Chapter 5

Identifying stories and preparing to build

Summary: Creating stories and learning how to analyse and negotiate with stories. Identifying functional requirements. Checking non-functional requirements and quality attributes. Managing the customer. Techniques for estimating resources.

5.1 Looking at the user stories

The business problem has been studied quite extensively, now and we are keen to make a real start. After all we have spoken about progress being measured in functioning software so we need to get coding!

The details that we have relating to the proposed system may seem quite daunting and we need to take a breath and try to find something simple and useful to do.

Looking at the list of stories that have so far been identified it may not be obvious where to start. The customer may be able to steer us towards what he/she thinks is important but they will look at things from a business perspective and the technical issues of building stories may not be clear to them. We, however, have to choose something that will create confidence and a sense of achievement early.

In the first case the stories will probably comprise short sentences which describe the purpose of a small element of software, thus they could be a statement such as:

Users can login to the system

or

Users can enter the customer details and validate data on entry

or

Patients can create an account

The key to an agile approach is to ensure that each story can be implemented in a short time period – 2 weeks at the most. It has to be a coherent and clearly identifiable software function that is not too complex. The problem is that the right level of complexity in a story can sometimes be hard to define and needs a lot of experience to be able to do it without too much trouble. For those with less experience it is important to approach the problem in a simple structured way.

In order to think about these issues in a clearer way it is useful to identify a sequence of actions and events that are recognisable from the perspective of a user interface. If this produces a large collection of inputs and events then is is likely that the story is too complex. Many users and customers think about their system in this way and it is important to try to identify how a system might work from that viewpoint. However, some stories will involve internal types of processing which has no explicit representation in the user interface.

In a large scale industrial application there may be many teams or departments which collaborate in the development of a large piece of software. Here the relationship might be one whereby a team acts as clients to another team and will develop stories which are of a more

technical and specific nature. The principle is the same, try to write down the story in clear language using well defined terminology and if your team is providing the software development effort to engage fully with the clients or customers in discussing the meaning, the relevance, the priorities and the costs of the stories.

In your project, however, you are probably the only team involved and your client will provide you with the key concepts for their business processes and business needs.

We'll take some simple examples of user stories from some of the systems we have built to illustrate the process.

Our starting point is the requirements document. In this we have identified a number of functional requirements and that is what we need to look at.

Each one of these has the potential to become a story. It depends on the level at which you described the requirement, we will assume that if they are too *high level* then you have broken them down into a series or sequence of simpler activities. For example the decomposition of the requirement to be able to set tests in the Quizmaster system to the sub-activities of setting questions and forming a test from a collection of questions.

The initial analysis of a story is to identify the business process that it refers to. To do this consider two basic things, what is being done and to what. In other words, there is some operation described in the story that is prompted by some intervention - normally a user action but it could be a signal from an external component or system, such as a sensor or something similar. This operation will affect some aspect of the system or its data and will

usually produce some observable effect.

Now we create a card for each story, this will provide some basic information about the story and allow us to plan out our work.

The first version of a story might contain nothing more than a name and a brief description of what it does. This might be enough if the programmers are experienced and have built very similar stories before. They may well be able to create suitable test sets for the story also from their recent experience of doing something similar. But what if you have never done this before?

We will use a structured approach to describing stories that will help us to develop unit tests, the subject of Chapter 8.

Most units identified from a requirements elicitation process with a customer will have a relevane to their business needs. In other words, the customer will understand the purpose of the story. This may not always be the case and in some projects the stories will be less visible or apparently relevant to the customer – they may be of a rather technical nature and thus less comprehensible to the customer. In such cases it is important to try to explain why it is relevant – what business value it will provide.

From a customer's point of view it is likely that they will interpret a story in terms of how it relates to some business function or part of a business function. In other words, the story will be involved in generating some useful result.

In order to explain this – and to exploit the most from it in terms of test set generation – it is often useful to think in the following way.

The story will be triggered by some event – a mouse click, a result or a message from another part of the system etc. This event then results in some processing, it could be a database update, a query or the computation of a result and do on. In order for the story to be effective it may need to communicate with other stories or a database and it may result in changes to other parts of the system, such as a database or a screen display.

Consider the *login* story, for example. The purpose of a login system is to manage the different classes of users and to ensure that each user has secure access to the part of the system that they are registered to use. Thus the typical system will involve a user inputting their user identity (User ID) and a password. The system then needs to check that such a user exists and that the password submitted matches the password allocated to that ID. On successfully checkiubg this the system should then provide the appropriate start screen for that category of user. Thus we have information going into the system (ID and password), a check being made against a list of IDs and passwords, a new screen is then presented. This sceen is either the starting screen for that user or an error report if the user is not recognised – this may then provide further options to the user to correct their entry information or seek further advice.

We can thus think of many stories in the following way.

The story is triggered by some input or set of inputs; the story may then need to check some

of the values of the input against some existing data – a look-up table, database etc.; the story then computes some results and outputs these somewhere – perhaps to a screen or to some other part of the system; finally the story might update a database or some other store.

Suppose the story is a simple *login* function:

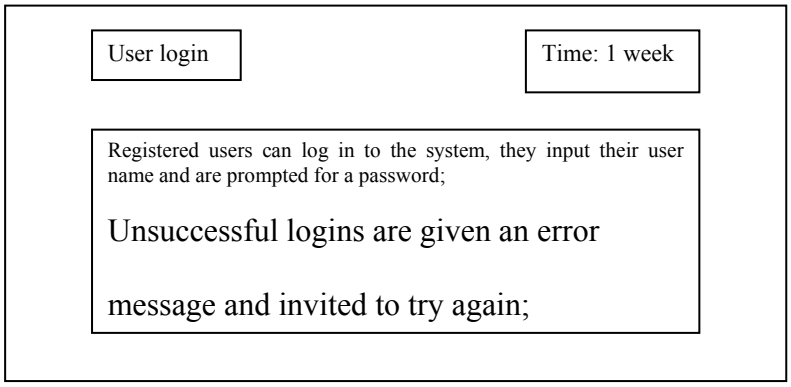

Figure 5.1 A simple login story

This is a minimal type of story card, the meaning of it is fairly clear and so it might not need elaborating. However, it is a good example to explain how a more sophisticated story might be specified.

First we look at the inputs that will trigger the story. There will, probably, be some screen that provides the login prompt that will consist of a data entry and password entry dialogue box.

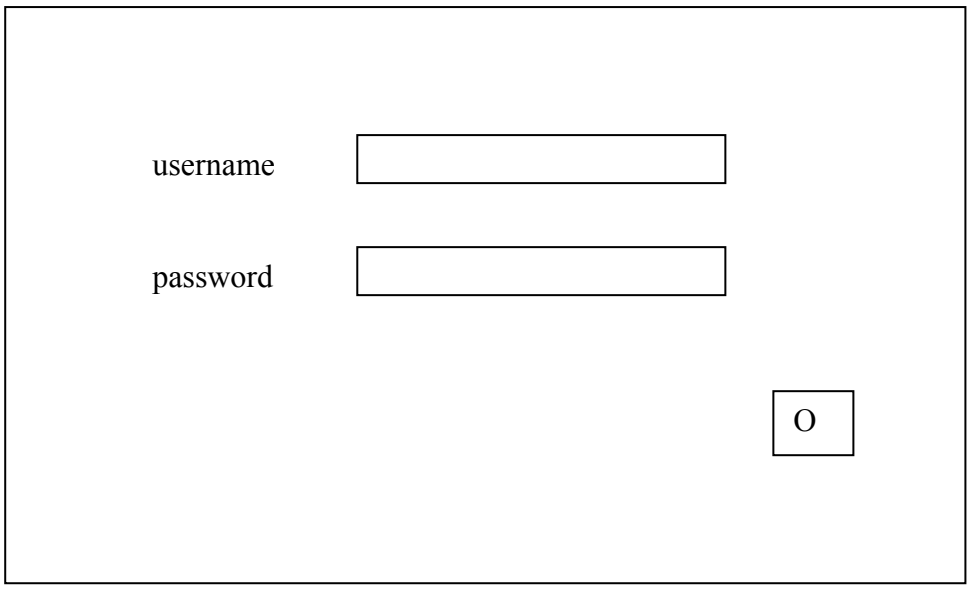

Figure 5.2 Login screen

There will be a database or ‘lookup’ table that contains the passwords for all the registered users.

The user puts in their user name and then their password. There may be some data validation for the user name in terms of checking for valid characters. The story will then have to check that the submitted user name data exists in the list of registered users an that the password submitted correspondes to that user. If this is the case then the login is allowed and user is then permitted access to the lparts of the system they are qualified to use.

If the password check fails then the error message is generated (it could be *no known user* or *incorrect password* depending on where the checking failed) allowing another try. The story needs to keep a track of how many unsuccessful tries are made before disabling the system for that person.

There are therefore three things to think about – the type of input data, the internal memory (database) that is consulted and the resultant output data. This information will be used in testing – the first being the information to trigger the test; the second defines the environment the test is run in; the third is the expected test output.

We need to think about simple tests in this systematic way if we are going to achieve the quality and delivery of products that we want.

One of the hardest things about XP is coming up with tests before we start to code. This simple approach can really help.

Not all stories will involve an internal memory check but many will and it can take many different forms: databases, special variables in the code, and so on. We need to know these when we do the tests or it might be the case that the code works when we test it in isolation but doesn't when part of the main system.

So we have a simple tabular description of each story. The cards define the following aspects of the story.

1. Its name.
2. What is the event that begins the story process.
3. What is the internal knowledge that is needed for the story.
4. What is the observable result,
5. How is the internal knowledge updated as a result of the story

6. What is the current priority of the story,

7. What is the estimated cost of the story.

8. What is the likelihood of the story being changed or dropped?

All this information is needed for testing. A story card should be created for each of these with the information described. A possible template for a story card is given in Figs. 3 and 4. Some of the topics will be discussed later.

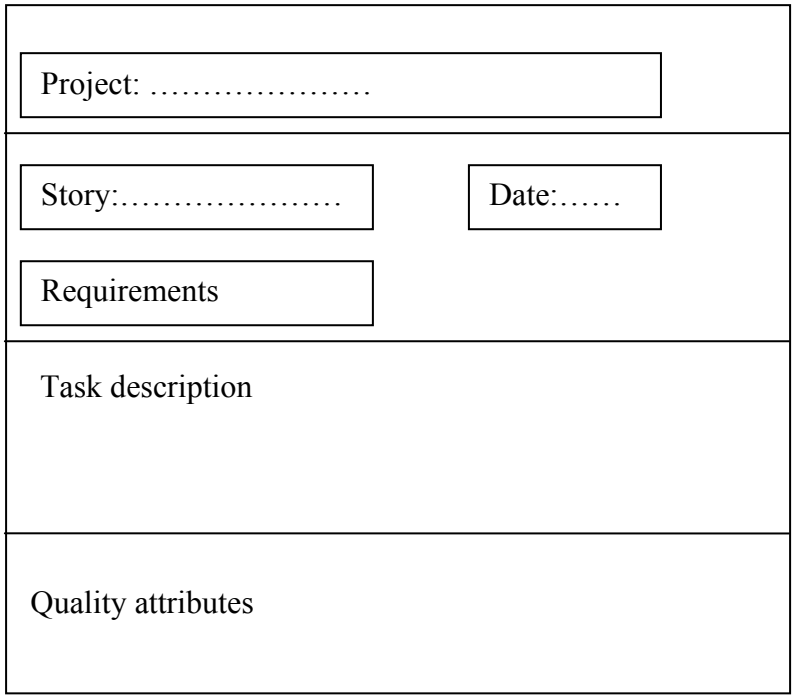

Figure 5.3. Story card template

On the reverse we can put some more useful information.

Initiating event/inputs	
Memory context	
Observable result/output	
Risk factor/importance	
Tests	
Associated stories	Date delivered

Figure 5.4 The reverse of story card

The design of story cards is topic worth discussing. Some would just contain the most basic statement of the story and little else. Clearly we do not want to introduce too much bureaucracy and any information included must be important and add value. We have to think about potential maintenance issues and provide just enough information to allow the programmers to understand what each story does and how it has been tested. Don't forget, when maintenance is undertaken the original programmers may have moved on and any information about what they thought the story should do and how they convinced themselves that it did it is invaluable.

It is also a good way to clarify what it is you are trying to do – if ever you get into a position where you don't quite understand what the issues are – write them down – using any

technique you are comfortable with – free text, bullet lists, mind maps etc. The process of writing it down should force you to clarify it – especially when working in pairs or groups. The cards we use seem to fulfill this purpose without being too complex or tiresome!

We will go through each component of the card and justify its role.

The name of the project is clearly needed if the story is to be related to a specific application being developed by the company. It may be that a given story is used in several applications and while this is a sensible strategy, reusing previous components that have been successfully built, it does bring with is some dangers in that the story might have been slightly modified for a different context. We need to avoid confusion – especially during maintenance.

The name of the story is an essential component of the card!

The date is also important – a story might be revised during different sessions with a customer and thus we need to be sure that we are using the most up to date version. Of course you should be using a suitable version control and management system to try to keep things in order.

Most stories will relate to a specific requirement, these were discussed in the previous chapter. The set of requirements and of stories will change over time and it may be that we scrap some stories that are no longer needed and introduce new ones as the requirements change. An important issue – and one that has no simple resolution – is the amount of detail needed in a story. In practice we should try to define stories so that their implementation and testing can be done within the natural time scale that we are working with - this may be a day

or a week or whatever the project needs and the programming team are comfortable with.

The task description is a set of short sentences that describe in terms the customer can understand what the story does. Too much technical detail or jargon will confuse many customers.

The quality attributes may seem an unnecessary ingredient at this stage but there should be some reference to these. A number of problems we have experienced in the past have been caused by a lack of clarity about these constraints. It is often the performance of the software that is unacceptable – a web page that loads too slowly or the usability of the story – for example a log in facility that looks very different to what is expected.

On the reverse of the card are the key issues relating to the resources needed to build and test the story.

These resources include the date that the story is to be delivered – this will be based on the time that the programmers estimate will be needed to build and test the story. The overall planning of the project will be a dynamic portfolio of stories and integrations that is adjusted as the project evolves and the stories will be situated within this framework.

The other information on the card assists in developing the tests.

When writing tests there are a number of important issues that have to be sorted out. Remember, we have no code yet, just the descriptions of the story.

To run a test we need to trigger the software with suitaable inputs. These will be the subject of the first component of the card. It might be that a direct user input cuses the story to start or a specific event or request from another part of the system. All of the expected inputs nd events need to be identified.

The next issue is the information that the story needs when triggered. Take the *login* story, for example, this receives a user name and password and has to consult a look up table or query a database to establish if the user's details are correct. This is the memory context that we need to set up for testing. Many stories will need such an environemnt for testing – but not all.

Naturally we will need to know what the expected behaviour of the story is to tell whether it is correct. It is important that this is written down somewhere – that way we can convince ourselves – and those that follow us – that the tests were properly written.

The next issue could be optional – some stories are more important than others – we have seen that the requirements can be clustered into mandatory, optional and desirable so the same will be true of stories that relate to he requirementa. However there is another issue and that is the risk associated with a story. Some stories are critical – in the sense that others many depened on them and that the system as a whole is threatened by failure of these stories. We should recognise this and give these stories some extra attention.

Another type of risk is associated with the liklihood of the story being superceeded by other stories as the requirements change. This allows the programmers and the customer to rationalise the order in which work is done – maybe do the stories that are fairly stable first – if that makes sense. Sometimes, however, you have to build some stories before they are

stable in order to explore the overall architecture of the solutions and to allow for some interaction between different parts of the system and for testing purposes – in the same way as you might write some scripts and stubs to help test partially built systems.

Now we write some tests which will be expressed in terms of what is applied to the story, in what environment it is run and the expected output. For example the log-in story will have a number of tests to explore, not only that it works how it should for legitimate users but also that unauthorised users are rejected and the number of rejections fits with the requirement. To do this some temporary database or list of users and passwords is constructed to check out the story if the main database is not yet ready. In such circumstances it will be important that the log-in function is carefully tested again when the proper database is integrated into the system later.

Finally we describe the stories that depend on the current story and the stories that the current story depends upon. This will help with planning out priorities amongst the stories and their integration into working systems.

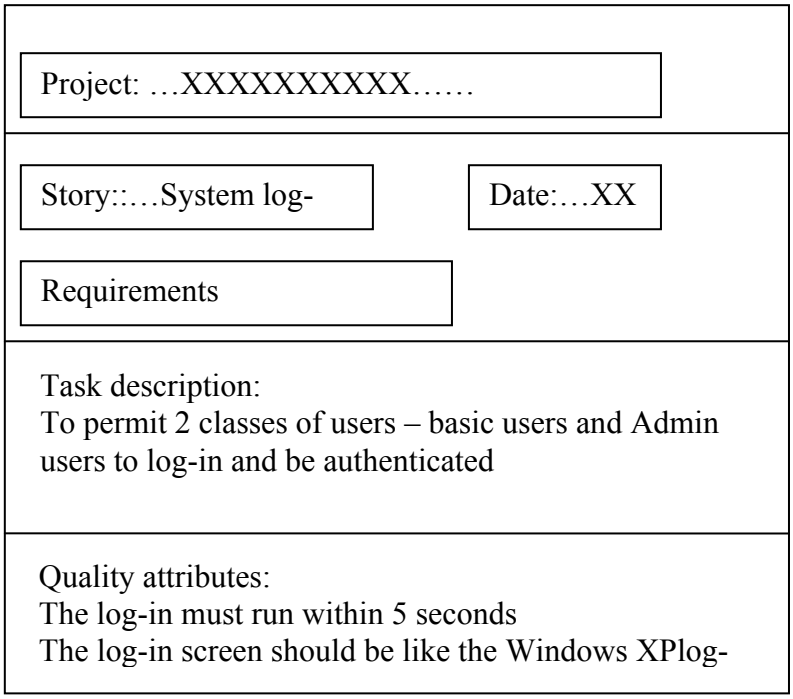

Figure 5.5 Log-in story card

On the reverse we can put some more useful information.

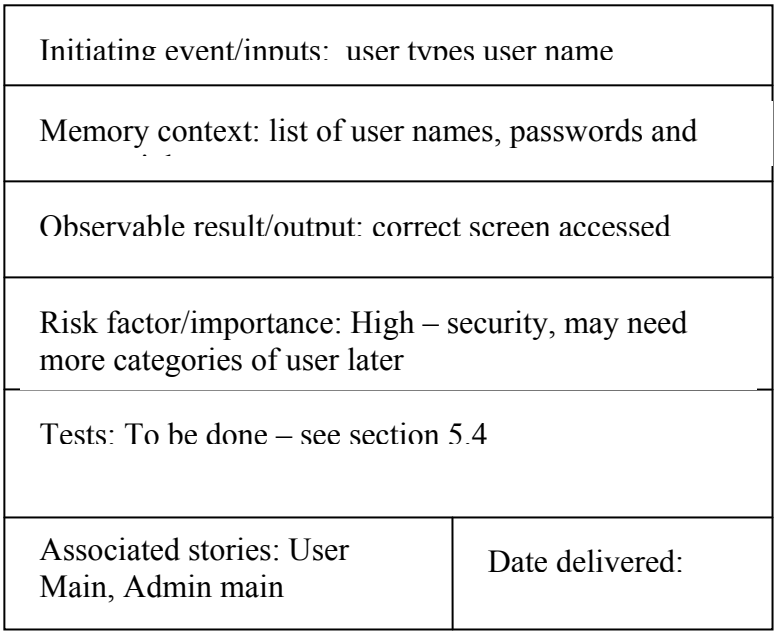

Figure 5.6 The reverse of the log-in story card

The XP method now tells us to write some tests for a story and then to code the story up. Writing tests, as we will see, is a sophisticated business and one of the weaknesses of much of the literature is that little advice is offered about how the tests are found, there is a lot of information about how to run tests and automate this process, but where the tests come from is something that seems to be left to experience - and this is something that you may not have!

Take one of your stories and write down some test cases. Don't forget what you are trying to do - to confirm that the code does what it is supposed to do and doesn't do anything else.

Can you think of any more examples of tests? Good testers think awkward, trying out unlikely scenarios and data in an attempt to break the code. If you are to succeed in XP you must all adopt this attitude. In traditional software engineering programmers tend to be too gentle with their own code, it's a psychological tendency which is hard to overcome, the influence of pair programming in XP is an attempt to prevent this. It's better for the programmers to find the bugs rather than the users!

We will discuss a more systematic way to derive tests shortly.

Now we are supposed to write the code and apply all the tests, correcting the code until they all pass. Naturally, we do this in pairs.

This is the basic XP process. How long did it take? Make a note of the time you spent on

writing the tests. This will give us an indication of what time it might take to write a similar story, and its tests. Hopefully, you will get quicker and better as you gain experience.

The story we have just discussed would need to communicate with other parts of the system and it is not worth showing the client this yet. There will be a user interface and a database, in all likelihood, and this class needs to be able to link in with them. While some pairs in the team are writing these basic units the others can be looking at the design of the interface and the database that will power the system. When we are clearer about these we will be able to write some system tests, link them together and see the results of our work, then we can show the client something that he/she would understand.

Although writing tests for simple classes such as this one is not particularly difficult things change when objects start communicating with each other and when a more complete architecture is being put together, This is where things can go seriously wrong.

In the previous chapter we looked at the requirements document and noted that it is not a static artefact but a dynamic entity that will change over the course of the project – as both developers and customers understand the problems better, as the business needs change and as resources, especially time, are used. However, maintaining an accurate list of stories both those that have been implemented and those that are believed to be required is vital in order to keep a grip on the project. Some proponents of XP might criticise the more formal ways that we do things. For example, just building stories without bothering with requirements document is a popular approach. This may work in some circumstances – perhaps you have a long term relationship with your customer and their requirements arise in a gradual and regular manner and the overall system architecture is stable. This is not the position that we

have ever experienced. Usually we have a fairly fixed deadline for completion and we need to stick to that as far as possible – meeting stage deadlines and agreed installments of software, otherwise the income needed to survive may not come through. Many customers are reluctant to pay unless they think that they are getting something of value.

The next section details some simple techniques for thinking about system tests. The basic idea has been used in industrial settings and has seen massive improvements (up to 300%) in the effectiveness of the tests compared to the original test method being used.

5.2 Collections of stories

The task of taking the list of functional requirements or stories and identifying and organising them into a coherent system can be achieved using the technique that we will describe next.

To gain a greater understanding of how all the stories fit together into a coherent system we need to think about how they relate to each other. For example, it may be that one story can only occur after another one has occurred, or it might be that at some point in the business cycle there is a choice between several stories.

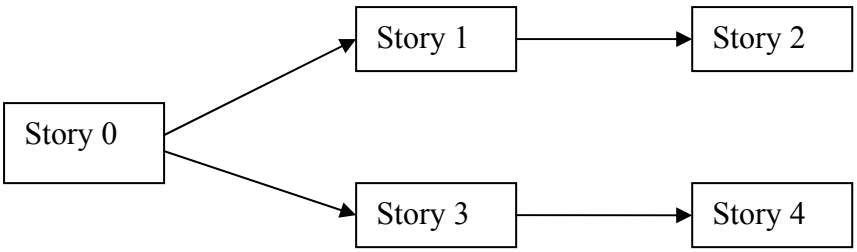

Figure 5.7 Collections of stories.

In this picture the initial story, story 0 is followed by either story 1 or story 2 (but not both at the same time) and then either story 1 is followed by story 2 or story 3 is followed by story 4. It might be then, that stories 2 and 4 are succeeded by further stories or the system returns back to the initial story.

In many cases each main story is associated with a user interface screen, there may be a whole screen to a given story or there may be many stories that can be driven from that screen. Although it is too early to plan out the detailed graphics of the screen it is still important to identify the key elements of the screen, the components that can be used by the user to instigate the process defined by a story, the extra information needed to be displayed for this and the result of the operation of the story displayed suitably.

We might break down a story into tasks which, when combined, provide a natural way to implement the story. One task might be to paint a screen - eg. a form, another might be to provide a data entry function which will connect to a task that performs some calculation woth the data. This might involve communicating with a database - to check with current data, and

then to communicate the result back to them screen. Once these tasks have been programmed and integrated together we have a coherent story to show the client.

It is sometimes a good idea to show the client some of your thoughts, on paper, of how the story relates to your interface ideas before you do much coding. This can then lead to a clearer understanding of what is required.

The system will respond to some *external* stimuli, these will be, for example, users interacting with a screen entering data, choosing options through mouse clicks, ticking boxes etc.; messages from some other system, perhaps the results of a query to a database,

This data is then processed in some way, perhaps it's just a simple calculation, perhaps the system needs to contact another part of the system, eg. a database, in order to proceed. The results of the computation may then be fed back to the user via a screen, or output in some other way or to another component. It is possible that the database needs to be updated as a result of this interaction.

This we have 4 essential actors in any system - an input actor, a processing actor, a *memory* actor, and an output actor. The memory actor - this could be managing a database - will be involved in reading and writing to the notional memory and commnicating with the other actors. The input actor reads the input from the screen or input device, the output actor deals with the output while the processing actor does the actual core computation.

We conclude this section with some examples of stories.

5.2.1 Pharmacovigilence

This project involved developing an on-line web-based system for medical workers to submit detailes of adverse reactions to drugs by their patients. It is part of the European Union's process of monitoring the safety and effectiveness of medication.

Project: Pharmacovigilance	
Story: Add entry via XML e-mail	Date: 5th of March, 2003
Requirements number: 6	Iteration:1
Task description: Imports a report in the XML-based ISCR format. If the software is to manage the e-mail input, the user may be required to specify a source from which to obtain the e-mail. The XML data will be entered into the same form used for inputting data manually, so that the user can check its validity and add additional information	
Quality attributes: Processing of input file should be quick enough to be unnoticeable to the user Should be 99% successful. Clear interface design	

Figure 5.8 Pharmacovigilence story 6 (i)

On the reverse we put some other useful information.

Initiating event/inputs: A request is made to import the data for addition from an XML e-mail on the Add form.	
Memory context: Source list and database available; Entry is added to the database	
Observable result/output: The result will either be a success confirmation or an	
Risk factor/importance: 3 (high) Change factor: 2 (medium)	
Tests: 6. Do nothing and exit [cancel] 7. Import non-XML file [error} 8. Import XML file that does not contain mandatory requirements [error] 9. Import invalid XML file [error] 10. Import empty file [error] 11. Import appropriate XML file 12. Failure to source e-mail from a secure repository [error]	
Associated stories: Addition of data manually, main	Date delivered:

Figure 5.9 Pharmacovigilence story 6 (ii)

Project: Pharmacovigilance	
Story: Output IHL line	Date: 4th of March,
Requirements number:	Iteration: 2
Task description: Exports a report in the IHL line listing format.	
Quality attributes: Operation should be completed in an acceptable time period. Should be 99% successful.	

Figure 5.10 Pharmacovigilence story 10 (i)

On the reverse we can put some more useful information.

Initiating event/inputs: A request is made for output in this format on viewing	
Memory context: The report exists in the database and the current user accessing the system is an administrator	
Observable result/output: The result will either be a success confirmation or	
Risk factor/importance: 3 (high) Change factor: 1 (low)	
Tests: 1. Request leads to output of IHL line listing	
Associated stories:	Date delivered:

Figure 5.11 Pharmacovigilence story 10 (ii)

5.2.2 Stamps system.

This project was delivered to a client who ran a retail organisation that sold rare and historic postage stamps to collectors.

<table>
<tr><td>Customer story card</td><td colspan="2">Project title: Stamps</td></tr>
<tr><td colspan="2">Date: 24/11/02</td><td>Project phase/iteration: Design</td></tr>
<tr><td colspan="2">Requirements Number: 2</td><td>Story name: Order Maintenance</td></tr>
<tr><td colspan="3">Task description (English):

This page is designed to give the user the ability to organize orders to existing customers

“Order Maintenance” contains links to the “Create New Order”, “Modify/Delete Existing Order”, “Print Order Letter”, “List Unreturned Orders” and “List Returned Orders” pages.</td></tr>
<tr><td colspan="3">Initiating event:

The user clicks the “Order Maintenance” link on the navigation bar.</td></tr>
<tr><td colspan="3">Memory context:

The “Order Maintenance” page is only used for navigation purposes and does not alter the database in any way.</td></tr>
</table>

Observable result: The “Create New Order”, “Modify/Delete Existing Order”, “Print Order Letter”, “List Unreturned Orders” or “List Returned Orders” page is displayed, depending on which link the user clicked on.	
Risk factor: 1 (Low risk).	**Change factor:** 9 (The system is being completely rewritten).
Related stories: “Stamp Dealer System Contents”, “Order Details”, “Print Order Letter”, “List Unreturned Orders” and “List Returned Orders”	
Notes:	

Figure 5.12 Stamps story 2(i)

Story name: Order Maintenance			
Resource estimates:			
Input	☐	**Simple**	**X**
Output	☐	**Average**	☐
Enquiry	**X**	**Complex**	☐
Reference file	☐		
Database	☐		
Function/object point total:		**Man-hours total:** 2	

Functional tests:

User clicks on “Order Details” link.

User clicks on “Print Order Letter” link.

User clicks on “List Unreturned Orders” link.

User clicks on “List Returned Orders” link.

Quality attributes:

The system should work in screen resolutions of 800x600 or higher.

It should be quick and easy to get to any other section of the system from this screen.

All operations should take a reasonable amount of time to execute (less than 15 seconds).

The interface should be easy to use and satisfy the usability heuristics described in [Nielsen, 1993].

Figure 5.13 Stamps story 2(ii)

Note that this story card is more complex than the previous ones since it also contains information about estimation of resources for the story.

This is based on a more traditional software engineering approach tpo estimation that relied on identifying 'function points' or 'object points' in the story. This is a mechanism for trying to ascertain the complexity of a story in terms of what it does – does it communicate with or query a database, is it simple piece of code or something that has some real challenges involved etc.? There is a considerable amount of data available about industrial projects which have been classified in this way but this data may not provide a reliable answer for our needs here. We no longer record this information on a story card sincee it did not seem to provide sufficient value – the estimates generated using this method were very inaccurate.

Customer story card	**Project title:** Stamps
Date: 24/11/02	**Project phase/iteration:** Design
Requirements Number: 5	**Story name:** Unit Maintenance

Task description (English):

This page is designed to give the user information about functionality relating to units, helping to clarify the way that the system behaves.

"Unit Maintenance" contains links to the "Add New Unit", "Modify Existing Unit", "Delete Existing Unit", "Browse All Units" and "Find Unit" pages.

Initiating event:

The user clicks the "Unit Maintenance" link on the navigation bar or the "Unit Maintenance" link on the "Stamp Dealer System Contents" page itself.

Memory context:

The "Unit Maintenance" page is only used for navigation purposes and does not alter the database in any way.

Observable result:

The "Add New Unit", "Modify Existing Unit", "Delete Existing Unit", "Browse All Units" or "Find Unit" page is displayed, depending on which link the user clicked on.

Risk factor: 1 (Low risk).	**Change factor:** 9 (The system is being completely rewritten).

Related stories: Stamp Dealer System Contents, Add New Unit, Modify Existing Unit, Delete Existing Unit, Browse All Units and Find Unit.

Notes:

Figure 5.14 Stamps story 5(i)

Story name: Unit Maintenance			
Resource estimates:			
Input	**X**	**Simple**	**X**
Output	☐	**Average**	☐
Enquiry	☐	**Complex**	☐
Reference file	☐		
Database	☐		
Function/object point total:		**Man-hours total:** 1	

Functional tests:

User clicks on the “Add New Unit” link.

User clicks on the “Modify Existing Unit” link.

User clicks on the “Delete Existing Unit” link.

User clicks on the “Browse All Units” link.

User clicks on the “Find Unit” link.

Quality attributes:

The system should work in screen resolutions of 800x600 or higher.

It should be quick and easy to get to any other section of the system from this screen.

All operations should take a reasonable amount of time to execute (less than 10 seconds).

The interface should be easy to use and satisfy the usability heuristics described in [Nielsen, 1993].

Figure 5.15 Stamps story 5(ii)

5.2.3 DELTAH (Developing European Leadership Through Action-learning in Healthcare)

This project is concerned with supporting leadership development activities for health service professionals. (see http://www.deltah.org)

Customer Story Card	Project Title: DPP
Date: 20 / 02 / 07	*Project Phase\Version*: Initial\1.0
Requirements Number: 1	*Story Name*: User can choose a language
Task Description: The user can choose between English, Dutch and Polish languages.	
Initiating Event: The user requests the 'language change' option	
Memory Context: Content is available in all three languages.	
Observable Result: The language of the content changes to that chosen by the user	

Risk Factor: Low	*Change Factor*: High
Related Stories: None	
Resource Estimates: X Input X Output O Enquiry O Database X Other File	O Simple X Average O Complex Man-hours Estimate: 2
Functional Tests: 1. Choose the current language 2. Choose the English language from either the Dutch or Polish language 3. Choose the Dutch language from either the English or Polish language 4. Choose the Polish language from either the English or Dutch language 5. Move to a different page [cancel] 6. If a page is not available in the chosen language [error]	
Quality Attributes: 1. The alteration of the language should be instant 3. The language that is being chosen should be obvious	

Customer Approval:	Date:

Figure 5.16 DELTAH story 1

Customer Story Card	Project Title: DPP
Date: 20 / 02 / 07	*Project Phase\Version*: Initial\1.0
Requirements Number: 2	*Story Name*: User can read background project material
Task Description: The user can find general background information about the DELTAH Project	
Initiating Event: The user requests the 'background information' option	
Memory Context: Background information is available in the current language	
Observable Result: The background information is displayed	
Risk Factor: Low	*Change Factor*: Low
Related Stories: User can read promotional material, User can save promotional material	
Resource Estimates: O Input X Output O Enquiry O Database	X Simple O Average O Complex

O Other File Person-hours Estimate: 1	
Functional Tests: 1. The material can be read 2. Move to a different page [cancel]	
Quality Attributes: 1. The material is grammatically correct 2. The material is accurate 3. The material is written clearly 4. The material covers all the background information of the DELTAH project	
Customer Approval:	Date:

Figure 5.17 DELTAH story 2

Customer Story Card	Project Title: DPP
Date: 21 / 02 / 07	*Project Phase\Version*: Initial\1.0
Requirements Number: 3	*Story Name*: User can read promotional material
Task Description: The user can find promotional material about the DELTAH Project	
Initiating Event: The user requests the 'promotional material' option	
Memory Context: Promotional material is available in the current language	
Observable Result: The promotional material is displayed	
Risk Factor: Low	*Change Factor*: Low
Related Stories: User can save promotional material	
Resource Estimates: O Input X Output O Enquiry O Database O Other File	X Simple O Average O Complex

Person-hours Estimate: 1	
Functional Tests: 1. The material can be read 2. Move to a different page [cancel]	
Quality Attributes: 1. The material is grammatically correct 2. The material is accurate 3. The material is written clearly 4. The material covers all the promotional material of the DELTAH project	
Customer Approval:	Date:

Figure 5.18 DELTAH story 3

We will see how this way of looking at things is both useful for planning out a program but also for testing it. It will be the basis for our system metaphor.

5.3 User interfaces

So far, the concepts that we have discussed are oriented towards the needs of the developers, when it comes to communication with the customer, however, it is essential that we use ideas that he/she can understand. Many people look upon a software system from the perspective of how it presents itself to them. In other words *the system is the inteface!*

People are all different and differ greatly in the way they think and behave when using a software system. The designers of a user interface would seem, therefore, to have an almost impossible task when it comes to trying to satisfy every possible user of the system..

There is now a considerable amount of research and experience when it comes to this area. We will briefly review some of the commonly proposed principles that are recommended for the design of good graphical user interfaces (GUIs). Note, however, that if your client has some special factors, perhaps some of the users are handicapped in some way or have other special needs, than these will have to be investigated carefully and may result in some specialist features being incorporated in the interface.

A useful general reference on user inteface design is [Schneiderman1998]

Most user interfaces consist of a collection of windows and screens. These have two main purposes, one is to present information to the use, the other is to permit the user to carry out

some tasks. Naturally, many windows are a combination of both types.

If we are presenting information then there are some important principles that should be followed:

i) the information presented should not be confusing, contradictory or misleading;

ii) the words, icons and other visual metaphors used should be clear and understandable, the use of obscure technical jargon should be avoided;

iii) the screens should not be cluttered, full of irrelevant and distracting images and text, it should focus on the task in hand;

iv) the information should be up to date and presented in a consistent manner;

v) if the user is expected to do something it should be made clear what.

If the window is designed to allow the user to carry out some task then other important characteristics are desirable:

i) the action required to carry out the task should be clear, help should be given if appropriate;

ii) similar tasks under similar conditions should require similar actions;

iii) feedback should be given, if the operation was successful then this should be clear to the

user;

iv) it should be possible to recover if the action was not successful, make sure that nerror message are clear and helpful;

v) too many alternative ways to do the same thing can be confusing;

vi) similar actions should be broadly consistent, so don't use radically different techniques for actions which are very similar but taking place in slightly different states.

How these windows are organised is crucial. Many simple interfaces can be modelled, as we have seen, by using a state machine or an X-machine (XXM) (see Chapter 6). This is well worth doing as it will relate easily to the user stories and tasks that we have been thinking about earlier. There is a balance between the desire to provide lots of information and the need to keep it clear and simple.

We also have to decide how many windows to use, too many and users find things getting tedious, too few and they may get confused. The correct level can only be found by extensive consultation and trials with prospective users or their proxies.

Since we are using XP we can expect to show our customer examples of the sort of interfaces we are thinking of using, this is an opportunity for some useful feedback. Remember, that many inexperienced custmers and users often think that the system *is* the interface.

XP stresses the need for simplicity but do not interpret this to mean that the interfaces must be

very simple, they should be good but that may not mean the same. Interface design is a sophisticated skill, do not underestimate how hard it is. Test out your ideas as much as possible on potential users or on others with a similar background. Some student friends from other departments and schools could be helpful in this respect. The more experiments you do with people the better will be the result. Don't forget that people may have very different opinions about the same interface. Set up questionnaires to get some evaluation from anyone who uses it, getting them to evaluate it on the basis of how easy it is to learn, how easy it was to carry out the key tasks, how well it kept them informed about what it had done and what needed to be done next, and whether it worked without crashing or failing in other ways.

Ask users to rate the key features on a 1(poor) ... 5 (excellent) scale.

Check with the non-functional requirements identified earlier.

The system doesn't stop with the interface. The system will be situated within an overall enterprise and work flows and interactions in the company may be involved with it. Some of the tasks will be manual ones and the introduction of a new system may influence these and perhaps change them. Customers should be aware of the implications of introducing the new system and should plan for it properly.

It is sometimes tempting, when designing an interface, to want to use whatever the latest technical feature that you have learnt how to implement. This will be a bad idea in many cases. Only use appropriate technoilogy, not technology for technology's sake. Adding complexity, whether from a programming point of view or from the users', will threaten rather than enhance the system.

Ask the customer or the users which input techniques they want to use in the different contexts. Do they prefer selecting from a drop down menu, clicking on radio buttons, filling in forms, using hot keys etc.? Study the sort of systems that the users are currently using and are familiar with. Keep things similar if at all possible.

There are many different types of widgets that can be used, depending on the technology and any toolkit used : buttons of various types, sliders, drop down lists, combo box and so on can all be successfully used.

Don't forget the help system, this might be a key feature for some users. It should provide some basic support to enable users to get back to a point where they can then continue. Think about the key tasks that are identified from the user stories. Perhaps use each one as the basis for a help system.

An on-line manual is also a good idea. This is discussed in Chapter 10.

5.6 Communicating clearly with the customer and building confidence

At each meeting - in our case weekly meetings are the only practical face to face opportunities - we need to provide the customer/client with a progress report. Because some of our clients need to keep their managers informed as well it is best to provide them with a document that summarises the current state of the project and where it is going to. This can be achieved by producing a requirements document and updating it regularly.

After a few weeks this can become quite a detailed description of the system being built. It is

suggested that a working requirements document is kept updated as the requirements change, in particular, as the collection of stories grows and the system architecture develops this should be recorded in a coherent way and the requirements document is the place for this. Because a student's life is a varied one and many other course and activities will be taking place concurrently with this project it is important to keep everything orghanised so that there is no confusion about what is being done and where one is going. This is why the requirements document is important. In many industrial companies it will be based around a standard template.

A suggested agenda for a regular client meeting/

1. Progress update - what has been achieved since the last meeting;
2. Review of system requirements;
3. Demonstration of new code and interface mock-ups;
4. Changes needed to existing stories/requirements;
5. New stories/requirements;
6. Plans for the coming week;
7. Next meeting - time and place.

There will always be a number of issues that can upset the best of developer-client relationships.

Many customers expect software to be produced rather faster than is possible. It is important to know how and when to try to manage their expectations. The benefit of regular increments and deliveries is that the customer becomes more aware of progress. It is not always possible – nor even desirable in some cases – to focus on delivering working pieces of code if they are

so incomplete as to prevent the customer/user from properly exercising it in a sensible context. The use of screenshots and mock-ups together with lists of stories and even XXMs will help the customer to understand where things have got to and where they are going. Some authors have tried to use UML diagrams to illustrate some of the system design but to limited effect. Static diagrams such as class diagrams are unlikely to be appreciated by clients. Use case diagrams contain so little information that they are also of marginal value. Some of the activity diagrams can be useful if they are carefuly explained. We have found that XXMs are readily understood by customers and popular with developers compared with all the other approaches that we have tried. Interestingly, UML diagrams are widely disliked!

Sometimes, despite the best intentions of all concerned, relationships between clients and developers can become strained or even break down altogether.

This has happened in Genesys. Maybe it's because Genesys is a company operated by students but some – admittedly a small minority – of clients do not treat them with much respect and can act quite unprofessionally. One scenario we have experienced is the small business client who wants something for nothing. They often have a poor understanding of their own business and can change their mind in eratic ways. One customer, involved in SMS texting services, kept coming up with different technologies to use halfway through the project because he had found a cheaper telecoms service. This resulted in having to restart to project because of the incompatibility with what had been built with the new service technology. This didn't just happen once but at least 3 times! It was a major challenge to the concept of XP.

Another customer signed off a contract that had been successfully delivered and then decided

he wanted a lot of changes later. He did not accept that this should be the subject of a new contract and this led to a lot of stress amongst the team. Sometimes you have to be firm – the customer is not always right! Of course, if it comes to legal arguments then that is an indication of failure – but there is always a point where you have to stand up for your side of the bargain and avoid being exploited unfairly. Under these circumstances it is vital that you can demonstrate that you have carried out everything in a professional way, that you can produce the evidence to support your position and that you have behaved in a reasonable way. This is where a good source of documentation: minutes of meetings, requirem,ents documents and contracts, e-mails, test data and so on. If you don't keep this information in a systematic way then you deserve to be criticised by a smart lawyer.

5.7 Demonstrating the non-functional requirements

The format of the requirements document that we will be presenting to our client was discussed in the previous chapter and examples are given in Appendix A. The important thing about this document is that it should be understandable to the client. It is built from the basic information in the stories together with some outline ideas of the how the system might look and work. Important non-functional requirements need to be specified and clear statements about how these are to be interpreted and tested included. It is no good saying that the system will be fast if we don't say what that means, for example, in a Web based system this might include the maximum acceptable page download times under suitable conditions, etc.

Although we have presented the requirements document and the story cards as two separate things they are very closely related. There will be an interplay between them. We might regard the requirements document as a *summary* of our current understanding of the overall system whereas the stories are a more detailed description of individual aspects of its

functionality with enough information to allow us to plan, test and implement each story. The requirements document will have extra and vital information about the proposed system, context statements, assumptions as well as global quality attributes and non-functional requirements. It is these that we turn to next.

Non-Functional Requirements

A non-functional requirement either describes how well the system should perform (a quality attribute) or a constraint or limit that the system must adhere to (a resource attribute). The non-functional requirements were defined in Chapter 4, and can be split into categories like *reliability, usability, efficiency, maintainability* and *portability etc*. Here we give some example statements that might be part of the requirements document.

Reliability:

Examples are:

For a single user, the system should crash no more than once per 10 hours.

The system should produce the correct values for any mathematical expression 100% of the time.

If the system crashes, it should behave perfectly normally when loaded up again with minimal data loss.

Usability:

A user should be able to add a new customer to the system within 1 minute.

A user should be able to add a new order to the system within 1 minute.

A user should be able to edit a customer's details within 5 minutes (will vary with details

type).

A user should be able to produce reports and statistics within 1 minute.

Efficiency:

The system should load up within 15 seconds.

The time taken for the system to retrieve data from the server should never exceed 30 seconds.

Maintainability:

The system should be designed in such a way that makes it easy to be modified in the future.

The system should be designed in such a way that makes it easy to be tested.

Portability:

The client system should work on the client's current computer network which is connected to the Internet and has Windows 95 or better.

The system should be easy to install.

These statements need to be refined into a more precise statement in order to make them testable. What, for example, does *easy to install* mean? we will look at this in the next Chapter.

From each story that we have discussed with the client we extract the key functional details. These are grouped in sections with other story lines that are clearly related.

These requirements are categorized on the basis of which are *mandatory, desirable* and *optional*. To do this we need to have an estimate of the time we might take to complete these and this will help us to make these decisions. The next section looks at the process of trying

to estimate this.

Naturally we must consult the client on which he/she thinks are mandatory etc. *We have to be realistic, however, you must not promise to do more than you can achieve in the time given.*

5.8 Estimating resources

If we have a model like the one above we can use something like the *function point* and *object point method*. Here we try to estimate the amount of effort required to build a story or a screen with its accompanying functionality.

To do this we look at each story and consider the functions contained in it. We then try to categorise what sort of function this is. We can find information which estimates the amount of effort each category of function might require, this data is being collected in industrial organisations and some of it is published. We include some examples here. It is a good practice to try to measure your own efforts for these functions to see if they agree with the estimates and to inform future projects.

Software cost estimation

We need to ask some basic questions at the *start* of the project and also at suitable review points *during* the project:

How long will it take?

What resources will it need?

How expensive will it be?

The standard approach is to use the techniques of Software Measurement, however, there are overheads involved in doing this and we need to consider to what extent it is wrth doing this.

During the course of projects we measure the following parameters:

lines of code (loc) produced over the timescale;
number of observed defects over the timescale;
number of person hours worked over the timescale;
amount of time spent on debugging over the timescale;
amount of time spent on requirements over the timescale;
amount of time spent on design/specification/analysis over the timescale;
amount of time spent on writing documentation over the timescale;
amount of time spent on testing/review over the timescale;
etc.

These are all measures of production volume, product quality and effort. If we have some previous experience and data of this type for old and similar projects we may be able to estimate the effort and time needed for the new project. In many cases the type of project is of a new type, the technology being used may be unfamiliar and the programmers may also be different to previous projects. This it is a difficult issue to decid what is best.

From the timesheets and documentation produced we should be able to find the following for the completed project:

lines of code (loc) per person-month (pm);

cost per 1000 lines of code (kloc);

errors per kloc;

defects per kloc;

pages of documentation per kloc;

cost per page of documentation;

number of requirements;

average kloc per requirement.

If we have this data then we might be able to estimate what the next project will need in terms of people and time.

But different types of project will require different amounts of effort, so we need to collect information about the type of project:

product functionality;

product quality;

product complexity;

product reliability requirements;

etc.

These are not always easy to measure, unlike the first set of measures.

We need to describe the software being built on the basis of the requirements in order to estimate the resources needed. There are several techniques, none of which are very precise.

If the new project is very similar to the previous ones things are much simpler. If it is a completely new type of project, perhaps involving a new technology, then it is much more difficult.

We can pick out a few simple principles from function point analysis that can be helpful as long as they do not become too time consuming. Function point analysis (FPA) was developed by Albrecht in 1979, [Albrecht1979] for business information systems development.

1) for each requirement/story we decide if it is one of: *input, output, inquiry, reference file, database.*
2) assign a weight to each requirement: *simple, average, complex*.
3) consider other influencing factors: reusability, adaptability and weigh them according to a suitable scale.

This is, to an extent, guesswork but if we have an idea of which requirements are hard to implement and which are easier it will help us to plan.

Assigning these attributes needs some experience but then what? Ideally there is a database of previous similar projects which can be analysed and conclusions on the effort required to complete a story made.

Object point analysis

This method was introduced by Banker, Kauffman et al in 1992 [Banker1992].

It is based on:

the number of separate screens - simple = 1 object point, average = 2, complex = 3.
the number of reports to be produced - simple = 2 object points, average = 5, complex = 8.
the number of modules that must be developed, 10 object points for each module.

A module will be any small coherent part of the system; it could be a screen or a story.

It is easier to calculate this from the high level requirements.

The COCOMO model, [Boehm1995], is another estimation process which is based on industrial data, see any Software engineering text such as [Pressman2005] or [Sommerville2006] for more details.

COSMIC FFP

This is an updated version of the Function Point Analysis.

It is based around a very simple definition of a piece of software functionality:

> a function Process is a unique set of data movements: input, output, read and write.

Actors trigger these movements, directly or indirectly. These movements are also identified at the lowest level in terms of the requirements, that is, we do not break them down into smaller functions. Events will trigger these functions and we consider them to run until they complete.

Then we sum up how many of these functions there are, ignoring their type or any other factor.

Such a simple method of estimating might be very useful for XP projects; it still requires the collection of data about a team's velocity in order to be useful, though.

Further details can be found at http://www.cosmiccon.com

Whatever technique is used there is a number generated – function points, object points, etc. What does this mean?

Well, it means nothing if there are no data from similar projects available. This is the key.

For each story carry out some resource estimate and then record this to generate some kind of useful data for the future.

As projects progress the reliability of these estimates will be discovered and some amendments can be made to the method.

What is important is that some method and review is attempted. We do not prescribe any particular approach since it is clear that there is so much variability in projects, teams and technologies that it is hard to come up with a *magic* solution to the problem. The key point is to measure something on a regular basis and use that information to help with developing an understanding about how future projects might progress.

We have developed an Eclipse plug-in, PlanClipse, to support simple day by day planning activities and feedback. It is available from:

http://www.genesys.shef.ac.uk/eclipse/planclipse/index.html

http://www.genesys.shef.ac.uk/eclipse/planclipse/2.2/org.eclipse.adpt.planner_2.2.1.jar

The basic idea is that as each story is completed – unit tests written, coding done and tested such that all tests are passed – it is recorded in the tool together with any decisions about those stories that are still to be done. This is done using a story planning chart. If the project has a fixed delivery date then there may some scope for adjusting the list of mandatory stories, reducing the number if things are taking too long and increasing them if things are going well. There are usually some desirable and optional stories that can be included if an earlier delivery is not needed.

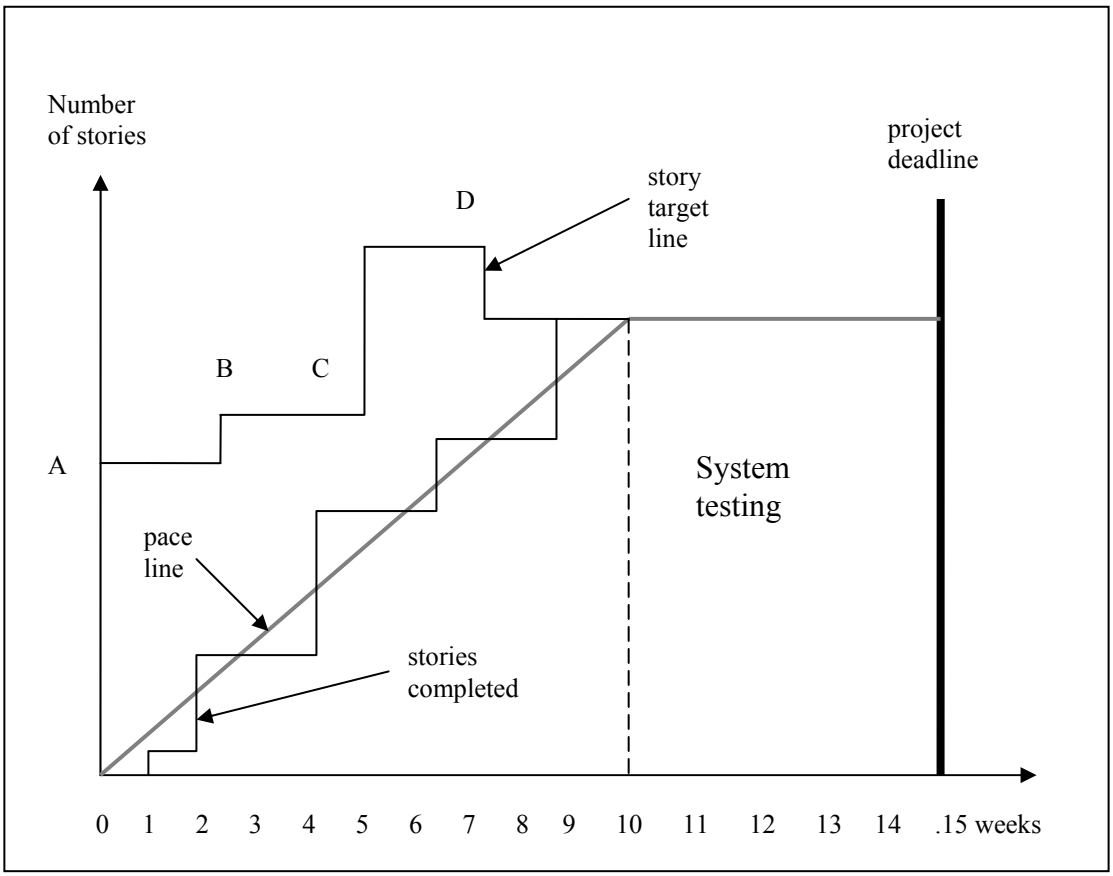

Figure 5.19 A story planning chart

The chart in Figure 5.19 plots weekly progress. Initially we estimate the number of stories that will be implements – as well as we can – in the time available for the project. This is point A. As the project develops we check of the number of stories completed – so initially there was good progress and after 3 weeks things were going well. As a result some extra stories were then planned, point B. The estimate of progress to completion indicated by the progress or pace line was drawn. The challenge is to keep above that line as far as possible. At C the number of stories was increased because it looked as though the plan was too pessimistic. Finally, the progress made by week 8 was reviewed another revision of the story

list at point D. This resulted in a reduction of the number of stories in order to allow for the time for system testing. There may be further changes as the project goes into the last few weeks but these must be considered in terms of the risks and benefits to the project – risks of over-running and threats to the quality of the system due to inadequate systems testing versus the benefits of better alignment with the current needs of the client.

It is absolutely vital that enough time is left for thorough systems testing before delivery – IF THIS IS NOT DONE THERE WILL BE SERIOUS PROBLEMS.

The benefit of this approach is that it is reactive and progress can be measured easily and tactics adapted to meet the demands of the situation.

5.9 Review

This chapter has looked at how the planning game and the use of story cards can help us to develop a detailed requirements document. Although the requirements are changing it is important to bring all the changes into a single document at suitable stages through the document. We considered how the different functional requirements can be integrated into a simple (extreme) model which helps us to think through how the system, will work overall. These processes will be repeated as the requirements emerge and change. Chapter 9 will examine some of the issues relating to this evolution of the system and the ways it can be articulated and managed.

Non-functional requirements were identified as a key factor in the success of a system. These

need to be thought about very carefully and clear and measurable statements made about them.

Estimating the resources needed to complete a project is notoriously hard and two techniques function point and object point analysis were briefly described. These can only be rough and ready guides until you get more experienced. Noting down as much detail about your team's performance and the time taken to do things will provide an ongoing and useful archive for future projects.

Conundrum

The company wanted an *intranet* that provided support for many of their business activities and also their personnel management. The site would contain information about the various company activities, a diary system and templates for administrative tasks such as the submission of illness and absence forms. The users would be able to log on remotely to carry out tasks as well as from within the company offices. The customer was able to maintain a very close relationship with the development team and had a clear idea of what the company needed. There were 3 teams using XP working on this project, each competing with all the others. Initially all the teams thought that the project would take 10 weeks. It didn't quite work out like that. When the first team delivered their first increment they discovered something important that had not emerged from the planning game. The company had a service agreement with a third party network solutions company which provided the computer system and the internet connection for the customer. This led to a serious problem for some of the teams and resulted in some failing to meet the 10 week deadline despite the careful planning.

What might have been the problem?

Exercises

1. Read Appendix A. These requirements relate to a real project that were successfully implemented using XP.

2. Prepare your own requirements document for your client for submission also to your tutor. The contents are specified below. Use the planning game to create the individual requirements for the document.

Your requirements statement should contain the following sections and paragraphs:

Introduction - a statement of the required system's purpose and objectives

Dependencies and assumptions - things that will be required for your system to meet its specification, but which are outside your control and not your responsibility

Constraints - things which will limit the type of solution you can deliver, e.g. particular data formats, hardware platforms, legal standards

Functional requirements - you are advised to priorities your requirements into those that are:

mandatory

desirable

optional

Non-functional requirements - with accurate definitions and an indication of how they are to be measured and the level required.

User characteristics - who will the users be?

User interface characteristics - some indication of how the interface needs to be structured and its properties.

Plan of action - defining milestones - key points in the project

deliverables - an indication of when increments will be ready

times when these events will occur.

Glossary of terms.

Any other information such as important references or data sources etc.

Here is a simple tabular template that could be used for some of the functional and non-functional requirements. It includes a column for trying to set priorities for the individual requirements and to identify the risk of change in the requirement, a difficult thing to estimate but worthwhile for planning purposes.

Number	**Description**	**Mandatory / Optional / Desirable**	**Priority (1-9)**	**Risk (1-9)**	**Function point**
1.					

References.

[Albrecht1979] A. J. Albrecht, "Measuring application development productivity", *SHARE/ GUIDE/IBM Application development Symposium,* 1979.

[Banker1992] R. Banker, R. Kauffman et al, "An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment", *J. Management Inf. Systems*, 8, 127-150, 1992

[Boehm1995] B. Boehm et al, "*Cost models for future life cycle processes: COCOMO 2* ", Balzer Science, 1995.

[Holcombe1998] M. Holcombe & F. Ipate, "*Correct systems: building a business process solution*", Springer, 1998 available on-line at: http://www.dcs.shef.ac.uk/~wmlh/

[Pressman2005] R. S. Pressman, *"Software Engineering a practitioner's approach*", McGraw Hill, 2005.

[Sommerville2006] I. Sommerville. *Software Engineering*, 8th edition, Addison-Wesley, 2006.

[Thomson2003] Thomson, C., & Holcombe, M. (2003). *Applying XP Ideas Formally: The Story Card and Extreme X-Machines.* In the Proceedings of the 1st South-East European Workshop on Formal Methods. Thessaloniki, Greece,, South-East European Research Centre, 57-71

[Thomson2005] Thomson, C., & Holcombe, M. (2005). *Using a formal method to model software design in XP projects.* In the Proceedings of the 2nd South-East European Workshop on Formal Methods. Ohrid, FYR of Macedonia, AMCT, Vol. 1 (3)