An Object Oriented Requirements Capture and Analysis Environment

Jean-Paul Gibson

An OO Requirements Capture and Analysis Environment

J. Paul. Gibson*
LORIA-UMR n^o 7503-CNRS &
Université Henri Poincaré
BP 239, 54506 Vandoeuvre-lès-Nancy
France

February 20, 1998

Key Words: Object Oriented, Validation, Specification.

Abstract

Requirements capture is the first step in the process of meeting customer needs. Building and analysing a model of customer needs, with the intention of passing the result of such a process to system designers, is the least well understood aspect of software engineering. The process is as much an art as a science. The object oriented paradigm offers a modelling technique which is effective at all stages of software development. In particular, the fundamental concepts of abstraction, encapsulation and classification are prominent, in many cases, in customers' understanding of their needs.

There has, in recent years, been widespread research in the development of formal object oriented models. Consequently, there is now a great opportunity for software developers to build mathematical object oriented models of customer requirements, with all the advantages that formality brings to later stages of development. Such models have been constructed in a number of specific case studies. However, there is no support, in the form of tools, for this type of engineering. This paper addresses many of the problems inherent in providing a general purpose formal object oriented requirements capture tool set.

*email: gibson@loria.fr

Contents

1 Requirements Capture: An Introduction

Requirements capture and analysis, within software development, is the first step in the long, often arduous, process of satisfying the needs of the customer. In short, it is the process of identifying and recording what is required. Unfortunately, the process must fulfil two very different roles:

- The customer must be convinced that requirements are completely understood and recorded.
- The designer must be able to use the requirements to produce a structure around which an implementation can be developed and tested.

The requirements interface between problem domain 'experts' (the customers, with potentially very little comprehension of computers) and solution domain professionals, who understand computer systems, languages, models and methods (but may have little prior knowledge of the problem domain). This makes requirements capture and analysis a not insignificant problem. However, as this paper argues, the process is made easier by the fact that many of the same principles of structure, organisation and method are common to both the problem domains and solution domains.

The development of any complex system requires at least guidelines, and preferably a method, to provide structure for the task. The fundamental principle of requirements capture is the improvement of mutual understanding between customer and analyst and the recording of such an understanding in a structured model. The construction and analysis of the common model provides the framework upon which structured development can proceed. Structure is fundamental:

- **Customer understanding is structured:**
  The customer views a problem and expresses this view in a structured way. Each customer views a problem in their own particular way — quite simply, a customer expresses their needs in the way in which they best understand them. For complex problems this understanding will most certainly be structured.
- **Analyst understanding is structured:**
  Analysts must merge the structured view of the customer with their understanding of the customer's view. The underlying requirements model must utilise the structure in the problem domain to help the analyst to identify ambiguities, contradictions and overspecification. Analysis of customer needs must be strutured and therefore the requirements model must be structured. Analyst-customer communication can improve mutual understanding only if the structure in the requirements model corresponds to the structure in the conceptualisation of the problem domain as held by the customer.
- **Designers understanding is structured:**
  The final output of the requirements capture process must be a model which can be understood by designers. The structure in the model must be easily mapped onto the implementation framework available to designers. Further, designers must not be asked to make

a large conceptual leap between problem domains and solution domains. Consequently, it is important that problem domain structure can be utilised by designers.

The requirements development process is summarised as a continual shift in the way in which the problem is understood and recorded. An analysis model records only the *current* understanding of the behaviour that is required. Modelling tools must provide a means of constructing and analysing a sequence of models which gradually evolve into a true recording of customer needs.

With simple systems, the mutual understanding between analyst and customer, as recorded in the *current* model, can commence at a reasonably high level. However, as customer requirements become more demanding (advances in technology mean that this is now a universal truth), the initial mutual understanding is necessarily very low. Further, the needs may be distributed across many individuals. The requirements capture problem is compounded by:

- Multiple views of customer needs.
- The need for a group of analysts.
- The dynamic nature of analysis — personnel and requirements.

The only realistic solution to these problems is a consistent method and set of requirements capture tools, based on a common modelling strategy.

2 Object Orientation: An Introduction

2.1 Background

Object oriented techniques and concepts have been shown to be applicable in the analysis phase of development [9, 2]. This should not be surprising since object oriented programming is often refered to as 'real world modelling' [3] which, in general, is what analysts are doing. The idea of applying object oriented methods, which initially appeared in the program domain [12, 11], to design and analysis corresponds to the way in which structured approaches, originated in the 1970's, gradually infiltrated each stage of software development [5, 6, 7, 14].

The object oriented philosophy does not throw away all the previous work on structured analysis: it re-uses many of the ideas and combines them in a coherent and consistent fashion. This paper argues that object oriented techniques make requirements capture easier (though not easy). Object oriented methods can be applied to different problem domains with much greater confidence in the underlying principles. Consequently, the analysis method gives prominence to the understanding of problem domain rather than model semantics. With traditional analysis techniques, the structure of the problem domain is often compromised. The object oriented appproach promotes the maintenance of problem domain structure throughout the whole development process. It is the conceptual integrity of the object oriented paradigm which provides the essential bridge between customer requirements and program design.

2.2 Fundamentals

Throughout the main body of this paper, object oriented concepts and terminology are widely used. This section gives a brief, informal introduction to the terminology being employed. One of the main problems with the object oriented community is the lack of common consent on the meaning of the basic terms. This paper addresses the inconsistency by initially stating what we regard as being appropriate definitions for each of the fundamental concepts.

Objects

The fundamental concept is the object, which combines both data and behaviour in a single entity. In general, the definition of an object must incorporate:

- A means of identification.
- An interface which encapsulates the object by forcing access to its behaviour (and state) to be through a well defined set of services.
- A semantics for the external services provided by the object — i.e. how an object in a particular state responds to a particular service request (by returning some result and/or updating its internal state).
- An internal state — in other words, a set of internal values upon which the behaviour of an object may depend.

It is important to distinguish between an object as a dynamic realisation (*instance*) of a class of behaviour and an object as a static *member* of a class. For example, a traffic light class may have three *members* (red, green and amber). However, there can be any number of traffic light *instances*. Throughout this paper, the term object is used interchangeably in both cases. The terms *instance* and *member* are used when clarification is required to remove the potential for ambiguity.

Classes

Two objects are said to belong to the same class when they exhibit the 'same behaviour' through their external interfaces. A class embodies the notion of a set of objects together with some common behaviour exhibited by its members. A class can define a parameterised set of behaviours. Each class member corresponds to a particular actualisation of the parameters (which we conceptualise as partitioning the object state). The real power in object oriented models arises from the class concept. A class definition must provide:

- An interface specification.
- A set of class members.
- A semantics for the external services provided by each member. An object can be said to get its semantics from the class to which it is bound.

Subclassing

A class A, say, is said to be a subclass of a class B, say, if every member of A is also a member of B. B is said to be a superclass of A. Subclassing must be reflexive and transitive.

A system class hierarchy is a set of subclassing relationships between classes in the system. This is often represented as a tree (or forest of trees) in which the classes are the nodes and the inter-nodal directed links represent the subclassing relationships.

Multiple subclassing occurs when a class has more than one direct superclass. In other words, a class has two superclasses which themselves are not related by a subclassing relationship. It is important that a general object oriented analysis environment caters for multiple subclassing as many problem domains exhibit this type of property.

Composition

An object can be defined as a composition of one or more component objects. The components of an object represent a partitioning of that object's state. The behaviour of a structured object (i.e. an object with components) is defined in terms of the services which the components provide. The composition relationship can be extended to the notion of class: when the state partition of all the objects of a class is the same (in other words, there is one parameterisation of semantic behaviour) then the class can be said to be composed from a set of component classes.

Polymorphism

There are two types of polymorphism which dominate object oriented modelling:

- **Genericity** — the definition of behaviour which is parameterised on classes: a generic queue, for example. A generic class definition provides a template for the generation of a set of class definitions. This feature is not unique to object oriented models, but it is important, nevertheless.
- **Inclusion Polymorphism** — in object oriented models, the external attributes of a class include all the external attributes of its superclasses. Consequently, when an object of one class is required (for whatever reason) then it should be possible to provide an object which is a member of a subclass of the specified class. For example, if asked for an integer then it is fine to reply with an even integer, provided the appropriate subclassing relationship is defined. This property is known as *substitutability* and arises in many problem domains.

Within our work, the notions of object, class, subclassing, composition and polymorphism (as outlined above) are considered fundamental. The formal treatment of these concepts is outlined in section 3.1.

3 An Overview

3.1 A Formal Foundation

However good object oriented methods are at modelling real world requirements, they do not necessarily provide a formal framework for mathematical reasoning and manipulation. Like traditional approaches, the diagrams common to object oriented methods are not formally defined. The strength of these diagrams is that the customer finds them easy to understand. This is also the root of their main weakness: by ensuring 'lay person' readability, the informal models lose much of their potential for mathematical reasoning and manipulation.

A formal model of requirements is unambiguous — there is only one correct way to interpret the behaviour being defined. Although the model must still be mapped onto the real world (i.e. validated by the customer), this mapping is in essence more rigorous than in informal approaches. Building a formal model requires a better understanding of the problem domain and a better undertstanding of how the problem domain is viewed by the customer. A formal model can explicitly model nondeterminism, when choice of behaviour is specified. Another advantage of a mathematical approach is that high levels of expressibility allow the definition of *what* rather than *how*. (In an ideal model, the object oriented framework would provide a structure upon which a solution *may* be built but it does not say that that is the structure upon which a solution *should* be built.)

Designers also benefit greatly from a formal requirements model:

- It facilitates rigorous verification practices.
- It encourages the development of automated verification tools.
- It explicitly identifies freedom of implementation issues.
- It encourages compositional re-use at earlier stages of system development, thus easing the burden on designers and implementers.

We aim to merge the advantages of mathematical modelling with the customer friendly object oriented diagramatic notations. We do this by encouraging the development of customer defined views of formal object oriented models.

3.2 A Customer-Oriented Environment

The analysis and requirements capture phases of software development should be *customer oriented*: it is generally agreed that customer communication is the most important aspect of the early stages of development [13, 18, 20]. The successful synthesis of a requirements model is dependent on being able to construct a system as the customer views the problem[8]: requirements validation is not possible if the models cannot be communicated to the customer.

Customer orientation also implies a certain amount of flexibility. [15] identifies the importance of *opportunism* in software development. This is even more true when customer interaction is considered. It is not suitable for us to enforce a highly prescriptive method:

- Working in a highly constrictive environment may discourage customers to communicate freely.
- Different customers will have different degrees of computer aptitude — we must provide the flexibility to cope with a wide range of users.
- Requirements capture, in essence, is as much an art as it is a science. Prescriptive methods are best suited to the solution of well understood problems.

However, although we advocate customer-oriented flexibility, it is necessary (as with all complex problems) to enforce some sort of structure on the requirements capture process. Customers and analysts benefit from an approach which is simple to manage and easy to understand. The development environment should always provide guidance when necessary, yet allow the users to experiment along the way. The conflicting natures of simplicity and flexibility must be balanced in a comprehensive set of tools.

Traditional analysis approaches do not 'use the customer' to their fullest potential: these methods are *analyst oriented* since all the methods and tools are problem domain independent. In traditional structured analysis methods, the customer features most prominently at the beginning and end of the requirements capture process. Initially, traditional approaches advocate an informal communication, together with a document transfer, from customer to analyst. The documents are those which the customer believes are valuable to the analyst. They may be provided in response to particular analyst requests, and may even constitute a set of informal requirements. The problem with this approach is that too much emphasis is placed on the documents and not enough time is spent interacting with the customer. The analyst then constructs a requirements model and then tests it against the customer's needs. Problems are identified and recorded (again in informal documents) and the process continues until the customer and analyst are satisfied that the requirements are complete and consistent.

Many problems arise from this approach because of the separation of the models from the customer during construction. The customer plays a major role only at the end of each cycle and is prominent when the analyst has problems understanding their requirements. Unfortunately, the problems that are most difficult to eliminate are those which arise from the customer misunderstanding their own needs. We aim to provide much more support for customers to validate their understanding of their requirements rather than validating their understanding of the models. In an ideal environment the analyst would identify the set of concepts with which the customer understands their needs, map these concepts onto the mathematical framework and let the customer construct their own requirements model. In practice it is more feasible to expect the customer and analyst to work together during the construction and refinement of requirements (as well as at the testing stages).

The environment must provide a set of tools which can be customised to different users. There must be a means for customers to create their own views (graphical representations) of the behaviour that they require, whilst maintaining mathematical rigour. Each stage of requirements capture would then be automatically available for validation (with no extra work required from the analyst) and the method consequently improves the interaction between customer and

analyst.

We recommend a predominantly object oriented approach since there is an inherent flexibility in such methods. Object oriented development is both bottom-up and top-down, supporting composition and decomposition at all levels of abstraction. Further, the scalability of object oriented methods means that different parts of the requirements model may be at different levels of abstraction without undue complication. Consequently, analysts and customers can move between system parts in a flexible fashion.

The structure of the problem domain should never be compromised to make the recording of the requirements suit the analyst. An analyst, on identification of a complex problem domain may suggest a better way for the customer to structure their understanding. This better representation may reflect a simplification, but if the customer does not agree with the suggested restructuring then no changes should be made to the model. Problems arise only when customers view their problems in a convoluted way and therefore make it difficult for the analyst to capture their requirements in a comprehensible fashion. In such an instance the analysts must attempt to educate the customer towards finding a better representation of their needs (if there is one). When the customer and analyst cannot find a mutually agreeable way of understanding the problem then there is no simple solution. Requirements capture is not complete until both parties are sure that they have a common understanding of the formal requirements model.

3.3 Visualisation

Graphics

Graphical views have long been used to represent large quantities of information in a simple and concise form. Humans have evolved a very complex mechanism for collecting and colating information that is presented graphically. Understanding the information depends on clarity of expression which, in turn, relies on meaningful structure. Graphical models can provide both these properties. Graphical views are prominent at all stages of software development because of their ability to convey structural aspects of a system.

All standard software visual models are particular types of graph — each model attaches meaning to the labelling of nodes and links and the relationship defined between connected nodes. Categorisation of graphical models is simply a grouping together of models in which the meaning attached to the views shares some commonality. It is precisely the meaning attached to graphical views which distinguishes different models.

Graphical Properties

Rather than attempting to identify a useful categorisation of graphical views used in software engineering, it is more beneficial to identify particular properties which are common to some views (but not others). There are three useful distinctions which should be made when presenting graphical information based on complex software systems:

- **Dynamic vs Static**
  Dynamic models represent properties of system as they change over its lifetime. Static models, on the other hand, represent properties which are constant during a system's lifetime.
- **Hierarchic vs Nonhierarchic**
  Some graphical views are hierarchic in the sense that there is an inherent ordering between nodes. In other words, the relationship being represented between nodes is transitive.
- **Scalable vs Nonscalable**
  Some graphical views are scalable: each of the nodes in the graph can themselves be treated as graphs (i.e. individual systems which can be subjected to the same analysis as the system in which they are found).

Throughout the main body of this paper (sections 4,5 and 6), these properties are identified in a number of different graphical views. We therefore propose that there is some consistent means, within our views, of highlighting these properties in any particular graph which exhibits them. This provides the user of the graph with an initial notion of the type of information which is being presented.

Modelling Languages

The underlying modelling language (semantic basis) is a major influence on the structure of a visualisation environment. Because the environment manipulates components in the language, this directly influences the environment's structure and form, though not necessarily its presentation to the user. A visual representation must be able to naturally model a conceptual system with the minimum amount of mental transfer and mapping on the part of the modeller (or viewer). We advocate an approach in which the fundamental modelling block is the object since it is considered the most basic construct available for representation. It is therefore important that our underlying semantic framework is also based on the object notion.

Visualisation Limitations

Until recently, computing costs were high and human costs relatively low: it was therefore justifiable that efficiency of machine utilisation took precedence over analyst/customer development time. However, with the dramatic decline in computing costs, it is now possible and highly desirable to optimize the human resource by providing highly interactive visual modelling systems.

The graphical capabilities of the machine being employed are now such that there is a real danger of *over-visualisation*. Analysts and customers must be made aware that pictures do not necessarily imply clarity. The nature of the brain is such that it can be effectively utilised only if the pictures are constructed in a logical (i.e. structured) fashion. Further, it is important not to overload the brain's pattern recognition ability with too much information — high resolution screens now make this very possible. As with non-graphical forms of modelling, the secret is to keep things as simple as possible.

3.4 Synthesis

Within structured software engineering methods, visualisation has played a role primarily during the analysis of models. The synthesis of models is largely done outwith a visual framework (although, syntax driven editors could be said to enforce pictorial structure on the specification being written). Models are, in general, constructed textually.

In our approach we aim to provide 2 types of visual synthesis mechanisms:

- **Complete** mechanisms are formally defined graphical manipulations which result in a new formal model which is well defined and subjectable to rigorous analysis.
- **Template** mechanisms are formally defined graphical manipulations which result in the generation of a textual specification template from which an analyst is expected to develop a complete model. Such a template will, of course, benefit the analyst during the modelling process.

Section 4.3 examines a number of proposals for visual synthesis mechanisms which are generally applicable in any problem domain which is modelled in object oriented terms. It is much more difficult to define visual synthesis mechanisms which are more problem domain specific. In practice, visual synthesis mechanisms need to be highly parameterised since optimum model presentation is a highly individual and contentious issue. This is a general problem with visual languages: there is so much more flexibility in how a specification is to be presented. We address this issue in section 5 by proposing a mulitple view approach.

3.5 Analysis

Visual Analysis

Model visualisation is an important part of requirements analysis, for both customer and analyst. There are three main aspects to analytic visualisation:

- Providing a means for the user to analyse a particular part or property of the model.
- Providing a means for the user to change the way in which the information is presented.
- Providing the user with a means to interact with the model through the visual representation.

These aspects are dealt with in sections 5 and 6.

Non-Visual Analysis

Given an underlying formal model of requirements, there is potential for automating certain aspects of the analysis process, for example: static type checking, completeness tests and consistency tests. In these cases (and others) it is unlikely that the user will want to be presented with a visual representation of this computer-intensive analysis.

4 A Constructive Approach To Requirements Modelling

4.1 Library as Language

A major strength of object oriented programming languages lies in their emphasis on the re-use of pre-defined components. The pure object oriented programming languages (Eiffel [16] and Smalltalk [12], for example) often have a very simple semantic kernel. These languages provide a semantic framework upon which programmers can construct their own classes of behaviour. Further, the languages are often accompanied by a large range of pre-defined classes (and class hierarchies). Once the fundamental language constructs have been mastered, programmers can concentrate on learning about the behaviours defined by library classes, combining predefined behaviours to create new classes and making these new classes available to others. In many respects, the underlying object oriented semantics can be 'forgotten about' and concentration placed on libraries. We asssume that a similar approach will be taken by requirements modellers. Consequently, within an ideal OO development environment, the notion of library browsing is given prominence (see sections 5 and 7).

4.2 System = Class + Hierarchy

To construct a model of requirements one specifies a class of behaviour together with a class hierarchy environment in which the model is to reside. A class cannot exist distinct from the set of classes it depends on to offer its behaviour. Further, the classes it depends on may also depend on other classes .... This potentially very large set of classes will exist within a class hierachic framework (specified as a set of subclassing relationships). The way in which this framework is used by the system model class can vary considerably. The following examples illustrate some of the subtleties which are involved in consructing a requirements model.

Example 1: Dependency in a queue of integers

The LIFO behaviour of a queue is well understood and simple to specify. A queue of integers, *IntQ* say, can be said to **depend on** the data elements which it stores, i.e. the *Integers*. One could believe that the construction of the *IntQ* requirements model is straightforward. However, one must also consider the classes which *Integer* **depends on**. The *Integer* class may offer an *is-even* service which returns a *Boolean* value. We then say that *Integer* **depends on** *Boolean*. There are now two extremes to the way in which the *IntQ* model is constructed:

- The dependency of *Integer* on *Boolean* is ignored and *Boolean* behaviour is not evident in the final model.
- The dependency of *Integer* on *Boolean* is recorded in the final model and *Boolean* behaviour is recorded.

It is necessary to ask which method is *correct*. We conclude that the answer is not so clear cut. Traditionally, a queue is a passive data carrier, i.e. the services offered by the contained

data elements do not play a role in the queue behaviour. Consequently, in this case, one can construct the *IntQ* requirements model without considering *Booleans*. This simplifies the model and makes it much easier to understand. However, in many other systems the behaviour being offered is achieved through utilisation of the behaviour offered by the components. Consider a strange sort of *IntQ* in which the integer element being pushed on is rejected if it is the same as the element that is already at the top of the queue. In this case two integers must be compared and the boolean result used to determine the subsequent behaviour. This model cannot be constructed without incorporating the *Boolean* class.

Example 2: Incorporating a Class Hierarchy

Consider a *Thing* class which has three subclasses *Thing1, Thing2* and *Thing3*, as illustrated in figure 1. The *Things* class hierarchy is prominent in the specification of new behaviour (in *NewClass*).

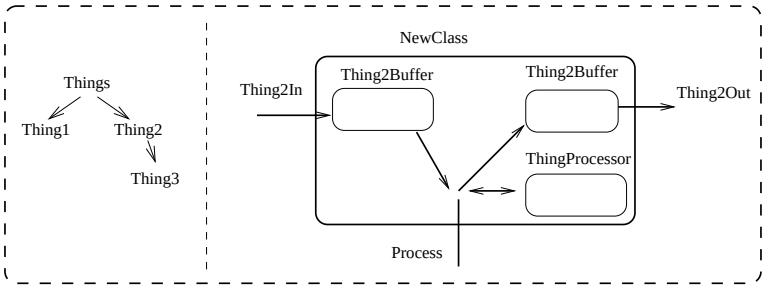

Figure 1: Incorporating a Class Hierarchy

The *NewClass* offers three services:

- *Thing2In* accepts a *Thing2* and buffers it for processing.
- *Process* takes a *Thing2* off the incoming buffer, transforms it into another *Thing2* using the *ThingProcessor* and puts it in the outgoing *Thing2Buffer*.
- *Thing2Out* extracts the *Thing2* element which is in the outgoing buffer.

When constructing the *NewClass* model there are a number of alternative hierarchic models:

- I) The system environment can be built without the *Things*, *Thing1* or *Thing3* classes. There are no subclassing relationships between the other classes in the system.
- II) The system environment can be built without the *Thing1* or *Things* classes.
- III) The system environment can be built without the *Thing1* class.
- IV) the system environment can be built without the *Thing1* or *Thing3* classes.

.

The precise nature of the environment of the system being constructed is dependent on a number of things, for example: the **dependencies** of the *Thing2Buffer* and *ThingProcessor* components, the contravariance or covariance properties required by the customer, and the internal processing mechanisms in *ThingProcessor*. It is not important to understand why such differences can arise (or the implications of these differences): it is important only that our environment allows the modellers flexibility in the way in which the class hierarchic environments of their models are constructed. Such flexibilty arises from our environment identifying the minimum set of classes (and class relationships) that must be supported by the model, whilst also permitting additional classes and relationships to be incorporated.

4.3 Alias Development

Complete Alias

The simplest means of specifying a new requirements model is to identify a class which already offers the behaviour which is required. For example, a *lightbulb* which can be *on* or *off* and provides the service *switch* can be specified by **aliasing** standard *Boolean* behaviour. In such a case it is much better for the *lightbulb* class to reference the *Boolean* behaviour and to handle the different naming conventions required by the *lightbulb* user. We must provide a simple visual means of creating aliases by forcing the modeller to rename services and, in some cases, internal state values.

Partial Alias

In some cases, it is possible that a class exists which offers all of the behaviour that one requires, together with additional unwanted services. One would then specify an alias class by renaming a subset of the services offered by the predefined class. This is known as a **partial alias**.

4.4 Extending An Existing Class Hierarchy

Two subclassing mechanisms should be available for generating a new model as a subclass (or superclass) of an existing class:

- **Extension**: this mechanism facilitates the addition of a service to an already existing class. For example, a *Stack* which offers services *push* and *pop* can be extended to offer a *count* service which returns the number of elements on the *Stack*.
- **Specialisation**: this mechanism specifies a subset of the members of an already existing class by defining a property which only this subset of members fulfil. For example, the class of *Integers* can be specialised to the class of *EvenIntegers* by specifying a divisible-by-2 property.

The inverse of **extension** and **specialisation**, namely **restriction** and **generalisation**, must also be provided as explicit visual synthesis mechanisms.

[9] offers many examples of modelling using these classification mechanisms. It shows that the mechanisms are very useful in the early stages of software development. However, in most instances, it is necessary to create a new class model which does not fit so readily in an already specified class hierarchy. The type of mechanisms used to visually create such models are collectively known as **composition techniques**.

4.5 Composition Techniques

Composition By Union

The simplest composition technique is a **union** mechanism in which a new class is defined as a conjunction of interfaces of already existing classes. In figure 2, the *NewClass* offers services *A* to *D* by passing them directly to the appropriate component classes. Component *Old3* illustrates the fact that it is not necessary for all the services of the union to be provided by the new system model.

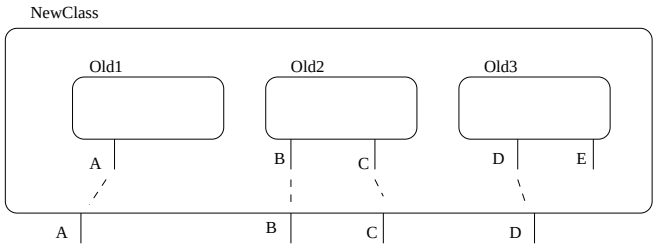

Figure 2: Composition by Union

In this example, the components (*Old1*, *Old2* and *Old3*) do not interact with each other. The behaviour of *NewClass* has been partitioned as a conjuction of the subset of services offered by its components. This type of composition is very simple. Within an object oriented approach such a **union** mechanism can quite easily be provided visually. However, like aliasing and subclassing, only a small percentage of requirements models can be constructed in this way.

Composition By Sharing

Consider the **union** example in figure 3.

The component *Old1* fulfils service *B* by using service `b` of *Comp1* and service `b` of *Comp2*. Similarly, *Old2* fulfils service *A* by using the `a` services offered by its components *Comp3* and *Comp4*. It is possible that the intended behaviour of the *NewClass* system is for *Old1* and *Old2* to *share* the same component instead of maintaining separate components. For example, *Comp2* and *Comp4* may be intended to be the same large database. A simple union mechanism does not facilitate the definition of this type of sharing. Consequently, we require the environment

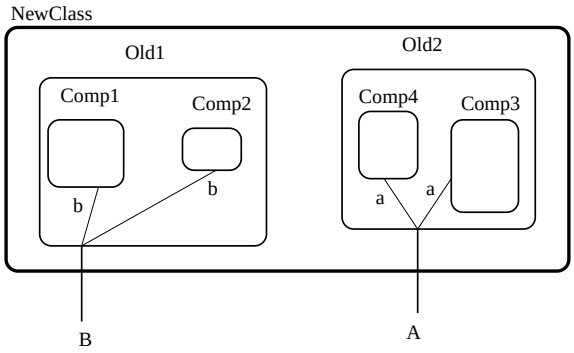

Figure 3: Another Composition by Union

to offer a **visual synthesis mechanism** corresponding to our *sharing* notion. This type of visualisation is represented in figure 4.

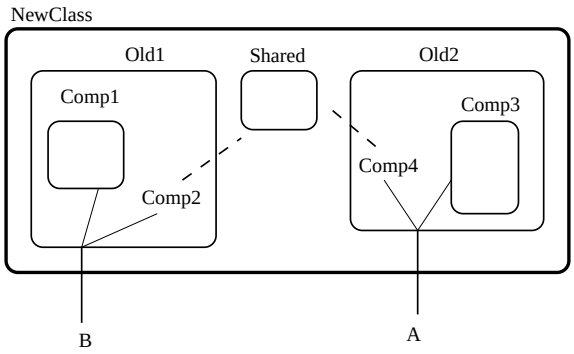

Figure 4: Composition by Sharing

In our experience, **sharing** is much more powerful than the simple **union** mechanism. We must be able to provide a **visual synthesis mechanism** which provides **sharing** functionality.

Composition By Internal Connection

Simple Data Flow

Consider a sysem of two stacks in which one wishes to provide three services:

- *push* an element onto the first stack.
- *move* an element from the top of the first stack to the top of the second stack.
- *pop* an element of the second stack.

Neither the **union** nor **sharing** mechanisms provide a means of visually synthesising this behaviour. Simple analysis of the problem identifies that *push* and *pop* operations can be passed

directly to the component stacks. The *move* service requires the result of a *pop* on the first stack to be *pushed* onto the second stack. Such a composition can be visually synthesised by specifying a simple internal data flow connection, as shown in figure 5.

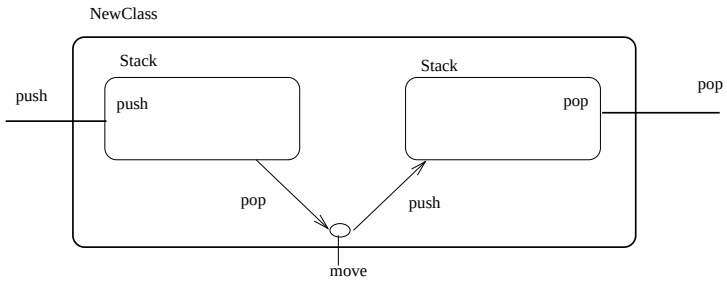

Figure 5: Composition by Data Flow

Data Broadcast

Consider a similar system to the one above in which there are again two stack components. The *NewClass* offers *push*, *pop1* and *pop2*. The service *push* puts the data element specified onto both stack components. The services *pop1* and *pop2* pop the top element of the corresponding stack component. This can be visually synthesised quite simply, as shown in figure 6

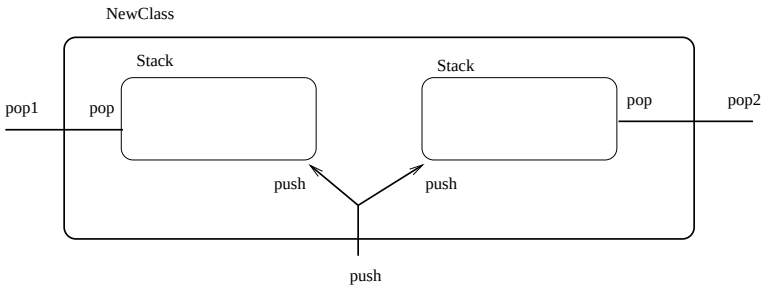

Figure 6: Composition by Data Broadcast

Template Mechanisms

The internal connection mechanisms (illustrated above) are useful in a small number of particular instances. The graphical syntax we have used for these re-usable mechanisms is not intended to be taken as *final*. We propose that for each re-usable composition mechanism, the client and analyst agree upon a particular graphical representation. In some cases, we may require more than one graphical representation for each composition operator (depending on the client or the point of view). The important thing is that there is a mapping from the formal text to the

graphical view(s), and vice-versa. Such standard views can then be thought of as defining a graphical specification language which is *client-oriented*.

In many other cases, the way in which a class uses its components is more complex than can be expressed in a standard view. Consider, for example, as system of three stacks in which a *transfer* service pops elements off two of the stacks, compares the two values and pushes the largest of the two values on to the third stack. It is not reasonable to expect a graphical syntax to be rich enough to express such behaviour in a coherent and concise fashion. More generally, such specific behaviour is usually defined textually. A graphical representation of this type of behaviour can usefully illustrate the dependency between a class and its components. This is illustrated in figure 7, below.

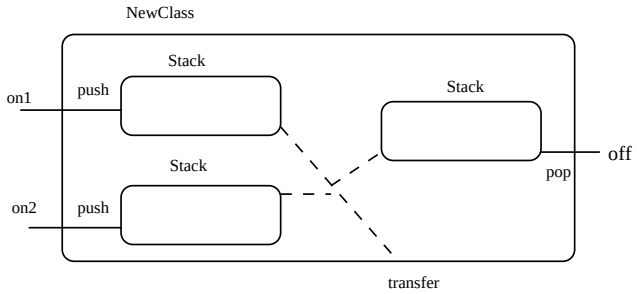

Figure 7: A Dependency Example

The dotted lines show dependencies, whilst the single lines continue to show direct internal connections. Visual synthesisation of such a model results in a specification template which can be used by the modellers as a frame upon which the complex dependency can be defined and checked against.

5 Visualisation: A Multi-View Approach

5.1 Standard Graphical Functionality

Graphs of labelled nodes and links make up the majority of ways of visually representing information in requirements models. In large, complex systems graphs can become difficult to manage — there is too much information to be replicated on the screen at any time, and the way in which it can be presented is quite arbitrary. Standard packages exist for navigation around, zooming into and compression of graphical images. It is important that such functionality is found in our environment.

5.2 Graphical Parsing and WhiteSpace

It is neither possible nor desirable to work only with graphical/visual specifications. Specifications are stored and analysed textually for a number of reasons:

- Mathematical reasoning and manipulation is restricted to textual specifications.
- Textual specifications are easier to parse.
- Graphical information is more *resource heavy* than textual information.

As with textual specifications, analysts and customers must be aware of the difference between specifications which are *parsing equivalent* and those which are *behaviour equivalent*. For example, consider the following two fragments of code:

- I: *read(x); x:=x+1; write(x)*
- II: *read(x);*
  x:= x + 1;
  write(x)

The whitespace used to indent the code plays no role in the specification of meaning, but it can help in its presentation. This type of presentation equivalence is also evident in visual specifications. For example, consider the three visual models in figure 8.

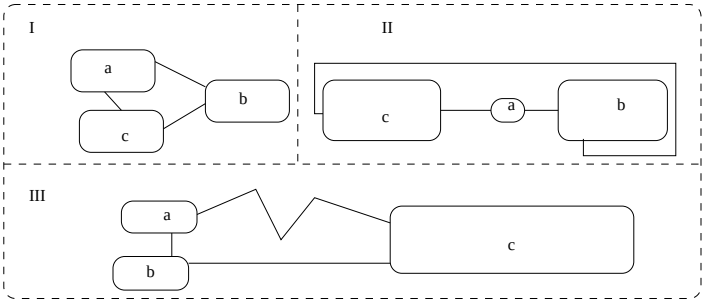

Figure 8: Whitespace Examples

Provided the orientation on the page, size of nodes, length of links ... are not part of the visual semantics then these three visual models are *parsing equivalent*. Clearly, the 'whitespace' in pictorial information, as with textual specifications, can help to improve the quality, in terms of how well it communicates its meaning.

Many computer languages have 'pretty printing' mechanisms which remove the arbitrary whitespacing and enforce a consistent and meaningful indentation strategy (in some cases the size, type and font are also manipulated) to produce a more presentable piece of code. Within our environment, it is important that the same facility exists for each of the different visual mappings. In other words, there must always be a default means of re-representing the graphical information in a way which is beneficial to the viewer. We call such a feature *pretty picturing*.

Consider a graph as presented in view I of figure 9. There are a number of different ways of improving figure `I` for the viewer:

- In view II the crossed links are removed.

- In view III the focus is placed on node D (with the size of the nodes decreasing as we move further from the focus), the orientation is placed down the page and the nodes which are more than 3 links away are not represented.
- In view IV the focus is on node C and orientation is from the centre out.

.

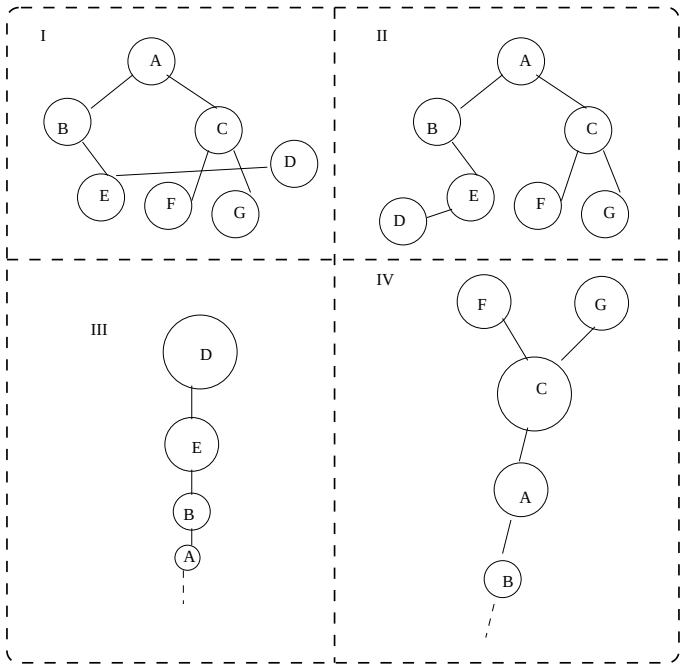

Figure 9: Pretty Pictures

The viewer should be able to change the visual presentation of a requirements model using a combination of predefined *visualisations*. Further, there must be scope within the environment for the viewer to define new ways of presenting the visual information.

5.3 Abstraction

Visual abstraction involves presenting only part of the model being visualised. Figure 10 illustrates abstractions of the information contained within a class hierarchy.

The three different views can be defined as visual abstractions of the class hierarchy presented above them:

- View1 is defined by identifying the tree to which B belongs.
- View2 is defined by identifying the descendants and ancestors of F.
- View3 is defined by identifying the tree which is rooted at B.

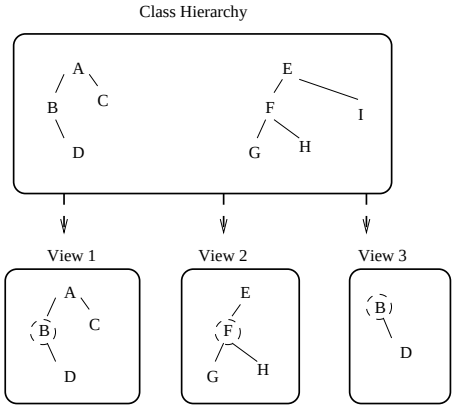

Figure 10: Visual Abstraction In A Class Hierarchy

. These are examples of the simplest types of visual abstraction — choose a property which is found throughout the model (in this case the subclassing relationship) and present a meaningful subset of these properties.

A different type of visual abstraction is concerned with the detail of information which is presented. For example, consider the sequence of diagrams in figure 11.

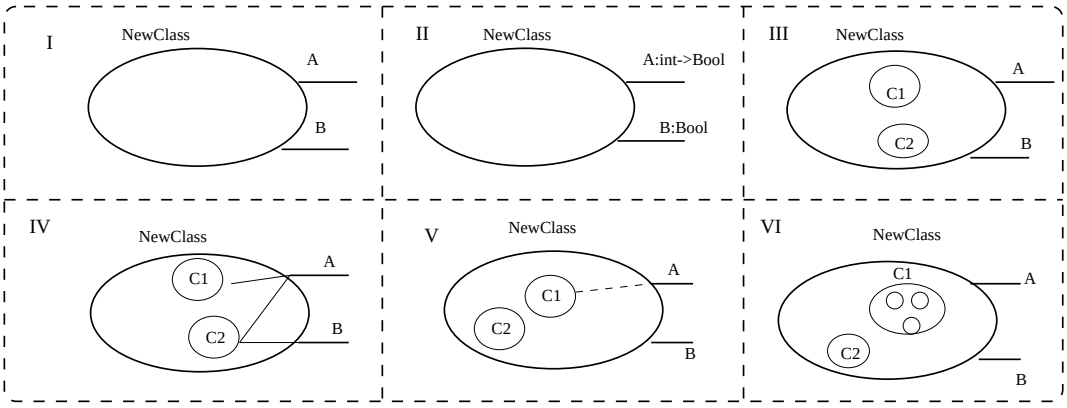

Figure 11: Visual Abstractions: Different Levels of Detail

Each of these diagrams abstracts away from some of the information in the model to emphasise other properties:

- Diagram I — shows only the untyped interface of *NewClass*, i.e. the names of the services *NewClass* offers.
- Diagram II — shows the type interface of *NewClass*, i.e. the names of the services together with the input and output parameter types.

- Diagram III — shows the internal composition of *NewClass*.
- Diagram IV — shows the dependency between *NewClass* and its components for each of the services offered.
- Diagram V — identifies, for each service, the components of *NewClass* which may undergo state transitions as a result of *NewClass* fulfilling the service.
- Diagram VI — represents the total composition of *NewClass*, i.e. the total composition of each of its components.

This list is not intended to be complete. It illustrates only the type of information which visual abstractions can be used to highlight. The viewer must always be able to choose the level of detail in the information that they are being presented with.

5.4 The General Visualisation Framework

Figure 12 shows the general framework of our ideal environment.

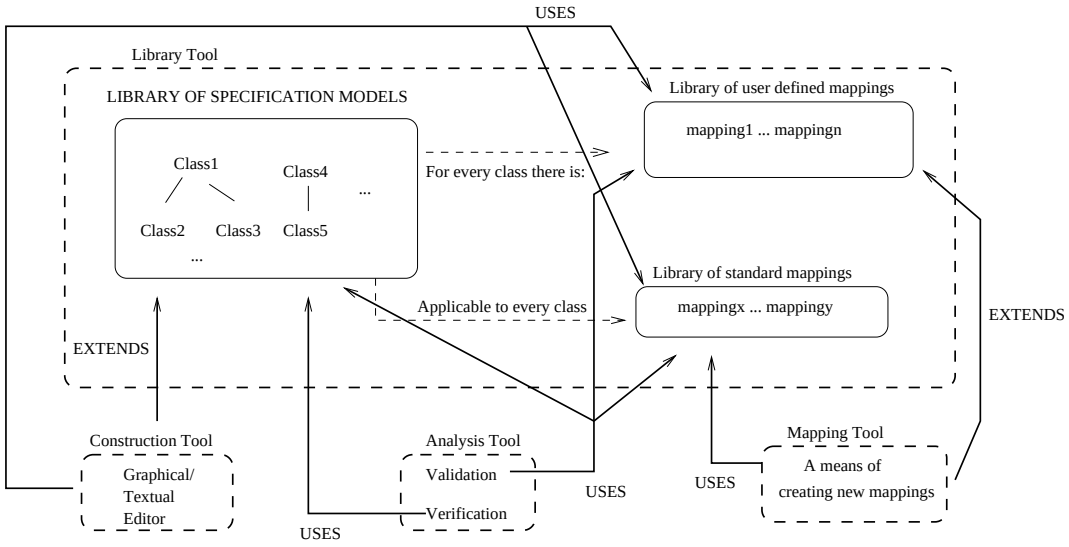

Figure 12: Visual Abstractions: Different Levels of Detail

Within the visualisation process the important aspects of such a framework are the mapping libraries and the mapping tool.

Standard Mappings

Standard mappings are those which are applicable to all specification models. The types of mapping which are important are:

- The presentation of the class hierarchy to which the class model belongs.

- The presentation of interface information.
- The presentation of composition properties.
- The presentation of dependency properties.
- The graphical representation of every member (object) of a class.
- The visual means of interacting with object representations.

The thesis by Gibson [9], and many of the diagrams in this paper, illustrate the type of graphical syntax which can be usefully employed. However, we do not consider the more difficult task of the definition of appropriate mappings which will automatically generate such diagrams from the specification models (and vice-versa, in constructive visualisation).

User Defined Mappings

The implementation of standard mappings is in many senses much simpler than supporting user-defined mappings. A simple example may illustrate the difficulties. Consider a queue of integers in which three elements, namely the integers *1*, *2* and *3* have been pushed on. The standard representation of such a queue could be as presented in figure 13.

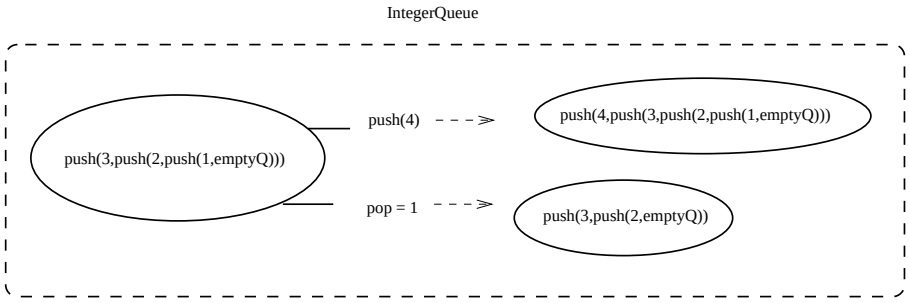

Figure 13: A User Defined Visualisation

In this figure, the state of the queue is represented by an expression within the queue node. The result of two service requests is shown as a simple state transition (which may also be annotated by the value to be returned to the service requester). There are two obvious problems with the standard representation:

- The viewer must be able to interpret the expressions which define the internal state values.
- For queues with large numbers of elements, it is likely that the state representations will be too complex for viewers.

To combat these problems the user may choose to provide their own mapping (the definition of such a mapping is not considered in this paper). Figure 14 illustrates a possible result of a user defined mapping on the *IntegerQueue* class.

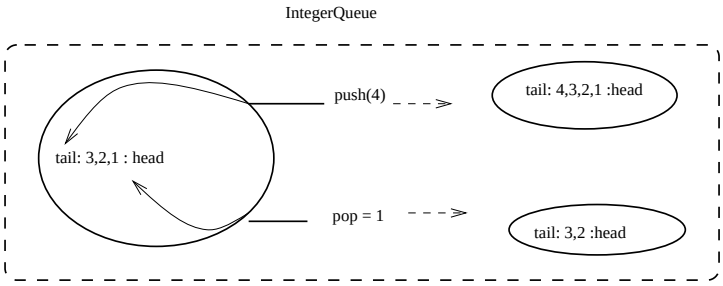

Figure 14: A User Defined Visualisation

The removal of the complex parentheses and the identification of the head and the tail of the queue improve the presentation.

The creation of user defined mappings is very much related to the concept of user interface design. Extra burden is placed on the analyst to provide mappings which are acceptable to the user. However, this extra work does lead to the development of rapid prototypes in which the interface of the system has been given as much consideration as the functionality which it offers.

5.5 System Presentation, Manipulation and Analysis

There are three distinct modes of operation in our ideal environment:

- **System Presentation**
  The creation of mappings is known as **visualisation**. It is the process by which relationships are defined between formally defined specification models (and templates) and graphical constructs (such as lines, boxes, circles ...).
- **System Manipulation**
  The creation of new classes of behaviour and adaption of already existing classes. This is known as **construction**. Construction mechanisms may utilise the visual mappings and may even be defined in terms of visual manipulations.
- **System Analysis**
  The development of requirements models is an evolutionary process. Initially, there will be many problems with the models which will gradually be removed by customer and analyst. The identification of errors and the testing of correct behaviour can be done only through thorough analysis. Analysis can be improved through the use of the available visual mappings and graphical animations.

Section 4 examined different construction mechanisms. Section 5 has examined the notion of visualisation. The next section of this paper completes the triangle by more clearly identifying the role of visualisation during analysis.

6 Analysis: The Validation of Static and Dynamic Properties

6.1 Static Analysis

Static analysis refers to the testing of properties of the model which are constant throughout its execution. We have already seen three different types of static analysis visualisations:

- **Class Hierarchy Diagrams**
  The class hierarchy is a fixed set of subclassing relationships which cannot change during system execution.
- **Dependency Diagrams**
  The internal structure of objects is defined by their components (and their interactions between the components). The way in which an object uses it components to fulfil behaviour can only be changed by the modeller. Once a system is constructed then these properties cannot change.
- **Interface Diagrams**
  The services offered by an object are defined by the class to which an object belongs. This cannot change during execution.

There are many possible ways of visualising this information — at different levels of abstration and in different presentation formats. The diagrams presented in earlier parts of this paper show only possible visualisations. It is important that there are a wide range of visual mappings available for the viewer, together with a means for the viewer to create their own mappings.

Static properties can be represented without the need for the viewer to be able to interact with the graphic being presented. This contrasts with dynamic properties, where viewer interaction is a very important part of the analysis.

6.2 Animation

Overview

Animation, is defined by the following simple algorithm:

> 1: Identify an object in a class by its initial state. 2: Represent the object and its state. 3: Accept an interaction from the viewer which corresponds to a service request which the object can fulfil. 4: Fulfil the service request by updating the object's state and/or returning some result. 5: For the new object, goto step 2.

At any point in this process, the animation can be stopped by the viewer.

There are two equally important aspects to the visualisation of such an animation process:

- How to represent an object and its state.
- How to represent an interaction with an object.

We have already addressed the importance of providing mappings for the presentation of object state. Within our approach there must also be flexibility in the way in which interactions are presented.

Experimentation

Experimentation is the phase that follows the construction of a new requirements model. The purpose of experimentation is to learn more about the system under study by subjecting its model to various interaction sequences selected from ligitimate inputs.

The process of constructing experiments is itself a modelling activity: one builds a model (or models) of the environment of the system being analysed. This can be done in an ad-hoc fashion by the viewer subjectively selecting interactions during each cycle of the animation. We must also provide facility for a more planned creation of experiments which permit the controlled exercise of the system through different simulation scenarios. There are a number of imprtant aspects to experimentation:

- *Full animation vs Statistics Gathering*
  The experiment together with the system model may be executed without interaction from the viewer. This auto-animation can either be presented to the viewer as-if they were involved in the visual interactions. Contrastingly, the viewer may not wish a full animation to be presented. In many cases the animation process is being used to check a set of predefined properties or for the purpose of gathering statistics. We must offer each of these facilities.
- *What vs How*
  Experimentation, as a closed model, can present the behaviour of a system as a black box — the internal state of the system can be abstracted away from and only the sequence of interactions need to be presented for analysis. In other words, the analysis is concerned only with *what* the system is doing at its external interface. This type of black box testing is fine in requirements models which are complete. However, whilst the modelling process is continually refining both *what* is being specified and *how* it is being specified, it is important that the viewer can choose to see different internal propertie of the model in question. In particular, we must provide facility for intercepting data being exchanged between subcomponent models and the means to relate the cause and effect of such internal interactions.
- *Nondeterminism*
  During requirements capture, the modeller often wishes to specify nondeterministic behaviour. There must be flexibility (during animation) in choosing random number generators or using an external algorithm or data file.

6.3 Archiving

It is clear that there must be mechanisms for storing and retrieving models (archiving). What may not be so clear is the need for archiving the types of analysis (both static and dynamic) that models have been subjected to. This is important because:

- It helps during the creation of a new system in which analysis plays a prominent role.

- It aids modellers to understand models which have already been constructed and tested.
- It promotes confidence in library models.
- It improves understanding of the validation process.

7 An Ideal Requirements Environment: An Overview of Functionality

The previous sections in this paper have addressed a number of issues concerned with the construction, visualisation and analysis processes inherent in requirements capture. In many sections we have touched upon the type of functionality that an ideal requirements capture environment should offer. This section brings together all these ideas into one coherent whole: an overview of the functionality offered by the environment tools.

7.1 The 'hypercard' approach

One of the advantages of the apple *hypercard* system is the way in which it copes with users with different degrees of competence. It provides layers of functionality which users can progress further into as their abilities (and needs) expand. At each layer, the user can access the functionality provided by the previous layer together with some additional functionality. The power of such an approach is threefold:

- **Confidence**
  Users can confidently work within the level which they are best able to cope without worrying about complexities at lower levels.
- **Safety**
  Users can be sure that the layers below the one on which they are working cannot be adversely affected by their work.
- **Semantic Clarity**
  The underlying semantic framework is clarified for the user — in *hypercard*, it is clear how the interface connects to the information model which is then implemented in the model language.

We advocate a similarly layered approach:

- **Customer Layer**
  Customers can perform analyses on predefined models which are represented in predefined ways.
- **Visualisation Layer**
  Visualisers can change the way in which models are to be presented to customers. There is facility for examining predefined mappings, altering existing mappings and creating new mappings. Visualisers need acces to the customer layer to test their mappings.

- **Modelling Layer**
  Modellers access the model specifications with the ability to change predefined specifications and create new ones. Modellers must have access to the customer layer to test their understanding of the models. Further, they may also require access to the visualisation layer to update visualisations in response to model changes.

The user must always be aware of the layer in which they are working. All the tools in our environment must provide a consistent user interface to all types of user (otherwise movement between layers would be made more complex).

7.2 Using Windows

Our environment requires a high resolution screen with windows facilities. The amount of graphical information that will be required to continually update may be very large. To simplify things, we advocate using a small set of different window types, each with distinct appearance and functionality.

A Class Hierarchy Window

The starting point for the construction or analysis of a requirements model is the library of predefined classes. In general, it is useful to be able to distinguish between:

- Generally Applicable Library Classes — behaviour which is well tried and tested and useful in a wide range of problem domains. Such classes are the fundamental building blocks for all modellers. There must be some form of security which protects such library classes from corruption.
- Problem Domain Library Classes — behaviour which is well tried and tested and useful within particular problem domains. Modellers will be knowledgeable about only a subset of these libraries.
- System Development Libraries — behaviour which is currently under development.

The class hierarchy window provides access to all three of these types of library. Each window is *focused* on one particular class. This *focus* can be selected in a number of different ways:

- Visually — by pointing to a specific class in the hierarchy.
- By Name — identifying a particular class in a particular library.
- By Usage — often one wishes to find a class which is used by another class.
- By Dependency — often one wishes to locate a class which depends on another.
- By Keyword — often one wishes to find a class which offers some sort of functionality. This can be done by searching the specifications for specific keywords.

Class Hierarchy Customer Mechanisms

There are six fundamental functions which must be provided by the customer layer of the class hierarchy window:

- **View Manipulation**
  In a large class hierarchy it is often necessary to search through particular areas of the graph and present only a subset of the information. The customer must be able move around the hierarchy, zoom in and out, change the orientation and alter the depth of the view.
- **Class Selection**
  By visually identifying a class in the presented hierarchy, the viewer must be able to create a new class hierachy window with the selected class as *focus*. The viewer must then be able to close any of the class hierarchy windows which are presently displayed.
- **Reviewing**
  Whilst carrying out a thorough browse of a class hierarchy, which may lead to the creation and deletion of many windows, it is often useful to retrace ones steps through the analysis. We must provide a mechanism for providing such a review process.
- **Archiving**
  The customer must be able to store and retrieve reviews.
- **Instance Generation**
  Given a selected class it is important to be able to generate an instance of the class. There are two ways in which this can be done. Firstly, the customer must be able to ask the system to provide (randomly) a member object. Secondly, the customer must be able to view all possible members and choose one themselves. In each case this results in the creation of an animation window.
- **Static Analysis**
  The customer must be able to see information concerning the interface of the selected class (in an interface window) and properties concerned with the internal structure of a class in a composition window.

Class Hierarchy Visualisation Mechanisms

Within the class hierarchy window, the visualisation layer must offer the following services:

- Browsing through visual mappings which are applicable to the *focus* class.
- Creation of new mappings.
- Changing existing mappings.
- Defining particular mappings as default to specific customers.

The means for providing such functionality is not examined within this document.

Class Hierarchy Modelling Mechanisms

Within the class hierarchy window, the modelling layer must offer the following functionality:

- Examination of the code of the chosen class with the ability to make changes.
- Creation of a new class as an alias, subclass or superclass of the selected class.
- Creation of a new class to be part of the library of the selected class but not related to the selected class in any other way.

These three different mechanisms should each be associated with their own particular type of modelling window.

Interface Window

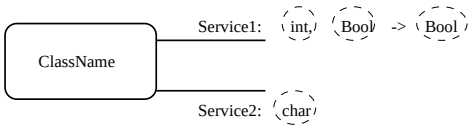

Figure 15: The Interface Window

Figure 15 illustrates a simple interface window. There is only one function which must be offered by this window at the customer layer, namely Class selection. The viewer must be able to select one of the classes which specify the input and output parameters of the displayed services. This selection results in the generation of a class hierarchy window with the selected parameter class as *focus*. In the figure, we must be able to select int, `char` or `Bool` classes in this way. The visualisation layer offers the chance to provide new representations of the interface information. No services are required at the modelling layer.

Composition Window

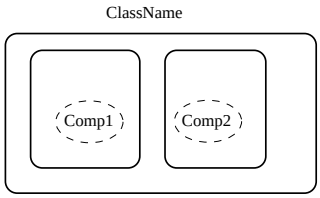

Figure 16: The Composition Window

Figure 16 illustrates a composition window. The customer layer must offer the viewer the option to select one of the component classes and view it in its class hierarchy, or view its

interface or view its composition. The visualisation layer must provide various mechanisms for the view to be changed or new views created. No functionality is required at the modelling layer because all changes to the specification must be done through the class hierarchy window.

Animation Window

The animation window provides the customer functionality for visual animation of the behaviour specified. Such functionality will cover all aspects covered by the experimentation section presented earlier in this paper.

7.3 Meeting The Ideal

Many of the features of our ideal environment are already present in a diverse range of CASE tools:

- Class hierarchy browsers/builders are evident in a wide range of object oriented programming language environments [12, 17], for example.
- Animation environments have been developed for programming languages and formal specification languages (see [21], for example).
- Visual mappings are a relatively new construct in the requirements capture field.
- Constructive visualisation mechanisms are comon in hardware modelling tools.
- Many formal languages have been developed to mathematically model object oriented properties and systems [9, 1, 19, 4]. There are a number of tools available for the synthesis and analysis of models specified in these languages.

Our environment is a conceptual prototype which points requirements capture in the right direction. It highlights the type of functionality which is useful to customers and analysts, and shows that visual mappings are the key to providing this functionality. We are currently implementing the graphical tools in JAVA and are testing our object oriented models and methods in the domain of telephone feature specification [8, 10].

8 Conclusions

This paper has presented a conceptual environment for the production of formal requirements models. Much of the approach is based on object oriented techniques because of their ability to bridge the customer-analyst-designer gaps. This paper, as it stands, is no more than an informal set of requirements on the functionality that the development environment should offer. Exact details of *how* such an environment is to be built are not given: it is beyond the scope of this work.

We have highlighted the triangular nature of requirements capture: the viewing, the modelling and the visual mapping. Emphasis has been placed on the need for an underlying formal semantics. Future work must concentrate on the generation of formal visual mappings. In this

ways the customer-accessibilty of the diagramatic representations can complement the powerful nature of mathematical modelling. That is the future of requirements capture.

References

[1] Robert Clark. Using LOTOS in the object based development of embedded systems. In *The Unified Computation Laboratory*. The Institute of Mathematics and its Applications (OUP), 1991.

[2] P. Coad and E. Yourdon. *Object oriented analysis*. Prentice-Hall (Yourdon Press), 1990.

[3] Brad Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley, 1986.

[4] E. Cusack, S. Rudkin, and C. Smith. An object oriented interpretation of LOTOS. In K.J.T. Turner, editor, *The 2nd International Conference on Formal Description Techniques (FORTE 89)*, 1989.

[5] Geoff Cutts. *Structured system analysis and design method*. Blackwell Scientific Publishers, 1991.

[6] T. DeMarco. *Structured analysis and system specification*. Prentice-Hall, 1979.

[7] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and techniques*. Prentice-Hall, 1979.

[8] J. Paul Gibson. Feature requirements models: Understanding interactions. In *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997. IOS Press.

[9] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Tech. report csm-114, Stirling University, August 1993.

[10] Mermet B. Gibson J.P. and Méry D. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997.

[11] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[12] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.

[13] IEE. *Special Collection On Requirements Analysis*. IEE Transactions on Software Engineering, 1977.

[14] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980.

[15] Ashley McClenaghan. Some thoughts on opportunistic design. Internal SPLICE document ash2, 1992.

[16] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[17] B. Meyer. *Eiffel: The Language.* Prentice Hall International Ltd., 1992.

[18] D. T. Ross. Structured analysis (SA): A language for communicating ideas. In *IEE Transactions on Software Engineering.* IEE, 1977.

[19] S. Rudkin. Inheritance in LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques IV.* North-Holland, 1991.

[20] K.J. Turner. SPLICE I: Specification using LOTOS for an interactive customer environment — phase 1. University of Stirling SPLICE Internal Technical Document, 1992.

[21] Adam Winstanley. *The elucidation of process-oriented specifications.* PhD thesis, The Queen's University of Belfast, 1992.