1994

Enhancing software reuse with intelligent documentation based on object oriented inheritance

Ian Sergeant
University of Wollongong

UNIVERSITY OF WOLLONGONG
COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Recommended Citation


Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
Enhancing Software Reuse with Intelligent Documentation
Based on Object Oriented Inheritance

A thesis submitted in partial fulfilment of the requirements for the award of the degree

Master of Science (Honours)

from

UNIVERSITY OF WOLLONGONG

by

Ian Sergeant, BSc. (Sydney)

Department of Computer Science

1994
I hereby declare that I am the sole author of this thesis.

I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis, or currently being pursued.

Ian Sergeant, June 1994.
Acknowledgements

I would like to thank my supervisor, Associate Professor N.A.B. Gray, for his patience and consideration throughout the preparation of my thesis. His suggestions and comments are always thoughtful and incisive.

I would like also to acknowledge Mayne Nickless for giving me time to attend meetings and seminars.

Abstract

To achieve software reuse has been an aim of software engineering since the late 1960’s. Software reuse increases programmer productivity, and can increase the quality of software. New technologies, such as object oriented programming, have been designed with the intent of enhancing the ability to reuse software, but have been applied with limited success.

This thesis focusses on the problems with reusing software using object oriented inheritance. It proposes that the documentation methods currently available for object oriented inheritance do not have the key programming factors that are claimed to enable reuse.

A methodology and environment for inheritance is developed that enhances software reuse. The Extended Viola Intelligent Documentation System (EVIDS) is an experiment in utilising the methodology and environment features proposed.
# Table of Contents

1. Introduction ............................................. 1
   1.1. Motivation ......................................... 1
   1.2. Overview .......................................... 2
   1.3. Summary ........................................... 3

2. Software Reuse and Inheritance ......................... 4
   2.1. Introduction ...................................... 4
   2.2. What is Software Reuse? ......................... 5
   2.3. History of Reuse .................................. 6
   2.4. Why Reuse Software? ............................... 7
   2.5. Enabling Software Reuse ......................... 8
      2.5.1. Programming .................................. 8
      2.5.2. Management .................................. 9
   2.6. Is Software Reuse Desirable? .................. 10
   2.7. Object Oriented Programming .................. 12
2.13. A-Kind-Of Inheritance and Models ........................................... 24
2.14. Problems with Models .......................................................... 25
2.15. Multiple Inheritance ............................................................. 27
2.16. Frameworks and Class Libraries ........................................... 28
2.17. Summary .................................................................................. 29

3. Documenting Inheritance ............................................................ 30
   3.1. Documentation for Reuse ....................................................... 30
   3.2. Requirements of Documentation to Enhance Reuse ............... 30
   3.2.1. The Software for Reuse must Work .................................. 30
   3.2.2. The Software must be Understandable ............................. 31
   3.2.3. The Software must be Locatable ...................................... 31
   3.3. Types of Documentation ...................................................... 32
   3.4. No Explicit Documentation ................................................. 32
   3.4.1. Source Code ................................................................. 32
   3.5. Copying Source Code Examples .......................................... 34
   3.6. Manuals – Written Documentation ...................................... 35
   3.7. Hypertext Documentation ................................................... 35
   3.8. Language Features ............................................................. 36
4.4.3. Programming for WIMP's, and DUNCE's
4.4.4. Exploiting User Knowledge
4.4.5. The Role of Constraints
4.4.6. Locating Software for Reuse
4.5. Controlling the Volume of Documentation
4.5.1. Powerful, General Systems
4.5.2. Naming Systems
4.5.3. Intelligent Documentation
4.5.4. The Documentation Yoyo Problem
4.6. Solving Problems
4.6.1. Naming
4.6.2. A-Kind-Of Inheritance
4.6.3. Abstraction
4.7. Summary

5. EVIDS - An Intelligent Documentation System for Inheritance
5.1. Intelligent Documentation
5.2. Viola
5.2.1. The Viola Language
5.7.2.1. Locating a Documentation Base Class ................. 87
5.7.2.2. Attaching a Documentation Class ..................... 88
5.7.2.3. Documentation Base Classes ......................... 88
5.7.2.4. Text Interface .......................................... 88
5.7.3. Documenting a Class Library .......................... 89
5.7.4. Slot Constraints .......................................... 90
5.8. Building a Simple Documented System ................. 91
5.8.1. Generating a Derived Class ............................. 91
5.8.2. Using Constraints ....................................... 92
5.9. EVIDS Support for Reuse Methodology and Environment . 94
5.9.1. Organising Properties and Behaviours ............... 94
5.9.2. A-Kind-Of Inheritance .................................. 94
5.9.3. Interaction .................................................. 94
5.9.4. Embedded Documentation ................................. 94
5.9.5. PROs, WIMPs, and DUNCEs .............................. 95
5.9.6. Inheriting Documentation ................................ 95
5.10. Writing Documentation with EVIDS ....................... 96
5.11. Limitations of EVIDS ........................................ 96
5.12. Extensions ...................................................... 96
1. Introduction

1.1. Motivation

Fifty years after the first simple subroutines were written for a computer, tertiary qualified software engineers write elementary routines daily. Many of these routines have been written elsewhere or could easily be written by someone less skilled.

New project team members, especially those relatively inexperienced in software engineering, are held back from working on production code until they have become familiar with the environment and the application. This is for fear of the mistakes they may make impacting on existing production code. The new team members have difficulty understanding and extending existing complex code.

Unfortunately, many technology advances have made the job of the new project team member more difficult. More varied and complex ideas can be expressed in the programming languages of today than in traditional procedural languages. This gives the skilled software engineer more power and flexibility in programming, but makes the interaction with the inexperienced programmer more difficult. The experienced software engineer can either use a simple subset of the language, ignoring the complex, powerful features, or the inexperienced programmer has a more difficult time understanding the existing code.
The motivation for this research is to allow software engineers to concentrate on areas where their skill is best put to use. They should be able to reuse code already written, and successfully delegate tasks on productions systems to programmers with less experience.

A longer term aim is to see programming novices independently using and combining well designed and tested components written by skilled software engineers. This makes the value of the work done by these engineers much greater.

1.2. Overview

The need for software reuse was first identified in the 1960’s with the software crisis. Software costs were increasing rapidly, and many software projects were not successfully completed.

Over the past two decades technologies to enable reuse, such as object oriented programming, have become widely used. Inheritance is a feature of object oriented programming languages not supported in other languages. It is one of the more controversial methods of software reuse.

Documentation is a form of communication. In software reuse documentation is the communication between someone who understands the software for reuse, and someone seeking to reuse the software. This communication is an essential part of software reuse.

This thesis claims that problems exist with the current methods of documentation available for documenting software reuse by inheritance. A methodology and environment for reuse of software by inheritance is proposed as the solution to these problems.
1.3. Summary

Chapter 2 provides a review of software reuse, emphasising the role that inheritance can play in software reuse. It proposes features of a programming environment that enhance reuse. These features are used to evaluate documentation and programming environments throughout the thesis.

Chapter 3 outlines the current techniques available to a person wishing to document software for reuse by inheritance and the problems associated with them.

Chapter 4 develops features of a programming environment and methodology for enabling software reuse by inheritance.

Chapter 5 develops EVIDS, an environment to support reuse using inheritance. It gives examples for using the system. It also shows how the methodology may be applied to other object oriented programming languages.

Chapter 6 contains an evaluation of the methodology and EVIDS developed here, including a comparison with other similar systems.

Chapter 7 presents conclusions, and shows how the results of this research can be incorporated into current software engineering technology.
2. Software Reuse and Inheritance

2.1. Introduction

Software reuse is an area of computer science that has received considerable attention over the past twenty years. It promises to make programming more efficient, and to reduce software development costs.

To describe software reuse an analogy is often made with industries that reuse components. Constructing a building entails assembling prefabricated components as specified in a design document. Electronics engineers select the components needed to construct a project from a catalogue. If it was required to build each component for each project, construction would take many times longer. Designs are also reused. New designs are modifications of existing designs with changes made to suit the required functionality and aesthetics of a new project.

Constructing computer software is still not a matter of selecting prefabricated components and combining them. This is despite the fact that between 40% and 60% of the code in applications performs essentially similar functions [Jones84].

Some software is reused. Most programming environments provide libraries of routines to perform common tasks. Programmers use ideas and experience from designing and coding one software project in later projects. It is
generally acknowledged, however, that software reuse has not lived up to its expectations [Bigge87].

The study of software reuse contains a large body of ideas for improving software reusability but only a small amount of empirical research on the amount of software reuse and how it was achieved. This ratio leads to a very wide range of potentially valid ideas.

2.2. What is Software Reuse?

Prieto-Diaz et al [Priet87] define software reuse as the use of previously acquired concepts and objects in new situations. This definition is general and emphasises the role of the programmer reusing software (the software consumer) over the programmer developing software for reuse (the software producer). Software reuse is, however, as much an issue for the software producer as it is for the software consumer.

Software reuse can be considered to be the use of program source code and software design in multiple situations. This definition places equal emphasis on the role of the software consumer and the software producer.

Software source code is information produced by a programmer that a compiler uses to generate an application. Reusing source code involves methods such as modifying an existing program, constructing a library of modules, using packages of components, or using common base classes in object oriented programming.

Software design is information known about the problem domain, and how that information translates into source code. Design can be expressed in words (written and verbal), diagrams (flow charts etc), and programs (code, class
relationships and comments).

2.3. History of Reuse

Software reuse has existed since the first computer applications. Libraries of sub-
routines for commonly executed procedures, like trigonometric functions, were
kept on punched cards. Software reuse became an area of research in computer
science in the late 1960’s [Nino90]. McIlroy’s paper [McIlr69] on mass produced
software components was the first to propose that software engineering could
become a process of assembling programs from reusable components.

The “software crisis” was identified in the late 1960’s. Software develop-
ment costs were increasing rapidly with no significant increase in programmer
productivity. The United States Department of Defense (USDoD) research identi-
fied software reuse as a key area for improving programmer productivity
[Freem87]. This had a large effect on design of the Ada programming language
which was commissioned by the USDoD, and the software technology for adapt-
able, reliable systems project (STARS) [Tracz87a]. It was anticipated that Ada
would be used for building reusable component libraries [Ichbi83]. The USDoD
sponsored a Ada Data Repository for the storage of reusable Ada components for
use by software developers [Conn87].

Many techniques were developed through the 1970’s that were of benefit
to software reuse [Freem87]. The Pascal programming language and structured
programming techniques allowed easier development of software components and
modules. The UNIX® operating system was developed which allowed the develop-
ment of small component programs, which could be combined by linking the out-
put of one program to the input of another. New programs could be created by
combining several existing ones using this technique[Kerni84b].
Jones [Jones84] identifies five issues in software reuse in 1983. Reusable data, reusable architecture, reusable design, reusable programs and common systems, and reusable modules. The problems of reusable data have been addressed by standards for data transmission and storage. The trend towards open operating systems, where a particular operating system runs on many hardware platforms reduces the problems of reusable architecture. Jones predicted that by 1990 50% of software development would consist of reusing previous work. This level of software reuse has not been achieved.

2.4. Why Reuse Software?

The main reasons for reusing software are productivity, software quality, and software consistency [Tracz87b].

Software reuse increases productivity by reducing the amount of time taken to develop an application. Design time can be reduced by further developing an existing design. Programming time can be reduced by using existing source code. Documentation and debugging time can be reduced as the reused software may already be documented and debugged [Nino90]. Maintenance time can be reduced as the maintenance programmer may already be familiar with the reused code.

Software quality can be improved as a new application can be built with existing software that is working and well designed. Economies of scale ensure that it is more economical to thoroughly design and test code that will be used more frequently. Improvements made to the existing software flow on to all the applications that reuse it [Spenc87].
Reusing standard toolkits increases software consistency. Software developed using the same toolkits can have similar behaviour. For example two applications developed using the same user interface toolkit will have similar user interfaces. An example of this is the Apple Macintosh Toolbox which provides routines for constructing user interfaces. All programs constructed for the Macintosh using the Toolbox have a similar look and feel.

2.5. Enabling Software Reuse

There is debate over whether enabling software reuse is a technical issue or a management issue. Aharonian [Aharo91] argues that management issues such as economics, technology transfer, training, legal issues, politics, tradition, and advancement in technology need to be considered to enable software reuse. Meyer argues that management issues are just the tip of the iceberg, and they will be overcome once the technical issues are resolved [Meyer87]. Some technical issues are part of the programming environment and others are the concern of management. Factors enabling software reuse can be grouped into those which are the concern of the programming environment, and those which are the concern of management.

2.5.1. Programming

The software for reuse must work. The consumer's required knowledge of the reused software must end at its documented external interface. It should perform its function reliably. Testing and use increases the reliability of software, as the more a piece of software is reused the more time can be economically allocated to its testing.

The interface between the consumer and the producer must be secure. If it is not possible to guarantee that software is error free then errors within the reused
software should be isolated within the reused software.

*The consumer must understand the software for reuse.* The software for reuse must be documented and encapsulated. The source code documentation must correctly document the functionality, external interface of the software, and the context in which it can be reused.

*The consumer must be able to locate software for reuse.* Reuse is enabled by matching what is available already and what is required in a new development Prieto-Diaz et al [Priet87] propose a classification scheme for software components. The classification of a component in their system involves determining what a component does, and in what context it can be used.

### 2.5.2. Management

*An organisation must be structured economically for software reuse* [Bigge87]. Corporations must have additional startup capital, as developing reusable software requires an initial investment for no immediate return. Software designed for reuse costs 20% to 25% more to develop than software that will not be reused [Tracz88c]. The value of the software for reuse must be estimated in advance, so that the cost of developing reusable software is not included in one project. This would make the price of earlier projects uncompetitive.

*Managers must reward programmers for reusing software.* A current measure of the productivity of programmers is the amount of lines of source code they write. As reuse involves more thinking time and less coding time, management must accommodate and encourage this [LaLon88]. In most corporate structures the programmer has nothing to gain by reusing software. It creates less work, and writing new designs and source code is usually considered a better job than
working with old software and existing designs.

**Barriers between local and foreign code must be eliminated.** The value of reused software increases each time it is reused. The not-invented-here syndrome and proprietary software make it more difficult to get full value from reusing software. If software is not shared between organisations then large corporations will get more economies of scale from software reuse than small organisations.

**A program must balance efficiency and generality.** To be reusable in more situations the software needs to be more general. A more general program tends to be less powerful and less efficient than a specific one [Bigge87].

### 2.6. Is Software Reuse Desirable?

So far a fairly optimistic picture of software development has been presented. All the reasons why software reuse has not occurred have been accompanied by methods or avenues of research which can be seen to overcome the obstacles. The possibility still exists that software reuse will not achieve what it promises. In this section the case is presented against software reuse.

**Greater time is needed to develop most reusable software than is saved in its reuse.** Only 10% to 20% of the total cost of developing a system is spent writing source code [Arthu83]. Much of the effort involved in developing software applications goes into the testing, debugging and training phases. Software reuse does not have the same time saving in these areas as it does in the writing of the source code.

**It is necessary to sacrifice elegance in order to reuse software.** Software pieced together will never provide seamless, elegant and efficient software. There will always be more redundancy, joins, and complexity in software reuse. Even
though using a framework may simplify additional code written for an application, the overall software package is more complex with more redundancy than an equivalent product written well from scratch.

*Technology advances with time.* Old software is bad. It lacks the knowledge of the problem domain that has been discovered since it was written. It also lacks the programming techniques, procedures and standards that have been adopted since it was written. Reuse perpetuates bad and inappropriate coding techniques and design.

*Programs are not developed from scratch.* Although there may be many hundreds of accounting programs already written, new ones are written from scratch infrequently. Just because there are existing programs with functional and design commonality, doesn’t mean that there will be enough new ones to justify developing a framework for reuse.

*Software development is not a shared activity.* Cox divides tasks into intangible or solitary activities and tangible or cooperative activities [Cox89]. Examples of solitary activities are solving an equation, writing a play, or inventing a joke. Cooperative activities include building houses, running a business, or buying groceries.

This chapter began with an analogy of the software industry to the electrical engineering and building industries. These two industries involve shared activities. The division of tasks is simple, and the construction is methodical.

Hoare [Hoare82] maintains that writing computer programs is a mathematical activity. Programs can be “derived from their specifications through mathematical insight, calculation and proof”.
Writing software may be an intrinsically creative activity, which cannot be shared. Since software reuse involves splitting the development of a application into parts completed by different users, it may not be well suited to software development.

2.7. Object Oriented Programming

Object oriented programming is a design methodology involving construction of a software system from a structured collection of objects [Meyer87]. An object corresponds to an entity within the problem domain of an application. A programming language can be considered object oriented if it supports the concepts of polymorphism, encapsulation, and inheritance [Pinso88]. An introduction to object oriented concepts and programming is found in [Hende92].

Software reuse is one the key benefits of object oriented software development [Wasse91, Snyde86, Nino90, Tracz88a]. Lewis shows empirically that reusing software in an object oriented programming environment offers greater productivity than reusing software in a procedural programming environment. He gave student groups different resources to complete a software project. The students with the object oriented programming language and code to reuse were most productive [Lewis91].

All of the three features of object oriented programming languages, encapsulation, polymorphism and inheritance, facilitate reuse and the expression of analysis and design information.
2.7.1. Classes and Objects

A class is a grouping of related methods and data in an object oriented programming language. A single class may have many object instances. A class may define many objects in a problem domain. Many clown fish in an aquarium system would be modelled by many clown fish objects in an object oriented programming system. The object instances all have their own positions within the aquarium, and their own distinctive markings. All the behaviours and important attributes of a clown fish are defined by a single class clownFish.

2.7.2. Abstraction and Encapsulation

Abstraction and encapsulation are complementary concepts [Booch91]. Abstraction involves expressing an object in terms of its external interface and encapsulation hides the details of its implementation.

Encapsulation allows modular software, and the building of software components. Each software component groups related functions and data. Reusability is enhanced because the user only needs to understand the object's interface and not its implementation.

The basis of encapsulation in object oriented programming languages like Smalltalk, C++ and Eiffel is a class. Each class has a public interface consisting of methods that can be accessed externally.

Encapsulation allows a module to be developed independently from the applications in which it will be used, by providing a division between the software for reuse and the application. This is essential for software reuse.

An example of using encapsulation as the basis for software reuse is found in the Integrated Toolkit for Highly Advanced Computer Applications (ITHACA)
project. ITHACA is an Esprit project to develop a “4th generation” object oriented system for use in an industrial environment. It is a collaborative project between several European universities and industry. ITHACA classes have standard encapsulations to enable their reuse. Applications can be generated by combining the classes using a scripting language [Niers91].

2.7.3. Polymorphism and Overloading

Polymorphism is used to refer to several features of object oriented programming. Here it is used to refer to the late binding of functions or overloading, where an object oriented language compiler or interpreter can call a different method depending on the types and number of arguments to the method.

Foote [Foote91] says that polymorphism’s contribution to code reusability is that it increases the likelihood that related objects will operate correctly in a variety of different contexts.

2.8. Inheritance

Inheritance involves a hierarchical structure of classes. A parent class in the hierarchy is called a base class, and a child class is called the derived class. A derived class inherits some or all of the properties and behaviours of its base class or classes, and then can add to or change them. A base class can also specify that descendants have a certain property or behaviour without completely defining its value. For example, a base class of type mammal could define that its descendants will have a behaviour “move”, without defining how they will move. These undefined properties and behaviours are called deferred.
2.8.1. Functions of Inheritance

Inheritance has three functions within an object oriented programming language.

Modelling a specialisation relationship between entities in a problem domain. A zoological classification system is a problem domain in which specialisation is used. A chimpanzee can be classified as a primate, which is a mammal, which is an animal, which is a living thing. In this inheritance hierarchy the chimpanzee inherits the properties of the primate and so on. Any properties of a living thing are also properties of a chimpanzee [Borni87]. Chimpanzee, primate, and mammal are classes in an object oriented programming system. Class chimpanzee inherits from primate which inherits from mammal.

Standardising the properties and access methods of a class. An abstract base class can define certain method names, and leave it to the derived class to provide the methods themselves. These programming techniques are used heavily in the Eiffel language to specify the standard interfaces of language types. Eiffel defines a base class comparable which defines a protocol for communication with all classes that inherit from comparable. The base class comparable defines deferred methods for greater than and less than operations. By inheriting from the comparable class, and implementing the deferred methods, the derived class has a well known interface. Other protocol classes defined include linkable (in a list), viewable, and storable. The proposed ANSI standard for Object Pascal defines protocol classes which consist in their entirety of deferred properties and behaviours.

Pure code reuse. Existing code can be adapted and reused without the need to alter it. The new class can inherit from the existing class, and only the differences between the existing class and the new class need be defined.
2.8.2. Inheritance and Frames

A frame is a generic specification of an object as developed in the area of Artificial Intelligence. The properties of the object are represented in the frame by slots, and each slot takes on a default value. A new object is specified by the slots that are different from the generic object.

Frames embody the idea of thinking about new objects in terms of their differences with a standard or preexisting object, and grouping their characteristic properties into slots. Generic objects representing an entity are constructed. The properties of the object that are likely to change, and the manner in which they are likely to change is noted within the frame. When a new situation is encountered, instead of interpreting it from scratch, a substantial structure can be selected, and the new situation can be fitted to it, noting the differences [Minsk75].

Frames have similarities with inheritance. A derived class notes the difference from a base class in an inheritance hierarchy. A new situation is described by its differences from a generic situation when using frames.

2.9. Classifying Types of Inheritance

2.9.1. Existing Classifications

Sakkinen [Sakki89] divides inheritance into incidental inheritance and essential inheritance. Sakkinen quotes Ian Holland as saying “Incidental inheritance seems to appear as a result of software engineering and program design. Essential inheritance occurs as a result of domain analysis and system design.”

Bar-David [BarDa92] divides inheritance into behavioural and implementational. He defines behavioural inheritance as an inheritance relationship where
the behaviour of a derived class is a superset of the behaviour of the base class, and all operations that are permitted on a base class will behave identically on the derived class. The return value of a derived class method is completely determined by the return value of the base class method. To achieve this he restricts behavioural inheritance to mean that a method called in a derived class and a base class, with the same parameters must return the same thing. This means either no methods may be overridden in a derived class, or if they are they still provide the same return values as the corresponding method in the base class. He defines implementational inheritance as all types of inheritance, including behavioural inheritance.

Bar-David proposes the division for the purpose of suggesting that behavioural inheritance be used to encourage software reuse, encapsulation and understanding. He claims that there is no need for a software developer to view the source code, because the behaviour of the derived class is defined by the behaviour of the base class.

Raj and Levy [Raj89] distinguish implementational inheritance from subtyping. Implementation inheritance is where inheritance is being used to access the functionality of a base class. Subtyping is used to express a subtype relationship between two classes.

Snyder [Snyde86] uses the term “inheritance” only to refer to the inheritance of implementation, and uses the term “subtyping” when inheritance is being used for subtyping.

A different definition for implementational inheritance is given by Waldo [Waldo91] as being inheritance that is characterised by classes where some of the functions of a derived class call on the functions of the base class for some of their
functionality i.e where a base classes defines more than just a protocol for the derived classes.

2.9.2. Proposed Classification for Inheritance

2.9.2.1. Need for New Classification

The existing classifications do not classify inheritance from the perspective of software reuse.

Bar-David’s model is too strict a model for a software producer to impose on a software consumer. The stricter the restrictions placed on inheritance, the easier it will be to make assumptions about the behaviour of a derived class, and the more design information can be seen from the inheritance relationships within an application. It is however questionable whether a class complying with Bar-David’s model of behavioural inheritance would be sufficiently reusable to justify the time spent designing it.

Subtyping need not correspond to a subtype relationship in the problem domain. The subtyping may only exist within the application.

2.9.2.2. The Proposed Classification

A classification of inheritance into *a-kind-of* and *compositional* is proposed.

A-kind-of inheritance is defined informally as inheritance that corresponds to an *a kind of* relationship in the problem domain being modelled. A-kind-of inheritance is the result of the analysis of the problem domain. The need for this informal definition will be demonstrated later. This type of inheritance would be called essential by Sakkinen’s division, but it is a specific form of essential
inheritance that results from a particular specialisation relationship in the problem domain. A-kind-of inheritance is where the derived class has as an a-kind of relationship with its base class in the analysis of the problem domain. The class salmon and the class catfish have an a kind of relation with class fish.

Compositional inheritance is the opportunistic inheritance to reuse source code from a class, where there is not an a-kind-of relationship between the classes in the problem domain. An alternative to compositional inheritance is often a "uses" relationship or an aggregation relationship in an object oriented program. Meyer presents the choice of whether to reuse code by aggregation or inheritance as whether it is better to inherit or to buy. Inheritance gives a simpler and more efficient interface but works against encapsulation. A derived class can access the procedures and data of its base class that are inaccessible to classes "using" it.

A class relationship like is a part of is also an example of compositional inheritance. Its use does not express a subtype, or an a-kind-of relationship.

Compositional inheritance involves composing a new class from a class or classes that have properties in common with the new class. Class shark could be derived from an existing base class whale. The property warm_blooded could be overridden or eliminated, and the property cold_blooded added. A shark is not a kind of whale, but they have some properties in common.

Compositional inheritance can also be used to take advantage of one or a few properties of a class. A sugar glider could be derived from a bird. Both being flying animals they have some properties in common. A sugar glider is not a kind of bird.

As with the terms behavioural and implementational, the terms compositional and a-kind-of have varied usage in the literature. Composition can be used
to refer to the composition of two classes using multiple inheritance.

2.10. Roles of Inheritance

Deciding whether to use a-kind-of inheritance or compositional inheritance is a trade off between structure and flexibility. Using a-kind-of inheritance guarantees consistency for classes in the hierarchy. A derived class is a subtype of the base class. Using a-kind-of inheritance lacks flexibility. A class can only be reused where every property of the base class is required and is appropriate, otherwise a separate hierarchy must be created.

Meyer [Meyer92] argues that to make an object oriented programming system into a practical programming system an inheritance method should allow the derived class writer control over what parts of it’s base class it inherits i.e to use compositional inheritance. He concedes that the base class could have been designed to eliminate the need for compositional inheritance, but that it allows a derived class programmer to correct an oversight in the design of a base class, and to reuse any part of the base class that is useful. The derived class writer is no longer using all of the design. This compositional inheritance deliberately reuses less design than does a-kind-of inheritance.

There are different situations in which the different types of inheritance can be used.

2.10.1. Roles of Compositional Inheritance

Compositional inheritance can be used as a replacement for cutting and pasting code. It is a replacement for copying a piece of existing software (from a colleague, or from a previous work) and making the necessary changes to make it suit
a new specification.

Compositional inheritance can encapsulate foreign code and make it usable in another environment. Old or incorrect versions of classes can be compositionally inherited, corrected and made available for reuse, possibly with a-kind-of inheritance.

2.10.2. Roles of A-Kind-Of Inheritance

A-kind-of inheritance expresses design information because it reflects the analysis of the problem domain.

A-kind-of inheritance allows division of programming responsibility. A-kind-of inheritance moves design decisions up the inheritance hierarchy, compositional inheritance moves them down. This technique can be used to reflect where the most design knowledge about an application is. In software project management the skills of the programmers on a team may be diverse. Tracz [Tracz88b] makes the observation that programmers can be divided into PRO’S and WIMPS. PRO’s are Perceivable Reliable, Omnipotent Software engineers. WIMPS are Well Intensioned Mediocre Programmers. In a team of programmers the Wimps are best employed where less design knowledge is needed. A-kind-of inheritance factors the design, and fundamental coding tasks in the base classes for the PRO’s, and leaves the writing of the derived classes, with less design to the WIMPS.

A-kind-of inheritance is useful with classes designed for reuse. To get a set of base classes reusable with a-kind-of inheritance is an iterative process, and takes additional time. To get a return for the additional time it is necessary to reuse the software many times. Unless the software is for reuse in many environments, it may not be efficient to spend the time iterating to create a base class
reusable by a-kind-of inheritance. Existing code cannot be reused if the software producer failed to anticipate the circumstances of the software consumer.

2.10.3. Using A-Kind-Of and Compositional Inheritance

Compositional inheritance and a-kind-of inheritance should be distinguished within a project design, and (if supported) within the resulting object oriented programming language. If an overall design is made first, it is possible to express the inheritance relationships which correspond to the entities in the problem domain using a-kind-of inheritance. It also will enable an overview of the design information modelled in the inheritance relationships in a system to be generated automatically with knowledge of only the syntax of the language.

2.11. Problems with Inheritance

When using inheritance, a particular method invoked could be at any level in the inheritance hierarchy. Tracing a running program can involve going up and down through the hierarchy. This can be confusing to follow, and can make it difficult to understand which class an error occurs in. This has been called the Yoyo problem [Taenz89].

2.11.1. Problems with A-Kind-Of Inheritance

A-kind-of inheritance can lead to having too many classes for reuse, as every class that is not strictly a kind of another requires a different base class. The base class programmer must completely anticipate the problem domain in which the software will be reused.
A kind-of-inheritance means having more disjoint classes and more duplicated source code as every class that is not really a kind of another class must have a separate hierarchy.

That A is a kind of B does not guarantee that class A is derivable from class B. This is because assumptions are made by every base class programmer that impose restrictions on what can be done in the derived class. Base class restrictions are essential to ensure a-kind-of inheritance, but there is no method in existing programming languages of allowing for exceptions [Carro92].

Solutions to these problems are proposed later.

2.11.2. Problems with Compositional Inheritance

Compositional inheritance can cause errors by using code in a way the code was not intended. The code may not have been tested in the context used, and is therefore more likely to fail.

Compositional inheritance enables source code reuse. It also allows a class to be reused in a way that the base class designer did not anticipate. A base class can be reused in many diverse contexts.

Compositional inheritance “freezes” the base class. A change cannot be made with no impact on the derived classes. A-kind-of inheritance allows a base class to be changed without impact on the derived classes as long at the change made is consistent with the modelled system. If a method “swimming_technique” is defined in class “salmon”, and inherited in class “atlantic_salmon”, then new research at salmon dams can be incorporated into the salmon base class, without impact on the derived class. If a very basic swimming technique method was inherited compositionally into class “platypus” then it may
become inappropriate when the method changes.

2.12. Programming Languages and Inheritance

There are features of programming languages that are suitable for compositional inheritance, and other suitable for a-kind-of inheritance. For example an a-kind-of inheritance feature should allow strict encapsulation of the parent class. A compositional inheritance feature would allow more flexibility to the child class.

C++ allows a derived class to have either a private base class, or a public base class. The use of a private base class is intended to model inheritance only for the purpose of code reuse. The use of a public base class declares the derived class as a subtype of the base class, and allows use of the languages subtyping features and type polymorphism.

Eiffel allows selective inheritance of various parts of the base class into a derived class. It allows methods in a base class to be eliminated or replaced by other methods. These features offer greater support for compositional inheritance.

The Sather language [Czype93] has one of its design purposes to distinguish inheritance used for subtyping, from inheritance used for code reuse.

2.13. A-Kind-Of Inheritance and Models

For inheritance to be useful for modelling a system, the model must be able to be expressed as hierarchies of classes. The hierarchy is arranged with most general classes at the top, and more specific classes at the bottom. Derived classes inherit all the properties of their base classes, and add additional properties. For example the species Atlantic Salmon inherits all the properties of the order Salmon, which in turn inherits all the properties of phylum Fish.
There may be many organisations for a particular system. Different analyses can result in different inheritance relationships being identified in a problem domain. Fish could be classified according to their colour, their shape, or their migratory habits. There is no guarantee of a correct analysis, and an inappropriate hierarchy may result from an intuitive organization.

2.14. Problems with Models

![Aquarium System Diagram]

Fig 2.1 - An Aquarium System

Fig 2.1 shows part of an inheritance hierarchy in an aquarium system. Although the properties of fish seem intuitively reasonable, none of those listed apply to all fish. An Australian Lungfish has no gills, a Porcupine fish has no scales, and a Grey Nurse doesn’t lay eggs. Other inconsistencies are lower in the inheritance hierarchy. In order to enable a-kind-of inheritance for general reuse the class fish would have to have no properties at all. This is not a very useful
class to derive from. It is very difficult to find a property that all fish would have without exception.

A-kind of inheritance requires that the base class writer create a perfect model. A perfect model that allows for every exception ends up very generalized and containing very little design information. The alternative is using compositional inheritance, and losing the subtype relationship, and other advantages of a-kind-of inheritance. Object oriented programming languages do not have the flexibility to express options and exceptions in a base class.

The problem could be fixed by defining less properties in the base class, and moving the properties further from the root of the hierarchy. Some useful information is lost to other users of the base class. The problem may be fixed by giving a more specific name to the base class, in order to limit the situations in which contexts in which it is appropriate to reuse the class. This is imposing further restrictions on the derived class programmer. Object oriented programming languages provide limited mechanisms to restrict the context in which a class may be reused.

A system base on the real world never fits perfectly into hierarchies [Pirsi76]. By the time the information in the top level of the hierarchy has been made general enough to cope with all possible exceptions we are either left with no design information or forced to use one possible design.

Whatever the cause of the problem, it is clear that even though the design of useful reusable base classes is an iterative process, and it takes many design iterations to get a good solution, that the solution will never be perfect. This cannot be regarded as the fault of the analyst. An environment for design and implementation of object oriented systems must accommodate this.
A complete object oriented program is an encoded representation of a model of a system. The model of the system is not always produced by the analysis phase of software construction, but may have been based on preexisting classifications. It would be difficult to eliminate errors in this area without extending the analysis time significantly. Therefore there is always the possibility of an inappropriate hierarchy resulting from an incorrect system model.

2.15. Multiple Inheritance

Multiple inheritance is where a derived class inherits from more than one base class. It is supported in a subset of object oriented languages. There is debate over whether multiple inheritance forms part of a real world modelling process. Wirfs-Brock et al [Wirfs90] claim that multiple inheritance is a complex mechanism which is required for modelling complex real world situations.

If the model is being constructed using a-kind-of inheritance multiple inheritance rarely provides a real world model. None of the common trivial examples given for multiple inheritance in the literature, like sea-plane, fall into the category of a-kind-of inheritance. This can be demonstrated in most cases if a change to the base class consistent with the real world model can force a change in the derived class that is not consistent with the real world model.

Nearly all multiple inheritance is classified as compositonal inheritance, as it is difficult to locate examples in a realistic system where an object can be classified as a-kind-of more than one other classes.

The current debate is presented in two successive issues of Computing Systems [Waldo91, Cargi91]. Cargill argues that no appropriate examples have been given of multiple inheritance to show why it is needed in a programming
language. All the advantages that have been given can be accomplished with single inheritance or aggregation. Waldo, in his rebuttal, argues that multiple inheritance is being used for a different purpose to single inheritance, and is useful in defining the protocol but not the implementation of a class.

2.16. Frameworks and Class Libraries

Frameworks are libraries of classes designed for reuse. An application built from the framework by creating new derived classes, configuring classes in the framework together, and modifying examples of using the framework. A framework consists of classes that collaborate and impose a model that the software consumer must adapt to [Johns93]. A user of a framework will usually adopt it as a base for their entire application.

Class libraries are groups of reusable classes, but are not necessary collaborative, and more often the classes have independent utility.

Examples of frameworks are Interviews, MacAPP, ET++, and Choices. Interviews, MacAPP and ET++ are written in C++ for the development of graphical user interfaces. Choices is a framework for presenting a consistent virtual memory model for an operating system [Johns91]. Examples of class libraries, include the class libraries shipped with most Smalltalk systems, the libraries shipped with C++ compilers and freely available class libraries like NIHCL from the American National Institute of Health.

Both frameworks and class libraries are intended to be reused many times and can justify additional effort to be put into iteration and documentation.

Graphical user interface (GUI) frameworks are the most common frameworks, possibly because the problems associated with modelling real world
situations do not exist in these environments as most GUI frameworks regulate the look and feel, and types of objects that can form part of the interface. The inheritance hierarchy therefore contains few exceptions.

When reusing a framework it provides common defaults for certain properties. One aim of a framework should be to specify as much of the design and properties as possible without causing the software consumer to override frequently. A framework designer must anticipate the use of the framework, and be sufficiently general to allow easy development of the application, but specific enough to express design and analysis information.

Part of the iteration involved in developing a framework is determining the needs of the software consumer, and to avoid the need for adapting the base class.

2.17. Summary

Object oriented classes may fulfill the need McIlroy saw for reusable software, but reusing software using inheritance cannot be regarded as use of black box components that he described. Inheritance can be used as nothing more than component reuse. A class can be regarded as a selection of routines that can be selected and used by inheritance. Inheritance can also enable reuse of design information. Reusing this design requires knowing more than just the interface and protocol of a class, but should not expose the software consumer to the detail of the implementation. The three factors enabling software reuse identified here (that it must work, be locatable and understandable) should be present in any system for documentation of inheritance. Appropriate documentation is required, and the forms of documentation available are discussed in chapter 3.
3. Documenting Inheritance

3.1. Documentation for Reuse

Documentation, in the context of software reuse, is the means of passing information from software producer to software consumer. Research into documentation for reuse spans the areas of computer science, computer-human interaction, language, and psychology.

3.2. Requirements of Documentation to Enhance Reuse

To enhance software reuse, documentation should form part of a programming environment that enables reuse. Documentation can assist in each of the programming environment features identified earlier as enabling software reuse. The software for reuse must work, be locatable and understandable.

3.2.1. The Software for Reuse must Work

The documentation for a reusable class should specify its functionality and the range of situations in which it will work. Software can only be designed and tested for a specific range of situations and these situations should be defined in the documentation.
The documentation must be responsible for what Wade [Wade93] called asset certification at OOPSLA '93. Reusable software is an asset. Certification involves testing and accurately documenting its functionality and the areas in which it was made to function. An electrical engineering component is delivered with documentation of its functionality and specifications of conditions under which it will function correctly. Asset certification is the equivalent for reusable software components.

3.2.2. The Software must be Understandable

The software documentation must be understandable for the software itself to be understandable. The expression and presentation of the documentation must be clear to the software consumer. It must be presented in such a way to communicate the ideas effectively. The software producer must be able to produce the clear documentation within a reasonable amount of time. The documentation should take advantage of shared specialised knowledge between the software producer and the consumer.

3.2.3. The Software must be Locatable

The documentation must assist the software consumer in the matching process between the software consumer's needs and the software producer's design.

The location of software involves the location of a class library or framework, and then the location of a class within the library or framework. The documentation must facilitate the location of the appropriate class for reuse. It must also identify when a class chosen by the software consumer is not, in fact, appropriate for reuse in a particular situation.
3.3. Types of Documentation

What follows is an examination of the current methods of documentation available to a software producer to document software to be reused by inheritance, grouped according to the style of the documentation.

3.4. No Explicit Documentation

Much of the documentation used by a software consumer when interpreting object oriented code to be reused does not come from deliberate attempts at documentation. Instead it comes from studying other peoples code, looking at the source code being reused, and obeying whatever restrictions the software producer has placed on his code using the language in which it is written.

3.4.1. Source Code

The parts of the software source code interpreted by the language compiler or interpreter are the definitive documentation of the functionality of a program. Brooks [Brook75] argues that the reusable source code must be available for the software consumer, otherwise the consumer cannot be sure that the documentation is correct. The source code shows the methods available, the relationship between classes, what methods are called and from where.

Programming languages vary in their ability to express documentation within the program. Features of a programming language such as allowing descriptive variables, method and class names enhance the programmers ability to use the source code for documentation. Object oriented programming languages allow more expression of relationships between data and methods than do procedural programming languages by grouping of functions and data into classes and
allowing the expression of relationships between classes.

Well written and designed source code contains a large amount of information about the application, framework or class library of which it is a part, and the problem domain for which is is constructed.

The programming style of the software producer, and the familiarity to the software consumer with that style, can greatly affect the understandability of the code. Style guides exists for most popular programming languages containing advice on programming style.

Ways of extracting the design from the source code are discussed by Biggerstaff in [Bigge87]. Techniques in reverse engineering source code extract design from the program structure, the functionality or processing done by the code parts, and the data and control flow[Byrne91].

Most programming languages have provision for programmers to write free text comments in the source code. The comments in the source code are usually written by the software producer at the same time as the other source code. This makes them likely to be accurate to the author’s intention (although the actual code may contain errors that make it behave differently). They may express the author’s intended context for using the code.

Comments can also express preconditions and postconditions of a function for example Figure 3.1 shows a typical C function declaration with documentation.
Fig 3.1 Documentation in the source code

The source code in Fig 3.1 shows that the function takes a value as a parameter, which is checked by the type-checking system to be a double, and the comment states that it must be positive and non-zero.

In reusing code by inheritance the base class source code can be examined to see its public methods, what classes can be accessed, the class and function names, and any information about the author's intention contained in the associated comments.

3.5. Copying Source Code Examples

Examples exist for the use of most software. Software not designed for reuse has already been used in the application the software was originally written for. These examples document the method for reusing the software. Other software for reuse is distributed with small examples showing how to reuse the software that can be adapted; for example the frameworks Interviews [Linto89] and ET++ [Andre89]. This example code is intended to be edited and adapted to another purpose.

In the case of documenting inheritance, source code that inherits from a class can be used as an example of how to inherit from a class, what properties can
be overridden, and how they can be used.

3.6. Manuals – Written Documentation

Manuals consist of written words, diagrams, indexes and cross-references.

Manuals are usually set out in either guide or reference form. The guide form can describe all the functions of the software, and can give examples of its use. The reference form indexes the features and functionality of the software.

Reference manuals and guides can be kept on line to allow electronic searches of their indexes, and online display of their contents. This facilitates the location of software for reuse in a software library.

In using inheritance for reuse a manual can outline the features of a class to be inherited. Diagrams can show class relationships and software design. Online searches can assist in the location of an appropriate class for reuse.

The techniques of how to make manuals understandable and readable are areas of research in themselves. An overview of this research is presented in [Weiss90].

3.7. Hypertext Documentation

Hypertext is non-sequential writing [Nelso74]. The writer links texts together to allow a reader to follow the links through many areas of the documentation. The reader can structure the way they read through the documentation. A hypertext system allows the user to see as little or as much documentation as they desire, and can offer additional information easily in areas selected by the user [Arthu83]. The links in a hypertext system (hyperlinks) can also be used to link the source code to the documentation. Hyperlinks can connect parts of the source code to
their associated documentation and examples.

Sametinger et al [Samet92] propose a hypertext system that allows the documentation to be easily accessed with the source code. Their system supports the inclusion of written and diagrammatic design of the source code. It allows a way to index and classify text of requirements statements, design diagrams, plans and schedules etc. Since all this information is accessible via hypertext, it allows the programmer to select the level of documentation required.

Garg and Scacchi [Garg90] propose a hypertext system to manage all information about a software project in a hypertext database. The source code of the project is a hypertext document in the database along with the detailed design specification, the functional specification, and the user and system maintenance guides. A user in the maintenance guide can follow hyperlinks to the relevant sections of the source code, and the design document. Creating hyperlinks while programming and designing the system ensures a relationship between all the software life cycle documents.

3.8. Language Features

Some programming languages have features to allow the compiler or interpreter to check the consistency of the software being developed. These checks can be of use to the software consumer in understanding how the software should be reused.

3.8.1. Type Checking

Type checking is used as a constraint mechanism in procedural and object oriented programming languages. The compiler or interpreter can verify that methods are called with the correct number and type of parameters. For example a software
producer can specify that a method requires a parameter of type colour. Colour is defined to be a type consisting of all the colours that the method can accept as a parameter. The compiler or interpreter can ensure that this function is used correctly by the software consumer.

The support for type checking varies between languages. The Viola language is typeless. The implementation of the language uses several types for data storage, and has built in methods to convert between them. The C++ language allows (weak) type checking to be performed at compile time [Joyne92]. Eiffel can type check all but certain constructs at compile time [Meyer92]. Smalltalk, being an interpreted language, performs only run time type checking [Pinso88]. Extensions to Smalltalk to allow strong static type checking are proposed in [Brach93].

3.8.2. Assertions

Assertions can appear as preconditions and postconditions, class invariants, check instructions and loop invariants [Meyer92]. Assertions allow the software producer to constrain value parameters at run time, usually in the coding, testing and debugging phases of software development before an application has been put into production use.

A value can be constrained at any place within the code using an assertion. Preconditions and postconditions are specialised cases of assertions which constrain the before and after values when calling a method.

Assertions document software by expressing the specification of a software component, and defining what it does [Meyer92]. A software consumer must ensure that the assertions about values are met when calls to the software
producers code are made. For example, a method that writes a value to a position on the screen, could make an assertion that the values specifying the screen position are actually on the screen.

In inheritance assertions made in methods in the base class that are inherited are assertions in the derived class. A method in a derived class which overrides a method in its base class may be expected to be constrained in the same way as the method in the base class.

3.8.3. General Constraints

The role of general constraints is covered in the next chapter. An example of a system to support general constraints is CCEL. CCEL [Duby92]. is an extension to C++ which allows constraints to be placed on C++ designs and implementation. CCEL can place constraints on design, implementation and style. The examples given by the authors are constraints in the redefinition of a method in a derived class. A constraint can ensure that if a class declares a pointer method in C++ that it also declares an assignment operator. A stylistic constraint is that a class name must begin with an upper case letter. CCEL works by having a program parse the source code for correctness.

3.9. Visual Languages and Software Visualisation

Visual programming is different from program visualisation. In program visualisation, the program is specified in a non-visual way, but other tools and techniques are employed to get a visual representation of it. In visual programming the graphics are a representation of the program [Myers89].
3.9.1. Visual Languages

Visual programming languages represent objects visually to a programmer that do not necessarily have a visual representation [Chang87]. Presenting these objects visually, is helpful to the user. Some visual languages also support visual interaction.

3.9.2. Software Visualisation

Software visualisation is concerned with providing a good high level of abstraction to present a program. Program visualisation depends on understanding the plans and intentions in the source code. A good AI system is required to recognise the abstractions, and display the behaviour of the code [Eisen92]. Software visualisation systems use visual media to enhance one programmer's understanding of another's work [Domin92]. Software visualisation systems involve either a static analysis of program source code, or dynamically tracing the program execution and highlighting events during the program execution.

In object oriented programming static analysis can reveal the relationships between classes in the software. Attempts have been made to extract design information from the program source code, in the way of which functions call which others, and which classes have inheritance relationships.

An example of a static analysis tool called the C++ Information Abstractor (CIA++) is described by Grass in [Grass92]. It is a system designed to construct system design from C++ source code; a process Grass calls software archaeology. It involves parsing the software to extract information including the use of variables, flow of control, and class relationships. The static analysis tools can only examine static syntactic and semantic relationships. They only provide limited
support for the analysis of dynamic behaviour of programs. Other tools are required for this analysis.

The simplest dynamic visualisation tools are source level debuggers. In both line based and graphical form, they are used by software engineers to trace and monitor program execution. \textit{dbx} is a source level debugger which allows a programmer to step through the execution of software, and to stop at any point. At any point data can be examined and changed. Other tools, like profilers, reveal information about the events in the execution of a program, for example the order and amount of function calls.

Cheng and Gray [Cheng92] propose a program visualisation tool designed for visualising object oriented frameworks. A similar system is proposed by Pauw et al [Pauw93]. The tool visually represents events in the life of objects. Their creation, destruction and communication events. The events to be monitored can be limited by the software consumer. The system involves adding a preprocessing stage to program compilation, that inserts code in method calls, constructors, and destructors to log events at run time.

Brown [Brown88] outlines a method of software visualisation in which a programmer annotates the source code to highlight the important events in the algorithm. He calls such an event a \textit{code atom}, an item of code that can be thought of as a single segment by the software user. From the perspective of understanding software for reuse, the software consumer must arrange the display and decide which algorithms to run. The software producer must annotate the program with markers, to note interesting events, or algorithm events, indicating that they will be of interest when running the program.
3.10. Artificial Intelligence

An aim in software artificial intelligence (AI) research is to have automatic programming, where a program functionality is specified in English or a high level specification language. Mid-term goals are programming by example, simple automated programming, and building a programmers apprentice.

The AI approaches require knowledge of the program domain, or shared knowledge between the programmer and the AI program.

AI techniques are not currently being used to document the inheritance relationship, but the ideas on building up knowledge of the problem domain are useful in that area.

3.10.1. The Programmer’s Apprentice

The concept of a programmers apprentice was started at MIT under the direction of Winston, Rich and Waters [Hunt86]. The aim is to provide a virtual assistant to keep track of details, and assist with the simple parts of the software process. It has many features of a programming environment, like being able to create program skeletons, and automatically testing revised programs.

The programmers apprentice relies on having knowledge of the programming environment and application in order to assist. This knowledge is represented within the programmers apprentice in structures called *clichés*. Clichés are similar to frames in the way that they represent knowledge.

3.10.2. Automatic Programming

An automatic programming system guides a naive programmer to describe a problem using natural terms and concepts of a domain, with informality, and omission
of details. An automatic programming system must be domain specific in order to interact effectively with the user and to draw upon domain specific knowledge during the implementation process [Barst85].

3.11. Documentation for Specialisation

Documentation for specialisation comes from the work of Snyder [Snyde86]. He maintains that there should be a well defined interface for code reused by inheritance, the same as there is a well defined interface for reusing entire classes, or reuse by aggregation.

3.11.1. Type Checking for Specialisation

Lamping [Lampi93] notes that a class has two external interfaces that should be consistent and documented. One is the client interface to users of the class. The other is a specialisation interface to those who use the class to build subclasses by inheritance.

He proposes a system that imposes type checks on the inheriting class. Any methods overridden in a derived class must be consistent with the methods in the base class. It does this by checking if methods being overridden are called by methods not being overridden. It uses this as a basis for type checking the inheritance. Classes can indicate how much of their state and functionality is relied upon by other methods.

The type correctness of the method inherited into the new class can be verified directly. Each inherited method is checked to make sure that the new class, which may not follow the specialisation interface protocol of the support-class, still adheres to the protocol relied on by the inherited method. In other words, a
user can be free to change a method not called by another method that is inherited and unchanged.

A protocol specifies what the interface to a class is. Not only what methods are available to call, but when and how they are available to call. Part of the protocol for a remove from set operation would ensure that the article was part of the set. As an extension of that theory, classes define the implementation of a protocol, and types describe formally the protocol implemented by a class.

3.12. Programming Environments

Attempts to merge documentation into software development environments are becoming common in object oriented programming. Documentation is presented with the software in a manner appropriate to the programmer interface with the software being reused.

3.12.1. Class Browser

A class browser forms part of the programming environments for some class based languages. They are designed as a method for allowing the software consumer to locate a class for their purpose from a library of classes or a framework.

They provide an interface for scanning the class names. The classes can be presented hierarchically based on the inheritance structure, and a text or graphical description of the class. The class names can also be presented with written documentation on the functionality and purpose of the class.
3.12.2. Cognitive Documentation

Fischer et al [Fisch88] describes two tools, *Codefinder* and *Explainer* for locating software components and getting explanations of program examples.

Codefinder takes a cognitive approach to location of software. It searches a hierarchical classification of terms describing code available for reuse. Links from this classification scheme back to the code for reuse, enable the Codefinder system to find code that is not matched exactly by a keyword search.

Explainer contains examples of the use of the code stored for reuse. It can “explain” the code by giving an overview description of the class, and the option for a more detailed explanation. It can give an example of how to reuse the class, highlighting parts of the class to illustrate the example.

3.13. Problems with Current Methods of Documentation

The problems of the current methods of inheritance are outlined below, classified by their conflicts with the programming factors for enabling reuse.

3.13.1. The Software for Reuse Must Work

*Encapsulation is not preserved.* Encapsulation is required for reuse, but inheritance in an object oriented language works against encapsulation. Any method of documentation which requires access to the source code by the software consumer compromises encapsulation. The debate between Brooks and Parnas in [Brook75] reflects a dilemma with encapsulation. Parnas argues that the programmer should be shielded rather than exposed to the details of the construction of a system. Brooks disagrees, arguing that errors in the interface cannot be discovered without breaking the encapsulation.
Encapsulation is hard to obtain using inheritance. Inheritance exposes the class to the software consumer. Snyder [Snyde86] points out that it is important for a class to define an external interface to its child class.

*The interface is not specified precisely.* Type checking cannot encode all the information about what a particular object is to be used for. Several elements may use the same type within a program that were never intended to be assigned to each other. Type checking is not sufficient to ensure correct use of a method or object.

Type checking is by its nature a formal specification of the interface of a class. It is limited in that it cannot capture the nuances of a class interface. Thus they are only an approximation of the interface [Lampi93].

### 3.13.2. The Software for Reuse Must be Understandable

*Complexity.* Having to refer to the source code for complete understanding of a class is complex.

A programmer does not necessarily want to sacrifice efficiency or clarity by using a subset of the language. In any organisation there is a range of programming expertise and styles. It it not desirable to limit the use of the programming language to be understandable to all.

*Language skills.* Technical software producers may not have effective written language skills to communicate information about the software for reuse, however desirable it may be for programmers to have these skills.

*Nobody reads the documentation.* Traditional documentation techniques can be hard to understand and are frequently ignored.
Rettig [Retti91] describes the Apple Macintosh System Software Users Guide as a bestseller that no one reads. Despite being a clear and comprehensive manual, human nature leads to the manual being the documentation of last resort.

*Abstraction* Every abstraction involves emphasising some features of an object being abstracted, and ignoring other features. An abstraction is necessary to get a clear picture of the functionality of an object.

Although visualisation systems attempt to provide an overview of functionality, they are limited by difficulties in getting the appropriate level of abstraction. In a medium sized application, there may be hundreds of events of a particular type. The person manipulating the visualisation system must have sufficient knowledge of the system to select the events, the parts of the code that are of interest, and which properties and behaviours of objects should be abstracted. When using a visualisation system to interpret a framework, it is only possible to visualise the features of the framework that are exercised with a demonstration application.

Papathomas [Papat92] points out the difficulty in providing abstract specifications from an objects implementation. He proposes a language to document the behaviour of objects, so that two objects with different implementation, but the same external behaviour are documented the same way.

*Limitations of Knowledge.* The AI technology is currently limited by the amount of information that can realistically be stored about a software project [Water85].
3.13.3. The Software for Reuse Must Be Locatable

Software matching and the naming problem Prieto-Diaz et al [Priet87] say that reuse is a matching process, and that successful software reuse relies on having a method for matching current requirements with existing code. Minimizing the time taken to find the correct match is an issue in enabling software reuse.

A name is a syntactic entity that denotes an object, and a naming system is a mechanism to answer queries of the form "what object is denoted by this name" [Bowma93]. Bowman classifies the systems used for naming in compilers to be functionally simple naming systems, where a word is used to represent a value, whether it be a stored value or an address of a routine.

Names within programs have become a more important source of documentation in languages which express information and concepts in a name. Some older languages restrict names to short meaningless strings. Most modern computer languages and compiler allow much longer names. This has caused a problem with identification of classes and methods from their names.

The naming problem is caused by the emphasis placed on the name of a method or class in determining its suitability for a particular task. In effect it can be used as the primary documentation for the class. The name cannot contain all the information required for class selection. The name of a class may be abbreviated, inappropriate or even misleading. A class named keypad can have very different functionality depending on its context. The name of a class does not always reveal its functionality.
3.14. Summary

It is the basis of this thesis, that none of the methods of documentation currently used, and detailed in this chapter, are appropriate for software reuse using inheritance; in particular a-kind-of inheritance. They do not provide an environment where the key factors for enabling reuse are present.
This chapter develops a programming methodology and programming environment useful for enhancing software reuse using inheritance.

Many of the documentation features outlined in the chapter 3 are useful in the documentation of inheritance for reuse. Many of the problems associated with them can be eliminated if kept in mind during the design of a methodology and associated programming environment.

4.1. Why is a Methodology Required

*Reuse doesn’t happen by chance.* Top down strategies for requirements analysis, specification and design are unlikely to result in any reusable software being developed [Niers91]. If software is to be reused effectively, it should be designed for reuse. A methodology for system design and programming is required.

*The software for reuse must behave predictably.* Inheritance is a very powerful and complex tool. It can be confusing and disruptive if used in a manner unexpected or not understood by the software consumer. A specific methodology makes the base class more predictable and less confusing.
Reuse by inheritance is different. General object oriented design and programming methodologies do not address the methodology required for reuse by inheritance specifically. Therefore extensions of the existing methodologies are required.

The interface to reusable software must be well defined. Lamping’s argument [Lampi93] for type checking can also be applied to the area of documentation. Documentation systems have focused on the needs of the client interface, and define the functionality at the client interface, that is documenting what methods can be called and their parameters. This documentation says nothing about the internal structure of the classes inheritance interface. Subclasses that inherit from a class are necessarily exposed to the classes internal structure. Therefore the documentation of inheritance must concern itself with different information.

4.2. Methodology

The aim of this programming methodology is to build software for reuse so that the documentation interface to the consumer can be presented simply, and where the software producer can influence the design and structure of the class resulting from inheritance. It is an extension of common object oriented design methodologies for object oriented programming, like those outlined in [Wirfs90] and [Booch90].

This methodology requires greater time for base class development than developing the same code without the methodology. If code is to be reused multiple times more work can be justified on the design and construction of the reusable parts. The time spent may be saved in further reuse.
4.2.1. A-Kind-Of Inheritance

A-kind-of inheritance is the first requirement of the methodology. There is no mathematical definition for a-kind-of inheritance. It simply reflects a real world relationship between the objects in the problem domain being modelled.

A-kind-of inheritance is important to simplify the documentation as it gives the software producer the ability to determine the context in which the software is reused. It is only in a-kind-of inheritance that the software producer can guide and set constraints on the consumer. Using a-kind-of inheritance allows greater potential for what can be done by a base class documenter, as more design decisions are made at the level of the base class writer.

When using a framework or a class library, where the boundary between the software to be reused and the application is well defined, it is reasonable to use compositional inheritance within the framework itself and within the domain specific code. Only a-kind-of inheritance should be used across the boundary between the framework and the application.

4.2.2. Organisation of Behaviours and Properties

The second requirement imposed by the methodology is the organisation of the behaviours and properties of objects into slots. This organisation is similar to the organisation of AI frames. Examples of this organisation are given in the next chapter.

Organising the properties and behaviours of objects into slots enables documentation to assist in the location of a class. A consumer can examine a classes properties and behaviours without compromising the encapsulation of a class.
4.3. Features of the Programming Environment

The aim of the programming environment is to give the software producer the power and flexibility to create for the software consumer relevant, descriptive and accurate documentation. The following features should be in a programming environment to enable this.

4.3.1. Writing the Documentation with the Software

The documentation and the software should be written at the same time. There should be minimum departure from the source code in writing the documentation. The graphics and hypertext systems previously described (Section 3.7) involved the integration of hypertext and graphics based documentation with source code.

This increases the accuracy and the reliability of the documentation, because the software producer never has to rewrite information that is present and obvious from the source code. The fact that the documentation is being written at the same time as the source code, and hopefully by the same person, means that the documentation does not rely on the memory of the producer, or his or her ability to communicate the documentation later to a technical writer.

4.3.2. Storing the Design with the Code

The design can be encapsulated along with the code. A method of achieving this proposed by Cybulski [Cybul92] was noted in the previous chapter.

Reusing source code by inheritance requires that the consumer be aware of the structure of the code. The consumer should not, however, have access to the code itself, as this breaks the encapsulation.
By keeping the software lifecycle documents linked to the source code the
design is more likely to correspond to the actual source code structure. This enables the true structure to be seen by a consumer not viewing the source code
directly.

4.3.3. Consumer Interaction

The documentation created by the software producer should interact with the con-
sumer. Allowing the producer to create interactive documentation gives flexibility
in the way the documentation is presented, and the interface of the reusable soft-
ware to the consumer. It allows the producer to specify constraints at edit time.

4.3.4. Constraints

4.3.4.1. Syntax

The documentation can interact with the consumer to assist in generating the resulting software application. The system operating at edit time can verify the syntax of user input in a similar way to a syntax editor, with a similar impact. Once a consumer is experienced with the class library and the intentions of the software producer it may be an encumbrance. The system can be used for learning a class library in the same way that syntax editing can assist someone learning a programming language.

Syntax constraints can force the software consumer to comply with the syntax of the programming language, or apply some other arbitrary syntax con-
straint, such as all function names should begin with a capital letter.
4.3.4.2. Functional Constraints

Constraints described in the previous chapter can verify by assertions that particular properties are true at particular points within the code.

Functional constraints at edit time can constrain things like what methods can be called, what objects can be created and what properties can be changed. Restrictions like these allow the base class programmer to determine the context in which these operations can be performed.

4.3.5. Hypertext

The design information that is expressed in the documentation system can be left to some extent to the software consumer. Hypertext is an approach that can be used for on-line presentation of documentation. Nielsen [Niels90] and Cybulski [Cybul92] describe a programming environment that allows a way to index and classify design diagrams, source code, plans and schedules using hypertext.

Hypertext documentation can also provide a method of partially documenting a derived class using the documentation of the base class. If the parts of the code in a derived class are inherited unchanged, the relevant links in the source code can be linked to the existing documentation.

4.3.6. Visual Documentation

Visual documentation allows the presentation of static graphics, diagrams, and flowcharts. It also permits dynamic examples of functioning code.
4.3.7. Inheriting Documentation

Documentation should be inherited from the base class to the derived class as well as the source code. Much of the documentation in a base class is useful as a basis for documenting a derived class. The documentation can be refined and added to at each level of inheritance. Improvements and changes to the documentation of the base class flows on to the derived class.

If the documentation for the base class is linked to classes and method, the documentation for the derived methods can be links to the same text. For example if a class is added to a hierarchy, at the most primitive level a hypertext link can be made between the derived class and the base class. On a more sophisticated level the methods of the derived class that are inherited can be linked back to their documentation in the base class.

This methodology allows greater accuracy in the documentation, as the person responsible for writing the code also is responsible for the documentation. The abstraction is also accurate as it corresponds directly to the source code, and the correct parts of the class can be emphasised, as these are also written by the software creator.

4.3.8. Knowledge

Programmer’s assistants, from syntax editors to programmer’s apprentices, have knowledge about the environment in order to assist in the development of the application.

The method preferred here, is rather than building in knowledge to an intelligent assistant, to allow the knowledge to be supplied by the software producer. It should provide language for the expression of this knowledge, and
provide a framework for the programming. It should allow a programmer writing the software for reuse to embed knowledge about the software into a form usable by an assistant.

4.4. Applying the Technology

Weiss [Weiss90] outlines how to do things wrong with computers. Sitting down at a blank screen, with no prompting for what to do. "If you want to do things wrong, make things invisible, and give no hints and no visible results." A system like this puts the entire burden for documentation on the software consumer.

Rettig [Retti91] outlines his solution to the problem of written documentation and manuals. He suggests first writing the software and documentation, and then considering how the documentation can be embedded in the software. In software applications the solution is to develop help screens and an intuitive user interface. The same ability must be given to the software producer developing software for reuse to develop an interface for the software consumer.

4.4.1. Task Oriented Documentation

Task oriented documentation assists the user in accomplishing a task [Retti91] and should be a key feature of an environment for reuse. Chin [Chin91] points out that a good consultant system is not one that just answers users questions, but one that leads them towards the answers. Talking from more knowledge to less knowledge requires intelligence in the user interface.

Task oriented documentation is an attempt to overcome the problems with reference documentation, where the documentation details the functionality of each of the components of a system.
Task orientation is more successful than feature orientation. The documentation should have the aim of completing a task, rather than documenting the software feature by feature. This is due to the fact that learners at every level of experience try to avoid reading. They learn by experimenting and making mistakes [Brock90].

4.4.2. Frameworks and Class Libraries

Frameworks can be large and complex. Interviews [Linto89] is a graphical user interface framework from Stanford University written in C++ for X windows. It contains over a hundred thousand lines of source code. For someone developing an application with Interviews it would be desirable for them to have experience developing with Interviews before, rather than just having read the documentation. The matching process is complex, and remembering all the details of a class to match from the documentation would be difficult.

Experience using a framework speeds the matching process. A consumer having used the framework before knows the context to best use the classes in the framework and how they interact. A programming environment can reduce the learning time by making inappropriate use of a class obvious to the consumer as soon as possible.

The matching process must also consider the relationship between the classes in the framework, so that classes derived from them have the correct relationship. An overview of the interaction between the classes is needed to identify an appropriate match.

As stated in the methodology, a guideline for developing a framework is that no compositional inheritance should cross the division between the framework
and the application code. If compositional inheritance is required, then the framework is not flexible enough, and additional generality or additional choices should be included in the base class. If the class "fish" was being built into a framework, the author may choose to build in more than one "fish" type class, perhaps with smaller categories connected by certain features that the application builder is likely to depend on. These multiple classes may (optionally) be implemented compositionally within the framework as long as the interface to the application is a-kind-of inheritance. This is necessary to preserve the design and authors intention. Alternatively the framework author may restrict the application to only construct certain types of fish (most likely the ones with standard properties).

Frameworks are intended to contain large amounts of knowledge and design information about a system. The consumer of a framework need know nothing about the implementation other than the interface the framework presents.

4.4.3. Programming for WIMP’s, and DUNCE’s

The idea that the tasks in programming a complex application should be divided between programmers on the basis of skill is not new. Baker [Baker72] outlines a methodology for having experienced programmers doing the central design, and core programming, thus leaving the easier programming tasks to clerical programmers. He points out that this allows the experienced programmers design to be used while allowing less experienced programmers to get experience on real applications. He notes that software designed in this way has a natural modularity, at the division between the experienced programmers code and the clerical programmers code.
By using the methodology described above, there is a division between the PROs and the WIMPs. The PRO writes the base classes to be reused by the WIMPs. The design and structure is determined by the PRO.

Dunce means Untrained, with No Computer Experience. The programming environment features allow the PRO to completely structure the interface to the DUNCE. The PRO can make it as simple as necessary applying technology developed for simple user interfaces.

A documentation system verifying the syntax of consumer input may have a similar impact. Programmers inexperienced with the framework or class library may be able to use it more easily. Once fully experienced with the class library and the intentions of the software producer verifying the syntax of input may be an encumbrance. This may not be the case if the software producer uses hypertext to reduce the time taken for an expert programmer to derive a class. The system can be used for learning a class library the same way that syntax editing can assist someone learning a programming language.

4.4.4. Exploiting User Knowledge

Utilising knowledge of users can assist in designing user interfaces and can speed the learning process for new users. There are two applications of this in the programming environment.

The software producer can take advantage of information shared between the producer and the consumer, for example he or she may take advantage of a consumer’s knowledge of a particular look and feel.

The software producer is producing code for reuse by inheritance, and therefore is familiar with the inheritance relationship. Therefore using inheritance
as a basis for documentation takes advantage of this familiarity.

4.4.5. The Role of Constraints

A constraint is a relation that should be satisfied. Constraints can standardise the way things look and work [Borni92]. Norman [Norma88] gives examples of types of constraints used everyday. Some of these are valid constraints in reusing software. Physical constraints correspond to physical limitations on operations, for example a large peg will not fit into a small hole. Semantic constraints are where it only makes sense for things to be constrained in a particular way. Someone must be in the drivers seat of the car before the car can be in motion. Cultural constraints can restrict behaviour in certain contexts. Logical constraints correspond to a no alternative situation. If the top light switch controls the light on the left, then the bottom light switch must control the light switch on the right.

In GUI design constraints can enforce the look and feel of an application. Graphical constraints and interdependence of values are necessary when designing a GUI. Fields must be big enough to fit the text contained within them. The window backdrop must be large enough to contain all the fields within it. Constraints can be used in AI to reduce the search space of an inferencing system by pruning paths.

Borning et al [Borni92] give examples of uses of constraints in problem domains, for example that a resistor in a circuit must obey Ohm's law. A constraint that two views of a set of data remain consistent (i.e pie graph and bar graph views).

Constraints can be required or preferential. In the case of a standardised GUI interface this allows a software producer to suggest, but not enforce, a
standard interface. This allows for cases which in the judgement of the software producer do not require an absolute constraint.

Constraints considered here are those which require the programmer to conform with them. Constraint based languages, where functional programming languages are adapted to allow the programmer to specify only constraints, and for the language to define the behaviour, are not considered.

4.4.6. Locating Software for Reuse

It is best to convey the understanding of a class as early as possible. This is best done by documentation before the class is chosen, but realistically, can only be done while the class is being created.

The environment working at edit time, and implementing constraints will let a consumer know, as quickly as possible if the source code being reused is appropriate for reuse in the consumer’s context.

4.5. Controlling the Volume of Documentation

Too much documentation can be overwhelming and time consuming, while too little can leave areas uncovered [Arthu83]. A method for controlling the volume of documentation in a programming environment is hypertext. Hypertext allows a brief overview or description of functionality to be given, and for more detailed information on areas of the consumers choice to be given on request.

4.5.1. Powerful, General Systems

A methodology of software development allows powerful systems that are general. More detail than can be used in derived classes can be encoded in the base class,
without the concern that the base class will be too specific for general application. Traditionally a base class supports a general case, but with a documentation interface the base class can be designed to support two or more specific cases.

Helping with the power versus generality dilemma assists greatly in modelling real world situations. In the earlier example of the real world fish hierarchy, the software producer could have written code to cope with the mainstream fish likely to be found in an aquarium system. He or she could have allowed for the exceptions within the documentation system.

The documentation system begins to address the power versus generality dilemma. A framework can have general application, and can still be powerful. The user of the software need only select the parts needed for the application. This is an efficient use of resources, as code not needed is not included in the final application.

If software reuse is to be successful, it must provide a large volume of reusable software. High functionality code is harder to reuse. Low functionality code makes it harder to fit a specific problem.

### 4.5.2. Naming Systems

Bowman [Bowma93] describes more complex naming systems, such as directory systems, where the attributes of an object are used to locate it. A preference system can be used where preferences can be given to the attributes of an object. The preference can be given by the quality of the information in the attribute and the attributes importance to the object, both within the system and to the end user of the system.
In the system described, an attribute corresponds to a slot. The description of an class can be constructed from the class name plus its slots. The slots are known to be either new slots in a particular class, slots that override existing slots, or slots that were defined in the base class.

4.5.3. Intelligent Documentation

Since the methodology and environment specify a-kind-of inheritance and task based documentation, the software producer can encode knowledge about the project when writing the base classes, for example information about protocols. It would not be possible for a computerised software apprentice or assistant to have this level of detail of knowledge about the project. Intelligent interactive documentation is knowledge based, and modular. It should be able to adapt its response to a user and the users current task [Tyler91]. It is combined with the source code of the system, and can directly assist in producing the source code for the resulting application. The software consumer can enter a more general specification, and not code.

4.5.4. The Documentation Yoyo Problem

The Yoyo problem as identified by Taenz [Taenz89] was described in Chapter 2. The problems involves that a consumer must follow the methods being called up and down the inheritance hierarchy. The environment features outlined can prevent a similar problem occurring in the software documentation. By linking the base class documentation that is not overridden to the current class, the documentation is all presented together to the software consumer.
4.6. Solving Problems

4.6.1. Naming

Naming systems can be improved by putting the name of the object in context, and by specifying the properties that the object has. Younger and Bennett [Young92] propose a mathematical model of naming in which they define the context of an name as a domain in which every name maps to a unique object.

A programming methodology that identifies behaviours and properties (called slots) in each class provides a significant overview of the class to a software consumer. Once this grouping is made, it can assist in placing the class in context. It can be determined from the inheritance hierarchies which properties have changed in an inheritance hierarchy. A class can be described by its ancestry, and the properties that have changed. The inheritance can map onto a difference relation, giving a metric of the similarity of two classes.

Large amounts of documentation for each class is not the solution to the naming problem. This is time consuming, and would cause much of the documentation to be ignored. Instead a useful goal would be to try and reduce the learning curve for the inexperienced programmer. A general overview allows initial selection of a class, with integrated documentation making it clear quickly if the selection is inappropriate. Cross referencing, and allowing abbreviated descriptions can assist, as can a system of constraints that make the software consumer realise quickly that the class is inappropriate.
4.6.2. A-Kind-Of Inheritance

The problems with a-kind-of inheritance modelling a perfect world were that the real world systems rarely fit perfectly into a-kind-of inheritance hierarchies. This left the software producer to write a class which was general enough for all uses, or the software consumer to use compositional inheritance to override the parts of the model that were unanticipated, or omitted by the software producer.

The environment that allows the documentation to determine the inheritance method can limit a base class to a range, allow several choices, or allow for exceptions. Examples of this are shown in the following chapter.

4.6.3. Abstraction

Any design notation that abstracts a problem ignores some details and emphasises others [Johns91]. The choice of what features to highlight is determined by the methodology, namely the inheritable behaviours and properties of a class. This presents the information most relevant to the consumer reusing software by inheritance, in a standardised form.

4.7. Summary

This chapter proposed a new methodology and environment for software reuse by inheritance. The new features allow for software producers to specify the interaction of the software consumer with the reusable software. Solutions for the the problems of abstraction, naming, and power versus generality with a-kind-of inheritance were proposed.

There is a need for a documentation system to demonstrate methodology and environment features developed in this chapter.
5. EVIDS - An Intelligent Documentation System for Inheritance

The Extended Viola Intelligent Documentation System (EVIDS) is an experiment in producing a documentation environment for reuse using inheritance. It includes many of the features of the environment described in the previous chapter, excluding some described in that chapter as having already been implemented and evaluated by other researchers.

EVIDS produces interpretable Viola code from a set of Viola base classes, their associated documentation, and user input from the software consumer.

The software producer writing a base class to be reused by a-kind-of inheritance, knows the functionality of the class, the ways to access it, and what parts of the code should be encapsulated and under what circumstances. EVIDS provides the software producer with a minimal and simple, yet powerful method of expressing these concepts to the software consumer, for the consumer to be guided, but not limited.
The features of EVIDS include:

- Facilities for the display of, and user interaction with documentation, including Hypertext, graphics, and sound.

- Allows inheritance of existing documentation classes so documentation does not have to be created from scratch.

- Automatic reuse of documentation where possible. If a property is not overridden in a derived class, the documentation of that property in the base class is used as its documentation in the derived class.

EVIDS is useful as a self-contained system, but the ideas contained within the system would best be used as part of a software development infrastructure.
This chapter first outlines the Viola language, and the extensions made to Viola to facilitate EVIDS. It then describes the file structures used by EVIDS, and an example of using the system for documentation.

5.1. Intelligent Documentation

Although EVIDS is described here as an intelligent documentation system it does not itself contain knowledge about any particular domain. Instead it provides the facility for the software producer to encode knowledge about the domain. Thus it provides a support environment for intelligent documentation.

5.2. Viola

EVIDS is implemented in Extended Viola and documents some of the classes included with the Viola package. Viola† is a package for construction of graphical user interfaces. It consists of an interpreted language and a built in class library.

A graphical user interface is expressed as a hierarchy of objects in Viola. An object at the root of a hierarchy is usually a backdrop window. Objects lower in the hierarchy make up the contents of the window, such as text fields, buttons, and scrollbars. An object like a scrollbar is in fact a hierarchy of objects itself, constructed from arrow buttons and a value bar. Syntactically the relationship between an object, and those beneath it in the hierarchy is called a parent-child relationship in Viola. This relationship is not an inheritance relation, but a way of representing the construction of a user interface.

† Viola was written by Pei Y. Wei at the Experimental Computing Facility at the University of California at Berkeley.
Viola contains some features and classes for functions other than GUI’s. In addition to classes and methods for windows and buttons etc, it has classes and methods to support http (hypertext transfer protocol), and internet TCP socket functions. It also has support for Hypertext Markup Language (HTML) text.

In Viola it is common to use inheritance to inherit properties as well as behaviour. This is in contrast to a language like C++ where inheritance is only usually used to inherit behaviours.

Viola is most frequently encountered as the language in which the Viola World Wide Web browser is written.

5.2.1. The Viola Language

Viola is a language with similarities to Smalltalk. It is an interpreted typeless inheritance based language. It could not be called object oriented or object based according to previous definitions as it has no support for polymorphism, and does not support strict encapsulation (any object may manipulate another, access its private data and, like Smalltalk, there is no concept of private methods).

5.2.2. Viola Slots

A Viola class is constructed from slots. Slots are the properties and behaviours of objects. For example a text display region would have colour, size, and font slots.

A pane (window) class, would have slots to define properties like its name, size, colour, content, icon bitmap, children, parents, and the way that children objects will be positioned within the pane.

Other slots exist for information about the object, for example the class type of an object, the object name, and its position in the visual display hierarchy
and inheritance hierarchy.

The terminology "slots" was adapted from the work on frames in AI research. In that case the properties of the frames were called slots [Bolsr77].

Slots in Viola cannot have deferred values.

5.2.3. Types of Slots

Slots in Viola can be either common or private. Common slots are inherited. If they are undefined in a derived class the slot is unchanged from the base class. Private slots are not inherited. Typically, private slots are properties like the class name, its children and parents. A private slot undefined in an object instance will have an undefined value.

5.2.4. Event Handling and Message Passing

Events that cause Viola to send a message to an object include user I/O events like buttonPress and keyPress which are sent on any mouse button being pressed, and on any keystroke being made respectively. Viola automatically sends messages to an object for initialisation and when an object is required to render or expose itself. Messages can also be sent between the objects using the send() function. A message that is not handled by an object is handled in the way defined by the base class. An object only differs from its base class in as far as it defines slot values differently from its base class. By default an object can be totally defined by the class type.

A message delivered to an object causes the script associated with the object to be executed. If a script does not provide a method to handle the event, then no action will be taken. Viola provides a usual() method which causes the behaviour that
would have occurred if no script at all was provided with the object. Conventionally, a script will call the `usual()` method for all messages not explicitly handled, to invoke the default behaviour.

5.2.5. Language Syntax

The language syntax involves building an object description from a built-in class type. A “Hello World” program in Viola would look like:

```text
\name {helloWorld}
\class {pane}
\content {Hello World}
```

Fig 5.2 A Viola hello world program.

This program declares an object description named `HelloWorld`. It is derived from the class `pane`. It overrides the content slot with the string `Hello World`. All other slots in the `pane` class are unaltered.

The program can be extended easily to add a quit button.
\class {vpane}
\name {helloWorld}
\children {helloWorld.display helloWorld.dismiss}

\class {textDisp}
\name {helloWorld.display}
\parent {helloWorld}
\content {Hello World}

\class {textButton}
\name {helloWorld.dismiss}
\parent {helloWorld}
\label {Dismiss}
\script {
  if (arg[0] == "buttonRelease")
  {
    quit();
  }
  usual();
}

Fig 5.3 A Viola hello world program, with a quit button.

The program defines three objects. The background vertical window pane (vpane). HelloWorld is the backdrop window for its two children. The pane is vertical, so objects within it will be arranged vertically. Its children are helloWorld.display, of type textDisp with a content of the “Hello World” string, and helloWorld.dismiss, of type textButton with the label
Dismiss. If it receives a buttonRelease message, it calls the quit() method.

![Fig 5.4 - “Hello World” in Viola](image)

### 5.2.6. Viola Built In Classes

Viola provides classes for text display windows, text input windows, buttons, hypertext capable windows, and classes with network functions. Viola classes are typically written as structures contained within C code, and compiled into Viola. Methods are written in pure C.

There is a difference in Viola between a class and an object description that can be interpreted. A Viola class can only be inherited from, and it cannot be used directly in an application. An equivalent object description can be defined by inheriting from a base class, but not defining or overriding any slots (except for the name and class slots).

The Viola class hierarchy is:

- **cosmic** /* Parent of all objects */
- **generic** /* Contains standard methods */
- **field** /* The base for GUI components */
5.2.7. Object Representation and Manipulation

Objects in the Viola system are created in one of three ways. An object can
1. correspond to an object representation on disk.

2. be a clone of another object, created by the \texttt{clone()} method in class \texttt{cosmic}.

3. be created dynamically by the \texttt{create()} method.

5.3. Viola and EVIDS

Viola was chosen as a basis for EVIDS because it is a language which supports and emphasises a-kind-of inheritance as its primary method of software reuse. This enables demonstration of a documentation system for a-kind-of inheritance without concern for other forms of documentation. It was designed for constructing user interfaces and the World Wide Web application. It natively supports features for advanced documentation, sound and graphics. It has key elements that enable it to be used as the basis both as a demonstration language for the documentation system, and the language to construct the documentation system. There were several extensions necessary to make it capable of supporting the basis of the documentation system.

5.4. Extended Viola

The Viola language was extended in order to support the EVIDS features, and to provide greater functionality for the documentation class programmer. The language is a complete superset of the original Viola syntax. EVIDS is constructed from a Extended Viola interpreter, and scripts written in the Extended Viola language. The Viola language and built in classes were extended in several ways.

- Inheritance of Viola classes is now allowed, and multiple layers of inheritance are possible. In the original Viola language only one layer of inheritance
from classes programmed in 'C' was allowed.

- A 'doco' slot was added as one of the primary description slots in the class (along with the 'name' and 'class' slots). This slot allows a file to be specified which contains the documentation description for a class.

- An interpreter for the documentation description language has been implemented.

- A document() method was added to the generic class (conventionally an ancestor of all classes). The function initialises the documentation system for a single class. A call to this method with a parameter of a class causes the documentation description to be read, and the Viola documentation code interpreted. This means that the initialisation and user interface to the documentation system can be written in Viola. This is convenient to provide a consistent look and feel to the user.

- Additional built in methods have been added to facilitate the building of the documentation system, and associated applications (such as view(), note(), and remember()).

- Classes can be dynamically loaded. This means a class that cannot be located internally within Viola is searched for on disk. If the file is located it is then loaded into memory, and is interpreted by Viola in the same way as an internal class. This eliminates the need to modify the Viola program every time a new class needs to be added.

The documentation classes referenced in the documentation description are written in Extended Viola. Therefore they can use the documentation features of the Extended Viola system. They can be inherited from other classes, and can be documented with the same documentation techniques as the Viola classes. The
writer of derived documentation classes is a software consumer in the same way as the derived class writers of applications.

Extended Viola has been ported into the System V Release 3 and System V Release 4 environments, on MIPS RiscOS and SUN Solaris computers.

5.4.1. Defining a Dynamic Extended Viola Base Class

A base class in Extended Viola is similar to the structure of a class defined internally in Viola. Each slot must be defined to be one of the internal Viola types. It can be marked as *readonly*, *writeonly* and *readwrite*. These properties refer to the ability of the object based on this class to change and access the value of the slot of the class. A derived class can set these parameters to any status it wishes (within the boundaries of the documentation description).

Each slot is marked as to whether it is a new slot in this class, or whether it overrides the slot that existed in a previous class. Classes defined with new names, that are not marked as new slots, cause an Extended Viola warning, and the slot is ignored.

5.5. Construction of the Documentation System

The documentation system is written in Extended Viola. By default a documentation description is loaded for each class. EVIDS loads and interprets the documentation description and Extended Viola classes. The product is a derived class, which may form part of the application.
5.6. File Layout

Four types of data files make up the system. The types are determined by the file suffix. A loaded base class (vc), the object description (v), the documentation description (vd), and the documentation object description (v). Note that the documentation object descriptions are not distinguishable from Viola object descriptions. Fig 5.5 shows that the Viola base class references a documentation description, which in turn references the documentation classes for the properties and behaviours.
5.6.1. The Base Class

The base class is written in Extended Viola. It is the software constructed for reuse. There is a special intrinsic slot that can be defined in a base class in Extended Viola, called doco. If it is defined it refers to a documentation description file for that class. If it is not defined then a warning message is issued that a
documentation slot does not exist, and Extended Viola attempts to load the documentation description with the same file prefix as the Viola base class. The Extended Viola method `document()` loads the documentation description based on the doco slot.

One of the base classes supplied with the Viola system is the `txtDisp` class. An outline of the properties and behaviours is shown in Fig 5.6.
5.6.2. The Documentation Description

The documentation description has one line for each slot in the Extended Viola base class. When the documentation description for a class is read, the Extended Viola object description corresponding to each slot in the base class is loaded and sent init and render messages.
A documentation description for the example class is shown in Fig 5.7.

like:

\init {txtDispinit}
\name {nameDoco}
\color {palette}
\width {txtDispWidthConstraint}
\height {heightConstraint}
\buttonPress {generalBehaviour}
\keyPress {keyPressBehaviour}

Fig 5.7 txtDisp.vd

Note there is an optional init slot, which is a class loaded before any of the other documentation classes. Its purpose is to be an introduction to the class.

5.6.3. Documentation Classes

The documentation classes are read as directed by the documentation description. The output from the documentation classes is expected to provide the slot behaviour or property in the derived class.

There is a set of documentation base classes provided with EVIDS. Not only the documentation classes provided can be used, but any new documentation classes built can also be incorporated. A library of documentation classes can be built up in much the same way as a library of source code classes.

EVIDS provides a framework for the documentation classes to be produced, and allows additional documentation base classes to be added at any time.
The documentation class for building a width constraint is formed by a class constructed from valueConstraint.

\name {txtDispWidthConstraint}  
\type {valueConstraint}  
\overview {The width of the object}  
\maximum {500}  
\minimum {100}

Fig 5.8 txtDispWidthConstraint.v

In this case, the base class valueConstraint uses the properties minimum and maximum defined in the derived class to use as the bounds for the value permitted.

In general, the documentation class can:

- Implement constraints (on value, syntax, properties, behaviour).
- Specify any interpretation of user input.
- Supply textual and graphical information.
- Place edit time restrictions on access to methods and data.

There should be three parts to a documentation class.

1. Overview. A brief written overview of the class.

2. Interaction. This section contains how the property in the derived class is obtained from the user.
3. Verification. This implements any constraints, or sanity checking on the input once it is obtained from the user.

The overview is the response when the object is sent an overview request. Conventionally this is a one line summary stored as an overview slot. The interaction and the verification are coded using any of the structures of Extended Viola. Conventionally a constraint responds to a `constraintVerify`. Other objects may have conventional responses, depending on their base classes.

5.7. Using EVIDS

Both the software producer and the software consumer interact with EVIDS. The software producer must produce documentation classes, and initialisation code. These classes and code define the interaction of the software consumer with EVIDS.

5.7.1. The Consumer Interface

5.7.1.1. Locating an Appropriate Class for Reuse

EVIDS provides default startup procedures. The base classes available for reuse are offered to the consumer. When one is selected, EVIDS interactively builds a derived class from the Viola built in classes. EVIDS allows modification to the startup scripts appropriate to the class library being documented. The software producer can add any interface to the startup procedures.
Fig 5.9 EVIDS class location. Finding a class.

Fig 5.9 shows the default EVIDS class location screen. It is simply a loaded hypertext, in this case the class hierarchy.
Class

txtDisp

Description
This class implements a multi-line, multi-font, hidden data embedable, text field. But does not edit very well (yet).

See the HTML class for an easier way to do formatted (deals with window resizing) and hyper text.

Show how to (to be links to tutorial sections):
* Transfer date in/out of text field from/to other objects.
* Set content with text loaded from file, and save content to file.
* Set content with text from output of external programs (ie: nroff).
* Get various state information about the text field (cursor position, number of lines...).

txtDisp

* name
* parent
* children
* content
* label
* width
* height
* minWidth
* minHeight
* maxWidth

The maximum permitted width of the field

Fig 5.10 EVIDS class location. Detail of a class.

Figure 5.10 shows the screen once a class has been selected. The top portion of the window shows the class documentation. The slots are shown in the centre, and the response of an overview call to the documentation class is shown in the bottom portion.
5.7.1.2. Creating a Derived Class

The interface for the software consumer hides the complexity of the file layout. The software consumer interacts with the system as defined in the documentation description and the initialisation code.

The interface for the software consumer is entirely specified in the associated documentation for a base class. This interface is alterable by the documentation producer. This description gives an outline of both the default interaction, and the examples of possible documentation methods.

5.7.2. Documenting a Class

The role of software consumer is played both by the base class writer, who reuses existing documentation classes, and by the derived class writer, who reuses existing Viola base classes.

Documenting a class is a matter of creating the documentation descriptions and documentation classes for each slot defined in the base class.

5.7.2.1. Locating a Documentation Base Class

The correct documentation can be created from scratch as an Extended Viola class. A better alternative is to select a documentation class as the basis for the documentation. The techniques for locating an appropriate class for reuse (ref 5.7.1.1) can also be applied here. Once the correct class is located, the base documentation class designed for reuse also may have a documentation description and documentation classes to assist in the construction of a derived documentation class.

The documentation description, which relates the documentation classes to the properties and behaviours of the class, can be created using any text editor.
5.7.2.2. Attaching a Documentation Class

Syntactically a Viola class can be documented by defining a documentation slot. This documentation slot references another file which contains description of how the class is to be documented. The syntax of the file is much the same as a Viola class. It contains all the slot names in the Viola class, together with other Viola classes that document them. It is not necessary to write all of these documentation classes from scratch, as many will already be written, either similar or close to what is required to document the class. In the case where a similar class exists to what is required to document a slot, the class can be inherited and the changes made. To make the documentation classes simpler to use, it may be desirable to document the documentation classes in the same way as the program classes. This is one of the design strengths of EVIDS.

5.7.2.3. Documentation Base Classes

Additional base classes, such as welcome, size, generalBehaviour are available to be specialised.

Other classes are available which have been used to document particular classes. In a production environment, libraries of documentation classes can be constructed.

5.7.2.4. Text Interface

An experienced user of EVIDS producing documentation can enter the text of the Extended Viola classes directly. It is not necessary for them to use the EVIDS documentation. An experienced user may find it quicker to use a text interface to enter the documentation classes and description, still using the existing
documentation classes as a base, but without using the EVIDS interface to the base classes.

5.7.3. Documenting a Class Library

Extended Viola provides the `document()` method to interactively read the documentation description and load the documentation object descriptions it describes. It also provides the method `view()` to show the properties of a class. `view()` extracts the properties and behaviour names of a class, and constructs a hypertext view of the class. Each property or behaviour is tagged. The context of the class is documented just by the names of the properties and behaviours of the calling class. HTML documentation for the properties and slots can be linked to the hypertext that `view` provides by the calling class recognising the tag. By default the EVIDS calling class calls the documentation description of the tagged class asking for an overview. This usually results in a brief description of the property or behaviour. These methods are provided in the `cosmic` class which is at the top of the inheritance hierarchy for all classes.

It is possible for the user to write their own location code in Viola using EVIDS. There is a foundation of location code available to document a class library.

The default view is compiled from the distributed hypertext documentation for the Viola classes, with links added to the `view()` of the class. Once the correct class is located, the `document()` method is called to initiate the interactive construction of a derived class.
5.7.4. Slot Constraints

EVIDS allows constraints to be put on the derived class writer as to what may be entered at edit time. Other object oriented languages provide the ability to have constraints at compile time.

Constraint classes can be constructed. The constraint classes can have slots which define the minimum and maximum value for a particular slot in the derived class.

Slot constraints are implemented by specifying a constraint documentation class. Constraint base documentation classes are provided for the common forms of constraint. A logical constraint in a GUI context is if the foreground colour is set to a particular colour, the background colour property cannot be set to the same colour. This constraint is implemented by a constraint documentation class inherited from the base class logicalConstraint.

In a GUI environment, constraints can be used to maintain relationships between consistency between data and the graphical display of the data.

One of the simplest constraint classes is valueConstraint. This is a base class for a documentation class which specifies values which constrain a slot. In the majority of cases the user will simply supply the values which the slots must be restricted to.

Another base class is syntaxConstraint. This reads some input, and provides a way for the derived class writer to validate the input. If the method implemented was a Extended Viola parser, this would be similar to a syntax editor.

All the constraint base classes defined in EVIDS call verifyConstraint and expect a true or false return.
The constraint documentation class hierarchy consists of:

```plaintext
constraint /* The constraint base class */
valueConstraint /* restrict a property to a value */
syntaxConstraint /* check input against syntax rules */
logicalConstraint /* check and/or of several properties/behaviours */
```

5.8. Building a Simple Documented System

5.8.1. Generating a Derived Class

The consumer interaction with the documentation classes to create a pane back-drop is shown in the figures. Fig 5.11 shows an “init” class. This is derived from class `welcome`, which gives a one line overview of the class. This class can contain hypertext links to further documentation for the class. Fig 5.12 shows a “size” class, which allow the user to visually set the size of the resulting window. The size class sets the width and height slots. If it is called to document the height first, it calls `note()`’s to store the width property, and calls `remember()`’s within the width class to set the width, or vice-versa. Fig 5.13 shows a menu to set the pane type. It offers a selection, since there is a limited range of pane types.

---

**Fig 5.11** An “init” class for pane

**paneWelcome**

Welcome to the world of window panes
Click to continue
5.8.2. Using Constraints

Another example to demonstrate simple constraints could be a bank account. Every bank account type has a restriction on the withdrawal amount. The withdrawal amount is a simple constraint on the account type.
This is based on the valueConstraint class that forms part of the documentation framework that forms part of EVIDS.

The methodology being used means that the writer of the bank account class knows the features of the accounts that can possibly be offered. The writer of the derived class is restricted in the types of accounts that can be offered by the writer of the base class.
5.9. EVIDS Support for Reuse Methodology and Environment

5.9.1. Organising Properties and Behaviours

The slots in Extended Viola correspond to the properties and behaviours of the class. The built in classes in Extended Viola already have their properties and behaviours organised.

5.9.2. A-Kind-Of Inheritance

Viola and EVIDS support a-kind-of inheritance. The classes built in to Viola are organised in an a-kind-of inheritance hierarchy. The base classes at the root of the hierarchy are the most general. The classes lower in the hierarchy are more specialised.

EVIDS, by supporting features that help solve some of the problems in using a-kind-of inheritance, encourages its use. For example the software producer can write documentation that allows for exceptions in EVIDS. This makes modelling real world systems with a-kind-of inheritance simpler.

5.9.3. Interaction

EVIDS facilitates interactive documentation. It supports fields for entering textual information, push button interfaces, visual examples, hypertext etc. All these methods of interaction can be used by the software producer.

5.9.4. Embedded Documentation

The documentation and the source code are combined from the point of view of the software consumer. The consumer sees the producer’s documentation as the
interface for the source code.

5.9.5. PROs, WIMPs, and DUNCEs

EVIDS separates the PROs from the WIMPs. The environment provides an explicit division between the parts of code that are written by the designer, and the parts of the code that are written by the inexperienced programmer. It prevents lower level programmers destroying the design, or circumventing the encapsulation of the system.

The documentation class determines what is contained in the slot definition in the derived class. Therefore the PRO has complete control over the code written by the WIMP or DUNCE.

5.9.6. Inheriting Documentation

Every slot in every class does not have to be documented. Slots that have been documented at one level of inheritance, and not overridden in an derived classes, inherit not only the slots of the base class but also the documentation.

The system works by searching the documentation descriptions referenced by base classes in the hierarchy. Starting at the class furthest from the root of the hierarchy, the first documentation description found for a slot is used.

By specifying documentation for the slot in a derived class, the documentation need not be derived, but can be specified separately at the new level of the hierarchy. This is not a recommended procedure, as the documentation can be changed by someone who did not define the original property or behaviour.
5.10. Writing Documentation with EVIDS

Techniques for writing documentation have been discussed widely in the computing literature. EVIDS permits documentation of the class library, the class, and the slots. Bad commenting techniques in normal source code should also be avoided in EVIDS, in particular repeating information that is already available via the class and property names [Kerni84a].

For EVIDS documentation to be more effective than more traditional forms of documentation it must interact with the consumer. The documentation should enforce restrictions, rather than stating them.

5.11. Limitations of EVIDS

The MIT group who developed X said in their early releases that they would not listen to criticism of the window manager that they has supplied for use with X, or any lack of features that the window manager had. The only complaints should be about what a window manager could not be written to do using the features provided in their libraries and the X protocol.

The same argument could be applied to EVIDS. EVIDS provides the environment and the philosophy behind class reuse by a-kind-of inheritance. If the documentation for a particular class or function, or a basis for it is missing it is not a criticism of the EVIDS system. The only valid criticism of the limitations of EVIDS is that a class or documentation could not be coded easily and simply.

5.12. Extensions
5.12.1. Changing the Base Classes

The base documentation classes provided with EVIDS can be changed to alter the look and feel of the user interaction. For example, currently a `valueConstraint` class prompts for a value in a text window, with the Viola standard look and feel. The base class could be changed to alter the way the value is input, while still using the same derived class to verify the constraint.

5.12.2. Other Object Oriented Languages

Other languages that support inheritance can be augmented with features to support an EVIDS environment. What is required is a syntactic extension to the languages to support methodology described in Chapter 4.

To support the grouping of properties and behaviour required by the methodology, a preprocessor could extract the grouping from the code, and pass the remainder of the code on to the language interpreter or compiler.

The documentation classes and the EVIDS documentation system could remain written in Extended Viola, or they could be rewritten in another GUI building framework.

A proposed system for C++ involves a syntactic extension to the language to add support for the methodology, while still using the Extended Viola language to provide the documentation classes. The documentation classes and EVIDS would have to be changed to produce valid C++ output.

5.13. Summary

This chapter outlines EVIDS and Extended Viola. Extended Viola is an extension to the Viola language performed by the thesis author to support the EVIDS
experiment. The main features and capabilities of the EVIDS system are based on interactive documentation, constructed from Extended Viola classes. Base classes provide reusable code to allow constraints, information and selections.

There are possibilities to extend EVIDS to other object oriented languages, and to change the Viola look and feel.
6. Evaluation

6.1. Performance

EVIDS operates at edit time, so there is no run time processing overhead. Extended Viola itself, which executes the documentation classes, operates quickly on a modern desk top workstation.

6.2. Shortcomings

6.2.1. Time for Development

EVIDS relies on the software producer investing the necessary time for the complete documentation of the class, and the class library. It is unlikely that this time investment would be justified for small scale reuse (reuse 1-5 times).

6.2.2. Scalability

Although EVIDS works well for locating classes within a framework or a class library, a problem still remains for locating an appropriate class among an even greater amount of classes for reuse. Fischer et al [Fisch88] propose that for software reuse to be successful there must be a large amount of reusable software. The problem of locating appropriate reusable software from large amounts of
diverse reusable code will become increasingly difficult.

The method proposed for class location by this thesis is that a class manager for a framework supply documentation for location. This is scalable only with the standardisation of documentation of all classes in different frameworks and class libraries. This standardisation by all software producers is not realistic in the short term.

6.2.3. Class Interaction

If there are two parts to understanding a framework, the class interface and class interaction, EVIDS is better at documenting the interface. EVIDS is concerned with locating, and inheriting from one class at a time. Although the software producer is free to document the interaction between the classes, EVIDS offers no explicit support for this. It follows that EVIDS is better at supporting a class library, then a class framework.

6.2.4. Debugging

It is still possible to write code within EVIDS that doesn’t work, or that contains bugs. Getting rid of these bugs is still a matter of using the Viola system directly.

6.3. Comparision with Similar Systems

6.3.1. Peridot

Peridot [Myers90] is an “experimental tool that allows designers to create user interface components without conventional programming”. One of its aims is to extend programming to non-programmers (DUNCEs). Peridot allows a
programmer to create user interface components by drawing what the interface should look like, and then indicating what actions should be associated with the interface. Peridot then generates user interface code, which can be incorporated into an application.

Peridot is highly customised to the graphical user interface development environment. EVIDS also works in this environment, because the existing Viola classes are intended for the development of graphical user interfaces.

Peridot can create constraints dynamically while designing interface buttons. If one button is drawn a certain size, Peridot will ask whether to apply a constraint to make similar sized buttons identical. These values are stored as active values (implemented as variables within Lisp) in Peridot. EVIDS allows any pre-existing value to be used as the basis for a constraint. Therefore the size of an existing button could be used to constrain another. Documentation classes in EVIDS are transient. They are only loaded and run in order to document an individual class. A value determined by one documentation class to be a constraint on button size, could not be applied to another button class developed later (short of the documentation class modifying the text of the subsequent documentation class!).

Peridot is a visual language. It uses visual representations of programs, and allows the programmer to manipulate objects visually. It is a simple visual language in that all the objects being manipulated have an intuitive graphical representation as they are all GUI components.

When using a visual language not only the DUNCE is limited by the facilities of the visual languages, the PRO is as well. EVIDS permits a more experienced programmer to enter the code directly, bypassing the visual interface. In the
same way a novice computer user may feel more comfortable with a Macintosh menu interface, but a serious computer user would feel restricted by it. When using EVIDS the restricted interface to the code is optional. The experienced programmer familiar with the software for reuse may always bypass EVIDS and program in the native language.

6.3.2. CCEL

CCEL is a extension to the C++ language to allow constraints. It was described in chapter 3 as an example of using constraints for documentation. It is described fully in [Duby92].

CCEL is closely mapped to the C++ language. EVIDS is currently closely mapped to the Viola language. The methodology and philosophy behind EVIDS could be adapted to any language using inheritance for reuse. Both systems require work to make them generally applicable to object oriented programming languages.

A constraint in CCEL requiring that all class names begin with a capital letter is shown in [Duby92].

```c
// Every class name must begin with a capital letter
CapitaliseClassNames {
    Class C;

    Assert(C.name().matches("^[A-Z]"));
}
```

Fig 6.1 A CCEL syntax constraint.
The class C maps to all classes in the system. The constraint takes the form of an `assert()` function, and a regular expression match. The properties of a class that can be constrained are enumerated in the CCEL language specification.

The same constraint in EVIDS would be enforced by the documentation class responsible for the class name slot. In the `textDisp` class presented in chapter five, this slot was documented by `nameDoco` class shown in Fig 6.2.
nameDoco.v

\name {nameDoco}
\type {syntaxConstraint}
\content {Name of the Derived Class}
\script {
  switch (arg[0])
  {
    case "verifyConstraint":
    {
      firstChar = nthChar(arg[1], 1)
      if (firstChar < 'A' || firstChar > 'Z')
      {
        return 0;
      }
      else
      {
        return 1;
      } 
    } 
  }
  usual()
}

Fig 6.2 A EVIDS syntax constraint
EVIDS can restrict any property of a class at edit time.

CCEL is separate from the programming language. A “lint” like program (Clean++) is run after the program is written, after the C++ language syntax is verified to be correct. A program written with all the class names in lower case, would then have the CCEL tool Clean++ run on it. The Clean++ tool would detect the violation of the CCEL constraint, and advise that it be corrected. At this point in time the programmer would have a choice. To edit and change all the required class names into upper case, or to proceed with the software compilation. If the programmer were to choose the latter option, the constraint is ineffective.

Viola has a constraint that the name of a child object in a visual hierarchy should have the name of its parent prepended to it. This constraint can be applied automatically with EVIDS, by supplying the correct prefix automatically. CCEL works at compile time, so the software would have to be familiar with the constraint from other documentation before beginning the project.

6.3.3. Marvel

Marvel is an intelligent assistant for software development and maintenance, described by Kaiser et al in [Kaise88]. Marvel operates in a persistent object oriented environment. It undertakes simple development, and manages simple tasks in the programming environment. Marvel can, when given a set of rules consisting of several conditions, determine a correct time to compile a set of files. The actions that Marvel can perform on the satisfying of conditions are called activities. The types of activities that it can perform include compile actions and edit actions.
Marvel also provides information access routines for querying a database of persistent objects. The default information access view of classes is a hierarchical structure. Class libraries contain classes which contain components. Kaiser et al describe navigating through this structure as "like navigating through files in a filesystem hierarchy". Marvel abstracts the information presented, with the user selecting objects to be examined in greater detail. There are also several other views available.

Marvel allows the user to follow links from code in one class to corresponding code in another class, for example linking a method call to its definition.

EVIDS does not provide any facilities for managing the programming environment. By providing the actual interface for the entry of code it provides greater control and restrictions over the edit phase of software creation.

EVIDS provides a standard view but permits extensions by the software producer. No links are provided in the code, but hyperlinks may connect parts of the documentation. Marvel provides several standard views for browsing classes and locating a particular class.

6.3.4. KBEmacs

KBEmacs is a knowledge based, emacs environment. It is an attempt by Rich [Rich90] to construct a programmer's apprentice. Clichés and KBEmacs are outlined in chapter three. The KBEmacs definition of clichés are called plans. A plan definition defines the role and the constraints for the cliché. It is the knowledge store for the programmer's apprentice.

Constraints are enforced by a code segment written as a plan calculus.
The constraint

\( (\text{DERIVED} \ {\text{the line-limit}}) \ (- \ 65 \ (\text{SIZE_IN_LINES} \ {\text{the summary}}) \) \)

enforces a size constraint on one variable relative to another. The constraint functions in the plan allow KBEmacs to automatically add code to comply with the constraint. In such a situation EVIDS will still present the software consumer with a small or no range of possible options.

The functions available in KBEmacs for coding constraints are very limited. In EVIDS the full expression of the extended Viola language is available to express the constraint.

The intelligent assistant obtains information about the project as the construction progresses. It becomes more intelligent the nearer the program comes to completion. The intelligent assistant does, however, start with some basis of knowledge. The information common to every application is known at the start of programming by the intelligent assistant. This basis of knowledge is contained in the base documentation classes available for use in EVIDS. This information is immediately available to an intelligent assistant. EVIDS requires that this knowledge is added to or manipulated by the software producer before it can be used at all.
7. Conclusions

7.1. So What's New?

Classification of programming features to enable reuse. The classification of the three programming features that assist reuse was first proposed by the thesis author in [Serge92] during early work on this thesis. A similar classification was later proposed by Wade at OOPSLA '93 [Wade93].

A-Kind-Of Inheritance. An a kind of relationship has been used to describe inheritance in many texts. Using a-kind-of inheritance to divide responsibility between expert programmers and novices is first proposed here. Using a-kind-of inheritance, but enabling overrides consistent with the author's intention, is also first proposed here.

A reuse methodology. Discipline in inheritance to enhance reuse is the subject of musings by many authors over the past half decade. The ideal engineering compromise has yet to be found, but grouping the behaviour and properties to provide component-like structures for reuse is new here, to the extent that it differs from the Viola language concepts, which were found to be similar.

Extended Viola. Extended Viola consists of approximately 4000 lines of new C code (unfortunately), and reuses 40,000 lines of C code written by Pei Wei. Roughly half of the added code increases the functionality of existing parts of
Viola, and the other half is new modules to extend the language. Extended Viola makes Viola a more usable language. Viola previously needed to be recompiled for every new class added. To add the diversity necessary to make the language successful would have resulted in an executable that was unnecessarily large.

**EVIDS** has some motivations and features in common with other systems, but the structure of EVIDS is currently unique. It incorporates new ideas for expert-novice interaction, and ideas from existing documentation methods applied in a different context. EVIDS itself consists of reusable base classes, and initialisation/location code written by the thesis author. It uses some HTML scripts written by Oscar Nierstrasz at the University of Geneva to generate nicely formatted hypertext.

### 7.2. Software Reuse

Software reuse is an active area of research in computer science. Productivity gains are being realised from software reuse techniques, but these have been less than anticipated. Most gains have been achieved in small scale reuse of generic components like those found in standard libraries. Object oriented programming facilitates larger scale reuse. Inheritance is one of the methods of software reuse in object oriented programming. It has been underutilised, partly due to the problems in documenting the source code to be reused.

### 7.3. Documentation for Reuse

Documentation is one of the features of a programming environment essential to enable reuse. In the case of inheritance, a greater scope is available to the documenter of software for reuse.
7.4. A Software Infrastructure

EVIDS will form part of a infrastructure for software reuse. Not all software reuse requires or uses inheritance, therefore not all software reuse will use EVIDS. With the move toward software development environments being shipped with programming languages, many of the techniques described in this thesis for using and implementing EVIDS will be useful in these environments.

A software infrastructure allows design, documentation and source code to be combined. Freeman [Freem83] notes that the reuse of program code alone has no value, and the real value is the reuse of the design and analysis. In a software infrastructure the distinction between the program code and the description of the design is not absolute. The aim is to incorporate the design, and the methods for reuse into a development environment.

7.5. Further Research

7.5.1. User Configurable Visual Environments

One of the motivations for the research was described in the introduction to this thesis as to enable novice or inexperienced users to be able to build an application by interacting with, collecting and joining software built by skilled software engineers.

Each class for reuse can be represented as an object in a visual language, which can be manipulated by the visual language programmer. The standardisation of the objects in the visual language and the way they could be manipulated could be defined by a class framework.
7.5.2. User Interface Considerations

EVIDS has been built using the look and feel of the Viola built in classes provided with EVIDS as proof of a concept. There are better designed interfaces designed for use in visual languages. The same Viola classes and language could be used to support any user look and feel, with only changes to the base class methods.

7.6. Conclusion

One of the obstacles to software reuse has been the lack of support in programming environments. Programming environments, and associated methodologies, like the ones developed in this thesis for software reuse by a-kind-of inheritance, can make the software development industry more productive.
References

Aharo91.


Andre89.


Arthu83.


Baker72.


BarDa92.


Barst85.


Bigge87.

Bolsr77.

Booch90.

Booch91.

Bomi87.

Bomi92.

Bowma93.

Brach93.
Brock90.


Brook75.


Brown88.


Byrne91.


Cargi91.


Carro92.


Chang87.


Cheng92.

Chin91.

Conn87.

Cox89.

Cybul92.

Czype93.

Domin92.

Duby92.
Eisen92.

Fisch88.

Foote91.

Freem83.

Freem87.

Garg90.

Grass92.
Hende92.


Hoare82.


Hunt86.


Ichbi83.


Johns93.


Johns91.


Jones84.

Joyne92.

Kaise88.

Kerni84a.

Kerni84b.

LaLon88.

Lampi93.

Lewis91.

Linto89.
M. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with
InterViews,” *IEEE Computer*, 22, 2, pp. 8-22 (Feb 1989).

McIlr69.


Meyer87.


Meyer92.


Minsk75.


Myers89.


Myers90.

Nelson74.


Nielsen90.


Niers91.


Nino90.


Norma88.


Papat92.


Pauw93.

Nelso74.


Niels90.


Niers91.


Nino90.


Norma88.


Papat92.


Pauw93.

Pinso88.


Pirsi76.


Priet87.


Retti91.


Rich90.


Sakki89.


Samet92.


Serge92.


Snyde86.


Spenc87.


Taenz89.


Tracz87a.


Tracz87b.


Tracz88a.

Tracz88b.

Tracz88c.

Tyler91.

Wade93.

Waldo91.

Wasse91.

Water85.
Weiss90.


Wirfs90.


Young92.