Chapter 5

Identifying user stories and the planning game

Summary:
Creating stories and learning how to analyse and negotiate with stories. Identifying functional requirements. Extreme modelling. Non-functional requirements and quality attributes. Techniques for estimating resources.

1. Looking at the user stories.

In the main the user stories will comprise short sentences which describe a sequence of actions and events that are recognisable from the perspective of a user interface. 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.

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 the only team involved and your client will provide you with the key concepts for their business process and business needs.

We'll take some simple examples of user stories from the *Quizmaster* system to illustrate the process. Note that these can be traced back to the appendix document from the previous chapter.

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. Recall that we gave a simple tabular description of each story. The columns 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,
6. what is the estimated cost of the story.
7. what is the likelihood of the story being changed or dropped?

A story card should be created for each of these and we put on the card the information described. A possible template for a story card is given in Figs. 1 and 2. Some of the topics will be discussed later.

Customer story card	**Project title** ______________________________
Date _____________	**Project phase/iteration** ______________
Requirements number _____________	**Story name** ___________________
Task description (English)	
Initiating event	
Memory context	
Observable result	
Risk factor	**Change factor**
Related stories	
Notes	

Fig.1 Story card template

Story name _______________	
Resource estimates input ☐ output ☐ enquiry ☐ reference file ☐ database ☐ Function/object point total __________	simple ☐ average ☐ complex ☐ Man-hours total _________
Functional tests	
Quality attributes	

Fig 2. Reverse of story card.

Now let's return to the Quizmaster project and consider the first requirement in the document in the appendix, requirement 1, Lecturers can add topics. This will be a story card. Figure 3 shows how a story card might be written, some of the details are missing, in particular the resource estimates which we will return to. Notice that the story card implies the existence of a database of some sort which will contain information about papers and topics, this is extracted through the questions about what memory interactions there are.

Customer story card	Project title Quizmaster
Date March 15th 2001 Requirements number 1	Project phase Initial Story name Lecturers can add topics.
Task description (English) A function whereby a registered lecturer can define a topic for a paper and this information is stored suitably by the system	
Initiating event A request is made through chosing a menu option from a suitable screen	
Memory context A record of papers and topics exists and will be updated	
Observable result Confirmation of success and the ability to generate a list of papers and topics at any time.	
Risk factor 1 (low)	Change factor 1 (low)
Related stories Lecturers can delete a topic	
Notes Mandatory	

Story name Lecturers can add topics.	
Resource estimates input ■ output ☐ enquiry ☐ reference file ☐ database ☐ Function/object point total __1______	simple ■ average ☐ complex ☐ Man-hours total _12_______
Functional tests 1. Define a topic when a paper is already defined 2. Define a topic when no paper is defined [error] 3. Define a n illegal topic type [error] 4. Do nothing and exit the function [cancel]	
Quality attributes The process of defining the topic and receiving confirmation of correct operation should be instantaneous The operation process should be clear from the interface design Trials of this function amomgst representative users should be 99% successful	

Fig. 3 A story card for a requirement.

Now we might identify another story which could be, for example:

3	Lecturers can edit details of a topic.

Customer story card	**Project title** Quizmaster
Date March 15th 2001	**Project phase** Initial
Requirements number 3	**Story name** Lecturers can edit details of a topic.
Task description (English) Lecturers may inspect the list of topics and papers and change the detail of the topic for a paper	
Initiating event Lecturer requests edit option	
Memory context Current list of papers and topics is updated to new list	
Observable result Updated memory changes confirmed	
Risk factor 1	**Change factor** 1
Related stories Lecturers can add topics.	
Notes	

Story name ______________

Resource estimates

input	☐	simple	■
output	☐	average	☐
enquiry	☐	complex	☐
reference file	☐		
database	■		

Function/object point total 1 **Man-hours total** 12

Functional tests

1. Carry out an update on an existing record
2. Carry out an update for a non-existant record [error]
3. Define a n illegal topic type [error]
4. Do nothing and exit the function [cancel]

Quality attributes

The process of updating the topic and receiving confirmation of correct operation should be instantaneous

The operation process should be clear from the interface design

Trials of this function amomgst representative users should be 99% successful

2. Extreme modelling (XM)

Extreme programming is a movement which tries to reduce the initial analysis and design stages so that the project can evolve without the baggage of lots of documentation and rigid design models. However, its success is dependent on identifying really rigorous system test sets since these are what constrains the software solution to a high quality and correct solution.

It is impossible to define stringent test sets without knowing some details of what the system has to do. This is the basic dilemma that we face and one that has not been fully resolved within the XP movement.

Our approach is to apply an extremely powerful modelling paradigm that is easy to use and has *proven* excellence in the construction of functional test sets.

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.

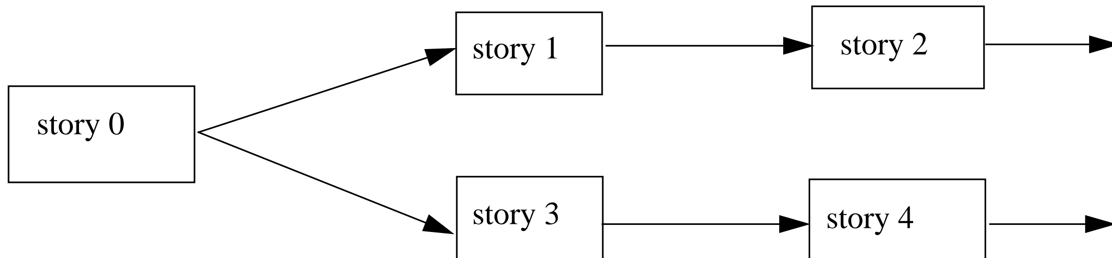

Fig. 3. Collections of stories.

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 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.

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. 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,

A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the refinement of the system as explained in a later section [Holcombe98], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this, [BOG00].

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

Table 1. Requirements table (part)

The table above describes some of the functions from the stories in this form.

Now consider some simple screens which may help us to visualise how it might work in practice.

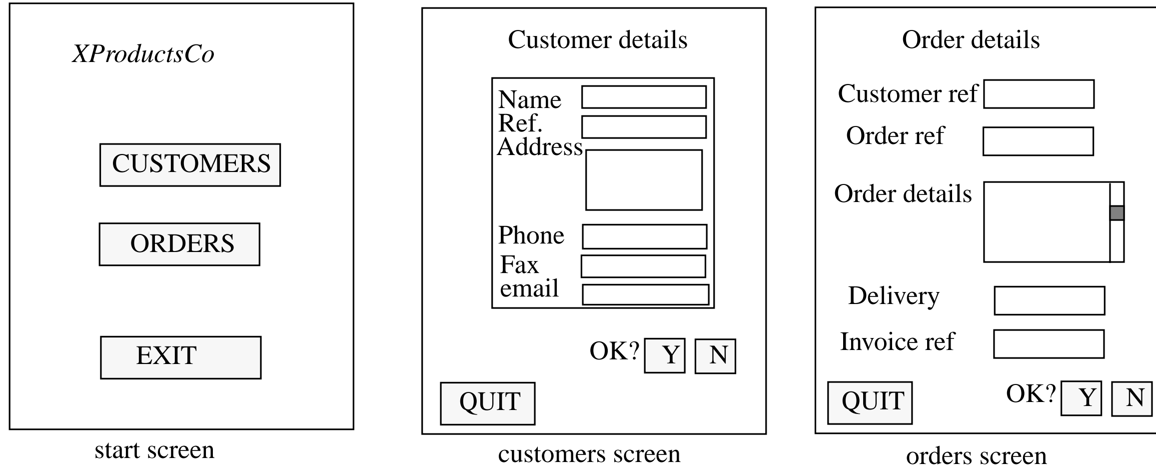

Fig. 4. Some simple sample screens.

The confirmation screens are not shown. Notice that we have filled in some of the data details which are not explicitly described in the stories.

The next task is to think about the order in which these screens will be deployed and the tasks that can be done on each screen. This information will be described using a state diagram.

From the diagram (Fig.5) one can see how the basic functions are organised. We build up a simple model like a state machine. Each state has associated with it an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, *refinements* that can be dealt with later and the work we do on test set generation here is built on them. The complete state machine is actually called an *X-machine*, a term first introduced in 1974 (so we use the abbreviation XM to mean both X-machine and eXtreme Modelling). These machines differ from the standard finite state machine in the sense that there is a memory in the machine and the transitions involve functions that manipulate the memory as and when an input is received. There are some more details about X-machines in Appendix C.

This is explained by referring to the requirements table and the story cards where the memory connection is described.

For example, the memory in this case is likely to be the database that contains the record of customers and orders.

The function click(customer) simply navigates between two screens and has no memory implications, but the function enter(customer) will involve some interaction with the database, it might search to see if the supposed new customer is in fact new before proceeding - generating a message if the customer is already on the database. This can be described by the abort(customer)

function which has been developed as a result of thinking about the way the system fits together.

The function confirm(customer) actually changes the database by updating the records with the information relating to the new customer.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

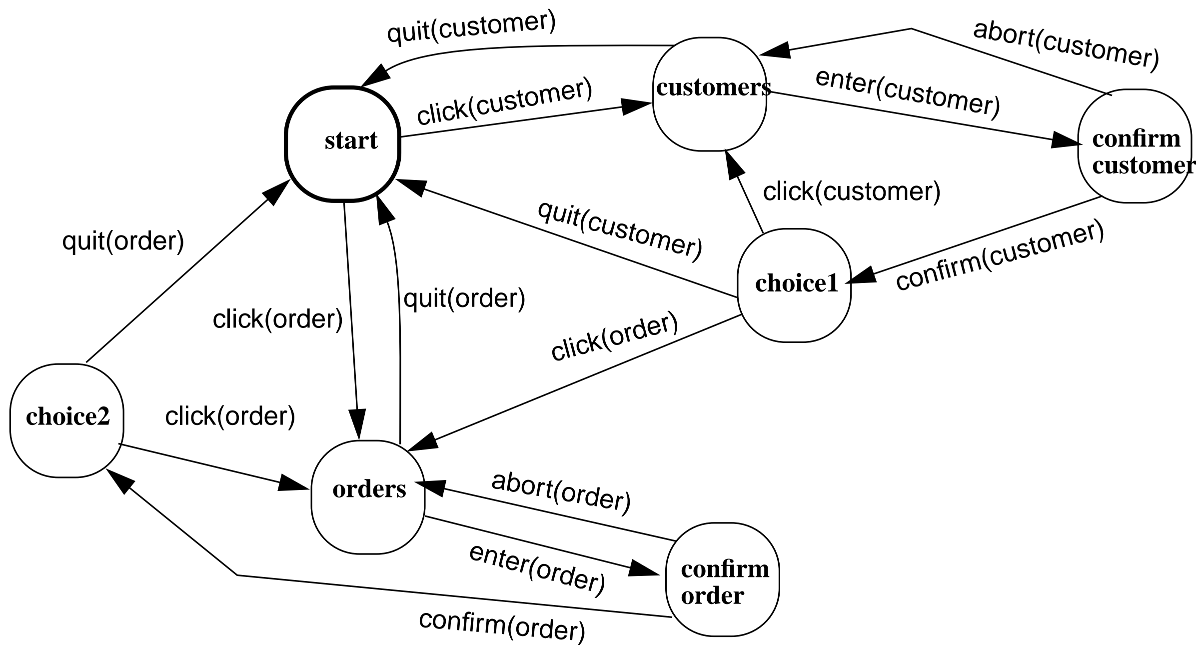

Fig. 5 X-machine diagram

Looking at Fig. 5 there are a few other things to note. We have created a specific user interface architecture which will constrain the solution in particular ways. For example it is not possible to quit the process of inserting the order details during this process. We can abort *after* completing it, however. This returns us to the **orders** state, then we can quit this screen.

The diagram can be used to define when and under what conditions different processes and stories are available.

3. The requirements document.

The format of the reqiuirements document that we will be presenting to our client was discussed in the previous chapter and an example is given in Appendix A. The important thing about this document is tha 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..

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 can be split into categories like *reliability, usability, efficiency, maintainability* and *portability etc*.

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 more than 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 got at least Windows 95.
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 categorised 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.*

4. 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.

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 approach is to use the techniques of Software measurement.

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 man 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 projects we may be able to estimate the effort and time needed for the new project.

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 waht 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 will look at the *function point method* (see the Function Point User Group - www.ifpug.org). There is also a more modern method called *object point analysis*.

Function point analysis (FPA) was developed by Albrecht in 1979 for business information systems development.

The principle underlying FPA is that the effort required to construct a software system is a function of the size and complexity of the product. The size is determined by the number and complexity of the requirements not the size of the code.

<u>Method</u>.

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.
Below are a list of influencing factors that could either make the requirements easier to implement or harder to implement depending on how much influence these factors have on the system. Each factor is allocated a number range and we try to estimate from this the likely weight of the factor.

Table 1: Input data (keyboard, screen, from other applications)

criteria	simple	average	complex
number of different data elements	1-5	6-19	>10
input validation	formal	formal, logic	formal, logic, database
ease of use	low	average	high

This means that if there are 4 data elements on a form and there is little data validation and ease of use is not an issue then the input data requirement is *simple*. If there had to be some logic based data validation then it would be *average* and if ease of use was a big issue then it would be *complex*.

Table 2: Output data (monitor, printer, to other application)

criteria	simple	average	complex
number of columns	1-6	7-15	>15
number of different data elements	1-5	6-10	>10
number of formatted data elements	none	some	many

Table 3: Inquiries (search and display)

criteria	simple	average	complex
number of different keys	1	2	>2
ease of use	low	average	high

Table 4: Reference file (tables and read-only data)

criteria	simple	average	complex
tables: number of different data elements	1-5	6-10	>10
tables: dimension	1	2	3
read-only files: number of different data elements	1-5	6-10	>10
read-only files: number of keys/sentence formats	1	2	>2

Table 5: Databases (managed by the application)

criteria	simple	average	complex
number of keys/sentence formats	1	2	>2
number of different data elements	1-20	21-40	>40
database already exists?	yes	-	no
new implementation?	no	yes	-

Influencing factors.
Having identified the key functionality of the requirements we need to factor in the effects of a variety of important aspects. These include the extent to which the application will interface with ther applications - often a source of interface problems, the organisation of data stores - distributed storage and procesing adds to the complexity of the design, the transaction rates expected - high transaction rates bring special considerations, being able to reuse some parts of an existing system, the amount of data conversion involved - perhaps we have to migrate data from an obsolete system. These issues need to be considered and the following table provides some details of how to do this. For those factors given a scale of 0 - 5 we define these by:

0 = no influence, 1 = occasional influence, 2 = moderate influence, 3 = average influence, 4 = important influence, 5 - strong influence.

For the scale 0 - 10 these points get double weighting.

Table 6: Influencing factors.

Factor
1) Interplay with other application systems (0-5)
2) Decentralised data (processing) (0-5)
3) Transaction rate (0-5)

4) Processing
(a) Calculations/arithmetic (0-10)
(b) Control (0-5)
(c) Exceptions (0-10)
(d) Logic (0-5)
5) Re-usability (0-5)
6) Data conversions (0-5)
7) Adaptability (0-5)

Re-usability is rated as follows: 0 = less than 10% reuse, 1 = between 10% and 20%, 2 = between 20% and 30%, 3 = between 30% and 40%, 4 = between 40% and 50%, 5 = above 50%.
We now consider all the requirements and create a table which summarises the function point position: here is an example:

Table 7: Example function point table - Quizmaster system.

Category	Number	Classification	Weight	SUM(FPs)
Input data	7	simple	3	21
	5	average	4	20
	1	complex	6	6
Inquiries	2	simple	3	6
	0	average	4	0
	0	complex	6	0
Total				53

Here there are 7 simple input requirements/stories each given a weighting of 3 making a function point score of 7 * 3 = 21. We do this for all the requirements/stories, only part of the full table is given see Appendix B for the details.

The next stage is to calculate the sum of the contribution of the influencing factors. If the sum of the influencing factors in N then the total function point contribution of the influencing factors is N/100 + 0.7

An example is given in the following table:

Table 8: Example influence factor table - Quizmaster system)

Factor	Score
1) Interplay with other application systems (0-5)	0

2) Decentralised data (processing) (0-5)	2
3) Transaction rate (0-5)	2
4) Processing	
(a) Calculations/arithmetic (0-10)	3
(b) Control (0-5)	2
(c) Exceptions (0-10)	4
(d) Logic (0-5)	2
5) Re-usability (0-5)	1
6) Data conversions (0-5)	1
7) Adaptability (0-5)	1

This influence factor value is then multiplied by the function point total from table 7 to give the final function point value for the list of requirements

The historical data on function point effort can now be used to calculate the resource required to implement a suitable system.
Data is available from a number of sources which can help us to make the time and resource estimates. In an ideal world these would be available from previous student projects but collecting reliable data on person months is always difficult.

Here is a table of person month data for various function point values derived form some commercial projects in IBM and VW.

Table 6:

Function points	IBM person months	VW person months
50	2.3	-
100	5.6	-
150	9.5	-
200	13.9	11.7
250	18.6	19.3
300	23.6	27.1
350	28.9	35
400	34.4	43

Table 6:

Function points	IBM person months	VW person months
450	40.1	51.1
500	46.1	59.6
550	52.2	68.2
600	58.5	77
650	65	86.1
700	71.6	95.3
750	78.4	104.8
800	85.3	114.6
850	92.4	124.7
900	99.6	135.2
950	106.9	146
1000	114.4	157.3

This data is based on averages taken over a variety of projects with many different project teams. It can only be a rough guide until you have established your own data.

It also depends on the programming language used, generally the number of lines of code per function point varies from 128 for C to 22 for Smalltalk. Java is probably around 30.

Given that the number of requirements that we have are more than we can deliver in the time scale we have to decide which ones are *mandatory* or essential, which are *desirable* but not quite essential and which are *optional*. The function point values can then be used to work out what requirements are feasible within the resources and time available. For these types of projects the amount of person months available for each team will vary from 2 - 3. So function point totals of around 50 are feasible.

Object point analysis.
This method was introduced by Banker et al in 1992.

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

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

The COCOMO model is another estimation process which is based on industrial data. See any Software engineering text such as Pressman, Sommerville, etc.

5. Dealing with change - refining stories.

Exercise.
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 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 prioritise 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.					