Reusing Requirements: Taking Advantage of What you Know

Suzanne Robertson
The Atlantic Systems Guild Ltd
suzanne@systemsguild.com
http://www.systemsguild.com/

When your friend who lives in the UK tells you that her new telephone number is 020 7262 3395, unless you are an idiot savant, you write it down. You expect to use the number many times in the future so, rather than just jotting it onto a scrap of paper you put it in your address book. Or you record it in your computer or your handheld personal organiser. Your friend lives in inner London and so do you, so when you phone her; you dial all the digits in the number. However, when you phone your friend from outside London you note that the leading zero (internal district indicator) is not necessary so you write the number like this (0)20 7262 3395. Now, because this way of writing down phone numbers is very familiar, if you give this number to anyone else in the United Kingdom there's a good chance they will know how to phone your friend.

But supposing you are phoning from overseas, then you will need to add a country code prefix (44 for the United Kingdom) and you will write the number as 44 (0)207 262 3395. One more thing that is necessary is the international code. This is different depending on the country you are calling from and is usually noted with a plus sign to remind you to dial the appropriate international access code if you are phoning from overseas. So the complete number is written as +44 (0)207 262 3395. Whenever you want to contact your friend, rather than ask for the number again, you reuse the knowledge you have previously written down.

The more examples of phone numbers we come across, the better we become at discovering patterns of reusable knowledge. For example if we see several London phone numbers we can observe that they are consistently recorded as:

International access code +
Country code nn
Optional internal district indicator (n)
District code nn
Number nnnnnnn

Without even thinking about it, we generalise our knowledge and reuse it to make sense of other phone numbers. But suppose we are given a telephone number like + 44 (0)1379 677418. Then we need to ask whether the district code (4 integers) and the number (6 integers) are correct because they do not conform to the pattern with which we are familiar. Given the answer that the telephone number is correct, we add to our reusable knowledge. Now we know that district codes can be 2 or 4 integers and numbers can be 6 or 7 integers. The more examples we come across the more we can add to our reusable knowledge about telephone numbers. And if we record that knowledge in some consistent fashion then it is also reusable by other people.

In our personal lives, all day every day, we reuse knowledge that we have previously discovered. If we did not, we would never have time to do anything new and

would probably drive ourselves and everyone around us crazy. So, consciously and unconsciously, we look for reusable patterns of knowledge to help our communications with the rest of the world.

What is requirements reuse?

The aim is to save time and effort and build better relationships by avoiding the duplication of work that has already been done. Requirements reuse is the ability to benefit from requirements knowledge that has already been captured. This does not mean that the all the requirements knowledge has to be formally defined in exactly the same way. If the knowledge is somehow recognisable and accessible then it is potentially reusable.

For example, suppose a colleague tells you that he has worked on a similar project to yours and has a list of all the stakeholders who were involved. You would use that list as the starting point for discovering the stakeholders for your project. There would probably be some differences, but if you have something to start with you can reuse all the work that your colleague did, tailor it for your own needs and save a lot of time. If your colleague has organised the knowledge in some consistent form like: stakeholder name, stakeholder role, stakeholder influence, stakeholder knowledge type……then there is a linguistic structure for talking about this knowledge and this increases the opportunity for other people to reuse it.

Any knowledge is potentially reusable; the interesting question is whether reuse provides a benefit. If something is recorded in a very obscure and ambiguous way then, rather than spending time trying to make the correct interpretation, it might be quicker to go back and ask the questions again. On the other hand, if the knowledge is recorded with some degree of consistency and formality, then it is more accessible and reuse is likely to be a paying proposition. Given that reusable requirements exist in many forms, the next question is – how do we discover whether or not we have any requirements worth reusing?

Reusing Functional Requirements

Functional requirements are related to the existence and use of functional data that a product (any product, not necessarily software) is required to create, reference, update and delete. Functional requirements specify what a product should do. These requirements consider the state of the data, the business rules applied to the data and the business meaning of the data. In 1983 Tom DeMarco suggested [DeM1] that, due to the complexity of functional requirements, we need to combine a number of different views to build a model that defines all the functional requirements. This composite model incorporates projections into three different spaces: function space, information space and state space. The good news is that over the last twenty years we have developed a variety of systems analysis models to capture these different aspects of functional requirements. We have used combinations of these models to capture functional requirements and we have built up considerable skills in specifying these requirements. The bad news – if we view it as such – is that we do have many different models that satisfy the same, similar or overlapping purposes. This, in itself, is not a bad thing because diversity fuels creative thinking. However it presents some problems if you are searching for consistency to aid in easy reuse.

Functional requirements models

Figure 1 is a summary of models that are commonly used to specify the functional requirements in the data, function and state spaces suggested by DeMarco.

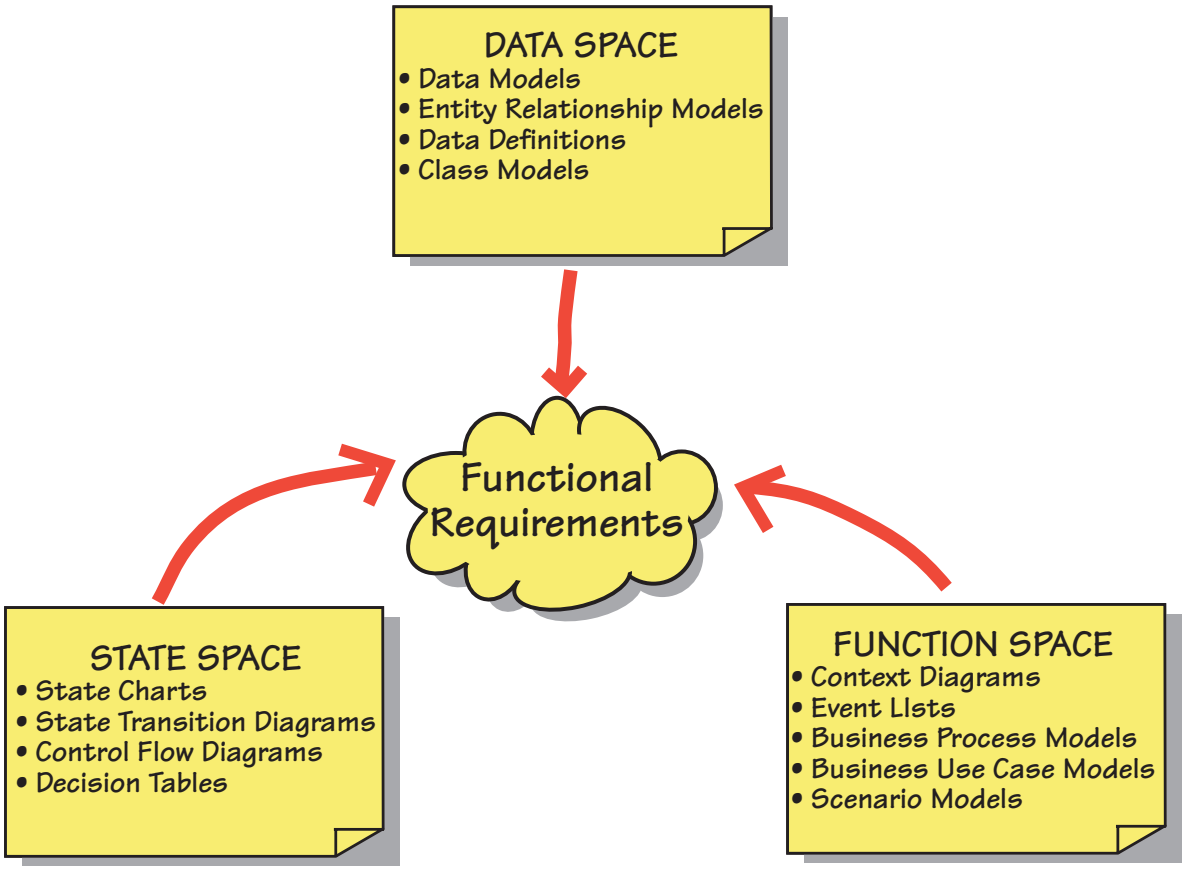

Figure 1: You can find reusable functional requirements in a variety of systems analysis models

There are many different ways to model the data space, some that are commonly used are: Peter Chen's Entity Relationship Diagrams and Unified Modelling Language (UML) Class Models – and there are many other variations with a variety of notations. Regardless of the notation, models in the Data space define logical groupings of the essential subjects, their attributes and the relationships between them. These models are a picture of the facts. They capture static requirements; the knowledge that has to exist and be maintained in order to carry out the activities in the particular domain.

Models in the Function space are dynamic. They are concerned with processes and activities. These models capture what happens when rules are applied to the data in order to make decisions, calculations, communications, evaluations. The wide variety of models available to define the function space include: Context diagrams, Event Lists, Business Process Models, Business Use Case Models, Scenario Models.

Models in the State space capture a different set of dynamics. These models define the steady states of existence of part or all of a system along with the transitions from one state to another. Once again there are many variations such as: David Harel's State Charts,

State Transition Diagrams, Systems Dynamics Models, Control Flow Diagrams and Decision Tables.

If you were specifying a taxi booking system, then models in the data space would specify entities like taxi and driver along with associations like which driver/s own which taxi/s. Models in the function space would focus on the processes and rules concerned with things like when a passenger books a taxi or when a despatcher schedules a taxi for a job. And in the state space you will discover that a taxi can be in a number of different states like vacant, occupied, out of service. Also you would see the triggers that cause the taxi to go from one state to another.

Regardless of the notation that you are using, if it is used consistently then there is the potential for reusing requirements knowledge.

Start with data models

If there exists some kind of a data model that covers all or part of the data space in which your project is interested, then it is a good starting point for looking for reusable requirements. The reason I suggest that you start here is because a business data model is at a higher degree of abstraction than the more dynamic models in the function and the state space, therefore it is more likely that you will find directly reusable requirements.

For example, figure 2 is a class model that captures some of the functional requirements for an organisation that sells books. The model identifies the classes such as customer and book order along with the associations between the classes. For example the class *customer* might have many (*) *book orders* and a *book order* is only associated with one (1) *customer*. Suppose that this model is part of the specification for a project that builds a web site for *customers* to buy *books*. The data model captures and communicates your knowledge about the business data that is germane to this specific project.

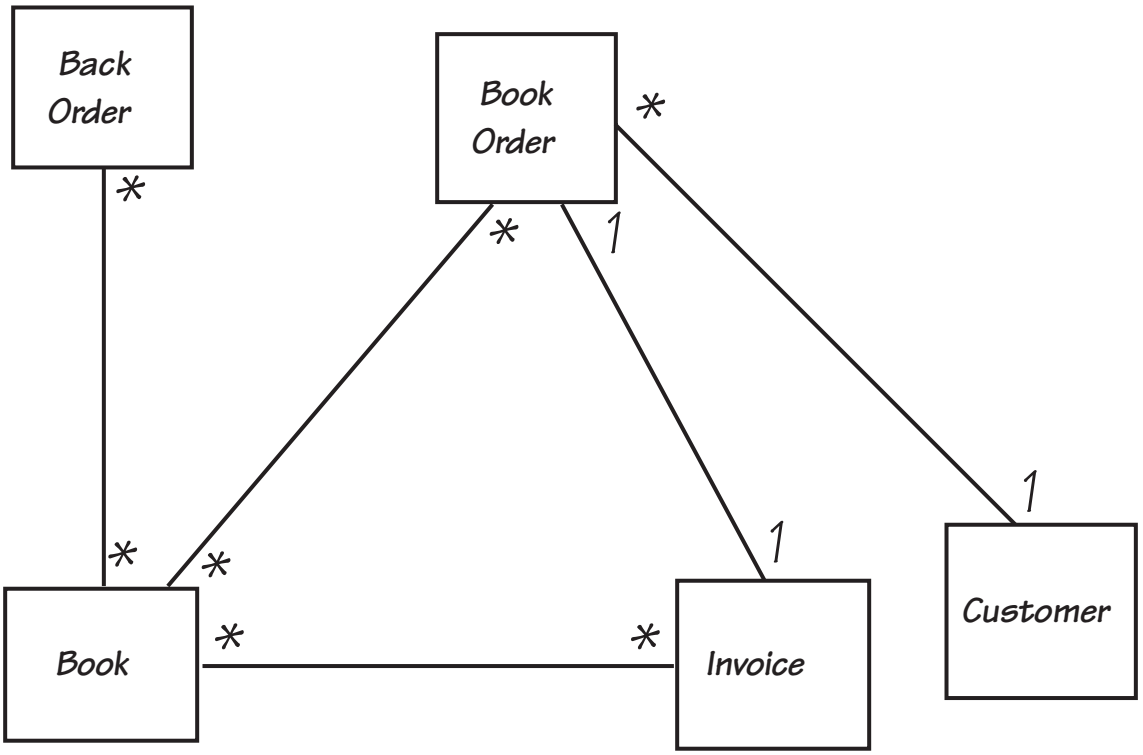

Figure 2: A class model that contains some potentially reusable functional requirements. This model is supported by definitions of each of the classes.

Now suppose there is another project concerned with analysing customer buying patterns and making recommendations about which books we should keep in stock and which suppliers we prefer. Clearly the new project can benefit by reusing some of the requirements that have already been defined by the first project.

If the first project has used a consistent formalism for modelling their data then it makes it easy for you to reuse the knowledge that they have captured. You can look up their definition of *Customer* and discover that it is:

Customer Name
Customer Address
Customer Credit Status
Customer Discount Rate

If you are lucky you discover that each of these elements of data is defined in a data dictionary. The definition of customer for your own project might be different; you might discover you have other elements of data that were not relevant to the first project. Also you might discover that some of the elements are not relevant to your project. But the first project has provided you with a valuable starting point.

You will add new classes and associations that are specific to your project. Given the focus of your project you will probably discover a class called *Publisher* that has an association with the class *Book* and as you see in Figure 3, you will add it to the class

model for your project.

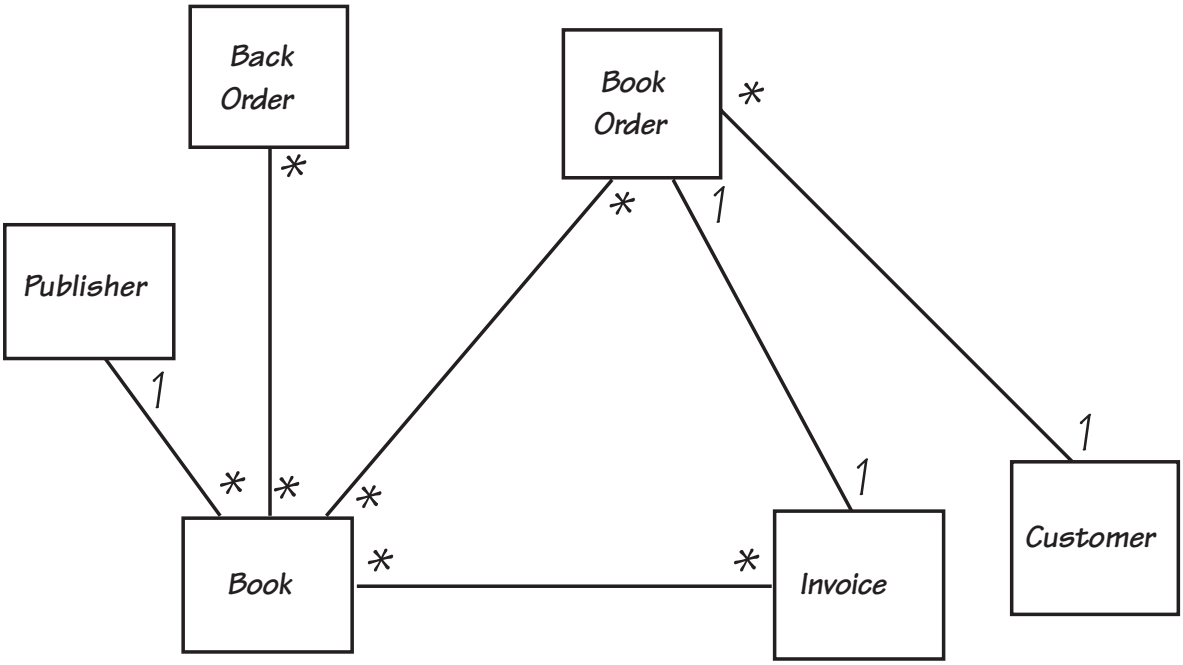

Figure3: The new class, Publisher, has been added to the original reused class model.

How much do you benefit from this type of reuse? Well certainly you would save the effort (all the meetings and reviews) invested by the first project. However you will only save the effort if the quality of the reused model is consistent and does not cause you to have to make interpretations or go and ask the same questions all over again.

Looking at the function space

Models in the function space focus on what the business does with the classes and associations in order to get the desired results. In the book-buying example, the class model tells us that the business needs to remember all the classes and the associations between them. A function or process model provides some information on what the business does with the data in order to achieve some desired result.

Figure 4 is a business process model that shows the details of the business response (or business use case) to the business event: *Customer wants to buy a book*. The graphic model identifies the processes and the interfaces between them.

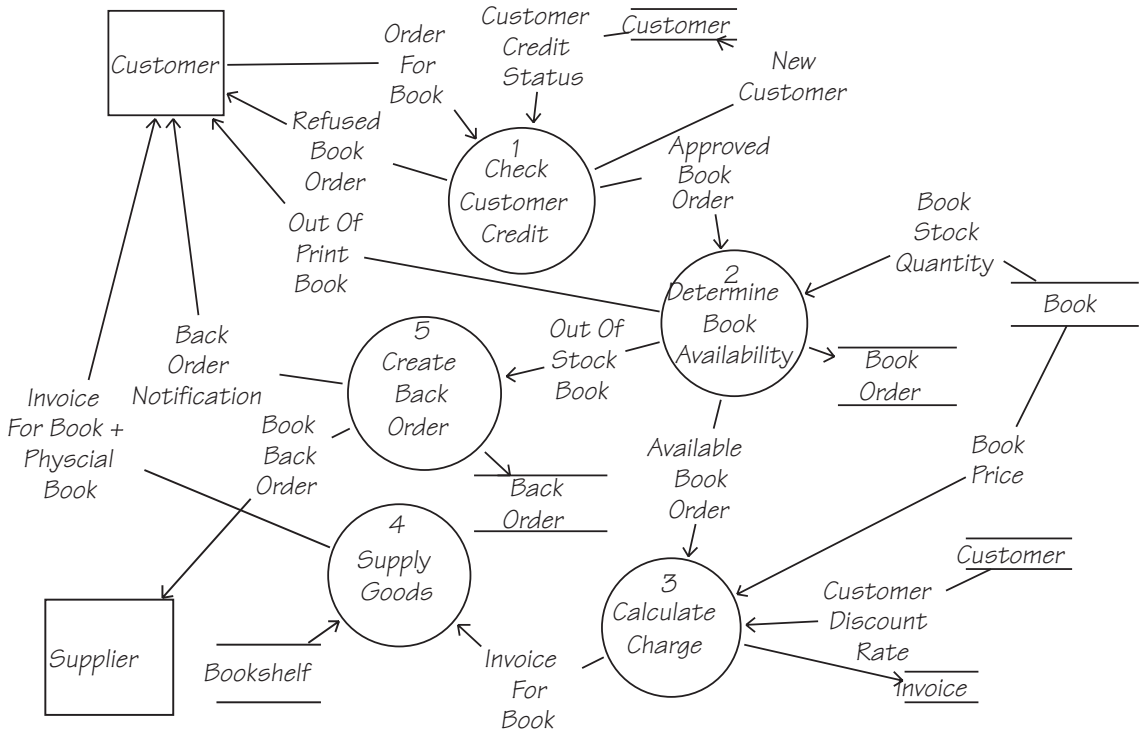

Figure4: This model shows the organisation's response to the business event: Customer wants to buy a book

The data flow notation [DeM2] in this process, introduced into mainstream use in 1978, is familiar to most systems analysts.

Instead of a graphic model, you can define the same business event response (or business use case) using a text scenario model as follows:

Normal Case Scenario for Customer Wants To Buy Book

Customer submits an order for a book together with a credit card number

If the customer's credit status is approved then approve the book order otherwise refuse the book order

Check the book inventory to find out whether the book is in stock.

If the book is in stock

Record the book order

Calculate the charge taking into account the customer discount rate

Record the invoice

Produce an invoice

Send the invoice and the book to the customer

If the book is not in stock

Create a back order

Send a back order to the supplier

Record the back order

Notify the customer that the book has been back ordered

Both the graphic and the text models, supported by data definitions and business rules, are potential sources of reusable functional requirements. In both examples we can see that there is some functionality to create a back order. This functionality is composed of a number of atomic functional requirements:

The product shall create a back order if the book is out of stock
The product shall record the back order
The product shall send a back order to the supplier
The product shall notify the customer if the book has been back ordered

It is reasonable to argue that perhaps it is not necessary to write individual atomic requirements if these requirements can be easily extracted from whatever analysis model you have built. However, sometimes the audience for the functional requirements is not prepared to accept a specification composed of models; hence it is often necessary to extract atomic requirements statements from the models. I will talk more about atomic requirements a little later in this report.

Models in the state space

You can build a state model to identify the states and transitions within the boundaries of any type of system. For example figure 6 is a simple state model for a lamp. The model identifies an on state, an off state and a broken state. The on switch is the trigger that causes the transition from the off state to the on state. If a fault occurs when the lamp is in the on state, then the system makes a transition to the broken state. Whilst this is a very simple system it illustrates the power of a state model in helping to discover missing states or transitions that in turn discover missing requirements.

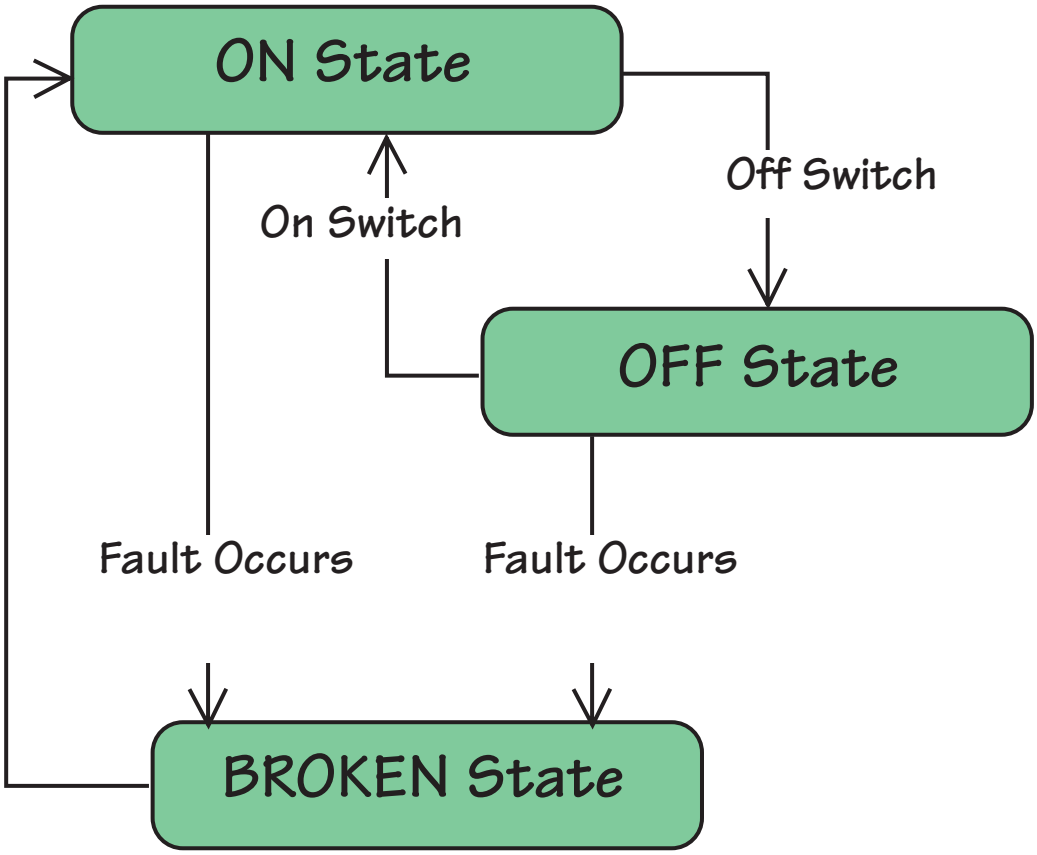

Figure 6. This simple state model illustrates the physical states of a lamp.

State models were originally used to model physical systems and systems in the field of engineering. But better understanding of the importance of state space has led us to use the models to understand the states of any system or class of subject matter.

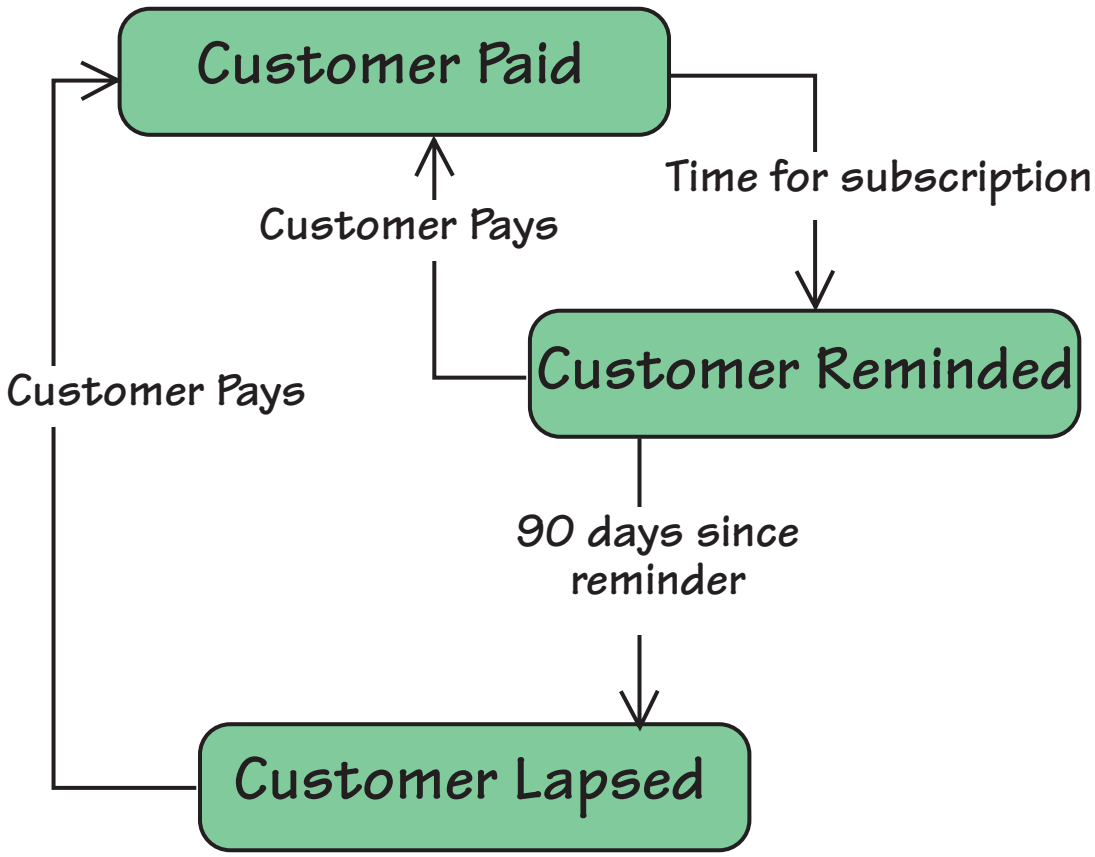

Figure 7: This state model specifies the states of existence of a customer within the boundary of a subscription payment system.

The example in figure 7 is a state model that shows the different states of existence of a class of subject matter called customer. We can see from the names of the states and transitions that this model is concerned with the customer from the point of view of his different states of indebtedness. If your project is concerned with customers and payments, then some of these states along with the supporting work that has been done might be reusable by your project.

Notice that models in the state space have a connection between models in both the function and data space. The data model for the customer payment system would have an entity/class called customer and the function model would contain rules for all the processing that takes place while the customer is in the various states along with the conditions for making the transition from one state to another. Each of the transitions can be viewed as a business event which leads to a business event response/business use case. Understanding of these connections between the data, function and state spaces helps with the search for reusable functional requirements. For example, if you are looking at a model in the state space you can use it to identify business events that will lead you to reusable requirements in the function space. Similarly, the discovery of a class/entity in the data space leads you to exploring the state space for that aspect of the data. So far I have focused on the idea of discovering and reusing functional requirements. There are many other aspects of requirements knowledge that are potentially reusable. The next one I will look at is non-functional requirements.

Reusing Non-Functional Requirements

Non-functional requirements are the properties that a system should have. They are sometimes referred to as qualities. It helps to think of them as the things that a system should be. An adjective is an indicator of a non-functional requirement. When you hear that the system should be fast, usable, attractive, secure, maintainable….. then it's likely that you have a non-functional requirement.

A checklist [Rob1] of non-functional requirements types acts as a driver to discovery. Types on our list are:

NON FUNCTIONAL REQUIREMENTS
10. LOOK AND FEEL REQUIREMENTS
11. USABILITY REQUIREMENTS
12. PERFORMANCE & SAFETY REQUIREMENTS
13. OPERATIONAL AND ENVIRONMENTAL REQUIREMENTS
14. MAINTAINABILITY & PORTABILITY REQUIREMENTS
15. SECURITY REQUIREMENTS
16. CULTURAL AND POLITICAL REQUIREMENTS
17. LEGAL REQUIREMENTS

An effective strategy is to use your knowledge of the functional requirements to trigger the non-functional requirements.

Connecting Functional and Non-Functional Requirements

One of the functional requirements from the earlier example system of a customer buying a book is:

The product shall notify the customer if the book has been back ordered

You can compare the functional requirement with each type on the non-functional requirements checklist. For instance, are there any look and feel requirements related to notifying the customer that the book has been back ordered? Maybe there is a specific form in which we make the communication, are there any special colours or logos that should be used for the communication? Are there any public relations implications related to telling the customer that the book has been back ordered? Answers to these questions all lead us to look and feel requirements.

Similarly from the point of view of usability requirements we can ask questions like how easy must it be for the customer to use the information that we provide about the back order? Will we have to train them? How will we know if we have produced information that is easy to use?

Performance questions focus on how quickly the product should notify the customer if the book has been back ordered? How many books do we expect to have on back order at any time? Safety related questions ask whether there are any safety issues as a result of a book being back ordered. In this example most unlikely, but there are many safety critical systems where safety requirements are of prime importance.

You can use the functional requirements along with each of the types on the checklist to discover appropriate non-functional requirements. In other words, your knowledge of the functional requirements is a trigger to discovering the appropriate non-functional requirements. You can help people to discover and reuse non-functional requirements that already exist.

Where to look?

Each type of non-functional requirement is the subject of some knowledge held by a specialist person or organisation. Some of the specialists have been defining non-functional requirements (although that is not what they have called them) in great detail for a long time. So if we can find the appropriate specialists for different types of non-functional requirement, then there is a good chance that they will be able to provide us with reusable requirements.

Non-Functional Requirements Types	Sources of Specialist Knowledge
Look and Feel Requirements	• Graphic Designers • Marketing Department • Advertising Department
Usability Requirements	• Usability Laboratories • Human Communications Interface Specialists • Usability Specialists • Psychologists • Organisations devoted to disabled groups eg: Institutes for Blind, Deaf, Wheelchair bound • Organisations devoted to specific age groups eg: Senior citizens associations Schools Universities
Performance and Safety Requirements	• Standards Associations for your Industry • Technical Specialists in your organisation • Health and Safety organisations • Ambulance Department • Fire Department • Doctors, Hospitals, Safety Officers
Operational and Environmental Requirements	• Architects • Facilities management specialists • Environmental groups • Recycling specialists
Maintainability and Portability Requirements	• Maintenance specialists for similar systems • Product designers • Technical specialists in portability technology
Security Requirements	• Police • Security Specialist Organisations • Your security department
Cultural and Political	• Your organisation chart

Requirements	• Your company goals • Travel literature • Specialists in other cultures' customs
Legal Requirements	• Your legal department • Legal precedents • Standards you have decided to observe

These specialists are particularly valuable to requirements engineers because their familiarity with their subject matter means that they know how to quantify the non-functional requirements. Someone without specialist knowledge might say there is a look and feel requirement that the message should be easy to read. But a graphic designer can quantify the ease with which different typefaces can be read on a screen and which ones can be read more easily on paper. A security specialist can quantify the security of a system by how long it takes to make a specific attack or how much force would be necessary to carry out a specified amount of damage. This idea for reusing non-functional requirements by locating the appropriate specialist relies on the requirements engineer knowing what questions to ask and being able to help the expert focus on the non-functional requirements that will support the functionality in question.

Scales

As discussed earlier, we have had a great deal of experience in specifying functional requirement. We have developed many models to help us and those models have provided us with ways of making functional requirements measurable. They provide us with scales of measurement. Non-functional requirements are typically more difficult to specify than functional requirements because we do not have readily accessible scales of measurement.

Certainly, some non-functional requirements are more obviously measurable than others. Performance requirements, for example, have an obvious scale of measurement that is some variation of time. We often quantify performance by specifying how long it should take for some specific task to happen. We do not need to search for a scale of measurement because time is a scale that we are used to, we have been using it all our lives. However other non-functional requirements like look and feel or usability or cultural require more effort to discover a suitable scale of measurement. Some examples of potential scales of measurement for non-functional requirements are:

Requirement Type	Suggested Scales of Measurement
Look and Feel	Conformance to graphic standard Conformance to colour Readability measurement
Usability	Amount of time taken to learn to do a specific task Amount of training needed to be able to do a specific task Error rates Time to perform a specific task
Performance	Time for a specific task to give a predefined result Volumes
Operational	Quantification of environmental conditions like temperature, weather, geographical distribution
Maintainability	Amount of time necessary to make defined changes

	Quantification of effort to port to another platform/environment
Security	Quantification of who can access a specific part of the product
Political/Cultural	Who accepts the product. Might be someone in the organisation or might be a defined cultural group.
Legal	Lawyer's opinion/court case

You can use these scales of measurement as a starting point and add to them as you discover scales that are germane to your own environment. Of course a scale of measurement is only one of the attributes of a requirement.

Atomic requirements

Systems builders, customers, users, designers, analysts, marketers….are all increasingly aware of the importance of requirements. As with any subject that is the concern of many different disciplines, there are many different interpretations of the meaning of key terms. In this case the term is "requirement" and it has become an elastic term – a hold all for a wide variety of meanings. It is well nigh impossible to reuse anything unless there is a consistent understanding of what we are talking about. Hence it makes sense to qualify our use of the term. A good place to start is to consider the concept of an atomic requirement, then, we can group atomic requirements into business requirements, technical requirements, system requirements or any other grouping that serves our particular purposes. So let's consider what we mean by an atomic requirement.

One of the attributes of an atomic requirement is a description like: *The product shall create a back order if the book is out of stock*. However the description does not tell the whole story, there are many other attributes necessary in order to specify all the details of the requirement. In Figure 8 you see an example of an atomic requirement. Notice that the requirement has a number of different attributes:

Requirement Number: A unique identifier for this requirement
Requirement Grouping: the parts of the business or the product to which this requirement relates
Requirement Type: Functional, Non-Functional, Constraint or Technical
Description: Makes the requirement readable and accessible
Rationale: Explains why the requirement is important
Source: Where did the requirement come from and who cares about it
Fit Criterion: Quantifies the precise meaning of the requirement so that any solution can be judged to either fit or not fit the requirement.
Customer Satisfaction/Dissatisfaction: the first indication of the requirement's priority.
Dependencies: The other requirements that depend on this one
Conflicts: Other requirements in conflict with this one
Supporting Materials: Other documents/sources that provide background information
History: When was the requirement created/reviewed/implemented/deleted..

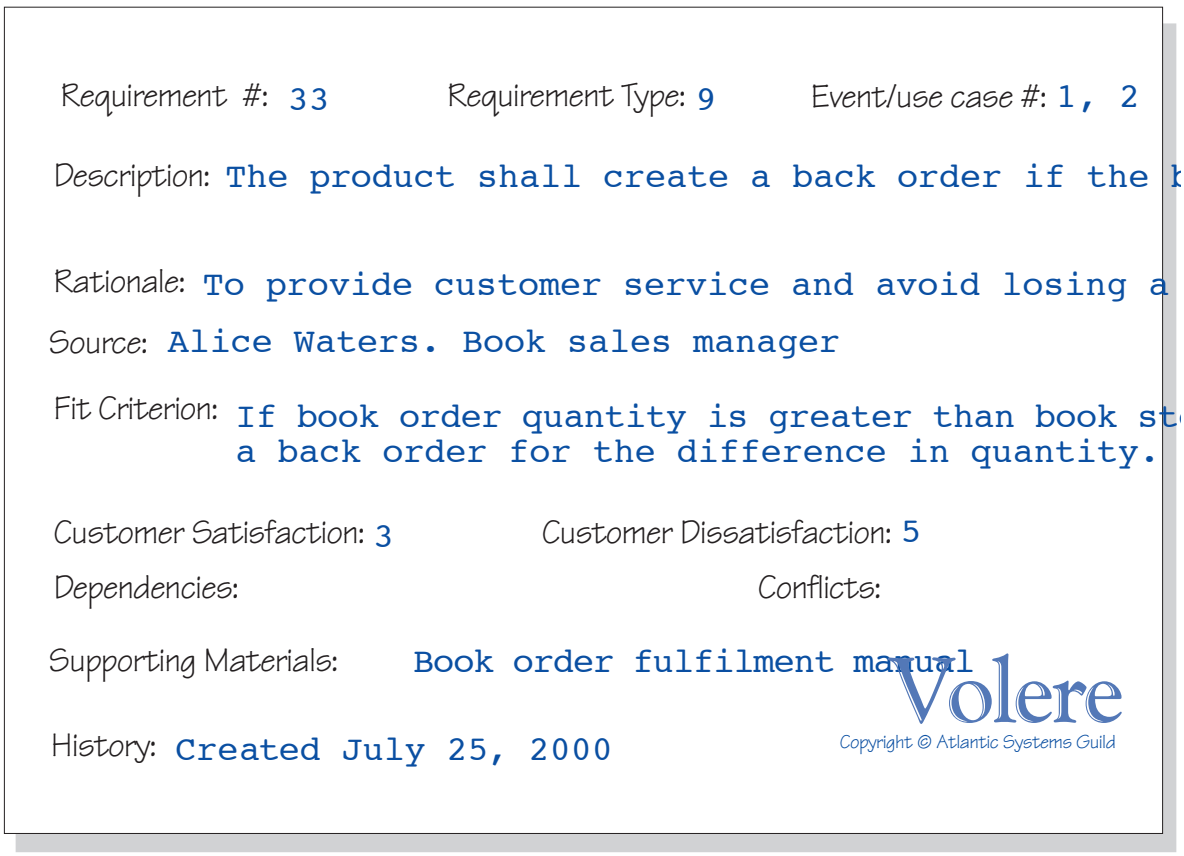

Figure8: This atomic functional requirement is made up of a number of attributes. Each attribute is necessary for a different reason.

Each attribute has a meaning and a purpose. All these attributes together make up the complete atomic requirement. If you have a discipline for consistently expressing your atomic requirements then you increase your potential for reuse.

The first opportunity for reuse is within a project. An atomic requirement often relates to more than one part of the business or more than one part of the product. For example, if you have an atomic requirement that specifies the rule for calculating interest then it is likely that there are a number of parts of the business that apply that atomic requirement. Similarly that atomic requirement might be implemented in several parts of the product that you build. By defining the atomic requirement once and then linking it to the appropriate business and product chunks you are taking advantage of reuse. You have a similar opportunity if you can reuse atomic requirements between projects, I will talk about some ideas for doing this later in this report. Apart from reusing atomic requirements there are other opportunities for reusing groups of requirements by identifying patterns.

Requirements Patterns

In 1977 a group of architects headed by Christopher Alexander, realised the potential of formalising patterns of knowledge and published a book [Ale1] of design patterns. This book, written over a 10-year period, is a distillation of the architects' individual

experiences. Each of the 253 patterns is a design solution related to a common problem in the field of architecture. For example there are patterns to guide you in how to design a pedestrian street, an entrance hall, a small meeting room, a vegetable garden, a cooking layout. The software industry has been paying attention to reusing patterns for a long time, but most of the attention has been focused on the reuse of code. More recently work has been done further up the food chain on the subject of reusable design. In 1995 Eric Gamma and his colleagues were inspired by the work of Alexander and his fellow architects. Gamma and his colleagues realised that, like the architects, they shared a great deal of reusable knowledge about their own field, so they wrote a book [Gam1] of patterns for solutions to common object oriented design problems. The patterns make it possible for object-oriented systems designers to take advantage of the other experts' experience by reusing these patterns as the starting point for their own designs. The work on design patterns generated a significant interest in discovering, communicating and reusing patterns in the field of systems design and to some extent systems analysis. [Rob3]. These experiences have led to an interest in the potential of reusing requirements-related knowledge.

As the architects illustrated, the work of defining, recognising, and reusing patterns takes time. First there is all the work of defining the best language for the subject matter. Next is the task of gathering potentially useful patterns. And then there must be a culture that makes reuse desirable and effective. Systems requirements analysis has made some progress in each of these areas. Today, instead of writing a free form, text-based specification, (or spending time inventing a new way of writing requirements) competent analysts use some kind of recognised modelling technique to provide a common communication language. The functional modelling languages that I referred to earlier in this report, provide a consistent way of defining groups (either data, function or state related) of atomic requirements. Then each group of requirements can be formally examined to see whether it contains any reusable patterns.

In 1993, starting with a project in Denmark, I formed an alliance with Kenneth Strunch [Rob3]. The work we have done indicates that the skill most necessary for identifying reusable requirements patterns is the ability to abstract. As well as a communication language and abstraction skills, requirements reuse also needs a supportive organisational infrastructure. Historically, systems development culture focuses on implementation and measures success based on how many lines of code are produced or how quickly we release a product. A culture that supports requirements reuse is one that focuses on abstraction skills and measures success based on how many requirements components are reused. So there are some cultural changes necessary in order to take advantage of requirements reuse. The projects that we have worked with have made some progress in defining, recognising and reusing the products of requirements analysis. The next sections describe experiences from these projects.

First Abstraction - Patterns Within A Project

It's easy to fall into the trap of repeating work because, on the surface, most tasks look different. We worked with a Danish insurance company that has a large and busy systems development department. The work of the department ranges from making changes to development of new systems for new areas of business. Some of the older systems are being redeveloped to take advantage of new technology. The company suspected that there is a great deal of overlapping business knowledge between different insurance systems. If

they could identify and define potentially overlapping knowledge then they could save time by reusing that knowledge when building new systems. The company decided to concentrate on identifying potentially reusable business requirements knowledge.

The first project to use this thinking was building a system for insuring cars, houses and contents of houses. This was known as the private insurance system. The analysts and users, a team of 8 people, started by building separate requirements models for cars, houses and contents. At the beginning, the products all seemed to be quite different and nobody had the knowledge to see the similarities. After a while, however, the team members brought the three models together and reviewed them. They consciously looked for similarities between them. They had conversations like: "insuring a house against flooding is much the same as insuring a car against being stolen. It's true that the prices, dates and conditions are different, but the concepts are the same".

Comparisons of the models caused many heated discussions about the details. But one thing was agreed. In spite of all the differences between the different kinds of insurance, there were more similarities than anyone had expected. And if those similarities could be extracted there was potential for creating some generic type of insurance model, which could cover all types of insurance - at least within their own context. The first step towards making the abstraction was realising that it was possible and useful.

The next step in trapping and defining the emerging pattern was to agree on terminology and representation. If insuring cars, houses and contents is similar then what shall we call the thing we're insuring? A number of new terms had to be invented and agreed before the effort could go further. Reaching agreement on which terms to use was surprisingly difficult. However, spending the time to have all the discussions proved invaluable in the long term because once a term was agreed everyone started using it and everyone knew its meaning. And we defined it in the project dictionary to make sure that we kept track of our decisions. Models, like the ones in figure 9, of specific types of insurance gave us a starting point for our discussions. In the insurance business different types of things like buildings, cars, cranes, antennas, contracts etc. can be insured against different kinds of damages like theft, fire, floods and so on:

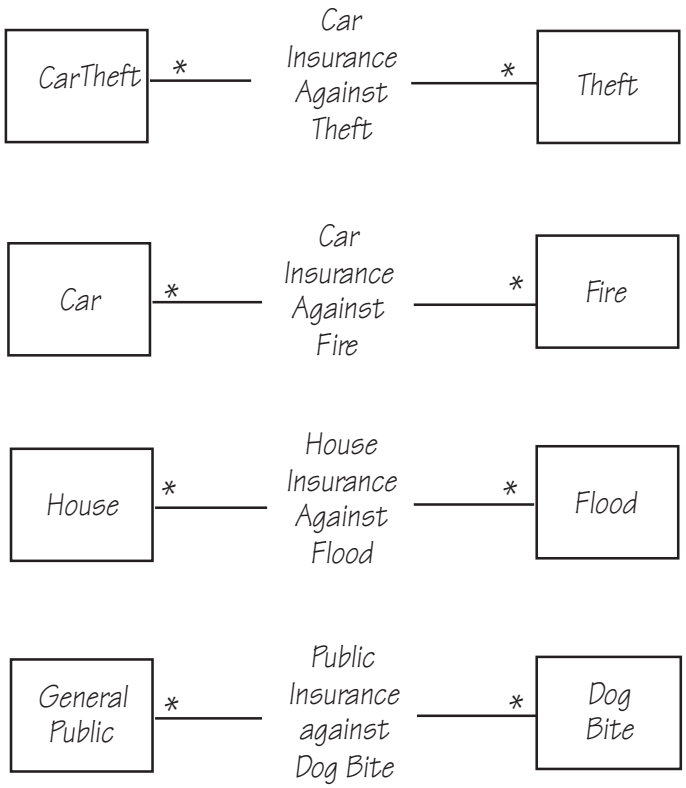

Figure 9: Each model specifically defines the rules for insuring a particular type of thing with a particular type of coverage.

After building and reviewing models of a few of these valid combinations, we started to see how the concept could be generalised. If the types of things that are insured are called ASSET TYPES, the types of damages are called COVERAGE TYPES and the valid combinations are called ASSET COVERAGES then a pattern like this emerges:

Figure 10: This generic model covers all valid business combinations of Asset Types and related Coverage Types. Every Asset can have an Asset Coverage connection with many types of Coverage. And every type of Coverage could be connected via an Asset Coverage relationship to many types of Assets.

The discovery of the generic ASSET, COVERAGE and ASSET COVERAGE triggered other insights. For instance whenever the insurance specialists design a new product for sale, they ask the systems developers to add the product to the existing computer systems. This involves a lot of work as each new product is thought of as a new combination of new components. So time is spent in analysing all the details of the new product, and adding them to the system. Now, instead of seeing a new product as a brand new set of components, the requirements analysts could see that it is a combination of asset coverages.

An asset coverage, remember, had already been defined as the rules for giving a particular type of asset a particular type of coverage. Every time the insurance specialists design a new product, they are defining the selling conditions for a combination of asset coverages. The experts agreed that the model in figure 11 defines the pattern that they follow.

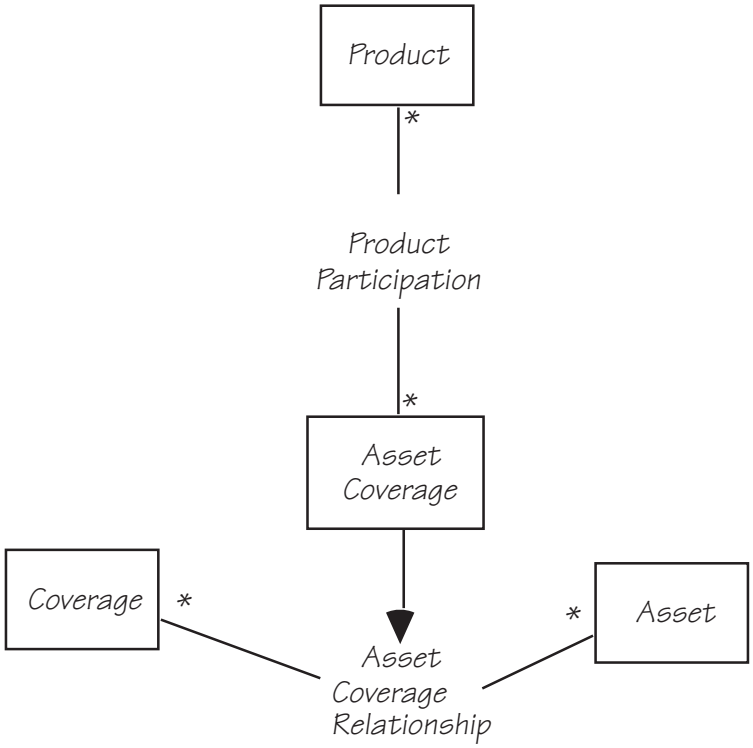

Figure 11: A generic model for the concept of a Product. A Product is a collection of Asset Coverages which are sold as a package.

The definition of these requirements patterns meant that they could be reused by both the insurance experts and the system developers. The system developers use the patterns to build a system that reflects the requirements patterns. So the computer system contains identifiable ASSETS, COVERAGES, ASSET COVERAGES and PRODUCTS. The system is built so that the users can add new instances of these business components. The insurance experts can define new products using any combination of asset coverages. The resulting, amazingly profitable, business value is that a new idea for selling insurance can be implemented by the insurers immediately, without necessitating any changes to programs.

The requirements patterns defined in the private insurance project helped that project group to capture knowledge for a business-oriented system. If requirements patterns could be reused within one insurance project, we wondered whether the same patterns could be reused by other projects within the same company.

Second Abstraction - Patterns Within An Organisation

The life insurance requirements analysts were about to start a new project and they agreed to see whether they could use any of the requirements patterns that the private insurance project had discovered.

To start with we came up against some barriers because life insurance specialists, naturally enough, view life insurance as something totally different from private insurance. Someone who specialises in a business area sees it as quite different from any other business area in the same organisation. And so it is from the point of view of the details. However from a more abstract point of view there are usually similarities. Our challenge was to help the life insurance specialists recognise similarities between their world and the requirements patterns derived from the first project.

We started by defining the context of the business study for the life insurance project. Then, working with the life insurance team, we compared the context data with each of the private insurance requirements patterns looking for similarities.

The team worked together in room lined with whiteboards. After 3 days and copious quantities of coffee and Danish pastries we had a great success. By renaming some of the components on the private insurance models it was possible for the life insurance project to reuse large parts of the models. For instance the life insurance analysts realised that an ASSET in the private insurance system is the equivalent to a PERSON in the life insurance system. And ASSET-COVERAGE in the public system is the equivalent of LIFE-COVERAGE in the world of life insurance. Once these patterns were recognised the detailed definition work done by the private insurance project could be reused, with some modifications, by the life project. And a great deal of time was saved. The reuse of the requirements patterns meant that the life insurance analysts only spent three days to produce the same amount of work that had taken the private insurance people six months:

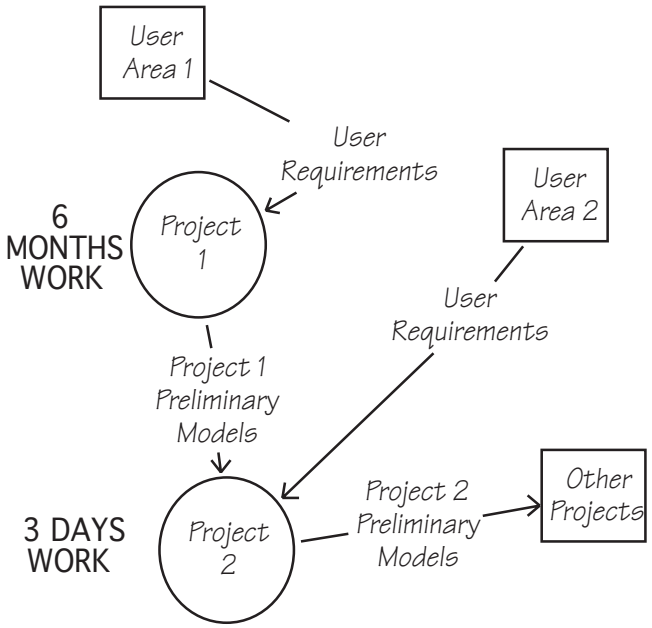

Figure 12: Project 1 did the pioneering work by producing understandable requirements models. Project 2 reused these models; hence their preliminary requirements analysis was completed much more quickly.

It should be noted that the productivity difference between the analysis of the life insurance and that of the private insurance had nothing to do with differences in individual skills. The private insurance analysts *discovered* the patterns while the life insurance analysts *used* the patterns. Discovering is a lot more difficult than using.

The timesavings are very impressive but they are only a small part of the potential savings. The longer term savings come as developers of other systems start to reuse the same requirements patterns. The computer systems will start looking more similar which in turn will have a great impact on the maintenance. It will be much easier for people from different groups to understand what the other groups are doing and it will be easier to reallocate resources. The patterns create a common way for an organisation to view the world.

The first two abstractions give examples of the pattern recognition and reuse taking place within an organisation. The next two abstractions are a natural extension. The use of a common language means that we could extend the idea of reuse so that organisations in the same field of knowledge can share patterns. The next level of abstraction ignores subject-matter boundaries and identifies generic, subject-matter independent, reusable patterns.

Third Abstraction - Patterns Within A Field of Knowledge

Right now, an insurance company in Indonesia is developing an insurance system. Another company in Australia is writing an insurance system. In Buenos Aires they are working on insurance and a firm in London is developing yet another system for insuring.

The curious thing is that each of these insurance companies, and many others all over the world, are going through the same time-consuming and money-eating procedure of specifying each system from scratch. This is especially wasteful when you consider that large portions of that system have already been specified many times before. One reason that the Indonesians, Australians, Argentineans and English cannot take advantage of previous analysis work is lack of a common specification language.

Another more serious reason is that the builders of one insurance system consider their system *totally different* from that of other organisations who are building insurance systems. This is a worldwide problem within the systems development profession. Our, "not invented here", culture does not encourage the identification and reuse of requirements patterns within a field of knowledge.

Suppose that you have already built a computer system for an insurance company. At this stage you know all the rules contained in your system for creating new policies, reviewing policies, producing reminder letters, reports and so on. You also know about the design of your database, your programs, your user interface and the telecommunications network. The system that you have built is specific to the requirements of a particular organisation.

Now suppose that you have the task of building another insurance system for a different company. This company uses different hardware and software and the way they do business is different. But, not surprisingly, some of the insurance related knowledge that you learnt when building the first system is also relevant to the second company's requirements. This thinking is very similar to the thinking used by the Danish life insurance

analysts when they identified similarities between their project and the private insurance project. However there is one big difference. The abstraction is at a higher level and consequently more difficult for company specialists to see. In addition to ignoring boundaries between projects, we are also ignoring boundaries between organisations.

By abstracting one system's business requirements, ignoring details that are applicable only to the way the system happens to be implemented, and deleting anything that is specific to the particular type of insurance or type of company, you can reveal a pattern which is general enough to be the starting point for all types of insurance. For example, all insurance policies have to be renewed. Although there some details of the renewal that are applicable only to specific types of insurance or specific companies, there are also parts of the renewal process which would apply regardless of what sort of policy is being renewed. Figure 12 illustrates the requirements pattern for a generic policy renewal. These generic requirements, once defined, could be reused by any company wishing to build an insurance system. However, in order to reuse the requirements, two things must be done. First comes the work of discovering and defining the patterns. Then companies have to agree that it is to their advantage to share their generic requirements patterns with other organisations.

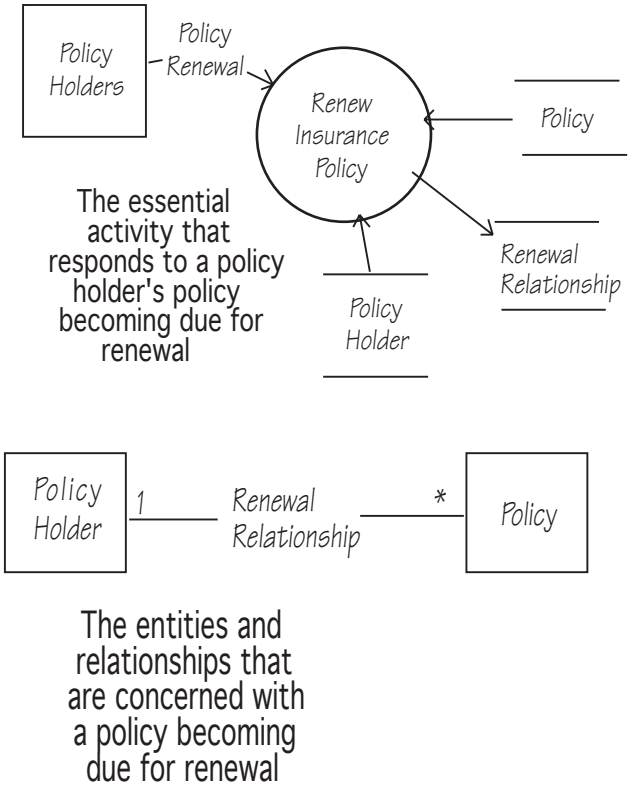

Figure13: There are some characteristics of a policy renewal that exist independent of the type of insurance being renewed. We could use this pattern as the starting point for a health, home, car, bicycle or any other type of insurance policy renewal.

Some organisations feel that they are sacrificing a competitive advantage if they share analysis knowledge with other organisations in the same field. However, other fields have proved that the real competitive advantage comes not from the requirements knowledge but from the particular way that the requirement is implemented. For instance, any musician can go to a music publisher and buy a copy of the music for Elgar's cello concerto, so the knowledge of the notes to be played is available to anyone who wants it. However it is the way that the concerto is played which makes a listener prefer one musician's performance to another. Lots of computer companies use the Intel Pentium 4 chip in their products. Yet two different products can look and behave very differently. The requirements for a spreadsheet are implemented by many builders of software products. But it's the way that those requirements are implemented and our perception of the company that we are dealing with that makes us choose one above the other. Providing the basic business problem is being solved, most of our preferences stem from the way that we are treated by an organisation. Once the subject matter is formally defined an organisation has more time to concentrate on human skills, the thing that really creates the difference.

Suppose our insurance company had access to a knowledgebase containing insurance requirements patterns. The patterns, a formal definition of the subject matter, could be the input to a project. The 6 months taken to analyse the requirements for the new system could be spent on working out competitive ways of implementing them. And they would be implemented more quickly.

The less we consider physical boundaries: inter project, intra project, inter organisation, the more concentrated requirements patterns can become. In our first abstraction we thought past the differences between components of a project. Our second led us to a concentrated view of the knowledge within an organisation. An abstraction which focuses on subject matter regardless of inter organisational boundaries has provided us with a third level of abstraction. It is useful to go further and abstract past differences in subject matter.

Fourth Abstraction - Patterns Between Fields of Knowledge

In his book, *Gödel, Escher, Bach,* Douglas Hofstadter [Hof1] illustrates how subjects that seem dissimilar on the surface can be shown to have many identical underlying patterns. Using this thinking, any specific subject can be abstracted to reveal potential similarities or patterns.

At first glance it is difficult to see similarities between the fields of health insurance and oil exploration. However, if one abstracts away from specific subject matter like oil wells and broken legs there are many things that the two fields have in common. For instance oil well leases have to be renewed and so do health insurance policies. So, for that matter do library books, fishing licenses, airline routing agreements, satellite broadcasting agreements and subscriptions to the fruit of the month club. If you compare the renewal rules you will find differences that are specific to the field in question. But you will also discover a pattern which is the same in each case.

You can tell that Figure 14 specifies the requirements for the renewal of a library book . The names of the processes and data make the analysis specific to a given type of subject matter. Similarly, Figure 15 is concerned with the requirements for renewing satellite broadcasting licences. Even though the subject matter of these two systems is very different, the two models expose some startling similarities. When someone wants to extend a library loan, the request is reviewed against the loan history. Similarly, the other

model shows us that a broadcasting licence request is reviewed against broadcasting history. Sounds as if there are some rules about renewals which compare a request with past performance and either refuse the renewal or conditionally accept it. This sort of thinking results in Figure 16, the requirements pattern for renewing anything.

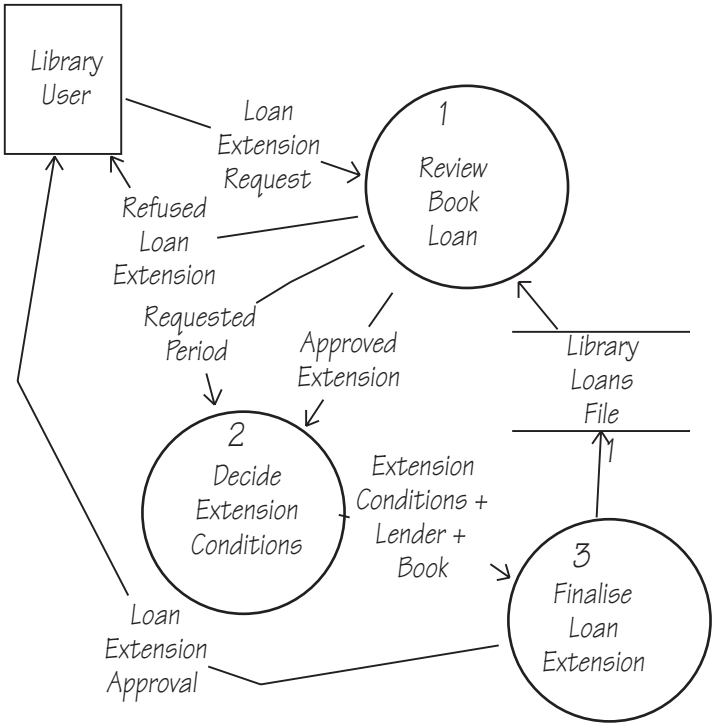

Figure14: The requirements for renewing a library loan

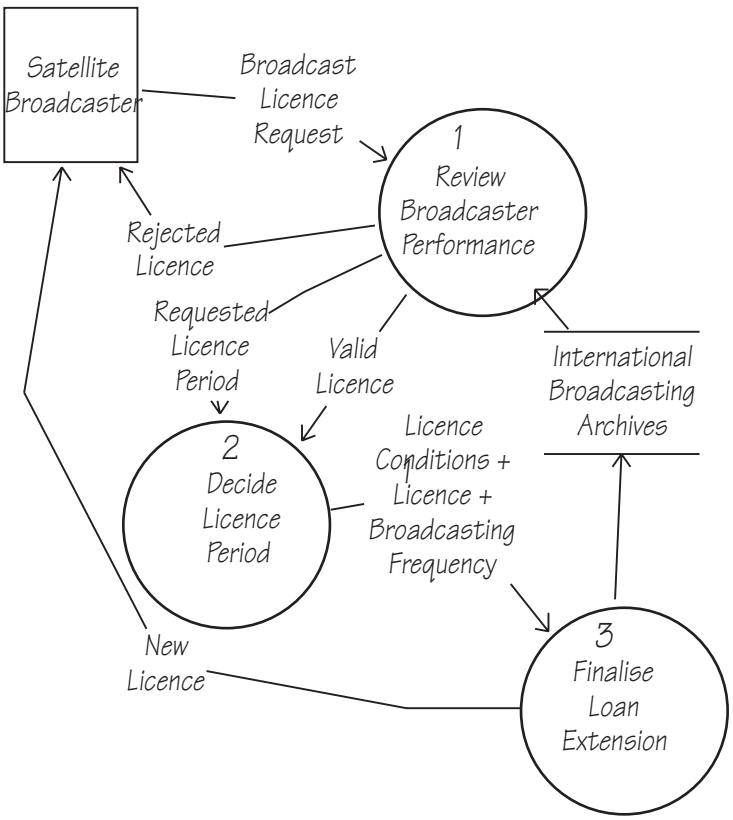

Figure 15: Broadcasting Licence renewal is interested in different subject matter

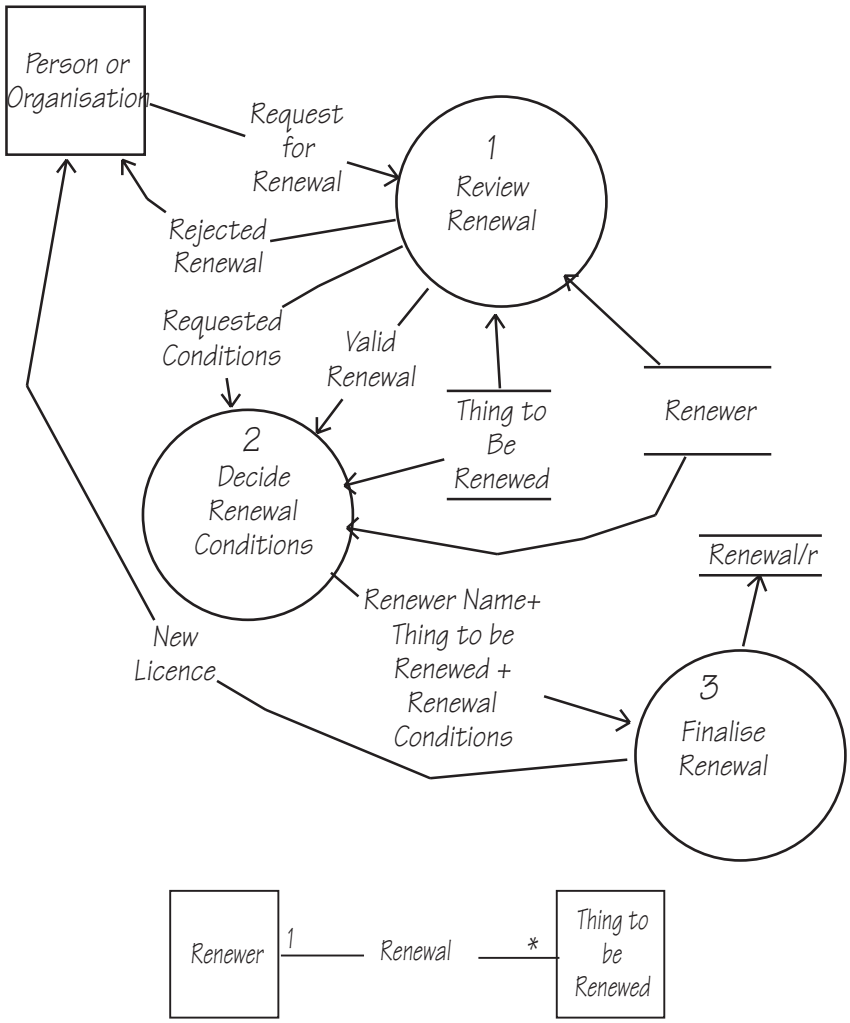

Figure16: This abstraction from the two subject matter specific models ignores subject-matter boundaries and focuses on the requirements for renewing anything.

You could use this generic requirements pattern as the starting point for building a system to renew computer rentals, home loans, airline landing rights or anything else that needs renewing. The requirements analyst would then add the policy and data that are specific to that field of knowledge in that organisation within that project.

Classifying Requirements Patterns

I have examined four levels of analysis abstraction and each abstraction focuses on knowledge patterns which are particular to that level. It helps to think about the different levels of abstraction by expressing them as a hierarchy of classes where lower levels inherit the higher-level policy. Our requirements classification hierarchy is concerned with classes of requirements patterns at various levels of abstraction. If we reuse these requirements patterns we avoid repeating analysis work.

We have arranged our requirements hierarchy into four levels. Our fourth level of abstraction is at the highest level of the hierarchy. This level ignores project, organisational, and subject-matter boundaries. It focuses on knowledge particular to a

domain. The third abstraction level inherits knowledge from the domain level and adds knowledge particular to a specific field. Similarly the second abstraction inherits all the knowledge from the domain and field level and adds knowledge specific to an organisation. Finally the first abstraction, being at the bottom of the inheritance path, inherits knowledge from all the other levels and adds knowledge specific to a particular project.

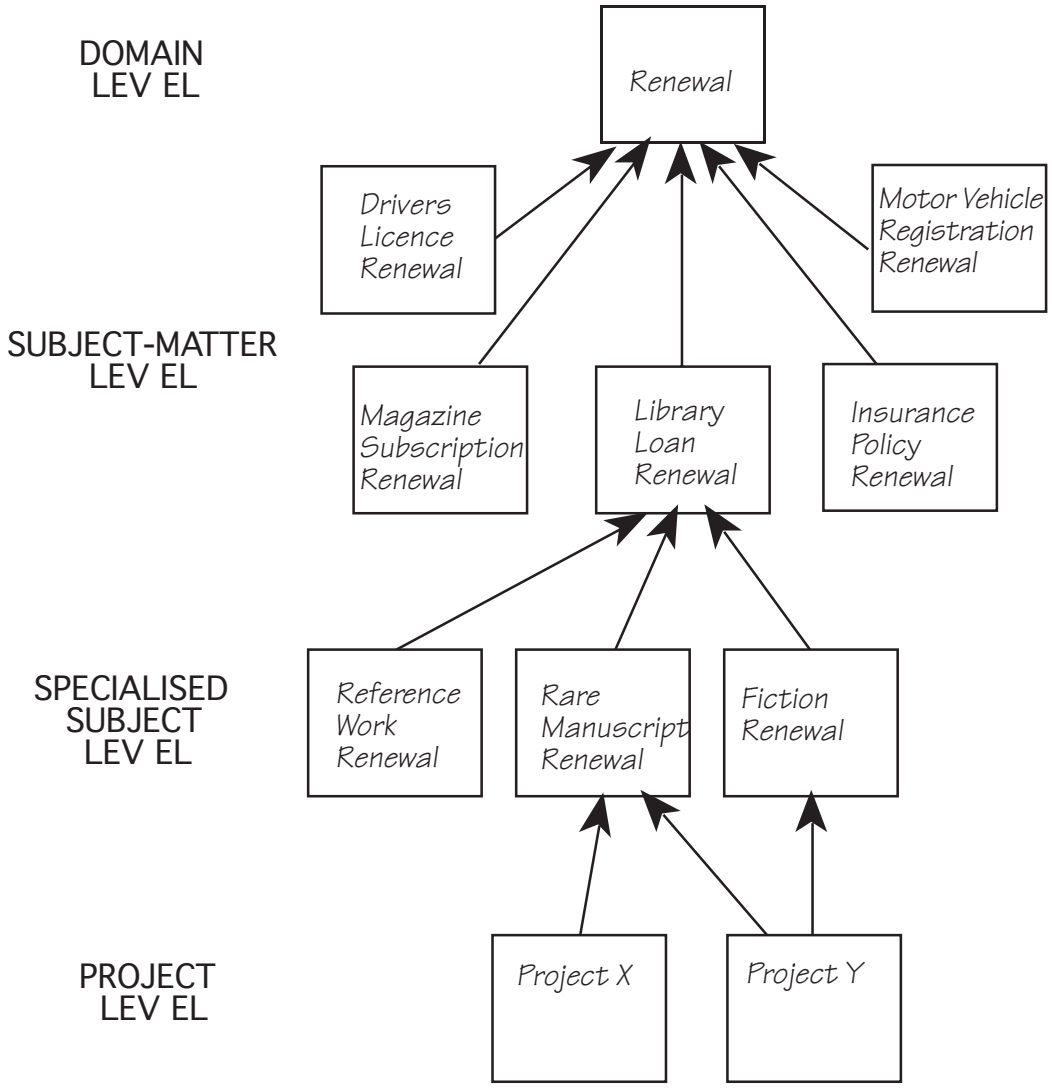

Figure 17: Projects X and Y have inherited the same policy for rare manuscript loan renewal. Tracing through the inheritance tree the projects have also inherited policy that is common to library loan renewals and renewals. Project Y has also inherited policy that is specific to fiction renewals.

An example of requirements inheritance patterns is illustrated in Figure 17. The requirements classification hierarchy is an inventory of patterns at 4 levels of abstraction. The domain level of abstraction captures knowledge at a level that is independent of projects, or specific subject matter. At the subject-matter level of abstraction the patterns contain details specific to a particular field of knowledge. The next level of abstraction, the

specialised-subject level, includes details specific to a detailed view of a subject. And at the project level of abstraction each of the patterns contains details specific to a given project.

Each level of the classification hierarchy is linked to the levels above and below. In our example the domain level is RENEWAL and it contains all the knowledge that is common to any type of renewal. In the next level down, the subject matter level, the LIBRARY LOAN RENEWAL, INSURANCE POLICY RENEWAL and MAGAZINE SUBSCRIPTION RENEWAL inherit all the domain knowledge about renewals. Then each subject-matter area, adds knowledge specific to itself. For example library renewals, adds knowledge that is specific to renewing in a library system. Going further down the tree to the specialised-subject level, suppose we are looking at a system that deals with RARE MANUSCRIPT RENEWALS. Then the specialised-subject level inherits all the knowledge about the library renewals and adds its own knowledge specific to rare manuscripts. At the bottom of the tree, PROJECT X inherits all the knowledge on the path we have followed and adds any extras that are specific to the way that this project deals with rare manuscript renewal.

Requirements inheritance patterns provide a way of categorising requirements patterns at different levels of abstraction. Requirements patterns expressed in this way provide an accessible path into a requirements pattern knowledgebase. If we are going to do a project for renewing rare manuscripts then we will reuse the requirements at the specialised subject level. However, if we are planning to build a library loan renewal system for compact disks and videotape then we will reuse the requirements at the subject-matter level. Of course each new project potentially discovers new requirements patterns which can then be added to the hierarchy. By formally expressing analysis knowledge in this way, we would be easily able to reuse well tested knowledge collected from many different fields.

Identifying requirements patterns

Given that there are all sorts of advantages in pursuing the idea of requirements patterns, the next question is "how do we quantify them?" Here are the measurables that we use to identify requirements patterns:

- Entities or classes within the context of the project
- Relationships or associations between the entities
- Groups of entities and relationships
- Business events or use cases to which the project must respond
- Behaviour patterns between processors

Identification of the entities and relationships within a context of study captures the overall flavour of the business policy. As in the case of the life insurance project, a data model from one project in an organisation can provide a starting pattern for another project. Sometimes two projects will be interested in the same data for completely different reasons. In this case only the high level abstraction of the entities will be reusable. The more the contexts of two projects overlap, the more the detailed definition of data content and relationships will be reusable.

As pointed out by McMenamin and Palmer [McM1] identification of a system's business events and event responses (also known as use cases) captures knowledge about the processes, the processing rules and dependencies between the processes. We use existing context diagrams and event lists as starting patterns for new projects.

Definition of a system's behaviour pattern focuses on environment rather than subject matter. A well-designed behaviour pattern between say a user and an interactive computer can be reused in a similar environment regardless of the subject matter of the system. State models and prototypes are the vehicles that we use to capture and communicate behaviour patterns.

We work with an up-to-date version of requirement analysis modelling that links process, information, event response and behavioural modelling. The technique combines the ideas of many systems engineering researchers and practitioners including Chen, Cockburn, DeMarco, McMenamin and Palmer, Harel, Jacobson, and Robertson (See bibliography for references).

Informal reuse- start with a stocktake

During his talk at the Cutter Symposium in Boston, May 2002, Alistair Cockburn remarked, "human beings are good at looking around". This looking around is what you do when you are trying to find requirements that you can profitably reuse. You spend some time looking around at what has already been discovered before you do any detailed requirements work.

At the beginning of a project try doing a stock take with the aim of discovering what requirements knowledge already exists in a reusable form. We use the table of contents of our Volere requirements template [Rob1] as a requirements knowledge checklist.
We look at each one of the knowledge types on the list and ask: do we have any relevant reusable knowledge of this type? Is it in a form that makes it practical to reuse? Here's a summary of the sorts of questions that we ask and the things that we look for.

PROJECT DRIVERS

1. THE PURPOSE OF THE PRODUCT

Are there any other projects that have the same or similar purpose?
Can we learn from or build on the purpose statements from other projects?

2. CLIENT, CUSTOMER AND OTHER STAKEHOLDERS

Has any other project done a stakeholder analysis?
Can we reuse the roles, responsibilities and names from other projects?

3. USERS OF THE PRODUCT

Are there any reusable user profiles from other projects?

PROJECT CONSTRAINTS

4. MANDATED CONSTRAINTS

Are there any constraints lists that we can use as a starting point?

5. NAMING CONVENTIONS AND DEFINITIONS

Can we make use of the data definitions done by other projects?
Are there any data/class models that we can use as a starting point?

6. RELEVANT FACTS AND ASSUMPTIONS

What were the assumptions made by other projects?
What assumptions did they miss?

FUNCTIONAL REQUIREMENTS

7. THE SCOPE OF THE WORK

Are there any work context models that have an overlap with our project?
Are there any business event response models that we can reuse?
Are there any business scenarios that we can reuse?

8. THE SCOPE OF THE PRODUCT

Are there any product context models for products that interface with ours?
Are there any product use case scenarios?
Are there any systems architecture models?

9. Functional and Data Requirements

Are there any reusable atomic functional requirements?

Non Functional Requirements

For each of the non-functional requirements types, are there any reusable atomic non-functional requirements?

10. Look and Feel Requirements
11. Usability Requirements
12. Performance & Safety Requirements
13. Operability Requirements
14. Maintainability & Portability Requirements
15. Security Requirements
16. Cultural and Political Requirements
17. Legal Requirements

Project Issues

18. Open Issues

Can I learn from other projects' open issues list

19. Off-the-shelf Solutions

Have any other projects used the same off the shelf software that we will be using?

20. New Problems

Have any connected projects experienced unexpected new problems?

21. Tasks

Are there some tasks that other projects missed?

22. Cutover

Does anyone else have a cutover plan we could use as a starting point?

23. Risks

Do we have a most likely/common risks checklist that we can use to get started?

24. Costs

Are there any costing metrics that we can reuse?

25. User Documentation

Can we reuse anything from other projects' user documentation?

26. Waiting Room

What can we learn from other projects' requirements prioritisation schemes?

27. Ideas for Solutions

Are there any design ideas we can reuse/learn from?

Formal reuse needs more organisation

Visualise yourself at the start of a project. Before you do any detailed requirements analysis you go to the requirements pattern book. Laid out before you is all the analysis that has already been done on projects similar to yours. You browse through the book looking at patterns for entities, relationships, events, contexts and behaviour patterns. You choose a number of patterns to use in your project. Some of the patterns you use in their current form, some you change to suit your special requirements.

In order to make the requirements pattern book a reality several things must happen. The patterns must be recorded using some common language that spans the culture of all the pattern users. The patterns must be organised using some readily accessible medium.

The pattern book must be kept up to date. To do this on any useful scale, you need a reuse lifecycle.

Your Reuse Lifecycle

A successful reuse lifecycle integrates essential reuse processes with your design for implementing those processes. The first requirement is to understand the essential processes for creating, using, maintaining and managing reusable assets. The second is to design an effective way of carrying out those processes. If there were no differences between organisations then the same implementation of the reuse lifecycle would suit everyone. However in reality, reuse lifecycles vary tremendously because of differences in many ethnographic factors like: size, skills, goals, geographical location, customs.

An effective reuse lifecycle is one that is based on the essential processes for creating, maintaining, using and managing relevant, understandable and accessible components. The first step is to understand the concepts represented by these essential processes. The next step is to design a reuse development lifecycle that implements these essential processes so that they work effectively in your implementation environment. The people, skills, infrastructure, hardware, software, machines and attitudes within your organisation are all part of your implementation environment. Differences will exist between the implementation of your development lifecycle and that of other organisations because you need a lifecycle which is environment specific. But, to successfully reuse requirements, no matter how you implement the development lifecycle it must contain the essential reuse processes in some form of implementation.

Essential Reuse Processes

We have had a look at some examples of requirements reuse. Now let's stand back and consider the essential thinking that is common to any requirements reuse effort. The context diagram in Figure 18 summarises the reuse lifecycle in its simplest form.

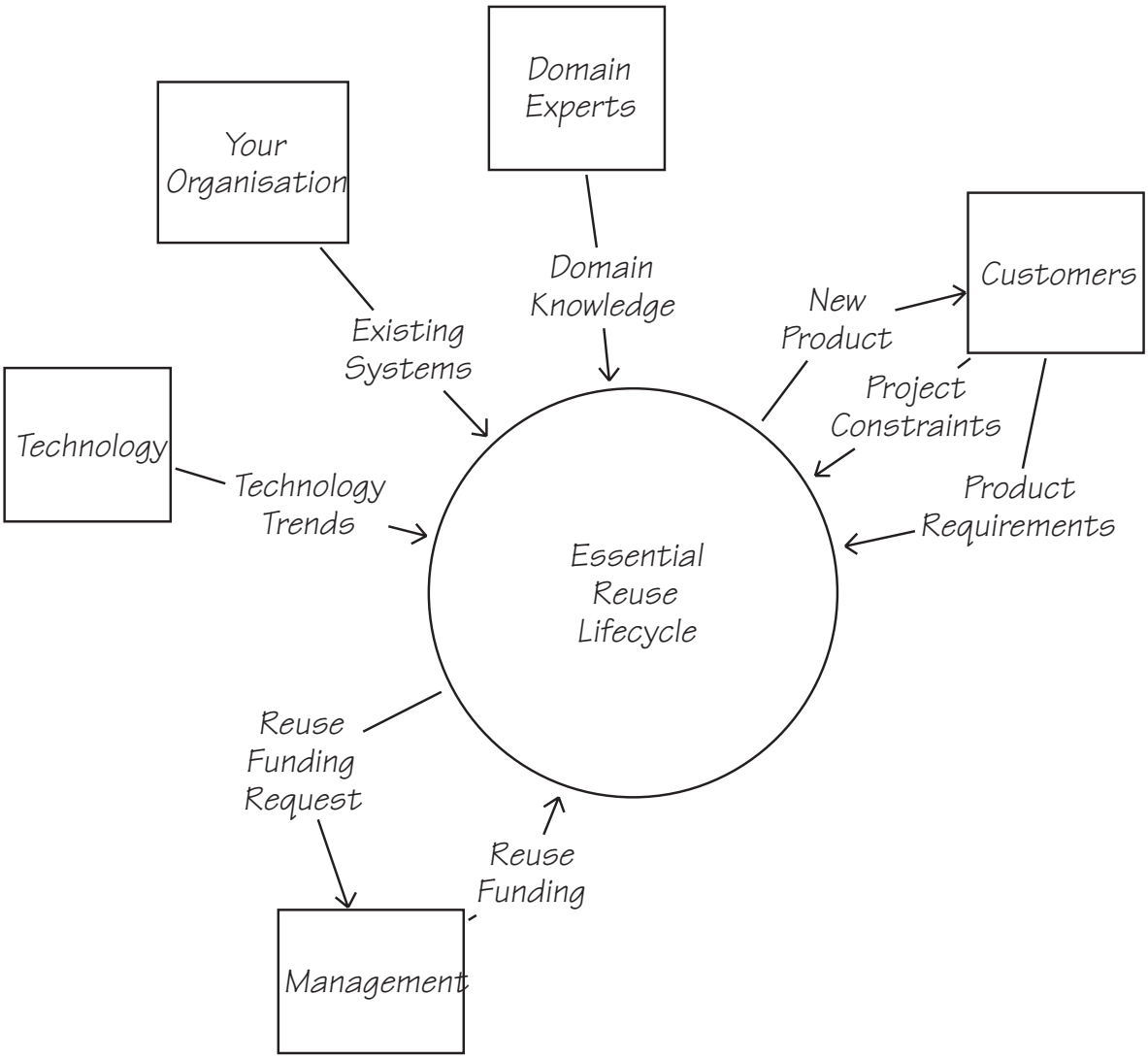

Figure 18: This context diagram summarises the reuse lifecycle by specifying the interfaces with other systems.

Your aim is to produce *New Products* that fulfil the *Product Requirements* within *Project Constraints* of time, cost and quality. Everyone always wants the highest possible quality but this increases the time to produce and the cost of the product. The promise of reuse is to decrease the time and cost at the same time as maintaining high quality. *Domain Knowledge, Existing Systems, Reusable Products* and *Technology Trends*, provide sources for building a library of reusable assets. Whenever you use an asset from your library instead of building from scratch, then you make a saving.

Looking at a more detailed view of the reuse lifecycle, we see that the four essential reuse processes are: *Creation, Support, Use* and *Management* of reusable components. Figure 19 illustrates how these essential processes are related to each other by defining the interfaces between the pieces.

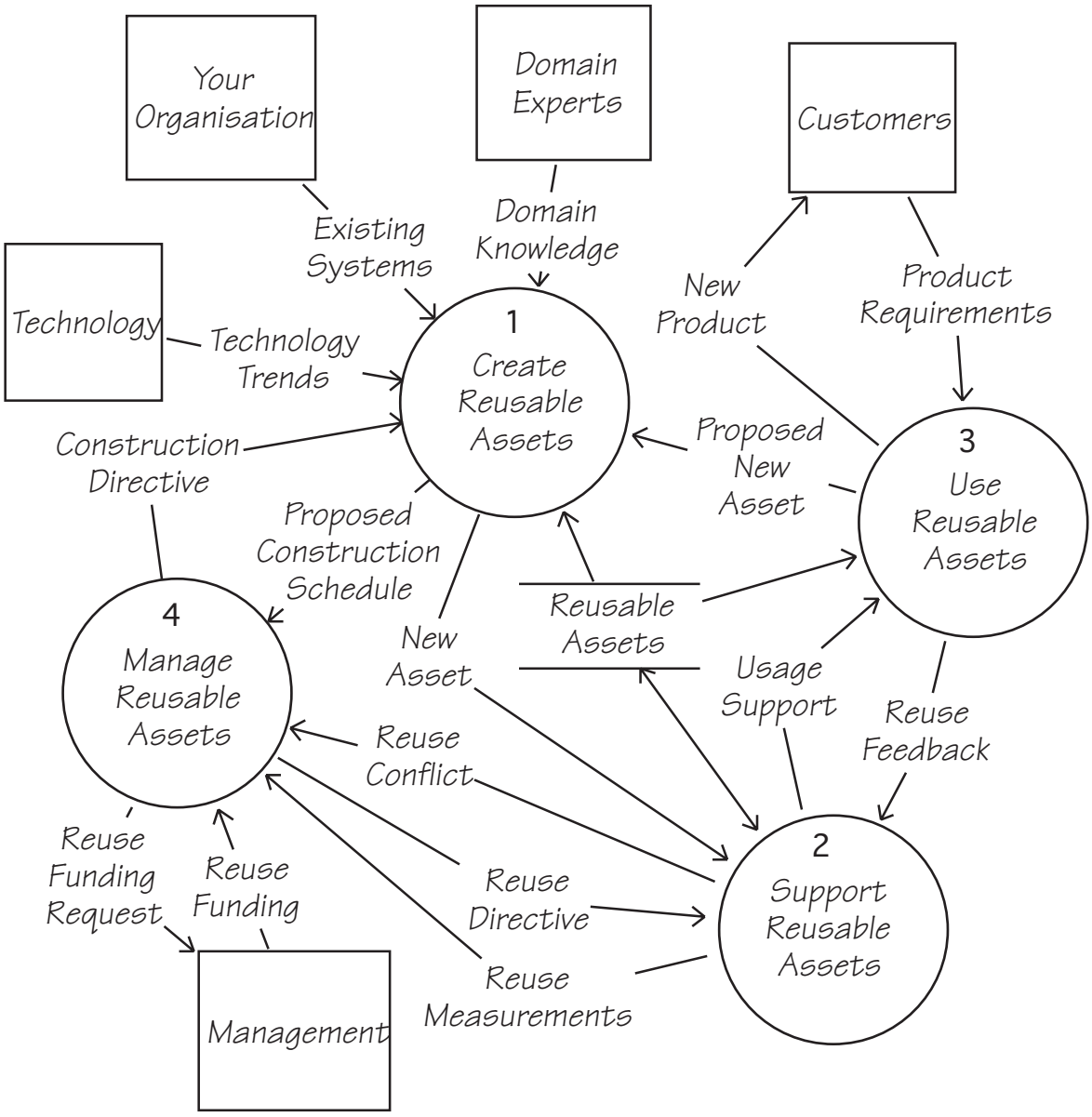

Figure 19: This model of the reuse lifecycle expands the context by showing the interfaces between the essential processes within the reuse lifecycle.

For example, the model tells us that under some set of circumstances the process that uses the assets (Process 3) sends some information about a *Proposed New Asset* to the process that creates the assets (Process 1). Each of the interfaces identifies a dependency between two processes or between a process and another system external to the essential reuse process.

These four processes identify the essential processes for a reuse lifecyle and the interfaces between those processes. The complete detailed essential reuse lifecycle model would also provide a detailed definition of each of the interfaces (data flows and stores) and each of the processes. For example the store of reusable assets is defined as:

Reusable Assets
= {{ Reusable Component + Creation Date}
+ {Reusable Pattern}
+ {Reuse Measurement Type + Reuse Measurement}
+ {Reuse Directive}
+ Reuse Classification Scheme}

From this we can see that there are many repetitions of reusable component, reusable pattern, reuse measurement and reuse directive and that there is a reuse classification scheme. Each one of the terms would be defined until it can be given a range of values.

The details of each process would also be specified. For instance, the process *2.1 Certify New Asset* on Figure 5 would be specified to give details of the rules to be carried out when taking a new asset and using information from the store of reusable assets to produce a certified asset.

Remember that the essential reuse lifecycle contains all the things that have to be done independent of the technology that you use. Your own reuse lifecycle is a design for fitting the essential lifecycle into your own implementation environment.

Designing Your Reuse Lifecycle

The design for your reuse lifecycle must specify who will carry out each of the essential processes, what technology will be used and how the interfaces between the processes will be implemented. Your aim is to allocate the essential processes to the most appropriate person/technology combination so that you are making the most effective use of your environment.

The ideal design is the one that allocates each essential reuse process to a different person or group of people. The design model would look rather like Figure 19 with the addition of the names of people and technology. This allocation means that the essential processes are not fragmented and each group can concentrate on one functional task which has minimal interfaces to other groups. For instance a group that is only responsible for creation of reusable assets can build assets that are focused towards the organisation rather than just one specific project. This group has the responsibility for keeping track of what is happening in the reuse industry and recommending the purchase of products like class libraries, and requirements patterns.

The support group is more concerned with the day to day organisation of the reusable assets. This group provides users of the assets with advice and sets up a mechanism for storing and accessing the assets. Perhaps your implementation environment already contains a tool that can be used to store your assets. Or maybe one of the tasks in your lifecycle design is to investigate the products that are currently available for building a reuse library. However it is not always appropriate to automate the storage of all assets.

The groups who use the assets are those who are building new products. These groups are concerned with producing a product within the constraints of the project. And that usually means fast. To make reuse work on a large scale, a project group must be supplied with appropriate reusable assets, advice on which assets are most appropriate and a mechanism for requesting and receiving new assets. If this mechanism does not work, then the idea of reuse will be little more than lip service and every project group will originate their own code.

Introducing Reuse

A requirements reuse lifecycle is a radical change in the way we build software, and as with any change there will be resistance. People who are expected to change the way they work need to understand why the change will be a benefit and how it will affect their work.

The Polish Railways are building a new system which involves developers and users at nine different sites. They wanted to share data and, even though the projects are separate, they wanted to reuse the data across all systems. The support group designed a lifecycle to implement the essential reuse processes (see Figure 20), that provides developers with a way of reusing data names and definitions that have already been defined by another project.

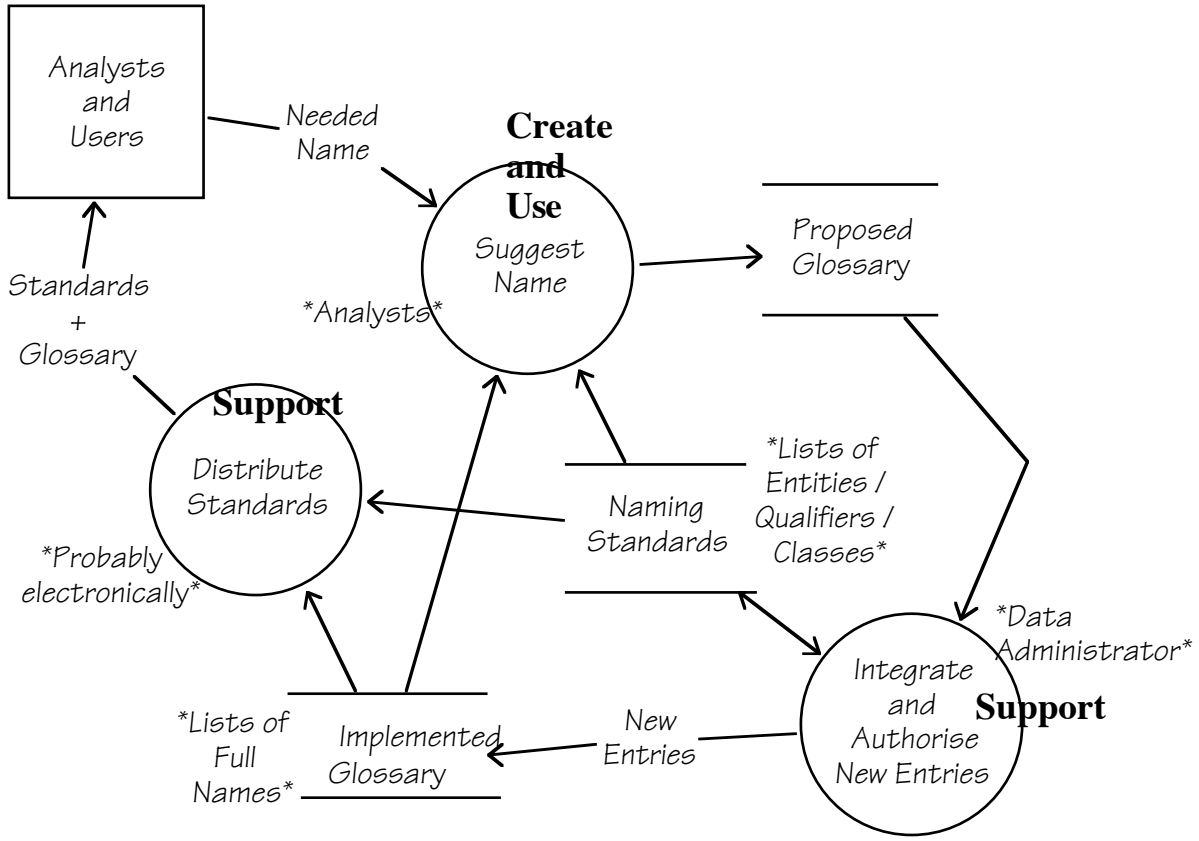

*Figure 20: This model is a design for a lifecycle for creating, using and supporting the reuse of data names. Information about the technology and people that will implement the lifecycle is enclosed in *asterisks*. The **bold** type identifies which essential processes will be carried out by each part of the system.*

The support group did a high level domain analysis to define a list of names for entities, classes and qualifiers. Whenever a requirements analyst or user on any of the projects requires a name they look in the implemented glossary to see if there is a name that they can reuse. If the name does not exist then the analyst proposes a name using the naming standards. Two of the support group are responsible for reviewing the quality of new names and integrating them into the glossary. The support group are helping the requirements

analysts and making them aware that they are already practising reuse and, as a result, experiencing much improved communication between projects.

The design for implementing a requirements reuse lifecycle varies tremendously depending on the people and technology in the implementation environment and the category of asset to be reused. The essential reuse processes, independent of the implementation environment, are an abstraction of the concepts of reuse. All of the essential reuse processes must be implemented somehow, using the facilities available within the implementation environment.

Towards Requirements Reuse

When you become more experienced in reusing requirements and you want to share the ideas across projects it makes sense to introduce a more formal requirements reuse process similar to the one that we discussed. However, the first step to reusing requirements is to have requirements reuse as one of the stated goals of your project. In other words, to build awareness within your team, to make it natural for people to ask – have we done anything like this before? And to make it respectable to spend some time thinking and looking around before doing some work that has already been done.

Informal requirements reuse within a project is the best way to get started. Using checklists and models so that you can talk about and search for reusable requirements helps you to develop your own common language. An essential element in becoming skilled at requirements reuse, is the ability to take an abstract view of known subject matter. We are spending a great deal of time in teaching and encouraging requirements analysts to improve this skill. Formal reviews of the requirements, apart from their usual purpose of trapping defects, are also a useful forum for encouraging abstraction and identification of requirements patterns.

Reusing requirements can provide a great deal of untapped business value. Communicating and realising this value depends on making requirements measurable. This in turn requires a consistent way of measuring the size of the requirements, the effort expended in gathering them and the savings realised in reusing them. The more consistent your approach to requirements gathering, the more potential you have for quantifying and hence putting a business value on your investment in requirements. Capers Jones in his work on conflict between software clients and developers [Jon1] uses function points as a way of measuring the size of projects. Function points is one metric that we can use to quantify requirements. We have discussed other possibilities like: the number of atomic requirements (functional and non-functional), the number of business events, the number of product use cases, the number of business classes/entities. These metrics, along with project resource statistics (time, people, budget) provide the input to doing a valuation of your investment in requirements. This works towards the idea that requirements analysis provides reusable requirements that are an investment that belongs to your organisation. And if you adopt the idea of requirements reuse then your investment is not left on the shelf gathering dust but is an asset from which you can realise value.

References

[Ale1] **Alexander, Christopher, Sara Ishikawa and Murray Silverstein**. *A Pattern Language*. Oxford University Press, New York, 1977.
This fascinating book shows how to make use of patterns when constructing buildings to make a more livable and beautiful house. We can borrow this idea for software construction and reuse successful patterns to lead to more usable and maintainable systems.

[DeM1] **DeMarco, Tom**. *Controlling Software Projects*. Yourdon Press, New York, 1982. Shows how managers can make software projects measurable by providing techniques making early and accurate estimates of time and cost.

[DeM2] **DeMarco, Tom**. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978. Systems analysis landmark book on how to build process models using data flow diagrams. Included here as one of the first books to introduce graphic modelling techniques, and hence ways of defining process patterns, for use by systems analysts.

[Gam1] **Gamma, Eric, Richard Helm, Ralph Johnson and John Vlissides**. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading Mass. 1995. ISBN 0-201-63361-2.
Reusable design patterns for common object-oriented design problems.

[Hof1] **Hofstadter Douglas R**.*Godel, Escher, Bach: An Eternal Golden Braid*. Penguin Books Ltd. England, 1980.

[Jon1] **Jones, Capers**. *Conflict and Litigation Between Software Clients and Developers*. Software Productivity Research, Mass. 1997.

[McM1] **McMenamin S. and J. Palmer**. *Essential Systems Analysis* .Prentice-Hall. Englewood Cliffs, NJ, 1984.

[Pat1] for a comprehensive list of books about patterns http://hillside.net/patterns/

[Rob1] **Robertson, James & Suzanne.** *Volere* 1995-2002.
The Volere requirements template is a scheme for identifying requirements knowledge. It can be downloaded from http://www.systemsguild.com

[Rob2] **Robertson, James & Suzanne**. *Mastering the Requirements Process*. Addison-Wesley, London, 1999.
A generic process for discovering, organising and managing requirements.

[Rob3] **Robertson, Suzanne & Kenneth Strunch**. *Reusing the Products of Analysis*. Second International Workshop on Software Reusability, Lucca, Italy, March 24-26, 1993

[Rob4] **Robertson James and Suzanne**, *Complete Systems Analysis*. Dorset House Publishing, New York, 1994.

Suzanne Robertson

Suzanne is co-author of *Mastering the Requirements Process* (Addison-Wesley 1999) a book that provides guidance on finding requirements and writing them so that all the stakeholders can understand them.

Suzanne is a principal of The Atlantic Systems Guild, a systems think tank. She has more than 30 years experience in systems specification and building. Her courses on requirements, systems analysis, design and problem solving are well known for their innovative workshops and business games.

Suzanne has varied experience as a manager, programmer, analyst, and designer. Since 1978, she has consulted, done research and taught in Europe, Australia, the Far East and the United States. She specialises in helping organisations to adapt modern systems development techniques to fit specific projects.

Current work includes research and consulting on patterns and the specification and reuse of requirements and techniques for assessing requirements specifications. The product of this research is *Volere*, a complete requirements process and template for assessing requirements quality, and for specifying business requirements.

Apart from her books, Suzanne is author of many papers on systems engineering. She is a member of IEEE and the Australian Computer and the British Computer Society's Requirements Group. She is the editor of the Requirements column in IEEE Software.