5.0 Prepare Project Plans

# 5.1 Describe methods of detecting dependencies between stories

Dependencies exist between user stories or product features when one story or feature appears as though it must be completely implemented and accepted before the implementation of some other story or feature can either be initiated or completed.

## Signs of dependent stories

* + "Instructor creates class", "Instructor adds assignment to class", "Instructor creates note for class". When the same noun or actor is used multiple times, it implies that some specific part of the system must exist before any of the stories can be implemented (in this case, Instructor and Class). Stories like these can often introduce technical dependencies.
  + The client may describe a set of features in the order they are used to seeing them performed. While occasionally the stories for these features will have to be implemented in this sequence, they can often be combined into larger features, or some workaround can be found.
  + Not all dependencies are based on functionality. External system dependencies are often not found in the language of the client or a story, but based on the team's knowledge of how the story will be implemented. For example, if implementing a story to view a list of students enrolled in a class, and the storage of student data was on a separate web service, the team could find their progress blocked by access to the separate storage system.

## Minimizing the impact of dependencies

**Separate technical dependencies from priority dependencies**

Functionality described as occurring in a certain sequence by the client may not actually be required to be developed in the same sequence. The client may have set a priority on those stories that reflects their existing methodology. The software team may be able to "decouple" those stories from each-other completely, or find some sort of workaround which allows the features to be implemented in parallel instead of in sequence.

Other times, seemingly independent stories may be forced into a certain implementation sequence due to technical limitations. For example, a team might feel as though they have to implement certain back-end functionality, such as setting up a database, as part of one story before moving on to others.

These two types of dependencies can be easily described with the create, read, update, delete functionality implemented as the basic behaviour of many systems. For example, the ability to create new users, update existing users, list all users or delete users. A team would likely hear from their client that the "create new users" story is the most important - people need to have access to the system before they can use it. They might also feel that the "creation" step is the largest and should be done first, since the ability to delete a user could be seen as depending on a user already existing. They might include the design and creation of the user database as part of that first story. This would force them to implement that story first, waiting until it was completely finished before moving on to the next one.

The team would have several options:

* Split out a smaller story or subtask which implements the minimum amount of background work required to create the back-end database. Then fill the database with test or dummy data. Once the smaller set of work is complete, move onto implementing the other stories in parallel
* Ask the client why the stories/features should be delivered in a specific order. If the ability to have users present is important (for example), while the ability to add new ones could be done later, the team might choose to simply import existing data, leaving the dynamic features for later.

**Merge dependent stories into a larger story**

In some cases, the team may find that a group of dependent stories have relatively small time estimates at the end of their estimation activities. Rather than spending more time attempting to decouple those stories from each-other in order to allow them to be completed by different programmers in parallel, they might choose to combine those smaller stories into a single, larger story which simply takes a single programmer pair a slightly longer time to implement.

**Split stories in a different way**

Most user stories end up split along "vertical" lines. For example, stories around class management might be split into "Create new class", "Update existing class", "Add student to class" and so on. These stories as worded could introduce some uncertainty around time estimation. Since all of these stories deal with the same set of objects, they are likely to be dependent on each-other, or at least one some piece of shared functionality, such as a database table and related objects used to save the Class data.

* **Estimate all setup in a single story**
  + One story is chosen as the “natural leader” story, to be implemented first and to have the initial setup and programming time included in its estimate. This allows all other related stories to be estimated independently. Which story this is depends on the team, but in this example it would likely be the “Create new class” story which would be implemented first, though this could change depending on client priorities.
* **Create internal story**
  + Instead of only using vertical story slices along functional lines, such as the “create”, “edit” and “delete” verbs, they could break apart the setup portion of the work into separate, internal story. This story, such as “Access class data in persistent storage”, is not intended to deliver value directly to the client, but to the team instead.

## Further reading

<http://itsadeliverything.com/user-story-dependencies-are-more-apparent-than-real>

<http://www.mountaingoatsoftware.com/blog/estimating-work-shared-between-two-backlog-items>

<http://tracks.roojoom.com/r/2218#/trek?page=3>

# 5.2 Describe methods of project estimation

When the project proposal and initial requirements gathering processes were employed it was necessary to estimate time to complete tasks and the project. It is difficult to estimate accurately but it is very important to try.

Estimating is an important skill. At project initiation it will be required to provide an estimate of time to complete.

## How to make an estimate

There are many ways to make estimates. Here is a typical method that usually produces decent estimates:

* Break down work into small tasks and estimate tasks individually.
* The sum of all tasks will be the time required to complete the full project.
* If any tasks are too big, break them down into subtasks and estimate on the subtasks. Accurate estimate are easier to achieve from smaller tasks.

There are a number of different methods for performing estimations of project timelines. For example:

* Estimate the number of methods. Estimate time per method. Multiply number of methods by time per method
* Estimate the size of the program, evaluate the competency of the development team and plug these values into a pre-made estimate sheet to calculate the estimate.
* Compare to a previous project and compare the estimates of a project with comparable complexity.
* Ask a company expert.
* Use planning poker for each task or unit of work.

A very common estimation method on agile project teams is the "planning poker" method.

## Planning Poker

The basic premise of planning poker is to get a group of developers to agree on an estimated amount for each task (usually per story).

Each developer gets a deck of cards with numbers on them. These numbers represent an estimated amount of time to perform a task. The typical planning poker desk consists of the number 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100. Depending on the tasks these numbers can represent months, days, hours, methods, etc. The unit of measurement for the team should be agreed-upon beforehand, and be used consistently throughout the project.

For each task or story:

* One developer describes the task
* Each developer selects the appropriate estimate cards for the task without revealing it to the others.
* Once all the developers have selected their estimate then all estimates are revealed.
* If all developers have the same estimate then consensus has been reached and the estimate for the task has been decided.
* If there are discrepancies in the estimates then:
  + The developer with the low estimate describes their rationale for the estimate
  + The developer with the high estimate describes their rationale for the estimate
  + Another round of estimates are performed
  + This process continues until consensus is reached on the estimate

It is important not to reveal your card before everyone has selected their card, otherwise you may be influenced by what others have chosen. This is especially important if there is a perceived imbalance in power between the estimation team members (if a supervisor or manager is present).

Developers have their own assumptions about what is and isn’t included in the task. By having the low and high estimators explain their rationale, it brings out assumptions that people have made. If someone assumes one thing and someone else assumes something else, their estimates will likely be different. By discussing the estimates, people can come to a common understanding on what is involved in the task.

A video describing an example planning poker session can be found below:

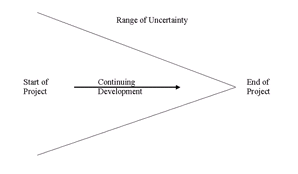
<https://www.youtube.com/watch?v=MrIZMuvjTws>

More information about planning poker can be found here:

<http://www.mountaingoatsoftware.com/agile/planning-poker>

# 5.3 Explain the importance of evidence based scheduling

Maintaining accurate statistics on previous projects is beneficial when estimating new projects. It is also important to maintain project statistics for determining if the estimated completion date is correct. As a project proceeds you will be able to estimate a more and more accurate completion date.



At proposal or project initiation time there is much uncertainty in the project. This makes it difficult to make accurate estimates. As the project progresses the amount of uncertainty is reduced. This amount of uncertainty is indicated by the open side of the triangle. As we continue throughout the project, the amount of uncertainty is reduced as indicated by the narrowing of the open end of the triangle. As the project progresses, better and better estimates of remaining work to be done can be made. At the end of the project the exact costs are known and this information should be used in estimating future projects.

## Time Tracking

Time tracking is an important tool in helping make better estimates, on a current project and on a future project. Time tracking is where time spent on each task in a project is monitored and recorded. This has the following benefits:

* It lets you know how much time has been spent so far on the project.
* Information can be used to more accurately estimate remaining time to complete the project.
* Information can be used on future projects for making better estimates

## Tracking Percent Complete

When working on a project it is always useful to know what percent of the project is complete. This can be used to determine if deadlines are achievable. If it is known ahead of time that a deadline will not be achievable then alternate measures can be taken to bring the project back on schedule by adding some more resources to the project or reducing the functionality to be delivered.

Percent complete should be calculated based on the actual values.

## Graphical Representation of Progress

There are a couple of graphs that are commonly used to indicate progress. These graphs give a quick and easy visual indication of how a project is progressing.

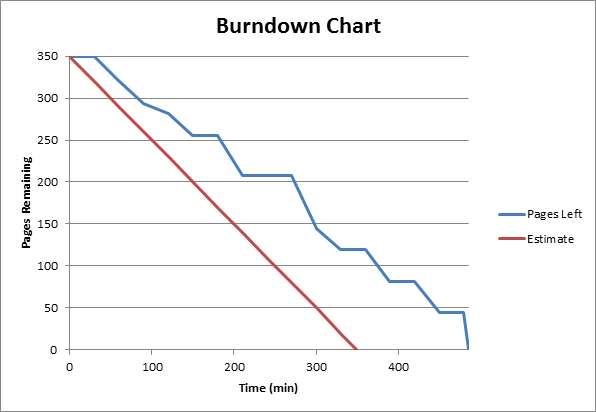
The most common one is an “Earned Value” chart. It looks like this:



(Note: on this chart, progress is tracked 30 minute intervals, and only counted pages as completed when the whole chapter is done. Usually, the x-axis would be dates.)

At regular intervals, we determine how much progress we have made. The progress we have made is termed “Earned Value”. Progress is usually tracked weekly. The progress needs to be an easily measured value, and it’s usually the same thing that we’ve estimated. There’s also a line plotted showing our estimate for the earned value. If our measured line is below the estimated line, things are going slower than we estimated; if the measured line is above the estimated line, things are going faster than we estimated. By fitting a line to our measured points, we can also predict when we’ll be done.

Another graphical method of showing progress is a “burndown chart”, and is typically used by projects using an agile methodology. It shows the inverse of the y-axis; the work that is left to be done:



The value on the y-axis is the amount of work that is still left to be done, often called a “backlog”. Again, we can fit a line to our actual progress, and where that line crosses the horizontal axis is when we expect to be done.

The average amount of work completed in a given period of time is often called the "project velocity", and serves as a good indicator of how project work will progress in the future.

## Further reading:

The importance of evidence based scheduling: <http://www.joelonsoftware.com/items/2007/10/26.html>

# 5.4 Define scope

“Scope” of a project specifies what the project involves.

Scope involves two things:

* What is involved in the project.
* What is NOT involved in the project.

It is important to identify both of these!

**Out of Scope**

Recording what is not in the scope is kind of tricky

It is important to note items that are not clearly not part of the project as agreed to by you and the client.

**Recording Scope**

Scope can be recorded in the following documents:

* Project charger
* Project plan
* Project Scope statement
* And/or requirements document.

A Project Scope statement is a document that usually consists of the following areas:

**Product scope description**

* A description of the scope of the product being delivered/developed.

**Product acceptance criteria**

* A description of how the product will be shown to be correctly completed when the project is done.

**Project deliverables**

* A listing of the things that are delivered to the customer as part of the project, as well as any documents that are produced during the project development (status reports, documentation – user manual).

**Project exclusions**

* A listing of things outside the scope of the project.

**Project constraints**

* Things that limit the project team such as time constraints (deadlines) and budget constraints.

**Project assumptions**

* Things that have been assumed about the project, what will impact the project.
* Example: an assumption that a new piece of computer equipment will be available at a certain time.

# 5.5 Explain the importance of scope control

## Scope Control

Scope control is the process of monitoring the status of the project and product scope and managing changes to the scope baseline.

The “scope baseline” are the things that have been agreed to about the scope.

Change is inevitable in any project and scope control ensures that all requested changes and recommended corrections or preventative actions are processed through a change control process.

There will be a defined process for how you handle requested changes to the scope. You need to record all the changes, ensure both the client and the developers have approved the changes. Changes may involve changes to the cost or schedule.

An advantage of using an iterative approach to software development is that each time through the development process, the customer is given a chance to provide feedback. By having a scope control process, they can’t make huge requests for changes without needing changes to the original contract.

Scope control processes vary between organizations. Sometimes there is a “Change Control Board” (CCB) that reviews all of the changes. Other times, there is an informal process, such as updating a story card.

Creating and adhering to a proper change control process can help project teams avoid "Scope Creep".

**Project Scope Creep** – uncontrolled changes which alter the original purpose of the project until it no longer resembles the original concept.

You can read about the effects and management techniques for scope creep here:

<http://www.techrepublic.com/blog/tech-of-all-trades/real-world-example-of-scope-creep/>

## Scope verification

Besides controlling the scope, we also need some way of having the customer accept the project as complete. All of the items identified as being within the scope of the project need to be accepted.

**Scope Verification** – process of formalizing acceptance of the completed project deliverables. Includes reviewing deliverables with the client and making sure the client is in agreement that it is adequately completed. Obtains a formal acceptance of deliverables by the client.

Scope control and scope verification is usually documented in the Project Plan, or else they are in separate documents that are referred to by the Project Plan (such as a Change Control Process document and an Acceptance Test Plan).

# 5.6 Explain requirements document

Requirements are a description of the needs or desires for a product. Each software requirement identifies a necessary attribute, capability, characteristic, or quality of the software.

REVIEW:

2 Categories of requirements are:

1. Functional (Behavioral)
2. Non-Functional (Non-behavioral)

The software project is complete when it meets all of cr defined. This may not be the same set of requirements as existed at the outset of the project; it is common for the client and project team to discover new or changed requirements as the project progresses. These changes and their impacts should be re-estimated and integrated into the project as schedule and budget allows.

There are different ways of specifying and documenting requirements:

* Agile – User stories
* Mainly Iterative (sometimes used by Agile) – Requirements Document

A formal requirements document uses a strictly defined, formal language for defining each individual requirement. A requirements document explains why a product is needed, puts the product in perspective, and describes what the finished product will be like.

## Requirements - primitive requirements

Before creating a formal requirements document, the software development team, their organization and their client must create an initial, primitive list of requirements. In many teams, this list of primitive requirements may have already been created in the form of user stories. Frequently, teams can take the individual acceptance criteria and story descriptions, converting them directly to the formal language required in a requirements document. In other teams, the requirements document may be their primary form of documentation, and so must be created from scratch.

Primitive requirements are often created as point form, basic sentences describing attributes the final product must have or restrictions placed on certain parts of the software.

Start by making a sentence for each requirement in the form:

1. Attribute to control
2. The relation (=, <, >, is, comply with, as defined in)
3. Values/units – number or standard (include the units)

**Example:** Phone number length equal to 10 digits.

In this case, “phone number” is the attribute, “equal to” is the relation, and “10 digits” is the value and units data.

In the case of nonnumeric requirements, the relation statement may be words like “is,” “shall comply with,” or similar text. Example: “painting is associated to artist”

When the primitive list is created, it doesn’t matter the order, the English used, categorization, punctuation, etc.

## Requirements - formalize requirements

It is important to create good requirements. The following are characteristics of good requirements:

* **Correct** – The requirement must accurately describe the functionality to be delivered.
* **Feasible** – It must be possible to implement the requirement.
* **Necessary** – must be needed by the system.
* **Unambiguous** – the requirement should be interpreted in just one way.
* **Verifiable** – it must be possible to verify that the software meets the requirement.

Formal requirements follow a well defined pattern. Generally, the pattern is:

With respect to [the feature being described], the system [will | shall | should] [the requirements or restriction of that feature].

The following should be considered when writing requirements:

* Requirements are written in an active voice.  
  For example: “The software shall display the rules when ‘r’ or ‘R’ is entered at the main prompt.”
* Keep the requirements short. Try to only put one function in each requirement.
* Use standard terminology when writing requirements.  
  The words “shall”, “should”, and “will” have specific meanings with regards to requirements. Be sure to use these words correctly.
  + **Shall** – indicates the requirement is mandatory. The functionality indicated in the requirement must be implemented.
  + **Should** – indicates the requirement is desired but not mandatory. (Optional)
  + **Will** – indicates a statement of fact. Nothing needs to be done for this requirements but it states how things are.

## Requirements - categorize

Finally, before creating the requirements document your requirements should be categorized into logical groupings. Categories will vary by project, but could include:

* **By user group**. All features intended for use by a specific subtype of user/actor are grouped together. This can prove difficult when the same requirements or features are shared by multiple groups of users/actors.
* **By feature**. All individual requirements around a single feature can be grouped together. Include details about how that feature might differ between user groups.
* **By system**. Many applications are made up of multiple different subsystems – group requirements by the major subsystems they directly apply to.

## Requirements - examples

The following are examples of requirements and an indication of whether it is a good requirement or not:

* The software shall display the total value of the user’s hand when each card is displayed.
  + **Good requirement (assuming there is a requirement that displays how to calculate the total value).**
* The user shall enter either lowercase or uppercase letters as input for all commands.
  + **Bad requirement** – this is not a requirement on the system. Better to say:
  + “The system shall accept both lowercase and uppercase letters as input”.
* The software shall save the open file to disk after a period of inactivity.
  + **Bad requirement** – there is no specification of the period of inactivity (makes it ambiguous and difficult to verify. Better to say: “The software shall save the open file to disk after 5 minutes of inactivity”.
* The software shall flash the “Traffic” LED with a 1 second period and 50% duty cycle when Ethernet packets are being received.
  + **Good Requirement.**

## Requirements document

The requirements document itself is straightforward once the list of formal requirements has been created, since a large part of the requirements document is the formal list of requirements.

A requirements documents usually contains:

* **An introduction or overview section -** (a written description of the product in nontechnical terms.) Include a narrative which will give readers an overview of what the product will do.
* The specific requirements– requirements should be separated into groupings which make logical sense based on the categorization determined while formalizing the requirement.
* Any appendices if required– this is a good place to append the traceability matrix created earlier.
* A glossary - if there are any unusual terms, acronyms or abbreviations you need to clarify.
* References - include a listing of external references used.
* An index - if your document is quite long.

Generally, organizations which require the use of a formal requirements document will use a common template to create the Requirements document. An example requirements document template can be found here: Requirements Document Template.docx

Once the requirements document has been created, it is important to present it to the client and have them either approve it or suggest changes or improvements. In some cases the client might not actually read the document so it’s important to use walk-throughs or discussions to have the client fully understand and approve the document.

# 5.7 Prepare requirements document

Refer to exercise.