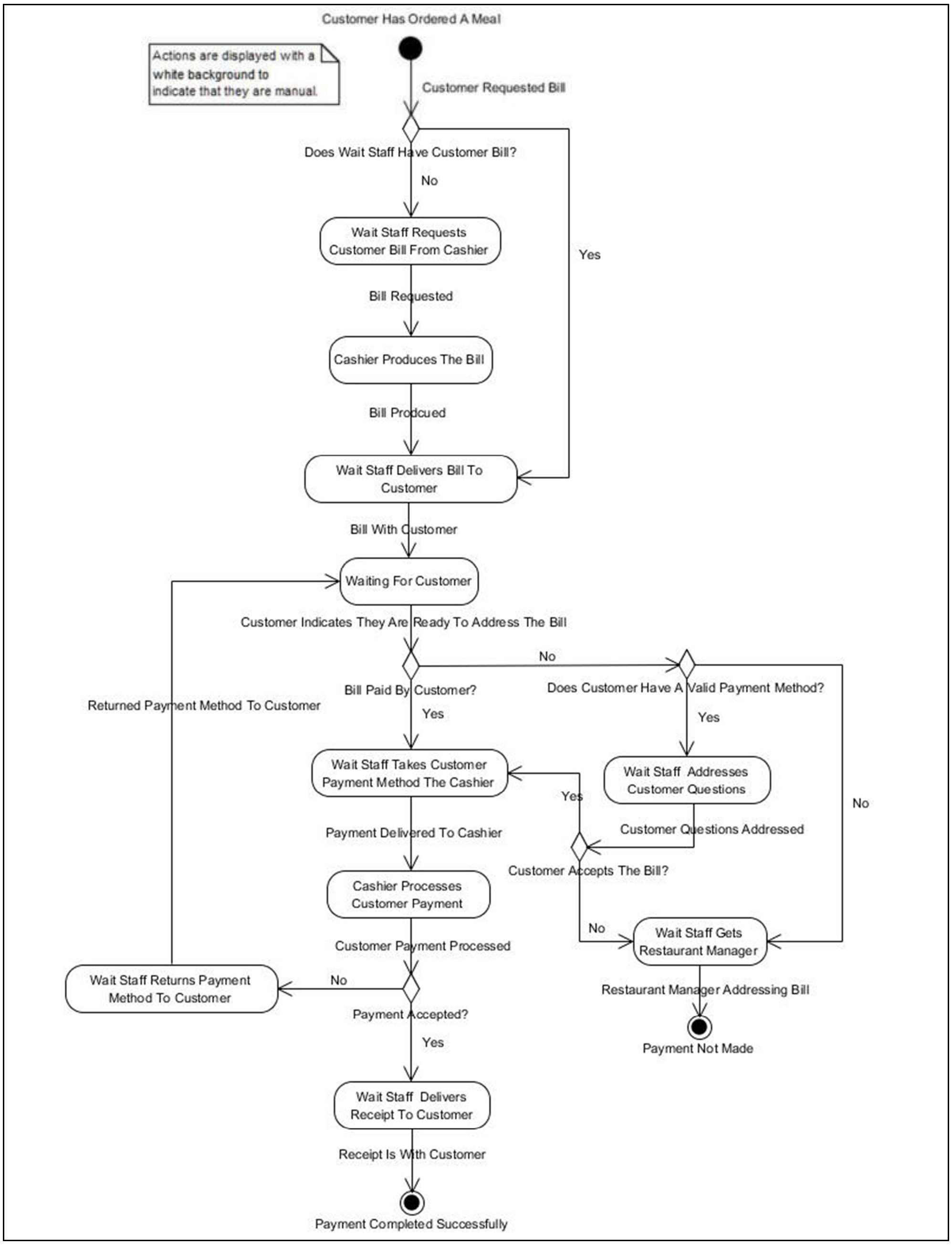

Customer Has Ordered A Meal
Actions are displayed with a white background to indicate that they are manual.
Customer Requested Bill
Does Wait Staff Have Customer Bill?
No
Yes
Wait Staff Requests Customer Bill From Cashier
Bill Requested
Cashier Produces The Bill
Bill Prodcued
Wait Staff Delivers Bill To Customer
Bill With Customer
Waiting For Customer
Customer Indicates They Are Ready To Address The Bill
Bill Paid By Customer?
No
Does Customer Have A Valid Payment Method?
Yes
Returned Payment Method To Customer
Yes
Wait Staff Takes Customer Payment Method The Cashier
Wait Staff Addresses Customer Questions
No
Yes
Customer Questions Addressed
Payment Delivered To Cashier
Customer Accepts The Bill?
Cashier Processes Customer Payment
No
Wait Staff Gets Restaurant Manager
Customer Payment Processed
Wait Staff Returns Payment Method To Customer
No
Restaurant Manager Addressing Bill
Payment Accepted?
Payment Not Made
Yes
Wait Staff Delivers Receipt To Customer
Receipt Is With Customer
Payment Completed Successfully

Figure 23 – Pay Bill Activity Diagram

- **A white background to the action icon indicates that the step is assumed to be performed manually.**

Diagramming the steps helps to validate the business process and assist with producing user stories. I find the diagram flow much easier to follow than reading the textual steps. However, the textual description of the steps assists with locating missing or inconsistent steps in the diagram. I find both are useful to capture an initial picture is the business process, but only the activity diagram is maintained once the system use cases are identified.

Business To System Mapping Example

The business steps have been agreed upon, but no development work has been identified. The business analyst can now assist with automating business steps by mapping the steps in the business activity diagram to system use cases.

I use an activity diagram to map business actions to system use cases. This diagram captures the action steps from the business use case activity diagram. (No other shapes or connectors are needed.) For each action in the diagram, identify if it will be performed manually or if it will be automated.

- ♦ **If a step is identified as partially automated then split it into 2 steps; 1 manual and 1 automated. Update the business activity diagram with the new actions and place them on the business to system mapping diagram.**

Place system use cases on the diagram that will implement the automated steps. Define the scope of each system use case while identifying the appropriate business steps that is will satisfy.

Make sure that system use case scope is defined so that it can be delivered in a single release.

- ♦ **We do not want to deliver functionality to a customer that does not provide benefit.**

For each automated business step, map the corresponding action to the system use case that will perform that step. Many steps may be automated by the same system use case. However, a single step should be performed by a single use case. List the actions in the diagram in roughly the order in which they are performed in the business use case activity diagram.

- ♦ **I use the realization relationship to represent that the use case realizes the business step.**

Figure 24 shows an example of a business to system mapping diagram.

- ♦ **The notation for the business to system mapping diagram is described in Appendix A - .**

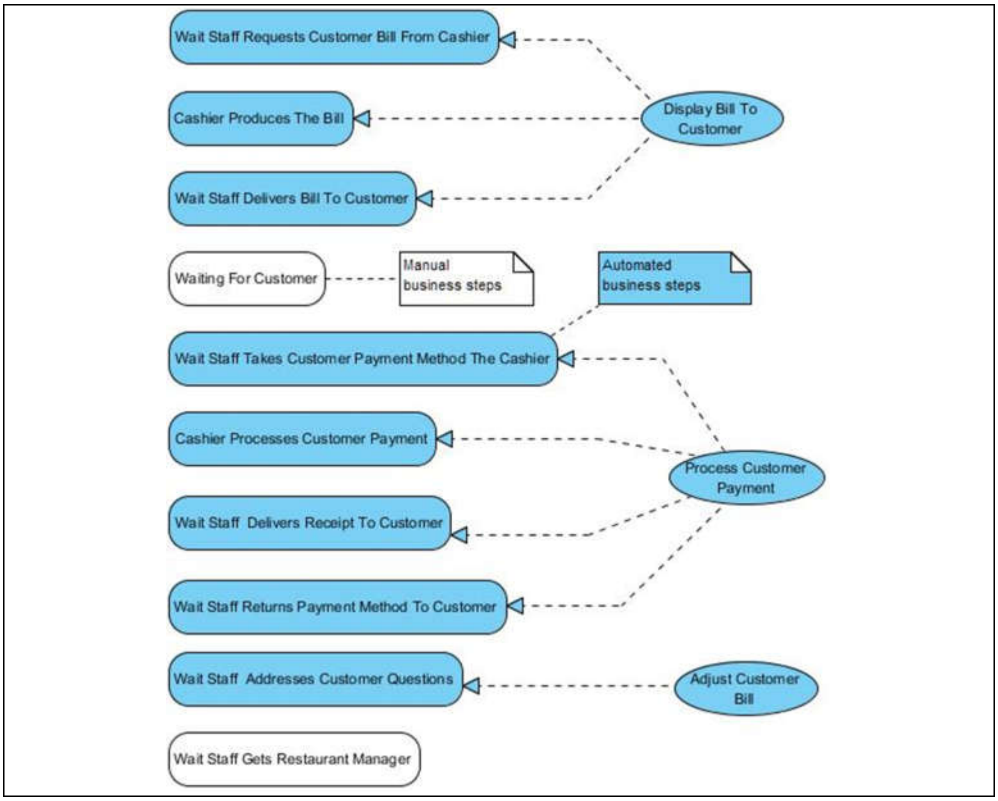

Wait Staff Requests Customer Bill From Cashier
Cashier Produces The Bill
Wait Staff Delivers Bill To Customer
Display Bill To Customer
Waiting For Customer
Manual business steps
Automated business steps
Wait Staff Takes Customer Payment Method The Cashier
Cashier Processes Customer Payment
Process Customer Payment
Wait Staff Delivers Receipt To Customer
Wait Staff Returns Payment Method To Customer
Wait Staff Addresses Customer Questions
Adjust Customer Bill
Wait Staff Gets Restaurant Manager

Figure 24 - Business To System Mapping Diagram

- **Automated business steps are shown as actions with a blue background. Manual actions remain white.**

Eight business steps have been identified as opportunities for automation. The other 2 steps will remain manual. Three use cases have been added to the diagram and linked to the business steps that they will capture.

The System Use Cases

The 3 system use cases have been scoped such that each brings some benefit to the business, by automating part of the Pay Bill epic.

Display Bill To Customer– Allows the customer to view and question their bill at any time.

Process Customer Payment – Allows the customer to pay their bill or ask questions about their bill.

Adjust Customer Bill – Allows restaurant staff to adjust the customer bill.

Business Step Mapping

The primary purpose of the business to system mapping diagram is to ensure that all automated steps in the business activity are captured.

Reading from the top of the diagram, the justification for automating each of the business steps is as follows:

- Wait Staff Requests Customer Bill From Cashier – Wait staff no longer need to get the customer bill because the Display Bill To Customer use case will automatically display it to the customer.
- Cashier Produces the Bill –The cashier no longer needs to produce the customer bill because the Display Bill To Customer use case will automatically display it to the customer.
- Wait Staff Delivers Bill To Customer – Wait staff no longer need to deliver the bill to the customer because the Display Bill To Customer use case will allow the customer to request their bill at the table.
- Wait Staff Takes Customer Payment Method To The Cashier – This step is no longer necessary because the Process Customer Payment use case allows the customer to select a payment method and enter their payment details at the table. This information, along with the bill total is automatically sent to the payment processing system.
- Cashier Processes Customer Payment – The payment processing system processes customer payments and sends the results of the payment request to the system. The cashier does not need to be involved because the Process Customer Payment use case handles the response from the payment processing system.
- Wait Staff Delivers Receipt To Customer – Wait staff no longer need to deliver the customer receipt because when the result of the payment request is received from the payment processing system, the Process Customer Payment use case displays the receipt to the customer. (Alternatively, an error message is displayed if something goes wrong with the payment.)
- Wait Staff Returns Payment Method to Customer – The Process Customer Payment use case does not take the payment from method from the customer, so it no longer needs to be returned.
- Wait Staff Addresses Customer Question – The Adjust Customer Bill use case allows wait staff to alter the customer bill, by changing items on the customer order. The wait staff will have to identify themself in the system in order to perform this function.

Note that the wait staff still has to wait for the customer to pay their bill before they can attend to the next customer and that if the customer is unable to pay their bill the process still needs the intervention of the restaurant manger. Hence, these steps remain manual.

System Use Case Diagram Example

When the system use case scope is defined and we have a complete mapping of automated business steps, the system use cases can be detailed.

Note that just because a bunch of system use cases are derived from a single epic, does not mean that all of those use cases need to be developed at the same time. A system use cases should capture a standalone and complete function that can be developed independently of other use cases. As shown in the following example, each system use case can be released in separate builds while still giving benefit to the customer.

System use cases are captured on system use case diagrams. The use case diagram shows the primary and secondary actors interacting with the use cases. The use case diagram capturing the implementation of Pay Bill feature is shown in Figure 25.

- **The notation for the system use case diagram is described in Appendix A - .**

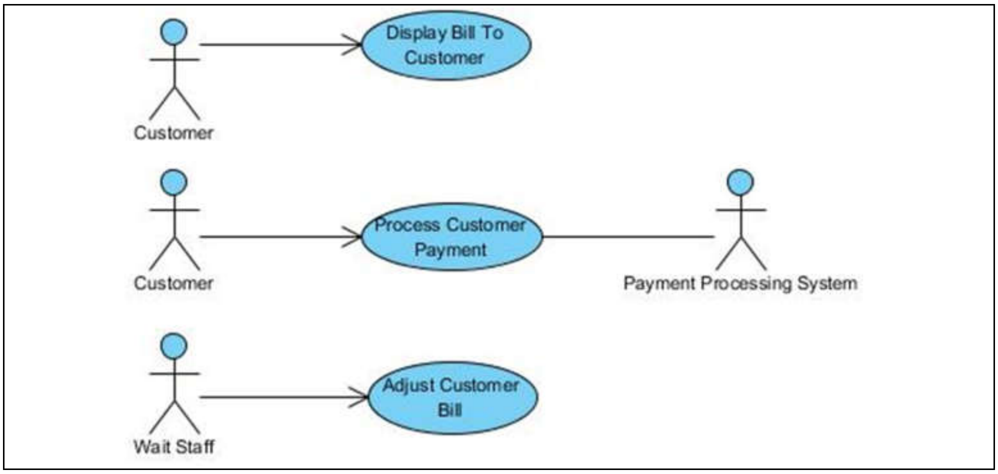


Figure 25 – Pay Bill System Use Case Diagram

The primary actor initiates the functionality of the system use case. Secondary actors interact with the use case to provide support that is required to complete the use case.

- **Do not confuse the use case diagram with the activity flow diagram. The use case diagram shows the scope of the use case in terms of its interaction with external actors. The activity flow diagram was created for this book, in order to show information input to and output from an activity.**

The Customer initiates the initiates the Display Bill To Customer use case. There are no secondary actors to this use case, which means that no other role interacts with the system during execution of the use case.

The Customer initiates the initiates the Process Customer Payment use case. There is 1 secondary actor to this use case - the Payment Processing System. This system validates and takes payment from the customer payment method.

The Wait Staff initiates the Adjust Customer Bill use case. There are no secondary actors to this use case.

I recommend keeping your use case diagrams as simple as is feasible. Includes, extends and inheritance relationships should only be used when they add value to the use case diagram. Do not use these relationships to try to *design* your use cases. The use case diagram is not meant for design. Their purpose is to encapsulate functionality and show the roles that interact with the use case. (You can use data flow diagrams to decompose functionality).

System Use Case Activity Diagram Example

A system use case activity diagram details the function of the use case by breaking down its functionality into steps. Each step is separated by an event. Events are the result of an external input to the system. The main flow through the diagram is generally shown running from the top to the bottom of the diagram. However, decisions and forks may cause branches in the flow to occur.

As with business use case activity, each diagram starts with a single start state and terminates with 1 or more end states. The start state represents the preconditions of the system prior to the use case executing. The end states represent each of the postconditions after the use case completes. The event exiting the start state initiates the system use case. This event is initiated by the primary actor. Intermediate steps are initiated by the primary or any secondary actor. Actions capture the steps is performed by the system.

A precondition is a state that the system must exhibit before the specified functionality is made available. If there are multiple preconditions then all of them must be true before the use case executes.

A postcondition is a state that the system exhibits after the specified functionality is complete. Often there will be a single expected postcondition and possibly several unexpected postconditions. An unexpected postcondition occurs when something unusual happens during execution of the activity diagram.

The system use case activity diagrams for the 3 identified use cases are shown in Figure 26, Figure 27 and Figure 28.

Display Bill System Activity Diagram Example

Figure 26 shows the behavior of the system that allows a customer to display their bill. There are 2 different paths through this activity diagram; 1) the expected path is where the customer makes a request to pay their bill and 2) an alternate path where the customer requests assistance about their bill.

- ♦ **The notation for the system activity diagram is described in Appendix A - .**

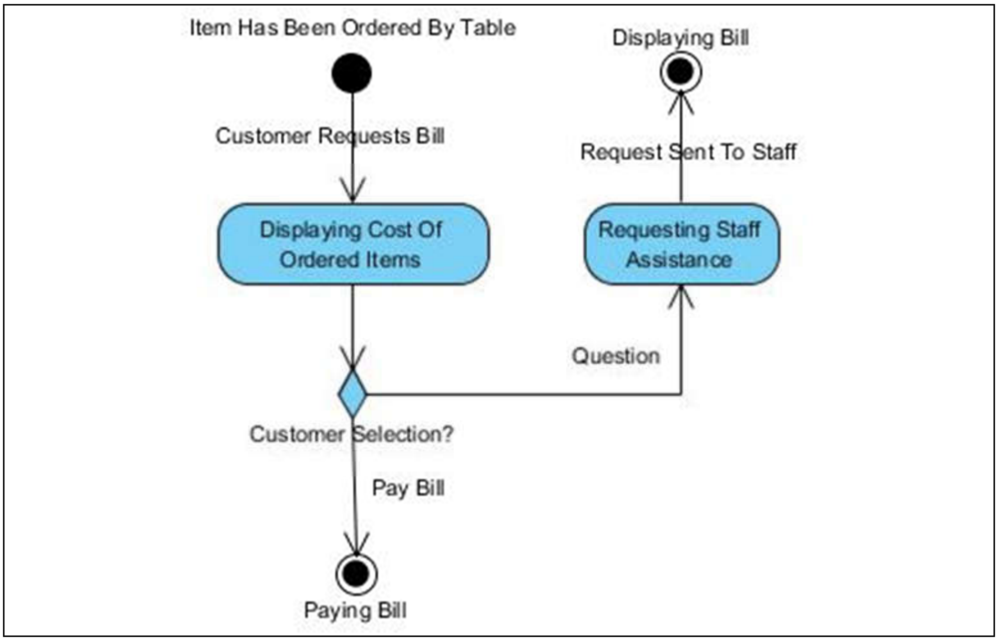


Figure 26 - Display Bill To Customer Activity Diagram

The following is a description of the steps in Figure 26.

- ♦ **Note that alternate paths caused by branches, are often shown flowing up the diagram.**

Preconditions:

- The customer has ordered items from the menu (When the customer has ordered from the menu, they owe money to the restaurant.)
- The customer can display their bill at any time and continue ordering from the menu

Steps:

1. The customer makes a request for the system to display their bill
2. The system displays a list of items that the customer has ordered and their cost, (including taxes)
3. The customer requests to pay their bill

The use case ends.

Postcondition:

- The customer is paying their bill (See Figure 27 for the bill payment process.)

A.1 Alternative Flow – The customer wants to ask a question about their bill

4. At step 3, the customer requests assistance from the system
5. The system sends a request for assistance to the staff

The use case ends.

Postcondition:

- The customer bill is still displayed (See Figure 28 for continuation of a request for customer assistance.)

♦ **By documenting the steps of each alternate path through the use case as its own separate thread makes it easier to break the use case into multiple user stories.**

Process Customer Payment System Activity Diagram Example

Figure 27 shows the behavior of the system that allows a customer to pay their bill. There are 2 different paths through this activity diagram. 1) The expected path ends with the customer successfully paying their bill. 2) The alternate path ends with the customer not paying their bill via the system.

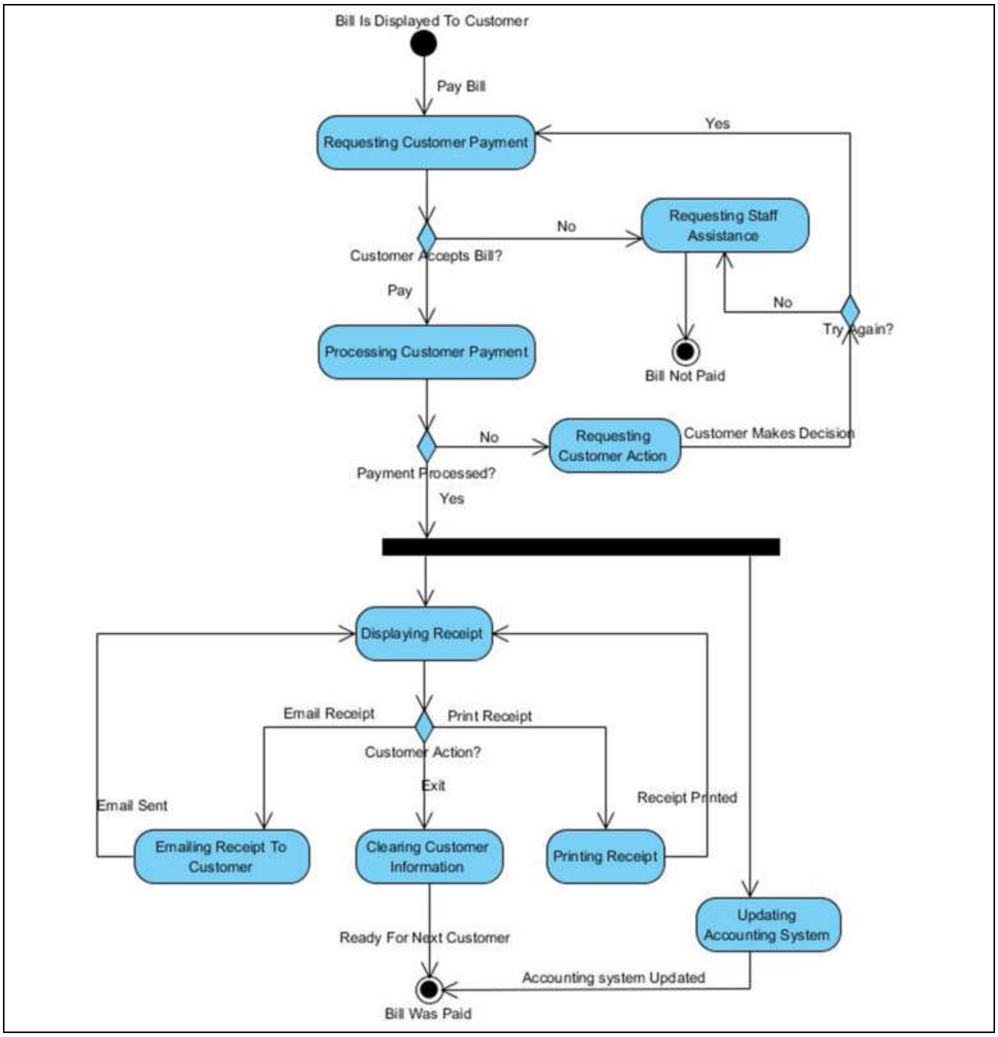


Figure 27 - Process Customer Payment Activity diagram

The following is a description of the steps in Figure 27.

Preconditions:

- The system is displaying the customer bill

Steps:

1. The customer requests to pay their bill
2. The system requests the customer payment information

- ♦ **The above steps are a UX decision. It is equally possible for the customer to enter their payment information before paying their bill.**

---

3. The customer accepts the bill and enters their payment information

4. The system sends a payment processing request to the online payment system
5. The payment is processed successfully
   and the system displays the customer receipt
   and the accounting system is updated
6. The customer exits the system and the system prepares the display for the next customer

The use case ends.

Postcondition:

- The customer bill was paid

A.1 Alternative Flow –The customer does not accept the bill

7. At step 3 the customer does not accept the bill
8. The system requests staff to assist the customer

The use case ends.

Postcondition:

- The customer bill is not paid (see Figure 28 for activity when the bill is not paid)

A.2 Alternative Flow –The customer payment could not be processed

9. At step 5 the system requests the customer if they want to use another payment method
10. The customer wants to try paying again
11. The use case continues from step 2

A.3 Alternative Flow –The customer asked for assistance with paying their bill

12. At step 10 the customer requests assistance
13. The use case continues from step 8

A.4 Alternative Flow –The customer prints the receipt

14. At step 6 the customer asks for a printed copy of their receipt
15. The system prints the customer receipt
16. The use case continues from step 5

A.5 Alternative Flow –The customer emails the receipt

17. At step 6 the customer asks for an email copy of their receipt
18. the system requests staff to assist the customer
19. The use case continues from step 5

Adjust Bill System Activity Diagram Example

Figure 28 shows the behavior of the system when the customer has requested assistance with their bill. There are 2 different paths through this activity diagram. 1) The expected path ends with a bill that was adjusted. 2) The alternate path ends with no adjustment made to the customer bill.

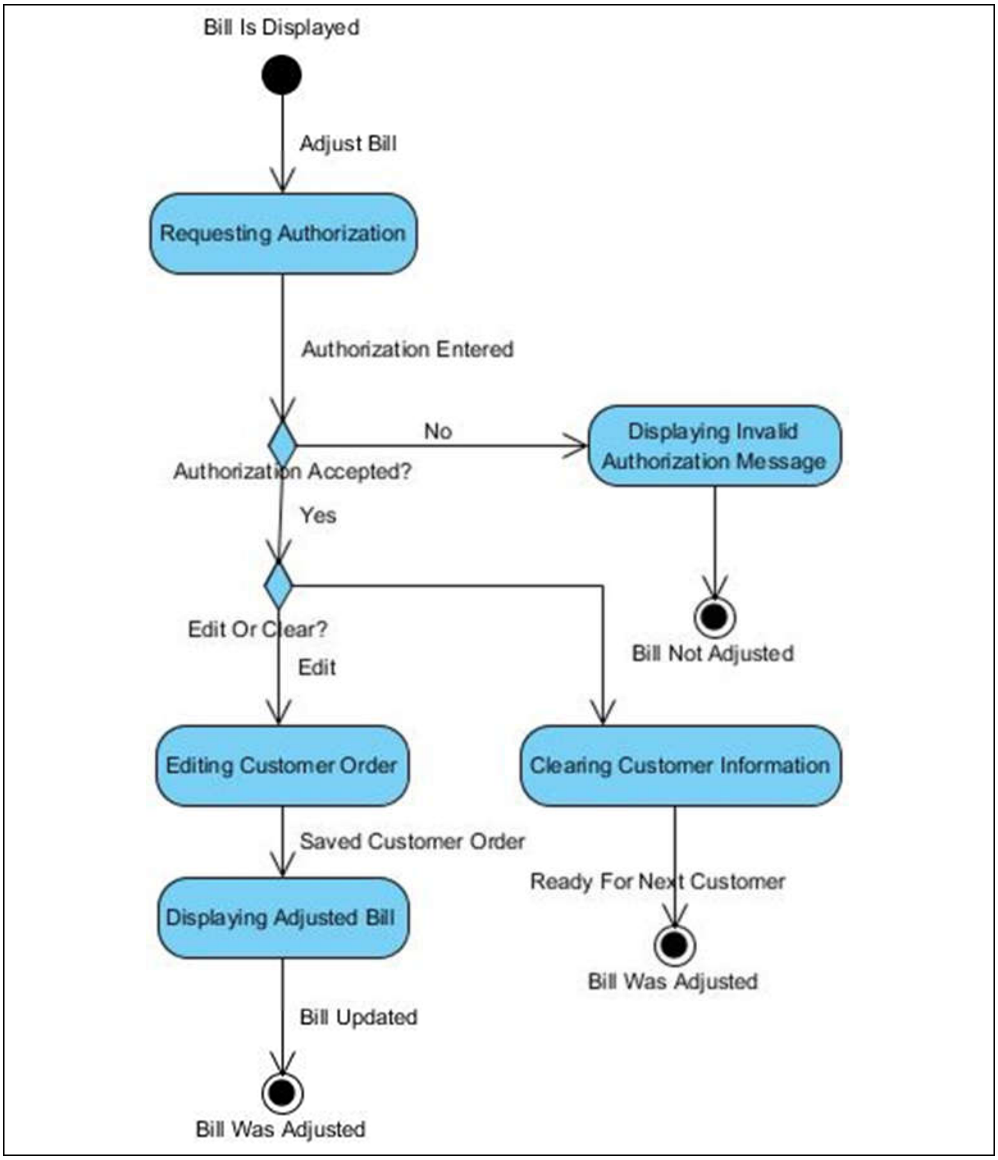


Figure 28 - Adjust Bill Activity Diagram

The following is a description of the steps in Figure 28.

Preconditions:

- The customer bill is displayed

Steps:

1. Wait staff requests to adjust the bill
2. The system requests authorization
3. The wait staff enters their authorization

- ♦ **I assume that the wait staff they have a pass key of some kind.**

---

4. The authorization is valid and the system displays and editable customer order
5. The wait staff edits the order and saves their changes
6. The system displays an adjusted customer bill

The use case ends.

Postcondition:

- The customer bill has been adjusted

A.1 Alternative Flow 1 –Authorization is invalid

7. At step 3 authorization is invalid
8. The system displays an error message

The use case ends.

Postcondition:

- The customer bill was not adjusted

- ♦ **Where the purpose for detailing business activity is to ensure that no business steps are missed. Likewise, the purpose of detailing system uses cases to ensure that all system functionality is captured.**

Detailing the use case assists with estimation of the work involved with its development. Breaking out alternate paths through the use case into separate flows helps to breakdown the use case into multiple user stories (where each user story can be completed in a single sprint).

User stories

The business analyst organizes the use case activity into sprint sized user stories. They may get assistance from the development team when estimating sizes, but they should have enough knowledge of the sprint capacity be able to understand how much work a single developer can perform in a single sprint.

- ♦ **Note that although a complete system use case should be delivered in a single release, the system use case functionality may be split over several user stories, each of which can be developed in different sprints.**

The Display Bill system use case main path is represented by this user story.

As a customer, I want to see the cost of items that I have ordered so far, so that I know the total cost of my bill at any time.

Each user story is documented in a user story template and it includes a link to the parent (epic) user story, a freehand textual description of the user story and acceptance criteria.

The overview field includes additional information that assists with the development of the user story. (Do not add extraneous information that is not relevant to development. For example, ask yourself if informing developers why this solution was chosen over an alternative bill payment method, is assisting developers or going to add confusion.)

There is exactly 1 user story per user story template.

A user story may (and normally will) include several acceptance criteria.

The Process Customer Payment use case is shown as broken into the user stories in Figure 29, Figure 30 and Figure 30. (Each user story is documented in a separate user story temple.)

Parent	Pay Bill Epic
Overview	Process Customer Payment (Main flow)
This user story captures the main flow of the Process Customer Payment use case. A customer who has ordered a meal makes a request to pay their bill.	
User Story	
As a	customer
I want	to be able to make payment for my order at any time
so that	I can pay my bill when I am ready.
Acceptance Criteria	
Given	that a bill is displayed to a customer
when	a Pay Bill request is received
then	the system will display a request for the customer payment method.
Given	that a payment request is displayed
when	payment information has been entered
then	etc...

Figure 29 - Main path user story

Parent	Pay Bill Epic
Overview	Process Customer Payment Request Assistance (Alternate flow)
User Story	
As a	customer
I want	to be able to get assistance
so that	I can ask questions about my bill

Acceptance Criteria	

Figure 30 - The customer does not accept the bill alternate path

Parent	Pay Bill Epic
Overview	Process Customer Payment Generate Receipt (Alternate flow)
User Story	
As a	customer
I want	to be able to email my receipt
so that	I can get an electronic copy of my receipt
Acceptance Criteria	

Figure 31 – Customer Requests Emailed Receipt Alternate Path

- ♦ **Although acceptance criteria are only shown for the Customer Pays Bill user story, every user story should be accompanied by at least 1 acceptance criteria.**

My preference is to write a user story for the main flow (happy path) through the use case, and add user stories for alternate paths. The main path is implemented first and alternate paths in the same or subsequent sprints. This allows a complete path from start to end state through the workflow, to be tested at the end of each sprint.

Acceptance Criteria

Acceptance criteria use the Gherkin format:

Given ..   When ..   Then ..

- Given – is a precondition that defines a current state of the system
- When – identifies an event that occurs while in that state
- Then – is the postcondition that describes the state of the system after the event has occurred

Each acceptance criteria specifies a condition that can be tested, by playing the role of an actor of the use case. Acceptance criteria are derived from the steps in the use case activity diagram. Each acceptance criteria describes an event shown on the activity diagram.

- The *When* statement captures the event that causes a transition from 1 state or action to a subsequent state or action. This is represented by the transition line from a precondition state to an action or from an action to another action or to a postcondition state.
- The *Given* statement describes what is happening during the action representing the precondition to the event.

- The *Then* statement describes what is happing during the action representing the post condition of the event.

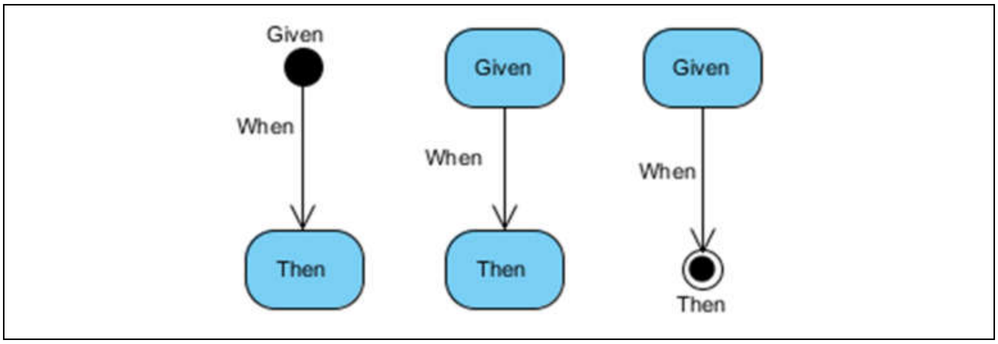


Figure 32 – Example Diagram Representations Of Acceptance Criteria

Figure 32 shows how acceptance criteria can be derived from a use case activity diagram. A start state or an action can represent the *Given* statement. A stop state or an action can represent the *Then* statement. The *When* statement is always represented by an event.

The format of an acceptance criterion is:

- Given (description of the start state or action)
- When (description of the event transition occurs)
- Then (description of the action or stop state)

♦ **If several transitions enter an action, each event may generate it own acceptance criteria. I.e. there may be several 'Given' statements for the same postcondition.**

An example acceptance criterion for the process customer payment use case is shown in Figure 33.

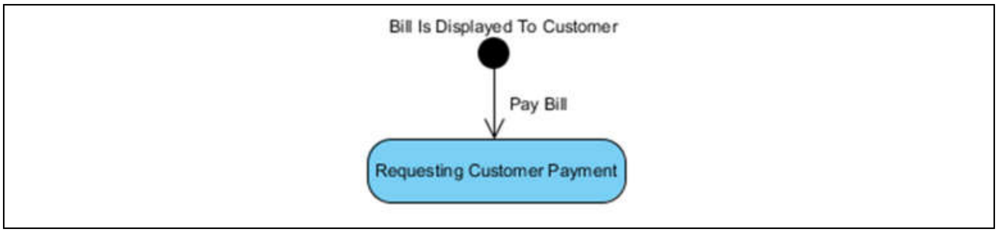


Figure 33 – Example Of An Acceptance Criterion

The following acceptance criterion is a textual representation of this Pay Bill event:

- Given that a bill is displayed to a customer
- When a Pay Bill request is received
- Then the system will display a request for the customer payment method

Decisions and Merges

If an event from the action enters a decision icon, then each transition out of that decision will generate separate acceptance criteria.

The *Given* statement for each acceptance criteria is the same (since the event exits from the same action in the diagram).

The *When* and *Then* statements vary because the decision generates several events each entering different actions.

Figure 34 shows 3 different acceptance criteria that share the same precondition (*Given* statement).

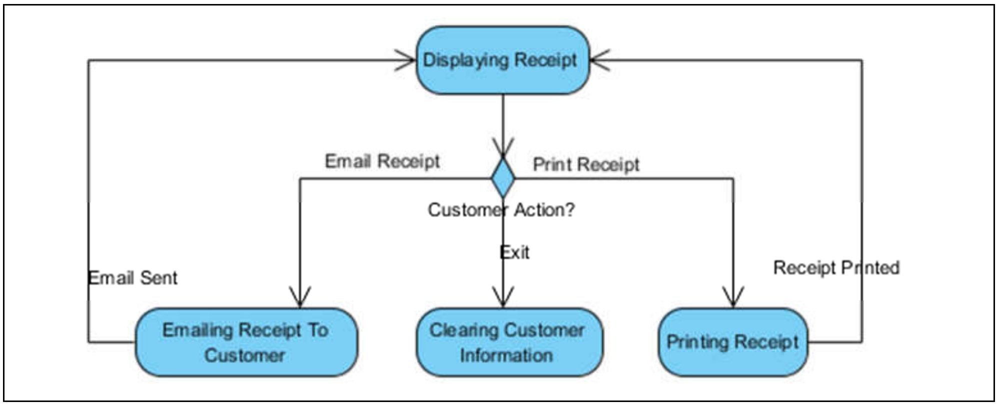


Figure 34 – Decision And Merge Example

- Given that the system is displaying a receipt to the customer:
- when the customer requests to email the receipt to their email address, then the receipt is emailed to the email address that was entered by the customer.

♦ **The user interface specification shows how a valid email address is entered into the system, it is not necessary to repeat that information in the acceptance criteria. The acceptance criteria should include reference to that UI specification.**

- when the customer requests to print their receipt, then the receipt is sent to the printer.

♦ **A reference to the system interface to the printer is included here.**

- when the customer exits the system, then all customer information is removed from the display and the main menu is displayed.

♦ **A reference to the UI for the main menu should be added here.**

Figure 34 also shows that emailing or printing the customer receipt returns the user to the receipt display. Displaying the customer receipt has 2 exiting events. These two events can be described with the following acceptance criteria.

- Given that the customer receipt is displayed, when the customer receipt was sent the entered email address, or the customer receipt was sent to the printer, then the customer receipt is redisplayed.

Forks And Joins

A fork in the activity diagram generates 2 acceptance criteria with the same precondition and same exiting event. The event splits into several paths, each with a different postcondition.

Figure 35 is an example of a fork taken from the Process Customer Payment system use case activity diagram.

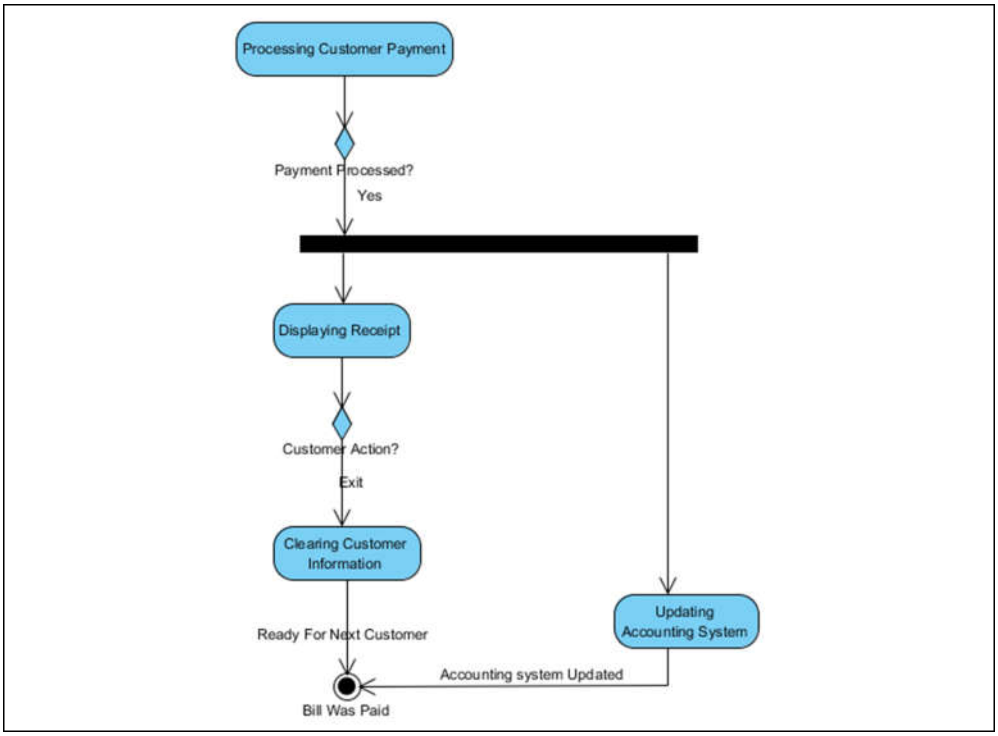

Processing Customer Payment
Payment Processed?
Yes
Displaying Receipt
Customer Action?
Exit
Clearing Customer Information
Ready For Next Customer
Updating Accounting System
Accounting system Updated
Bill Was Paid

Figure 35 – Fork Example

The fork occurs when payment is processed successfully. The customer receipt is displayed and the payment is sent to the payment processing system.

The 2 acceptance criteria for these 2 paths are:

- Given that customer payment is being processed by the payment processing system
- When customer payment was processed successfully
- Then the receipt for the customer payment is displayed to the customer
- Given that customer payment is being processed by the payment processing system

- When customer payment was processed successfully
- Then a record of the transaction is sent to the restaurant accounting system.

- ♦ **A reference to the accounting system interface specification should be added here.**

Both acceptance criteria may be combined, so that the *Given* and *When* statements are not repeated.

- Given that customer payment is being processed by the payment processing system
- When customer payment was processed successfully
- Then the receipt for the customer payment is displayed to the customer

And

- Then a record of the transaction is sent to the restaurant accounting system

Both paths rejoin when the customer has left the table and when system has finished processing payment for the previous customer at that table.

- Given
    - that the accounting system is processing a customer payment, when process is complete
- And given
    - the system is clearing the previous customer information, when it is ready for the next customer
- then the bill was paid and the next customer may start ordering

7.4.1.3 Summary

Eliciting business needs is generally the most important activity that is the responsibility of the business analyst. Errors that occur during elicitation may propagate through the complete development process.

Elicitation starts by interviewing stakeholders in order to capture business needs. These business needs are documented with epic user stories. An epic is analyzed to drive a business process. The business process is captured with a business use case. The business analyst works with subject matter experts to identify opportunities for automation. System use cases are identified and these capture the to-be automation of the business process. The business process steps that are to be automated are mapped to the system use cases. System use cases are detailed with system activity diagrams. User stories are created from paths through the system use case activity. User stories breakdown use case functionality into manageable chunks. Finally, acceptance criteria are derived from the events captured in the activity diagram. These are used by quality assurance to create test cases.

The state of user story elicitation is determined by the priority of the parent epic. Ideally, user stories are detailed just in time for the next sprint. However, my experience is that it can take several sprints to prepare a user story, by which time its priority may well have changed. This is why I recommend as many as 2 sprints worth of user stories are made ready-for-dev and available for the sprint planning activity. (I.e. the business analyst tries to work 2 sprints ahead of the development team.)

You may have noticed that there is a lot of overlap and duplication between detailing the system use case and documenting the acceptance criteria. If the quality assurance team is happy with working from the system use cases and their activity diagrams, you can skip writing textual acceptance criteria. Quality assurance can derive test cases directly from the activity diagrams.

♦ **I find that writing out the use cases as text is a great help in finding omissions and ambiguities in the activity diagram.**

Documenting the system use case steps is probably the most time consuming part of elicitation, it is the most detailed activity, and the steps in the activity diagram are liable to change often. Because of this, ideally I want a tool that allows me to document the system use case activity and automatically create acceptance criteria from the diagram

---

♦ **This tool does not exist.**

In reality, the diagrams are created in one (modeling) tool and the user stories are created in another (backlog) tool. In my experience, these tools do not communicate. Hence, there is going to be some manual duplication in order to keep both tools up to date. This is why acceptance criteria should be written just in time for the next sprint.

In the absence of an integrated single tool that offers both diagramming and prioritized user stories, I recommend using a 3rd tool to capture all the system use cases and related user stories in one place. This content management tool needs to support sharing of diagrams (from the modeling tool) and hyperlinks to user stories (from the backlog tool). Document all use case details and user story acceptance criteria in the content management and content sharing tool. Keep the user story information in the backlog to a minimum. Instead of putting information into the user stories, use links to the content management tool wherever possible.

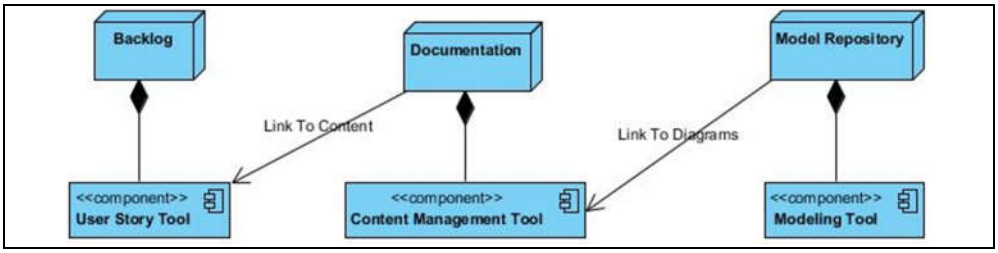


Figure 36 - Elicitation Tools Deployment

Figure 36 shows the arrangement of the 3 tools I use to manage user stories and use cases. User stories are stored in the backlog. The user story tool allows the user to view documentation from a user story. The content management tool manages documentation and user interface mockups. The content management tool can view diagrams from the modeling repository. Model diagrams are managed by the modeling tool. In this manner, a developer is able to open a user story in the backlog and by opening a link within that user story, view its documentation and user interface information in the content management tool. From the content management tool, the developer is able to view diagrams from the model repository.

7.4.2 Groom Backlog

Figure 37 shows the actors, inputs and outputs of the Groom Backlog activity.

Responsible Actor – Product Owner

Contributors – Business Analyst, Quality Assurance

Inputs – User stories and epics

Outputs – User stories and epics

Also known as backlog refinement meeting, the purpose of the Groom Backlog activity is to organize and clean the epics and user stories in the product backlog.

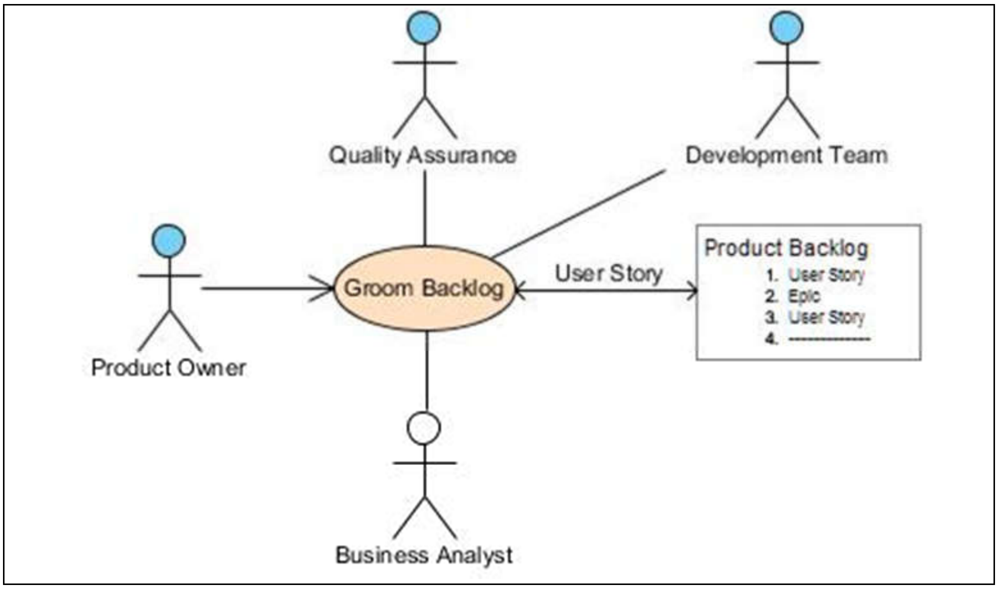


Figure 37 – Groom Backlog Use Case

The product owner is responsible for prioritizing and preparing user stories for upcoming sprints. The business analyst and development team support prioritization by estimating effort and complexity of user stories.

Starting with the highest priority user stories, the product owner identifies incorrect or missing information. The business analyst identifies what needs to be done to complete the user story, and moves on to the next story.

Quality assurance contributes information about defects and their impact on the current customer experience.

Ultimately, the product owner decides the order in which epics are to be delivered and which user stories are important and which are no longer needed.

Epics are updated as business needs change. Obsolete and duplicate user stories are removed, conflicting information is resolved and user stories updated. User stories are prioritized according to the order that returns most value while keeping the customer satisfied.

The priority of a user story may be calculated based on 4 factors:

- Value to the customer – What does the customer want done first?
- Cost to implement – Can the feature be put in the next release and how effort will it take to develop?
- Expected revenue – Is the feature going to be profitable?
- Risk – How sure are we that we know that implementing this user story is going to satisfy the customer?

♦ **In my experience, priorities are often dictated by the customer or company management, without the use of any figures on which to base those priorities.**

Because use cases are analyzed in isolation, it is possible that conflicting or duplicate user stories may be created. Backlog grooming will identify duplicate or conflicting user stories and remove or change those user stories, as appropriate.

7.4.2.1 Example

During the current sprint, the product owner prioritizes the user stories that have been derived from the Pay Bill feature. Stories are ordered with the highest priority on top of the list.

The product backlog is ordered as follows:

1. Display Bill To Customer main flow
2. Process Customer Payment main flow
3. Adjust Customer Bill main flow
4. Clear Customer alternate flow
5. Payment Failed alternate flow
6. Print Receipt alternate flow
7. Email Receipt alternate flow
8. Request Staff Assistance alternate flow

The business analyst and UI designer add detail to the Display Bill To Customer main flow user story. They then work through the list adding detail to use stories until enough user stories are ready for the next 2 sprints. This provides a buffer in case an epic slated for the next sprint is canceled.

7.4.2.2 Summary

Grooming the backlog ensures that high priority stories are not overlooked and that obsolete user stories are removed. Keeping the backlog to a minimum size reduces the amount of searching, sorting and filtering that needs to be performed.

The smaller the backlog, the less the cost of maintaining the user stories in the backlog hence lessening the chance of requirements being overlooked.

7.4.3 Maintain Requirements

Figure 38 shows the actors, inputs and outputs of the Maintain Requirements activity.

Responsible Actor – Business Analyst

Contributors – Quality Assurance

Inputs – User stories and epics

Outputs – User stories and epics

The purpose of the Maintain Requirements activity is to keep a record of the product functions and data, so that:

- The business analyst understands the changes to the software when new user stories are applied to the product.
- The business analyst is able to address questions about the current solution.

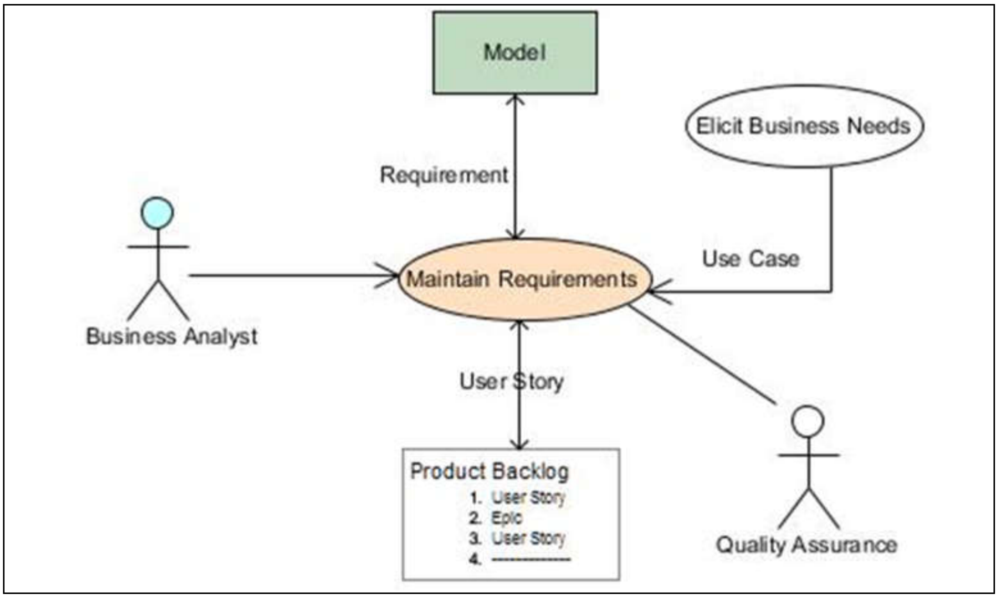


Figure 38 –Maintain Requirements Use Case

The business analyst is responsible for maintaining a model of the product. This model includes:

- a business use case view that captures the business needs
- a system use case view that captures the product requirements
- a logical view of the data used by the product
- a deployment view containing diagrams of the solution architecture

Maintain Requirements is a continuous low priority activity (much like a system batch job running in the background). However, the model is updated as user stories are analyzed as part of the Elicit Business Needs activity. This analysis causes use cases, activity diagrams, classes and non-functional requirements to be created or updated, as necessary. When a user story has been pulled into a sprint, the updates to the model from that user story are considered complete, and the business analyst works on a subsequent user story.

The business use case view is updated as new business needs are identified.

The system use case view is updated as user stories are added to the product backlog.

The logical view is updated as time and necessity permits.

The deployment view is updated as the architecture changes.

Any updates to the model that are caused by changes to user stories that are in development or done, can be given a low priority.

Updates to the model only need to occur as new user stories dictate (or defects are found). In this manner, the model is unlikely to ever be complete. However, the model should be good enough to be able to understand impacts to existing use cases, product functionality and data that are the result of new user stories.

Updates due to user stories that are scheduled for the next 2 sprints always take priority over other model updates.

The benefits to maintaining non-essential parts of the model, as time permits are:

- By maintaining the business view, the business processes are known.
- By maintaining system use cases, less time is spent determining changes to the existing functionality as a result of new user stories.
- By maintaining the logical view, the data used by system interfaces is known.
- By maintaining a deployment model, a picture of the system architecture can be shared with interested parties. This view is most useful to the deployment manager and operations.
- Quality assurance can use the system use case activity diagrams to create test cases.
- The writer can use the model to assist with user documentation.
- By maintaining traceability within the model, the business analyst understands the data that is impacted as a result of changes to system functionality. This ensures that no data elements or their dependencies are missed when a system use case is implemented.

The most benefit I get from maintaining a model is when I am addressing questions about the current product build. The business analyst should be the 'go-to' person for not only for questions about new product features, but also for questions about existing functionality. It is impossible to remember all product functionality. I use a combination of the model and the current product release to determine the product functionality. The model helps me locate where the answers to the questions can be found in the product. Using the product validates that my understanding is correct.

- **Most of the defects that I find are a result of comparing what the model states that the product is supposed to do, against using the product to see what it actually does.**

7.4.3.1 Logical View

I discussed the business and system use case views in the section on eliciting requirements. In this section, the logical view is discussed. The logical view is considered optional, because it not necessary to complete the user stories. However, the information in the logical view is very useful to understand the product.

The logical view captures the externally observable data used by the system and the relationships between that data. The benefit from maintaining this view is that a consistent set of data and the dependencies between that data are captured. This information is difficult (if not impossible) to determine from using the product.

- **An alternative is to find data dependencies in the code.**

The UI designer may find the data model useful when specifying field sizes and possible values for information that is displayed on the user interface.

Quality assurance may use this model to test valid values for data used by the product.

The data in the logical view can be derived from the system use case activity diagrams. This is achieved by analyzing system use case activity to identify the information used by that system use case.

- **The functionality for the high priority user stories takes precedence over updating the data in the logical view.**

For each action in the system use case activity diagram, identify what system data is impacted by this action. Capture this data by modeling it with class diagrams. These class diagrams persist throughout the life span of the product.

- **This is the most stable view of the model. In my experience, functionality changes much more frequently than the data used to support that functionality. Once a relationship is established between data elements, that relationship is rarely deleted.**

By maintaining a logical view of the data, less time is spent understanding impacts to externally observable information (such as user and system interfaces). A logical view of the model not only shows data, but also dependencies between that data. Capturing dependencies shows us what the impacts are to existing features when a change to a user story is introduced.

♦ **In my experience, changes due to unknown dependencies are the biggest cause of defects in the product. Especially as more and more features are introduced into the product.**

7.4.3.2 Example Logical View

Consider the Process Customer Payment system activity diagram shown in Figure 27. The system data that is impacted by this use case is shown by the Activity To Data Mapping diagram in Figure 39.

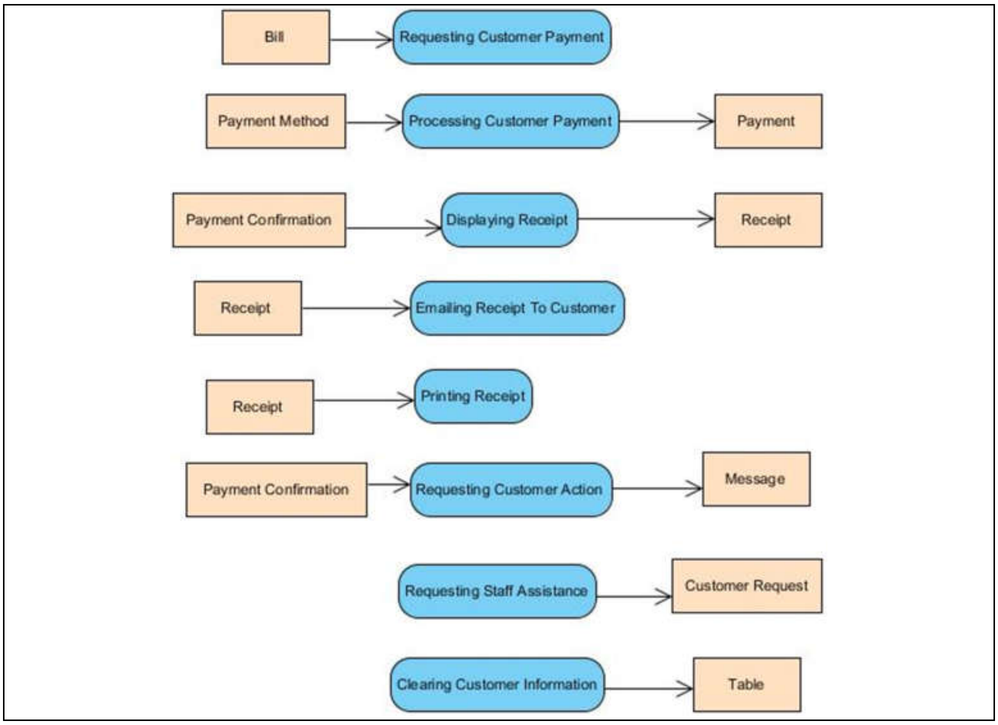

Bill
Requesting Customer Payment
Payment Method
Processing Customer Payment
Payment
Payment Confirmation
Displaying Receipt
Receipt
Receipt
Emailing Receipt To Customer
Receipt
Printing Receipt
Payment Confirmation
Requesting Customer Action
Message
Requesting Staff Assistance
Customer Request
Clearing Customer Information
Table

Figure 39 – Process Customer Payment Activity To Data Mapping Diagram

Create a system activity to data mapping diagram for the Process Customer Payment use case.

♦ **This is not a standard UML diagram, so I use an activity diagram.**

Place the actions from the Process Customer Payment activity diagram onto the activity to data mapping diagram.

For each action on this diagram, identify the data objects that are input to or output from the action. (In this example, many object classes have already been defined in the logical view.) If the data does not already exist in the logical view, then create a new object and connect it to the action. If a class already exists in the logical view, then put an object instance of that class on the diagram and connect it to the action.

- ♦ **I find arrowheads are useful to indicate whether the class data is an input to, or an output from the action, although this is not absolutely necessary.**

The mapping between the Process Customer Payment use case actions and system data is as follows:

- In order to request customer payment the bill is required. A Bill object is placed on the diagram to represent an instance of the Bill class. (The Bill class has previously been defined in the model).
- A Payment Method is needed to process customer payment. The Payment information is sent to the payment processing system. Payment Method and Payment objects are added to the diagram.
- A Receipt is produced in response to a Payment Confirmation being received from the payment processing system. The Receipt is emailed to the customer or sent to the printer. Receipt and Payment Confirmation objects are added to the diagram in the appropriate places.
- Payment Confirmation represents a message from the payment processing system in response to a processing request.
- A Message may be displayed to the customer. A Customer Request may be displayed to wait staff. The Message and Customer Request objects are added to the diagram.

Update The Logical View

Using the objects on the activity to data mapping diagram, the logical view classes are updated.

For each object on the diagram:

- Where the object is an instance of an existing class, update the class to capture any changes due to actions connected to the class instance.
- Where the object is not an instance of an existing class either, create a new class and give it attributes that reflect the information input to or output from each action to which it connects.

Once all the data has been captured, from the logical view open the class diagrams that were changed. Merge or split classes in order to make the diagram compatible with your best practice for specifying data (such as 3rd normal form).

Relationships are added between related classes, as required.

Figure 40 is class diagram in the logical view. It shows data as attributes of classes, and data dependencies as relationships between those classes.

- ♦ **The notation for the class diagram is described in Appendix A - .**

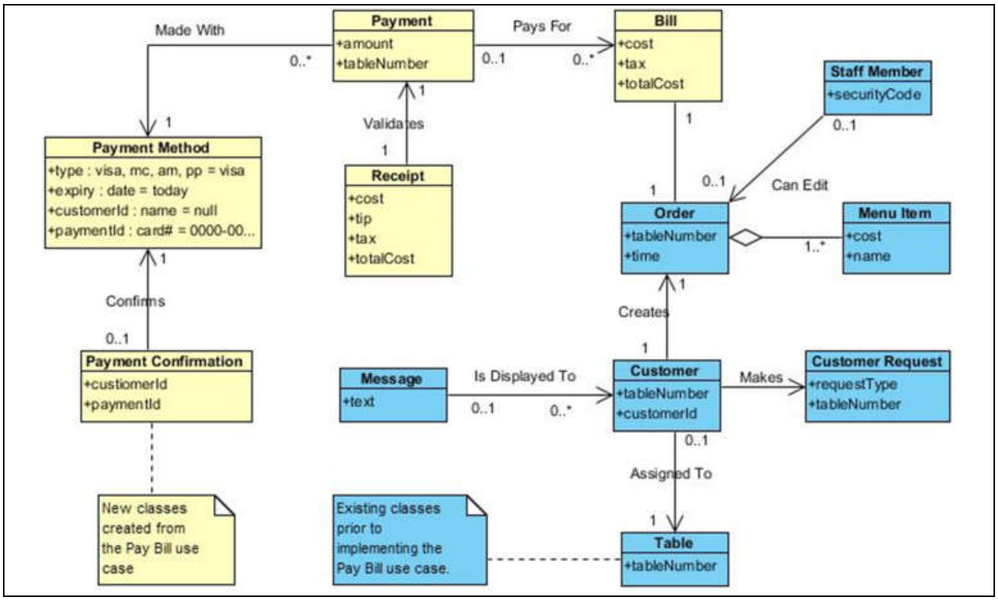


Figure 40 – Automated Menu Ordering System Updated Class Diagram

Classes with dark shading (blue) already exist in the logical view of the AMOS. Classes with a lighter shading (yellow) represent new classes that have been added to the logical view, as a result of analyzing data used by the Process Customer Payment use case.

Payment Method, Payment, Bill, Payment Confirmation and Receipt classes have all been added to logical view as a result of data shown on the activity to objects mapping diagram Figure 39.

The class attributes capture all externally visible data. Data is externally visible if it is visible on the user interface or passed through a system interface.

- ♦ **Only data that is visible to actors is captured in the logical view, where actors may be roles or systems.**

The amount of detail you want to capture depends on the benefit that the project gains in return for the effort of maintaining that data. I recommend that classes should include at least a named attribute for every item of data that is externally visible.

- ♦ **Note that data in a logical view is never hidden; hence the '+' symbol preceding each attribute.**

If your business analyst is comfortable with class diagrams, then you may find benefit in adding the following information to the logical view:

- Initial Value - defines the value shown on the user interface when that data element is first displayed

- Multiplicity – identifies how many of each class participate in the relationship
- Operations – can be used to represent operations that are performed on the data by a user or system interface
- Type – defines the possible values for each attribute (These can be useful to developers and testers.)

Figure 41 shows how these details are documented in the Payment and Payment Method classes on the menu system class diagram.

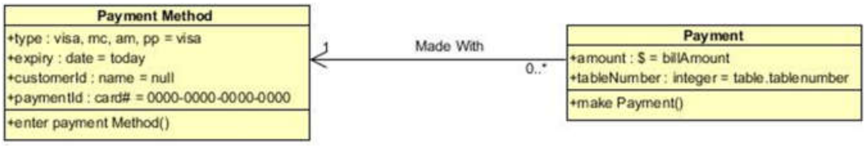

Payment Method
+type : visa, mc, am, pp = visa
+expiry : date = today
+customerId : name = null
+paymentId : card# = 0000-0000-0000-0000
+enter payment Method()
1
Made With
0..*
Payment
+amount : $ = billAmount
+tableNumber : integer = table.tablenumber
+make Payment()

Figure 41 – Detailed Class Diagram

♦ **This example shows how to populate class details and is not intended to represent good design.**

The types of payment method are Visa (visa), MasterCard (mc), American Express (am) or PayPal (pp). The initial value for payment method type is Visa (this is what is displayed to the user by default).

Payment method expiry is a date with an initial value of today's date.

The customer id is their name and it is initially displayed with an empty value.

The payment id is a 16-digit card number. The card number field is initially filled with zeros.

Payment amount is in dollars and the initial value that is displayed is the same as the dollar value of the bill.

The table to which the payment is applied is the customer's table number.

The relationship between the payment and the payment method shows that:

- the system allows a payment to be made with exactly 1 payment method
- the same payment method may be used to make many payments

Two operations have been identified:

- enter payment - a payment method is entered into the system
- make payment - a payment is made by the system (using the payment processing system)

Because the model is maintained, it is now possible to trace the data in the logical view, back to the use cases that use this data. This allows us to determine what existing information is going to be impacted by upcoming user stories.

- **New user stories may impact existing use cases. Using the use case activity to data mapping diagram, the data that is affected by a changing use case can now be mapped back to the user story that caused those changes.**

7.4.3.3 Traceability

Traceability is much like dependency (in that it is a relationship between model components), such that if 1 component changes then there is a potential impact to the related components. The difference between dependency and traceability is that, whereas dependency relationships are across components at the same level of abstraction, traceability relationships flow from a component at a higher level of abstraction to components at a lower level of abstraction.

A relationship between two classes in the logical view is a dependency, and a relationship between an action in a use case diagram and data that is realized in the logical view, is traceability.

By maintaining traceability between the model components, it can be determined what parts of the product are impacted by a feature change.

Figure 42 is a traceability diagram, showing the dependency links between different component types of the model.

Composite relationships indicate a parent child relationship. Aggregate relationships indicate that a traceability link is required to maintain the relationship.

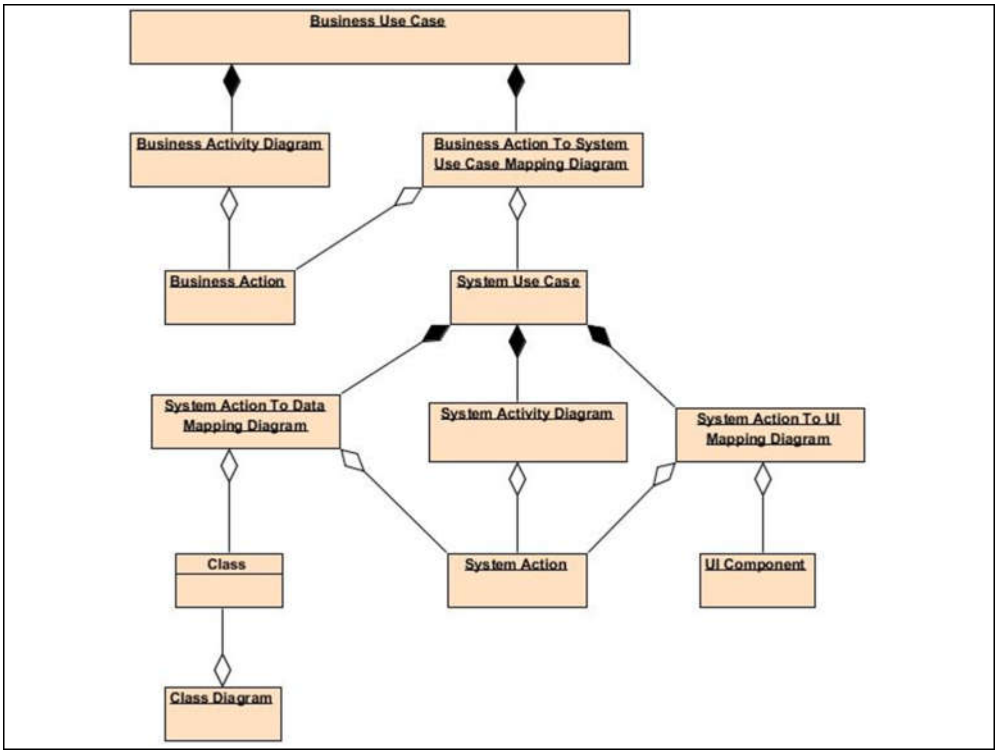

Business Use Case
Business Activity Diagram
Business Action To System Use Case Mapping Diagram
Business Action
System Use Case
System Action To Data Mapping Diagram
System Activity Diagram
System Action To UI Mapping Diagram
Class
System Action
UI Component
Class Diagram

Figure 42- Traceability Diagram

Traceability is maintained throughout the model as follows:

- Business Use Case -> Business Activity Diagram
  This is a parent child relationship that is supported by all UML modeling tools.
- Business Activity Diagram -> Business To System Mapping Diagram
  The business to system mapping diagram is a child of the business use case. This relationship is also supported by UML modeling tools.
- Business Activity Diagram -> Business Action
  The business use case view shows the connections between business actions on the business activity diagram.
- Business Action -> System Use case
  This traceability is captured within the business action to system use case mapping diagram.

♦ **Since no tool to my knowledge explicitly supports the mapping diagrams, the traceability is maintained manually.**

- System Use Case -> System Activity Diagram
  This is a parent child relationship is supported by all UML modeling tools.

- System Activity Diagram -> System Action
  The system use case view shows the connections between system actions on the system activity diagram.
- System Activity Diagram -> System Action To Data Mapping Diagram
  The diagram can be made a child of the system use case. The relationship is supported by UML modeling tools.
- System Activity Diagram -> System Action To UI Mapping Diagram – The activity to UI mapping diagram can be made a child of the system use case. This relationship is supported by UML modeling tools.
- System Action -> Class
  System actions are mapped to classes in system action to data mapping diagrams.
- System Action -> User Interface
  System actions are mapped to interfaces in system action to UI mapping diagrams.
- Class -> Class Diagram
  The logical view shows relationships between classes in the class diagram.

These traceability links are generated as a result of the following the Quality with Agile through Pictures process. Unless mandated, I recommend maintaining the manual trace links for features only as they are modified by new user stories. Other traceability relationships are automatically maintained as features are updated.

♦ **If I discover a broken link or out of date model element while investigating a question about the product, I will normally fix the problem at that time.**

7.4.3.4 Summary

The model adds quality to the product by helping to ensure that requirements pulled into development are:

- Accurate – include the right information
- Complete – nothing is missing
- Consistent – do not contradict existing requirements
- Necessary - satisfy a business need
- Verifiable – can be tested

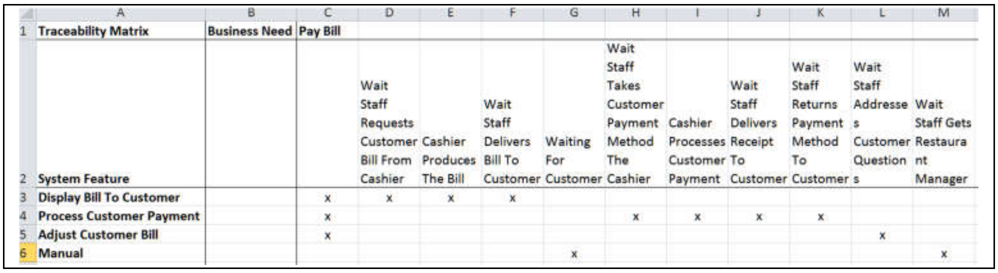
Traceability provides the ability to assess the impact of a change to the model. For example, the impact of a new business need can be assessed by following the traceability links from the impacted business use case through to system use cases and to the logical model.

When traceability is mentioned, the common reaction is to imagine a matrix (using a spreadsheet tool such as Excel).

- ♦ **At the time of writing, searching for traceability on youtube only returns videos showing traceability matrices.**

This process uses diagrams to manage traceability. Diagrams take up more space, and traceability is spread over several diagrams. However, traceability diagrams are less work to maintain, easier to view and fall naturally from the process.

Consider the following traceability matrix between business needs and system features.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Traceability Matrix**	**Business Need**	**Pay Bill**										
2	**System Feature**			Wait Staff Requests Customer Bill From Cashier	Cashier Produces The Bill	Wait Staff Delivers Bill To Customer	Waiting For Customer	Wait Staff Takes Customer Payment Method The Cashier	Cashier Processes Customer Payment	Wait Staff Delivers Receipt To Customer	Wait Staff Returns Payment Method To Customer	Wait Staff Addresse s Customer Question s	Wait Staff Gets Restaura nt Manager
3	**Display Bill To Customer**		x	x	x	x							
4	**Process Customer Payment**		x					x	x	x	x		
5	**Adjust Customer Bill**		x									x	
6	**Manual**						x						x

Figure 43 – Example Traceability Matrix

Figure 43 shows the business to system mapping diagram of Figure 24 represented in an Excel spreadsheet.

- ♦ **You decide which is easier to read and maintain, Figure 43 or Figure 24**

7.4.4 Design Architecture

Figure 44 shows the actors, inputs and outputs of the Design Architecture activity.

Responsible Actor – Solution Architect

Contributors – Business analyst, Development Team, Quality Assurance

Inputs – Requirement

Outputs – User Story, Component

The purpose of Design Architecture is to maintain a system architecture and design that satisfies current and future system requirements.

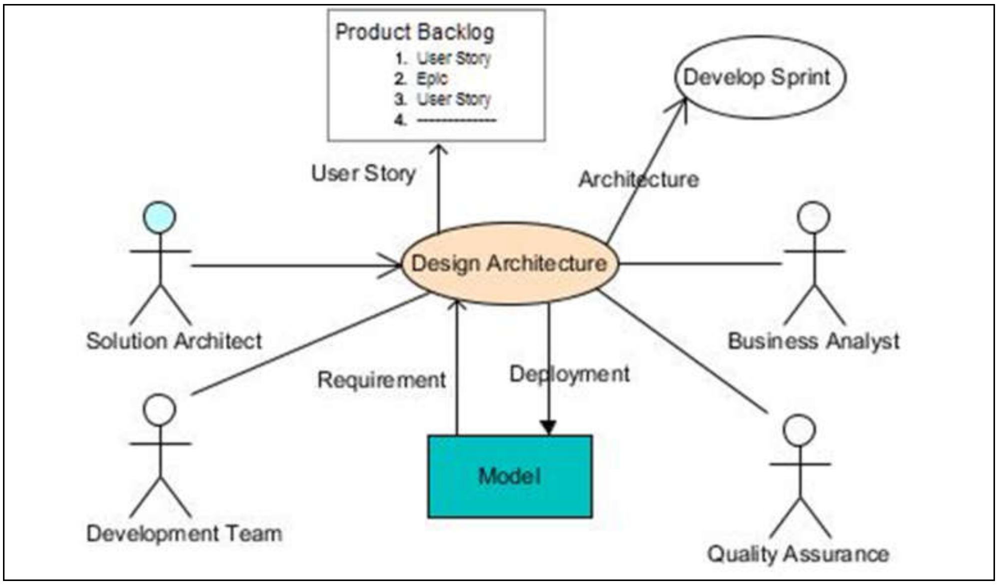


Figure 44 – Design Architecture Use Case

The output from this activity is a set of hardware and software components that comprise a foundation on which the product is built.

The impacts due to system architecture changes must be fully understood by the development team. The development team will also contribute to this activity with their recommendations.

The business analyst may need to update user stories in the product backlog that reflect changes occurring as a result of new technology. The business analyst updates the deployment view in the model to reflect changes in the system architecture.

Quality assurance will need to understand how architecture changes are going to impact their existing test cases.

Changes to the system architecture are captured by user stories. These user stories identify the software components that are impacted and the reason for making the change. Components may change as a result of the following:

- New requirements – such as changes to external system interfaces
- New technology – existing commercial software components may no longer be supported, or an upgrade reduces the development effort of future software builds
- Upgrades to system hardware – the system hardware is updated to meet future non-functional requirements
- Refactoring the software design – existing software is restructured in order to make maintenance more efficient

- ♦ **Refactoring refers to the restructuring of software without changing its behavior, in an attempt to reduce software maintenance and increase system performance. Architecture and refactoring user stories have similar origins (they originate from within the project team and I generally do not trace them to a specific business need), therefore I have included both types under the Design Architecture activity.**

Any changes to the design must still meet the functional and non-functional requirements, which are captured in the model of the system.

The solution architect is responsible for solution design changes.

The development team is responsible for changes due to code refactoring.

The resulting user stories are estimated and pulled into a sprint during sprint planning. Architecture changes are timed such that the work performed by the solution architect does not disrupt the development sprint cycle.

In Scrum all architectural design work is included as part of the user stories that are taken in during sprint planning. This activity assumes that the architecture design is performed externally before being submitted in user stories.

Once development starts, the solution architect still works with the development team to tweak the design. The flow labeled Architecture between the Design Architecture and Develop Sprint activities indicates that the architecture may still be modified during the sprint.

7.4.4.1 Design Architecture Example

The solution architect identifies changes to the product system design in order to accommodate the Pay Bill feature. This includes the introduction of a payment system interface. Figure 45 is a deployment diagram showing the interfaces to the AMOS.

- ♦ **The notation for the deployment diagram is described in Appendix A - .**

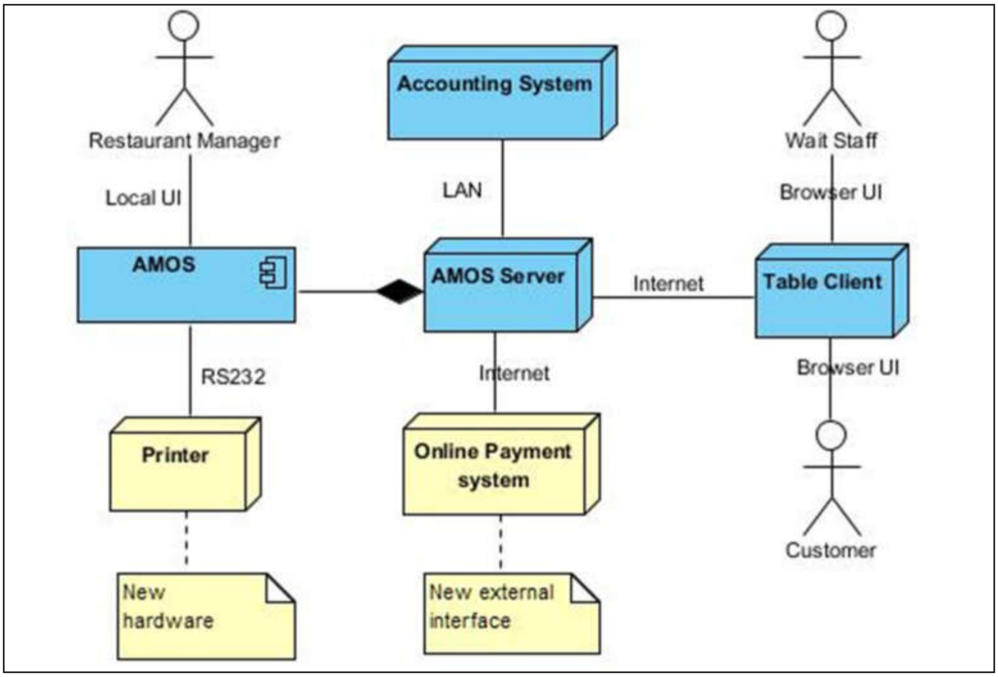


Figure 45 - Automated Menu Ordering System Deployment Diagram

Components shown with darker shading (blue) represent architectural components that make up the existing AMOS system. Components shown with the lighter shading (yellow) represent new interfaces to be added to the existing architecture.

The diagram shows hardware and software components of interest. (I indicate software components with the component icon and hardware components with the node box). Hardware connections are labeled with the interface technology connecting the systems.

Connections between software and hardware indicate the where software components are hosted.

Software to software interfaces may be shown if useful information is conveyed by the connection.

- **As with any other type of documentation, only show information if there is an identified audience that finds it useful.**

Notes are added to the diagram for clarification.

The solution architect has introduced an interface to an Online Payment Processing system. This system is where secure customer payments may be made over the internet. (This payment processing system provides the ability to send money using any major credit card or PayPal.) The interface to the online payment processing system is over the internet.

A printer has also been added to the system, so that customers may obtain a hard copy of their receipt from the system at their table. The printer interface is standard RS232.

- **Since I do not claim to be a system architect, there is no reason to believe that this architecture is in any way optimal, or even accurate. In fact, a diagram maintained by a solution architect might include a lot more detail about Commercial Off The Shelf (COTS) software, operating systems, and port numbers.**

The diagram also shows which stakeholders have a user interface to which systems. (This is not standard notation for a deployment diagram, but some people may find it useful.)

7.4.4.2 Summary

Well-defined and documented solution architecture encourages quality by providing a common framework upon which all development is based. Changes to the solution architecture occur in one place and are propagated across the whole solution (instead of being implemented in a piecemeal fashion).

7.4.5 Define User Experience

Figure 46 shows the actors, inputs and outputs of the Define User Experience activity.

Responsible Actor – The UI Designer

Contributors – Product Owner, Business Analyst, Quality Assurance

Inputs – User Story

Outputs – User Interface Design, User Story

The purpose of Define User Experience activity is to specify changes to interface components. A product UI design is included with the user stories that impact the user interface.

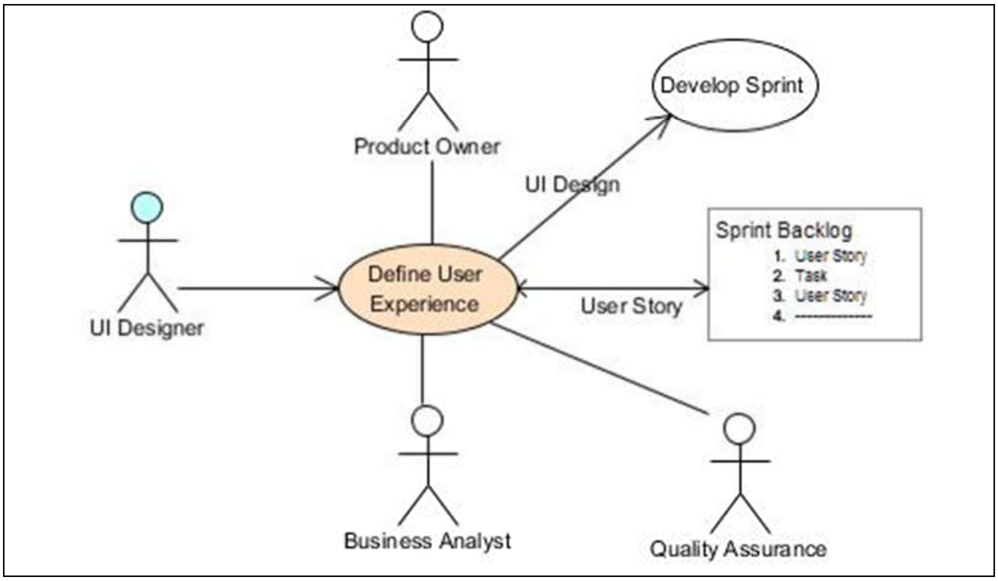


Figure 46 - Define User Experience Use Case

The UI designer provides guidelines for screens that are created or changed as a result of user stories. Guidelines are often in the form of mockups of a screen, or screen components and instructions for navigation between those components. The mockups include size, colour, font and position of text and components that are on the screen.

The product owner provides guidance and their opinion of the design.

The business analyst updates the user story descriptions to include additional information about the UI design.

Quality assurance uses information about the screens to add detail to test cases.

Scrum recognizes UI design as part of the sprint with developers designing and building the user interface. With an external UI designer mocking up screens instead, the user story is blocked from completion until the UI design is complete. This is why I recommend that user interface mockups are included with the user story in time for sprint planning. However, the UI designer should be working with developers to make tweaks during the sprint. The UI design is shown as a direct input to the Develop Sprint activity to indicate that changes may still be happening during the sprint.

7.4.5.1 Process Customer Payment Example

In order to understand the changes to the product user interface due to the Process Customer Payment use case, consider the actions on the use case activity diagram. Each action may have an impact on the product's user interface. For those actions, map them to a user interface components that are used to fulfill that action.

Figure 47 shows the actions from the Process Customer Payment use case and maps them to user interface screens.

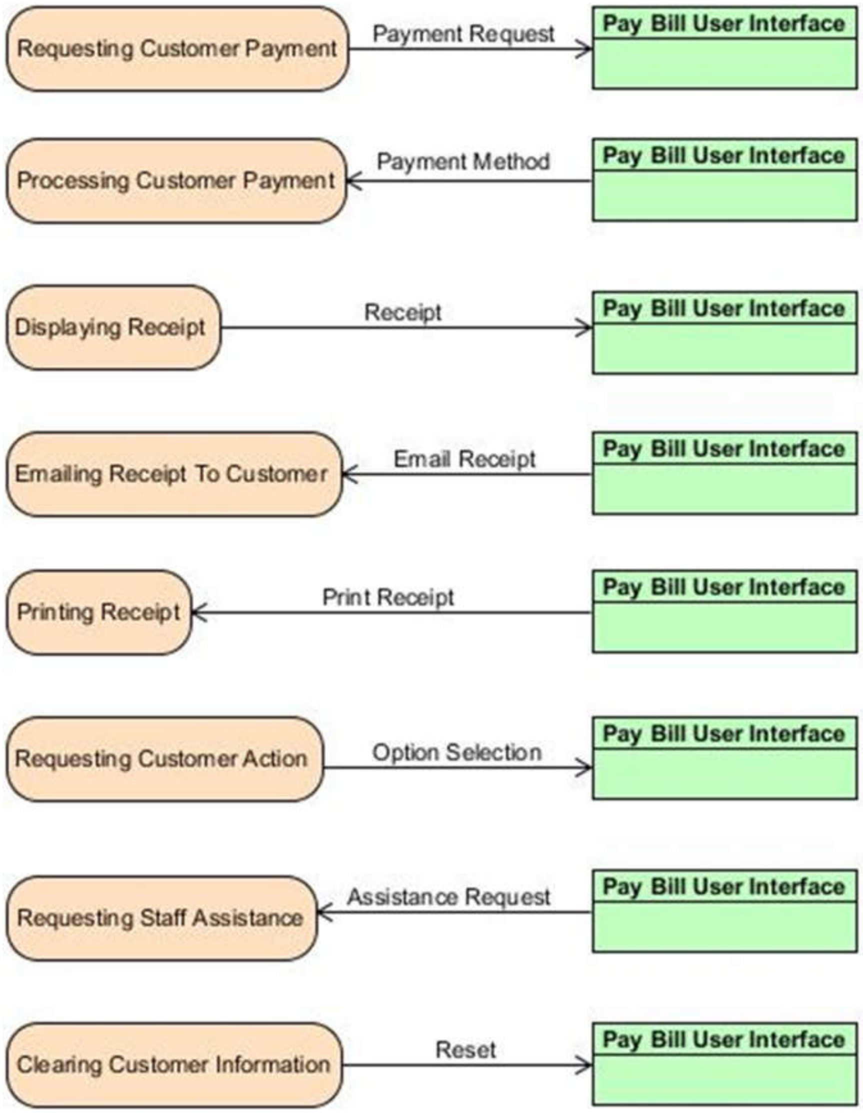


Figure 47 – Example Activity To User Interface Mapping Diagram

The activity to user interface mapping diagram is created as a child of the Process Customer Payment use case. There is no standard diagram type for this purpose, so I create an activity diagram.

The actions included in the system use case activity are placed on the mapping diagram. For each action identify the user interface components that provide input to the action, and the user interface component that is updated by the action.

Add objects to the diagram to represent the user interface components and connect them to the appropriate action. The connector between the action and the user interface:

- is given a direction to indicate whether the UI is being read from or being written to
- is labeled with a name that indicates the screen component that is being used

Sequence Diagrams

Sequence diagrams are another useful way to represent the primary and secondary actor's interaction with the product's user interface. Each action is captured by an event between an actor and a product screen.

Figure 48 shows the user interaction with the main path of the Take Customer Payment use case.

♦ **The notation for the sequence diagram is described in Appendix A - .**

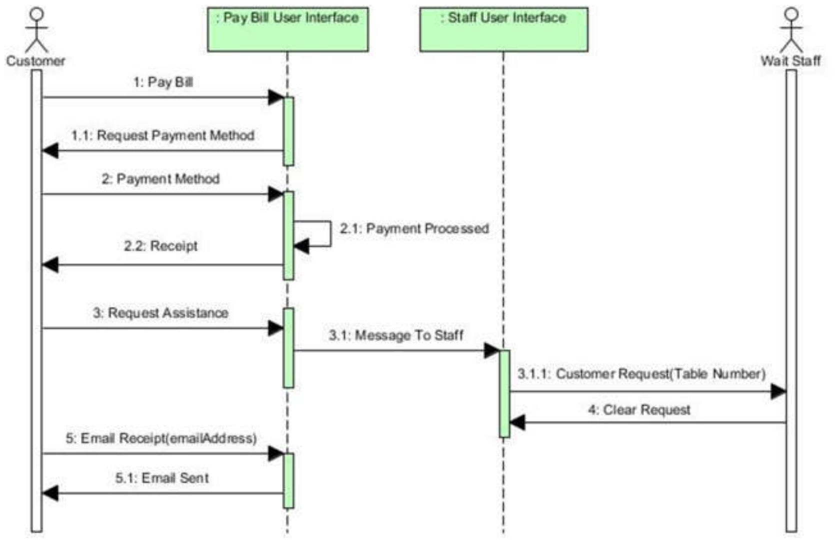


Figure 48 – Take Customer Payment User Interface Sequence Diagram

The sequence diagram is created as a child of the Take Customer Payment system use case. (This diagram uses standard UML notation.)

The lifelines (running vertically down the page) represent actors or user interfaces of the product. In this use case, the customer is the primary actor and wait staff a secondary actor.

♦ **The sequence diagram connects classes. In UML, an actor is a type of class. To add user interfaces to the diagram, you can create a class to represent each user interface.**

The flow of events through the diagram is read in sequence from top to bottom.

- The Customer makes a pay bill request from the Pay Bill screen. The system displays a payment method request to the Customer.
- The Customer enters their payment method details into the Pay Bill screen. (Some system processing happens on the backend at this point.) The system is informed that the payment was accepted and the receipt is displayed to the Customer.
- The Customer requests assistance from the system and a message is displayed on the Staff User Interface screen that includes the customer table number. The Wait Staff views the customer request. The Wait Staff clears the request from the Staff User Interface.

The system use case sequence diagram shows one or more paths through the use case. Each path is drawn as a separate timeline on the diagram.

Each path starts with an external event being recognized by the system. These events connect lifelines, shown vertically in the diagram. Each received event cause one or more events to be generated, until the flow ends. The sequence of events is read from the top down as they are displayed in the diagram. (The sequence can never go backwards.)

♦ **UML includes extensions to sequence diagrams that allow for the representation of loops, alternate paths and several other components that add complexity to the diagram. I keep analysis sequence diagrams as simple as possible, because I want everyone on the team to get the same understanding of the diagram, and I want maintenance of the diagram to be kept to a minimum.**

New paths can be shown on the same diagram as a sequence of events that is disconnected from previous events. Again, they are initiated by an external event.

An internally generated event (such as Payment Processed) represents actions that are performed internally by the system. Their details are not discussed on the diagram.

7.4.5.2 Summary

A common user interface design provides quality by sharing a consistent look and feel across all company products. This gives the impression of a professional organization, makes it easier for users to switch products and reduces the learning curve for customers.

7.4.6 Plan Sprint

Figure 49 shows the actors, inputs and outputs of the Plan Sprint activity.

Responsible Actor – Development Team

Contributors –Product Owner, Business Analyst, Quality Assurance

Inputs – User Story

Outputs – User Story

The purpose of the plan sprint activity is to populate the sprint backlog with a prioritized set of user stories that will be developed during the sprint.

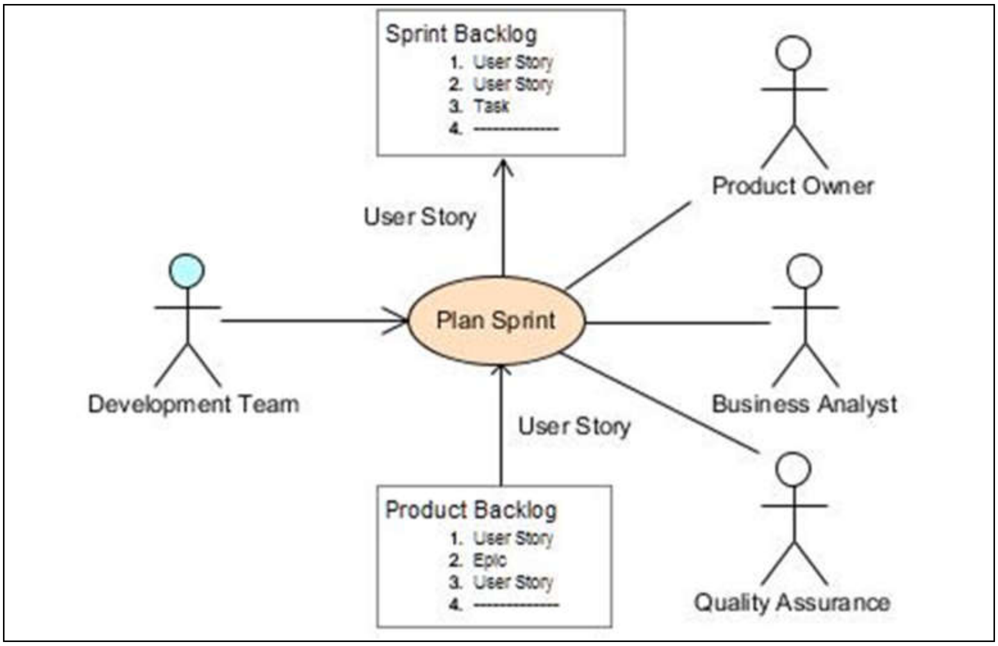


Figure 49 - Plan Sprint Use Case

The development team is responsible for scheduling a meeting to plan the next sprint. This meeting occurs immediately before the start of the sprint. Its objective is to populate the sprint backlog with user stories that will be developed in the next sprint. The product owner decides on the priorities of user stories that are candidates for the sprint backlog and sets the goals for the sprint. The business analyst assists with estimation of the work involved and complexity of each user story. Quality assurance provides information about any defect user stories that are considered for the sprint.

- ♦ **Although the development team is responsible for the success of sprint planning, it is often the Scrum master that schedules and organizes attendees for the meeting. In the absence of a Scrum master, I may be tasked with preparing sprint planning meetings.**

The details of the Sprint planning activity are out of scope for this book. For more information, see references to the Scrum sprint planning ceremony.

7.4.7 Develop Sprint

Figure 50 shows the actors, inputs and outputs of the Develop Sprint activity.

Responsible Actor– Development Team

Contributors – Business Analyst, Product Owner

Inputs – User Story, Architecture, UI Design

Outputs - Component

The purpose of the develop sprint activity is to create an incremental software build that for potential release to the customer.

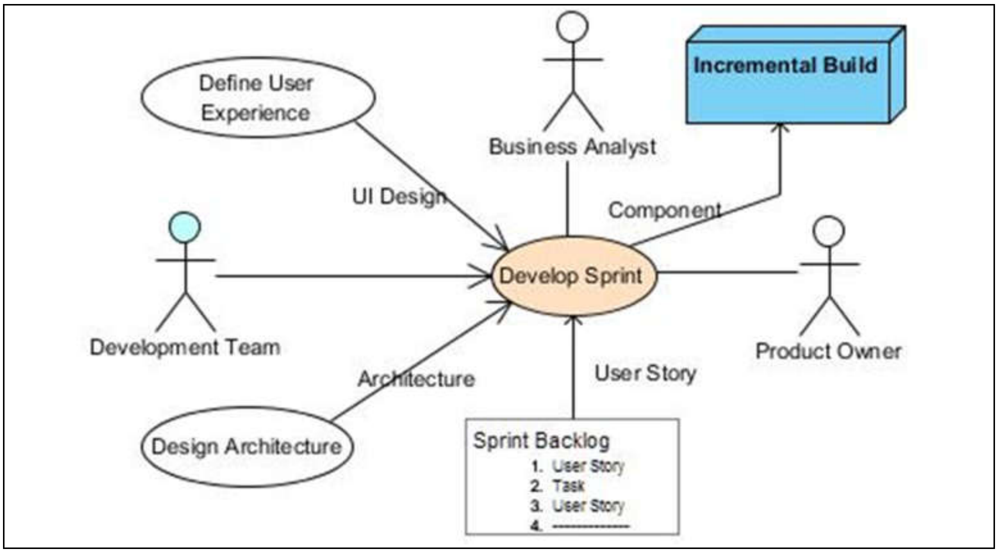


Figure 50 – Develop Sprint Use Case

The development team delivers a software build from the user stories in the sprint backlog. The business analyst and product owner provide support this activity as required, by answering questions about the user stories. The business analyst and product owner may change the user stories in order to clarify their intent, but any changes should not change the work effort that was estimated during sprint planning.

The architecture and UI Design are shown as external inputs to this activity to indicate that the UI designer and solution architect work with the development team during the sprint.

Develop Sprint is a time-boxed activity, ideally with the same development time allowed for every sprint.

The details of the Develop Sprint activity are out of scope for this book. For more information, see references to agile software development.

7.4.8 Review Sprint

Figure 51 shows the actors, inputs and outputs of the Review Sprint activity.

Responsible Actor – Development Team

Contributors – Business Analyst, Product Owner

Inputs – Incremental Build

Outputs – User Story

The purpose of the Review Sprint activity is to allow stakeholders to review the work that was performed in the previous sprint and modify the product backlog from reviewer feedback.

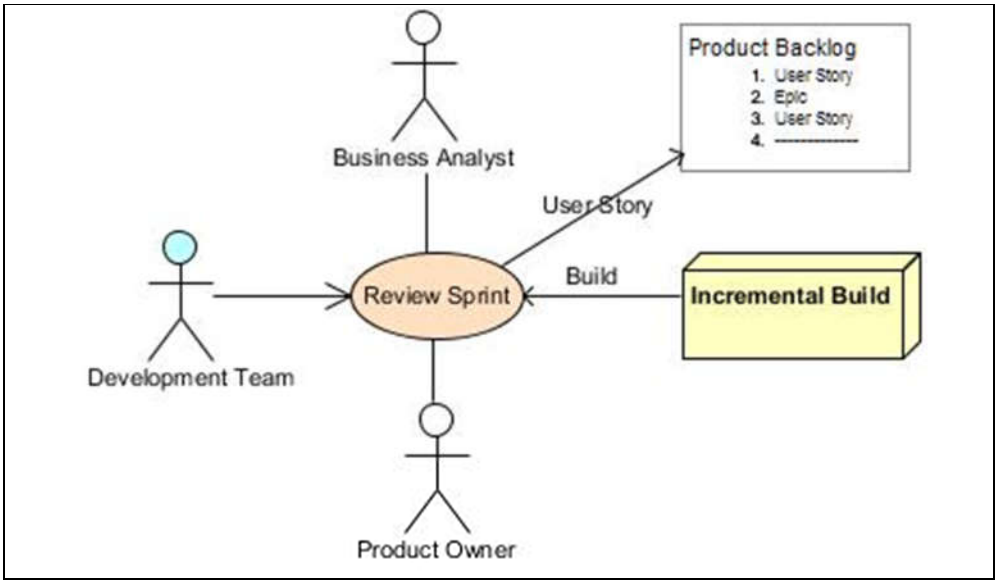


Figure 51 - Review Sprint Use Case

The development team schedules a review of the incremental build in order to demonstrate what was developed in the previous sprint.

This demonstration is given to any interested stakeholders. The business analyst and product owner attend this meeting and provide support by explaining the user stories that were developed. New user stories may be created and existing stories updated, from stakeholder feedback.

- **Scrum.org states that 'key stakeholders' are invited to this meeting. My experience is that anyone with an interest in the success of the product is welcome to attend the demonstration.**

The details of the Review Sprint activity are out of scope for this book. For more information, see references to the Scrum sprint review ceremony.

7.4.9 Learn Lessons From Sprint

Figure 52 shows the actors, inputs and outputs of the Learn Lessons From Sprint activity.

Responsible Actor – Development Team

Contributors – Business Analyst, Quality Assurance, Product Owner

Inputs – User Story

Outputs – User Story

- **Scrum defines the responsible role for the lessons learned meeting to be the Scrum master. I have never worked on an agile project where the Scrum master hosted any development team meetings. This meeting may be scheduled by any role on the project team, but it is the development team is shown as gaining benefit from the meeting.**

The purpose of the Review Sprint activity is to identify opportunities for improvement in future sprints.

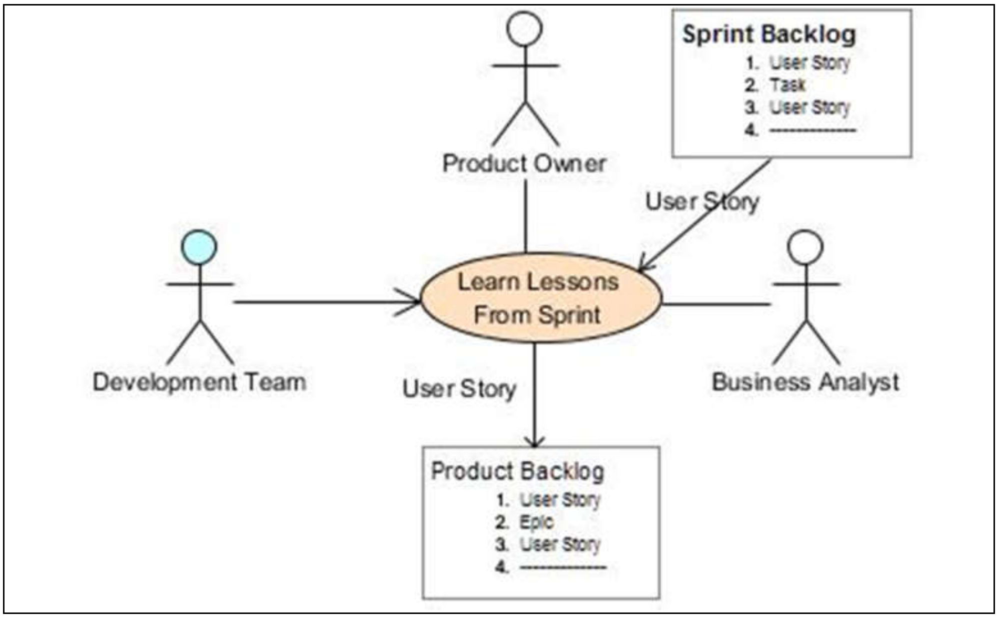


Figure 52 - Learn Lessons Use Case

A representative of the development team conducts a meeting where the project team discusses lessons learned from the previous sprint. Anyone involved with sprint development may contribute to this meeting, including quality assurance, the business analyst and product owner. A representative from the development team (often the team leader) hosts the meeting. Action items from the previous meeting are reviewed and closed, as appropriate. Attendees write down what went well and what could be improved and new action items are taken from the meeting. User stories may be updated or created in the product backlog to capture the results of the meeting.

The outputs from this meeting are action items in the form of user stories.

The agenda for the sprint retrospective meeting follows the following broad outline.

- What worked well the last sprint that we want to continue doing?
- What didn't work well the last sprint that we should stop doing?
- What should we start doing to improve the process?

The user stories that are the result of the meeting can be taken into a future sprint (if they affect developer time) otherwise they can remain in the product backlog and assigned a due date.

- ♦ **Many articles about the sprint review meeting recommend that a separate issue backlog is not maintained. This is, because causes confusion over priorities and may cause the action items to be forgotten. This is why they share the product backlog with development user stories.**

The details of the Learn Lessons From Sprint activity are out of scope for this book. For more information, see references to the Scrum sprint retrospective ceremony.

7.4.10 Test Build

Figure 53 shows the actors, inputs and outputs of the Test Build activity.

Responsible Actor– Quality assurance

Contributors – Business Analyst, Product Owner

Inputs – Acceptance Criteria, User Story

Outputs – Test Case

The Test Build activity is concerned with validating that an incremental software build meets its requirements. The results of this activity will help determine whether a software build can be considered for production.

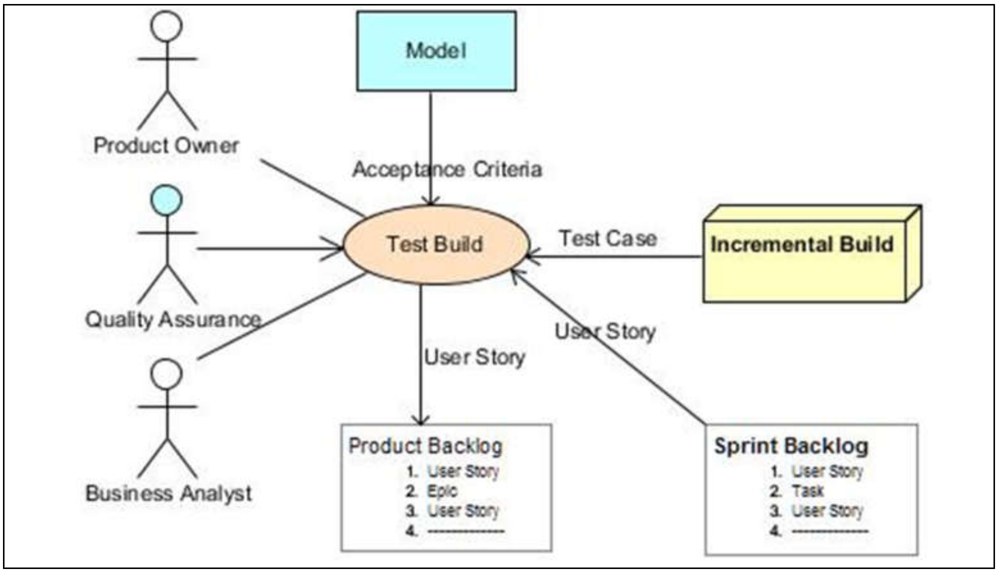


Figure 53 – Test Build Use Case

Scrum recognizes testing as part of the sprint with the development team verifying software that is developed during the sprint. Testing in the Quality with Agile through Pictures process refers to validation of the product. This allows people who are not part of the development team to approve the product prior to release.

Testers use the acceptance criteria from the model and details in the user stories, to create test cases.

- ♦ **A textual version of the acceptance criteria may be attached to the user story.**

Test cases describe a series of steps and the expected outcome(s) of executing those steps. The results of executing those steps are captured in the test case, as test results. When an unexpected test result occurs, quality assurance writes a defect user story detailing why the software did not behave as expected.

- ♦ **Many Scrum project teams like to create bugs in order to distinguish between defects and new requirements of the product. Defects are considered to be a type of user story, with a different format, but the same lifecycle. Both have the same result in that they produced work for a development team.**

The business analyst provides support to testing by answering questions that quality assurance has about the user stories.

Defect type user stories should be traced to the epic from which the functionality being tested was derived (if applicable). This implies that an epic can only be closed once all of its child user stories have been delivered to production.

- ♦ **The epic may be considered done when all of its child user stories are done. However, it is not closed until all acceptance criteria for the child user stories are validated.**

Errors discovered in production are not defects. Since the product has already been validated, it is assumed that all the features were tested and working properly when it was released. Therefore, any changes to the delivered product are considered to be a change to the requirements, and they are captured by user stories.

- ♦ **An epic that is done may have a child user story attached to it at any time. The epic returns to an open state.**

A user story that is created as a result of testing will typically be of the format shown in Figure 54.

Preconditions	Describe the state of the system and data values prior to executing the test
Steps	List the steps that a user will perform in order to reproduce the error
Actual Result	Detail the results in terms of what is observed by users or other systems
Expected Result	Details what was expected to be observed and the what was different

Figure 54 – Defect Type User Story

Although the content is different, user stories that are created from defects follow the same lifecycle as user stories created from requirements. One reason for labeling these user stories as defects may be for collecting metrics about how well the process is working. Figure 55 shows creation and closure of user stories and epics.

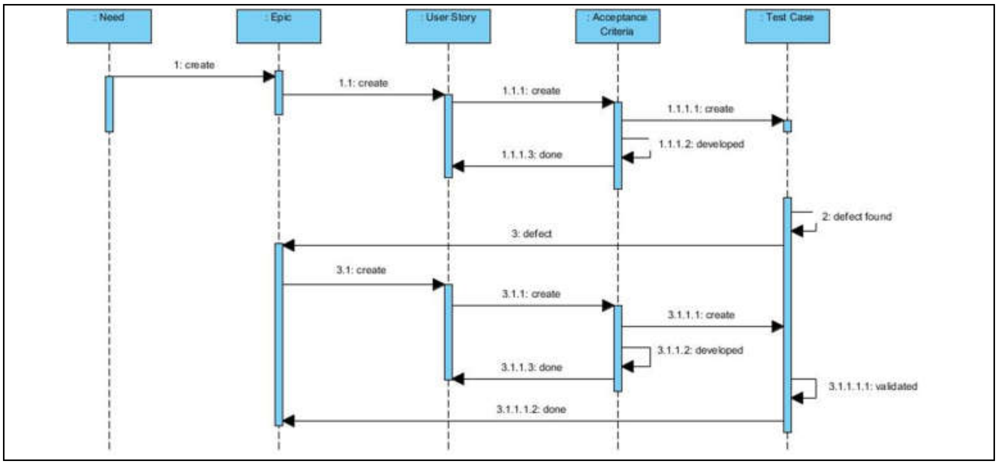


Figure 55 – User Story States Sequence Diagram

The sequence diagram shows the lifecycle of an epic from creation to closure. It starts when a business need is identified and ends when there are no open child user stories or defects.

1. An epic is created as a result of elicitating business needs.
2. A user story for development is created from the epic.
3. Acceptance criteria for the user story are created.
4. A test case is created from the user story and its acceptance criteria.
5. The user story is developed and its acceptance criteria are tested.
6. The user story is done in a sprint and becomes part of the incremental build from that sprint.
7. The incremental build is validated with a test case and a defect is found.
8. The defect is traced to the epic from which the test case was derived.
9. A (defect) user story is created from the defect and traced to the epic.
10. Acceptance criteria are created for the (defect) user story.
11. A test case is created (or the existing test case is re-used) for the acceptance criteria.
12. The (defect) user story is developed in a subsequent sprint and becomes part of a incremental build
13. The (defect) user story is done.
14. The (defect) user story passes validation during testing of the sprint’s incremental build.
15. All user stories traced to the epic are now done and all test cases are validated.
16. The epic can be closed.

When a defect type user story is created, the business analyst will update the model, if necessary. If the user story was generated as the result of a defect in the requirements (yes, it does happen) then the impacted use cases are updated and the business analyst follows the Maintain Requirements activity.

7.4.10.1 Test Case Example

Quality assurance needs to validate that an incremental build meets the Pay Bill acceptance criteria for the sprint in which that build was developed.

In this sprint the Process Customer Payment (Main flow) user story was developed (this user story is detailed in section 7.4.1.2). Some of the acceptance criteria for this user story are:

- Given that a bill is displayed to a customer, when a pay bill request is received, then the system will display a request for the customer payment method.
- Given that a payment request is displayed, when payment information has been entered, then (etc.).

A simple test case for these acceptance criteria might look like that in Figure 56.

Prerequisites: The bill is displayed to the customer; Credit card is recognized as valid by the payment processing system						
Step #	**Action**	**Inputs**	**Expected Outputs**	**Actual Outputs**	**Result**	**Comments**
1	Pay Bill selected	N/A	Bill payment request screen is displayed	Bill payment request screen is displayed	Pass	Paypal is not displayed as an option
2	Make Payment is selected	Credit card (number, expiry date, name, code)	Payment authorized screen is displayed	Payment not Authorized is displayed	Fail	It was verified that the credit card is valid
3						
Recommendation	Failed: Confirm that payment processing system is accepting valid credit cards.					

Figure 56 - Example Pay Bill Test Case

The testers have verified that the credit card is valid, yet the system is still displaying a message that the payment method is invalid. As a result of the failed test case, a user story is generated to investigate why the valid credit card is rejected. The user story looks like that shown in Figure 57.

Parent	Pay Bill Epic

Overview	Valid Payment Method Not Processed
As a result of entering a valid payment method when attempting to pay a customer bill, the system is displaying a Payment Not Authorized screen. Investigate why the Payment Authorized screen is not being displayed.	
User Story	
As a I want so that	customer my valid credit card to be accepted I am able to pay my bill
Acceptance Criteria	
Given when then	that a payment request is displayed a valid payment information has been entered the Payment Authorized screen is displayed

Figure 57 - Defect User Story Example

This user story is added to the product backlog and taken into the next available sprint. Note that the user story is linked to the parent epic.

7.4.10.2 Summary

Having an independent quality assurance organization ensures that there are no conflicts of interest when it comes to documenting defects. A tester who has knowledge of how the code was developed is influenced by that knowledge. A reviewer who helped write a user instruction manual should not validate the text that they wrote (since they already believe that they know what it says).

Ideally, test cases are written by people who are removed from development. Test cases should only be written against the requirements. Higher quality requirements ensure higher quality test cases. Higher quality test cases lead to better knowledge of the quality of the product.

- ♦ **Higher quality testing does not improve the quality of the incremental build. You cannot inject quality after something has already been built.**

7.4.11 Deploy Build

Figure 58 shows the actors, inputs and outputs of the Deploy Build activity.

Responsible Actor– Deployment Manager

Contributors – Business Analyst, Quality Assurance

Inputs – Incremental Build

Outputs – Release

The purpose of the Deploy Build activity is to give a customer access to the features that are part of a validated incremental build.

- ♦ **Deployment is often performed by a development/operations team. My experience with dev/ops is limited. As a result, the content of this section is kept to as minimum.**

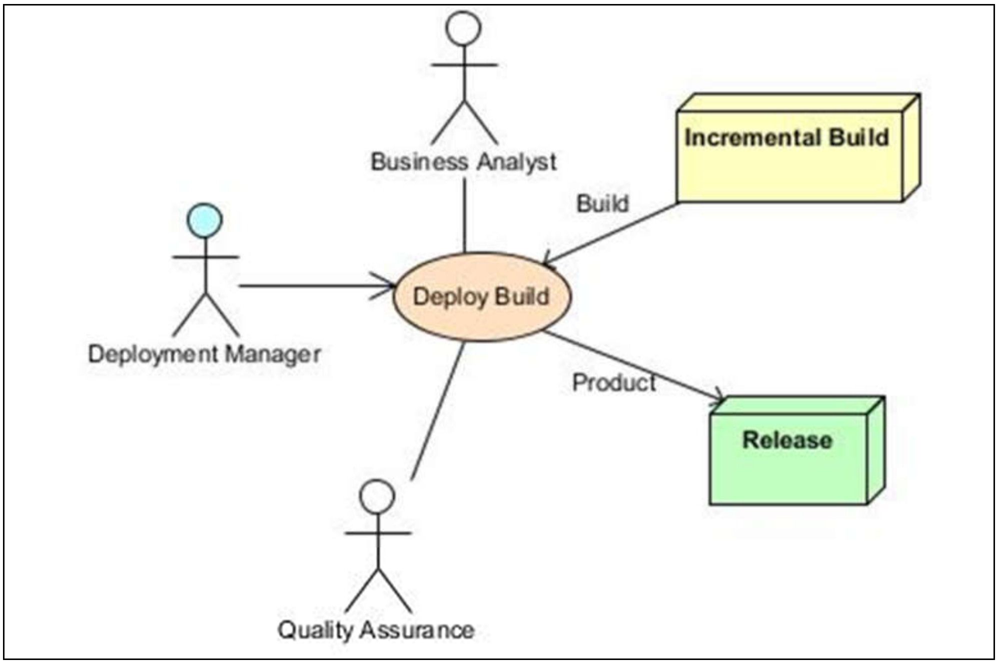


Figure 58 - Deploy Build Use Case

The deployment manager is responsible for deploying an approved incremental build into production. Although the software in the incremental build is the same for all customers, the architecture to which this software is deployed may vary by customer. (For example 1 customer may be using a Windows based architecture and another using OSX or Linux.) The deployment manager needs to be aware of these variations in deployment and will have developed procedures for each variation.

The business analyst and quality assurance supports deployment by ensuring that the release performs as expected.

I have experienced 2 variations of support to deployment.

- When the release is deployed to production, there is nothing that the business analyst or quality assurance can do to prevent the customer from using the system even if it is broken. In this case, the deployment manager is responsible for verifying the product after it has been deployed to the customer.
- When the release is deployed to a backup (or fail-over) system the business analyst and quality assurance can perform a full test of the product before the customer is given access. Only after approval will the backup system be

switched to production and the production system becomes the backup system.

The deployment manager provides continuous support to the customer after deployment of a release, and not just during deployment. As a result, deployment is considered a continuous activity from one release to the next.

7.4.12 Document Release

Figure 59 shows the actors, inputs and outputs of the Document Release activity.

Responsible Actor – Writer

Contributors – Business Analyst, Quality Assurance

Inputs – User Story, User Interface

Outputs – User Instructions

The purpose of the Document Release activity is to create instructions for people who use a product release.

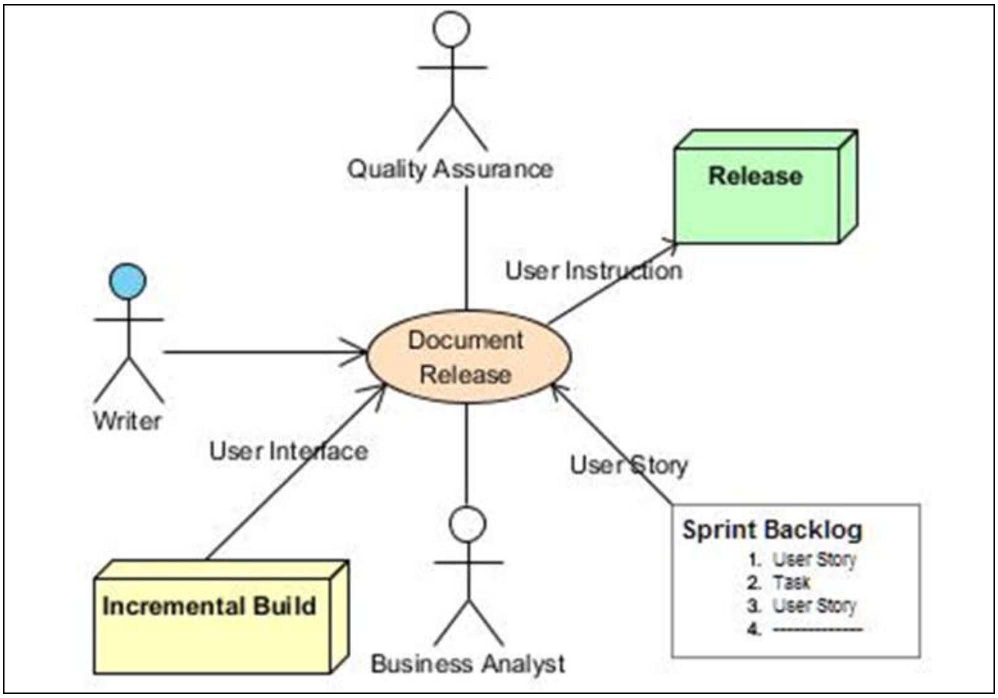


Figure 59- Document Release Use Case

The writer produces user instruction manuals from the user stories and from an incremental build that is due to be released. The business analyst contributes to the user instruction manuals by explaining the user stories. Quality assurance contributes to the user instruction manuals by explaining their findings during testing of the build.

The writer identifies changes to existing user instructions from the user stories that were completed in the sprint. The writer executes the incremental build in order to understand the user interface and document its usage.

- **Not shown in the diagram, but the writer will often create defect user stories when the product does not match the user stories.**

The user instructions and release notes, that are produced by a writer become part of the release that goes out to the customer. As such, the outputs produced by a writer should be validated by the business analyst, and anyone else who contributed to the release.

User documentation is released to the customer. Quality documentation that is accurate and follows company branding standards, adds to the quality of the released product.

7.4.13 Hold Daily Scrum

Figure 60 shows the actors, inputs and outputs of the Hold Daily Scrum activity.

Responsible Actor – Development Team

Contributors – Business Analyst, Quality Assurance, Product Owner

Inputs – User Story

Outputs – User Story

The purpose of the daily Scrum meeting is to bring members of the project team together in order to update the sprint status and talk about their plans for the working day.

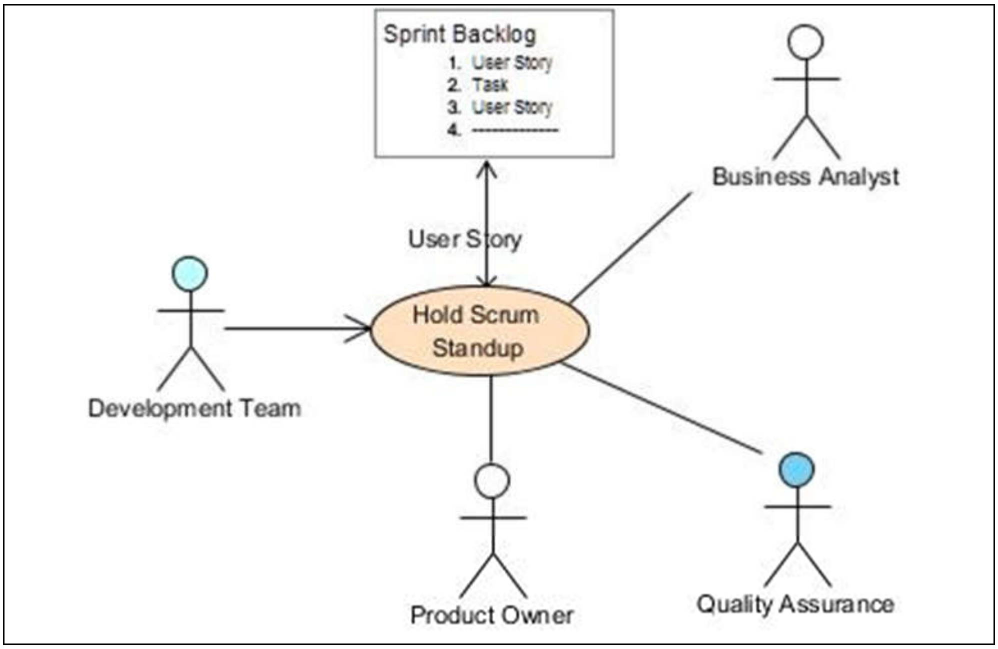


Figure 60 - Hold Scrum Standup Use Case

The daily scrum meeting is held as early as possible in the day such that all contributors can attend. Holding the meeting early allows the team to formulate plans for the current working day.

A representative of the development team (often the team leader) hosts the meeting. Contributors to the meeting update their status in the sprint backlog. Progress of user stories in progress is discussed. Finished user stories are done. New user stories are started as others are closed. The participants announce what they intend to do before the next daily standup and any issues that are slowing their progress. Issues that are preventing progress (blockers) are assigned to attendees as action items. Blockers are issues that are outside of the project team's control. The assignee will make the blocker a priority for the day, since it is influencing sprint development.

The business analyst and product owner should assist with clearing out blockers that are affecting the current sprint. The product owner and business analyst discuss stories that they are working on. This gives the team a heads-up of what is coming down the pipeline.

Quality assurance discusses test cases that they are working on and any issues that they found with the previous incremental builds.