Software Reuse Guidelines

Muthu Ramachandran
School of Computing
Leeds Metropolitan University
LEEDS, UK
m.ramachandran@leedsmet.ac.uk

Abstract

In this paper, we discuss the general area of software development for reuse and reuse guidelines. We identify, in detail, language-oriented and domain-oriented guidelines whose effective use affects component reusability. This paper also proposes a tool support which can provide advise and can generate reusable components automatically and it is based on domain knowledge (reuse guidelines represented as domain knowledge).

1. Introduction

Software component reuse is the key to significant gains in productivity. However, to achieve its full potential, we need to focus our attention on *development for reuse*, which is a process of producing potentially reusable components. We know clearly the difficulties that are faced when trying to reuse a component that is not designed for reuse. Therefore, the emphasis of the research described here is on development for reuse rather than *development with reuse*, which is a process of normal systems development (i.e., existing form of reuse). The process of developing potentially reusable components depends solely on defining their characteristics such as language features and domain abstractions. Reuse guidelines can represent such characteristics clearly. Therefore, we need to formulate objective and automatable reuse guidelines.

There have been previous studies on reuse guidelines (Booch 1987; Gautier and Wallis 1990; Braun and Goodenough 1985; Dennis 1987; Hooper and Chester 1991; Tracz 1990; Hollingsworth 1992; Weide et al. 1991; Meyers 2004), but these authors emphasise on general advice including documentation and management issues; their guidelines are sometimes unrealisable and contradictory. More recently, Meyers (2004) has clearly described with examples from C++ components on how best to design components interfaces.

In this paper, we will explore the general area of development for reuse and discuss how we can formulate realisable and objective reuse guidelines. We will also review some of these existing guidelines and present our guidelines. Why do we need such objective and realisable reuse guidelines? They are important for:

- Assessing the reusability of software components against objective reuse guidelines.
- Providing reuse advice and analysis.
- Improving components for reuse which is the process of modifying and adding reusability attributes.

Reuse guidelines can be categorised into many classes (described in a later section, Figure 2). In this paper we mainly describe the following two categories:

1. *Language-oriented reuse guidelines*: Most existing programming languages including object-oriented languages provide features that support reuse. However, simply writing code in those languages doesn't promote reusability. Components must be designed for reusability using those features. Such features must be listed as a set of design techniques for reusability before design takes place.

2. *Domain-oriented reuse guidelines:* Guidelines that are relevant to a specific application domain. We discuss more on this in a later section of this paper.

The language we have chosen for study is Ada, and the application domain chosen is components of abstract data structures (ADS). The main reason for choosing Ada is because of its explicit technical support for reuse, features such as the packaging mechanism, generics, support for abstraction, exceptions, parameterisation, building blocks, and information hiding. The reason for choosing ADS as the application domain is partly because, as computer scientists, we might be considered domain experts ourselves in this area and partly because it has been extensively studied and documented. These components are the fundamental building blocks for many applications.

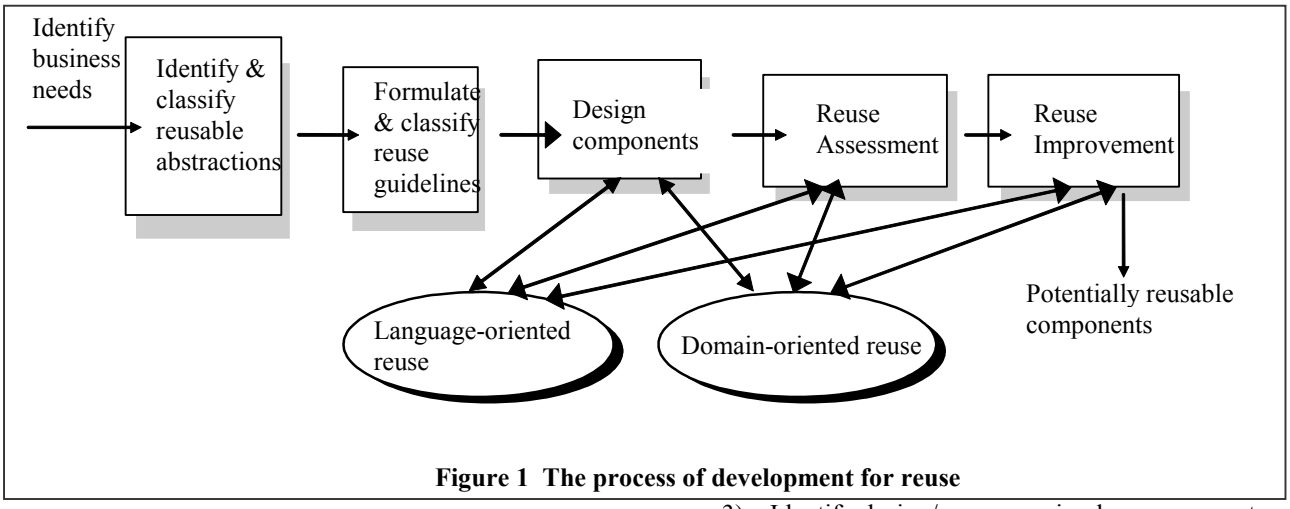

Figure 1 The process of development for reuse

2. Development for reuse.

An objective of the design process is to produce components which are potentially reusable. These components form building blocks for future development (over the long term) and are applicable for various situations and perhaps across application domains.

In development with reuse, reuse is desirable but there need be no resources expended in creating new reusable components. Development for reuse implies expending resources specifically to increase the reusability of components. In many cases, this process might follow development with reuse where components generated during normal system development are made more reusable by generalisation and improvement.

Our notion of the development for reuse process is shown in Figure 1, in which there are a number of stages to be followed which start from identifying an application domain, identify & classify reusable abstractions, domain-oriented reuse, language-oriented reuse, design components, assessment for reuse, improvement for reuse, and deliver potentially reusable components. The idea is to identify a number of frequently reusable domain-specific abstractions (using classification or by interview with domain experts) and then to apply domain-specific and language-specific criteria that are defined by the reuse guidelines.

The development for reuse process falls into a number of stages.

1) Identify domain. Domain analysis has been identified as essential for effective reuse. The first step is to identify a specific application domain and define its boundary.
2) Identify and classify (frequently) reusable abstractions. To identify potentially reusable components, the reuse assessor must know what the important domain abstractions are and how frequently these abstractions are used in systems developed for that domain. There is not much point in devoting a lot of effort in producing a reusable domain abstraction if that abstraction is rarely used. Domain classification helps to identify effective reusable abstractions. This stage involves interviewing domain experts, surveying domain literature and studying existing systems.
3) Identify design/ programming language constructs that support reuse. Selecting an appropriate language is an important part of development for reuse. We should be able to express our reuse guidelines effectively using language mechanisms.
4) Study and formulate language reuse guidelines (rules concerning language support for reuse). This emphasises the effective use of language features for reuse. This process includes studies of existing techniques and appropriate modifications to them.
5) Study and formulate domain reuse guidelines (rules concerning the domain characteristics for reuse). This emphasises the reusable domain abstractions that are identified in the application domain. Guidelines should not just be general advice but should be specific and verifiable for creating potentially reusable components. Design/ redesign components based on these guidelines.
6) The next step, known as reuse assessment is a process of assessing components based on the number of guidelines satisfied against the total number of guidelines that are applicable, and then produce an assessment report. This is where we need to automate this process. The outcome of this process is to make sure that the components designed for reuse satisfy some of the key characteristics.
7) The final step, known as reuse improvement is a process of modifying and improving these components for reuse by adding attributes of an abstraction for reuse. This process is based on the assessment report produced during the previous step. The reuse improver must know what attributes of an abstraction must be generalised to make it reusable. Again, an automatic reuse improvement is essential. Finally, produce potentially reusable components.
8) Automate, where possible, these two processes of assessing and improving components for reuse.

It is unrealistic to expect reusable components to be produced as a side-effect of normal systems development. The reasons for this are partly technical and partly managerial. Technically, the notion of what constitutes a reusable component is not well-understood and engineers working on a project cannot be expected to wrestle with this problem while developing to a given set of requirements. Furthermore, the requirements for a particular project may be such that components have to be very specific in order to satisfy them.

Furthermore, a project manager's principal responsibility is to deliver the required software system on time and within budget. Creating reusable components requires additional effort to be expended which is of no immediate benefit to that project. The project manager cannot reasonably be expected to give reusable component production a high priority.

Thus, we believe that the normal mode of production of reusable components should be to take existing components and to add reusability to them. This extra cost for reuse must be an organisational rather than a project responsibility. Reusability is an attribute which can be added at any level from the specification through to the implementation. In our work, we are principally concerned with the reusability of compilable components. However, we believe that the approach discussed is equally applicable to formal specifications and software designs.

Design for reusable component is dependent on the effective use of the programming language used to implement the component and application domain knowledge. Application domain knowledge allows the abstractions in a domain to be identified and encoded as a set of reusable components. The objective of our work is to use language and domain knowledge to assess, with automatic assistance, the reusability of a component and to suggest to the software engineer how that component may be made more reusable.

3. Reuse Guidelines

Development for reuse requires that the language features must be used effectively. The objective of language-oriented reusability is to exploit the use of language support for reuse and to capture the domain knowledge efficiently. There have been experiments conducted to show that experienced programmers can reuse better than novices (Soloway and Ehrlich 1984). The idea is to formulate a set of objective reuse guidelines (derived from experts, existing systems and literature) which can assists Software Engineers when creating components for reuse. It needs to be like a cook book on software reuse.

The major technical problems of development for reuse are:

- How to identify the characteristics of a reusable component?
- How to assess and improve reusability attributes of a component automatically?
- How to encode and analyse application domain knowledge?

The work described here addresses these problems and hence considers factors affecting reusability such as language factors and domain factors. We believe objective and realisable guidelines will help to solve these problems. Existing studies on creating reusable components (Tracz 1990; Hollingsworth 1992; Weide et al. 1991; Gautier and Wallis 1990; Booch 1987; Dennis 1987; Braun and Goodenough 1985; Meyers 2004) fall into the following classes:

1. *Highly Conceptual* studies which try to be language independent but very abstract. For example, all such studies say reusable components should be:

- Highly cohesive, meaning that they should represent a single abstraction.
- Loosely coupled, meaning that they should be largely independent of any other abstraction.

There are other three such criteria proposed by (Gargaro and Pappas 1987) specifically for Ada programs. A reusable program should be:

- Transportable
- An orthogonal (context-independent) composition, and
- Independent of the runtime system.

More recently, Hollingsworth (1992) proposed a set of discipline for constructing high-quality components:

- Correctness
- Composability
- Reusability
- Understandability

Similarly, Weide at al. (1991) and Tracz (1990) have proposed a framework based on a highly abstract ideas, known as the 3C model:

- *Concept*: a statement of what a piece of software does, factoring out how it does it; abstract specification of functional behavior.
- *Content*: a statement of what a piece of software achieves the behavior defined in its concept; the code to implement a functional specification.
- *Context*: aspects of the software environment relevant to the definition of concept or content that are explicitly part of the concept or content.

These are interesting principles on program design. Our objective was to identify understandable, measurable, objective and automatable reuse guidelines.

Our work has taken the existing studies as a starting point and has attempted to produce a scheme for classification and to produce more detailed and practical guidelines on the way in which language and domain features affect reusability. Our main aim was to formulate reuse guidelines that are practical and objective (as much as possible), domain-specific, comprehensive, classified, support design for reuse, and automatable. Figure 2 shows a classification mechanism for reuse guidelines.

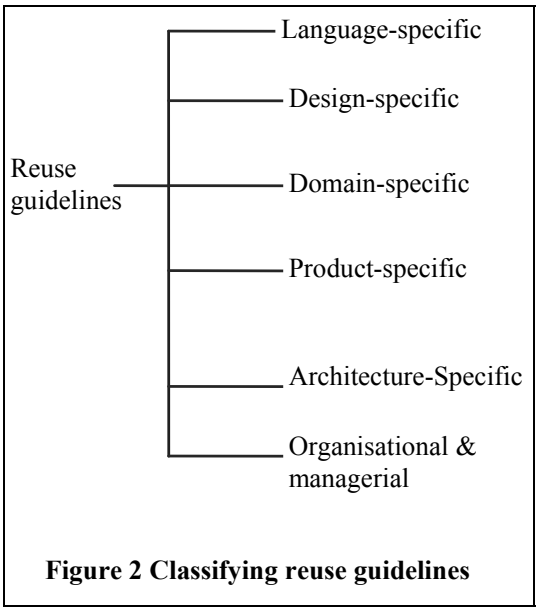

Figure 2 Classifying reuse guidelines

Reuse guidelines are classified into:

- Language-specific which deals with language support for reuse. How to make use of language features effectively. For example when to use private and public in most of the current OO languages.
- Design-specific deals with design principles that support reuse such as OO and other component paradigms like COM/DCOM/.NET/EJB.
- Domain-specific deals issues on how to identify and classify components for a specific application domain.
- Product-specific deals with ad hoc reuse guidelines emerging from experience for a specific product or a product line.
- Architecture-oriented deals with how to design an architecture which supports reuse explicitly. This includes various types of architecture.
- Organisational & managerial deals with how to set up a reuse programme and how to train and motivate engineers on reuse.

4. Domain-Oriented Reusability

Domain analysis and modelling deal with identifying reusable abstractions and architectures for the development of a **family of software systems** as opposed to the traditional system analysis methods and knowledge-based systems that have concentrated on developing a single and specific problem. Domain modelling has been widely studied in recent years and it plays a major role into software reusability research. For example, the Draco system (Neighbors 1984) is based on the domain analysis towards constructing software parts from existing components.

Neighbors (1984) points out that "the key to reusable software is best captured in domain analysis in that it stresses the reusability of analysis and design, not code". The Draco system has a domain language for describing programs in each different problem area. A domain analyst represents analysis information about a problem domain in terms of objects and operations in a domain language. A domain designer specifies different implementations for these objects and operations in terms of the other domains already known to Draco.

Other interesting approaches include, CAMP (1987) project on missile application domain, Booch's (1987) work on designing reusable components of abstract data structures, and more recently Maiden and Sutcliffe's (1992; 1993) work on reuse of requirements specification and architectures based on analogy and examples. There are a number of other approaches stated in Prieto-Diaz and Arango (1991) and Schafer et al (1994).

Wartik and Prieto-Diaz (1992) provides a detailed account on comparing various approaches to domain analysis based on a number of criteria. Most of these approaches to domain analysis (McCain 1985; Prieto-Diaz 1990; Lubars 1991; Simos 1991; Moore and Bailin 1991) consider organisational issues such as business analysis, infrastructures, workproducts, data collection and analysis, and classification rather than specific technical problems. Among these approaches, commonalties are stronger than differences. These are based on informal techniques using ad hoc approaches.

A conclusion is that neither Neighbors' nor other existing works address the issues of “how to conduct a domain analysis process and how to identify reusable components”. The success of domain-oriented systems depends on the evolution of new techniques to do the domain analysis and modelling the system. We need to address the following issues:

- How to identify frequently reusable abstractions?
- What are the domain roles?
- How to classify the application domain?
- What is the best representation technique?

In this research, we have tried to bridge the gap between application domain knowledge and language knowledge. The idea here is to use reuse guidelines for knowledge representation, and to provide analysis and advice on reusable abstractions in the domain of abstract data structures. Our approach is to support development for reuse encoding the application domain knowledge and language knowledge in the form of reuse guidelines.

In our approach to domain analysis, we have identified the following guidelines when supporting to identify frequently reusable abstractions:

- A specific set of domain roles.
- Practical and objective reuse guidelines to represent the application domain knowledge and language knowledge, and to provide reuse analysis and advice.
- A rule-based approach for the domain representation.
- Methods for assessing and improving components for reuse.

The domain-oriented system should support the following roles:

1. *Identifying the abstractions that exist in a domain.* To identify potentially reusable components, the reuse assessor must know what the important domain abstractions are and how frequently these abstractions are used in systems developed for that domain. This will help the designer avoid producing a component that is rarely used.

2. *The attributes of an abstraction for reuse.* Advice is necessary on what attributes of an abstraction must be generalised to make it reusable. The domain analyser and improver must know the attributes to be generalised within that domain so that it enhances reusability of that abstraction. For example, if a component of a dynamic abstract data structure is to be generalised then the system should check for generic abstraction.

3. *Advice on structural information*. It is necessary to provide advice on structural information on existing abstractions and on newly required abstractions. For example, it is not always clear how to select the most suitable abstract data structure for a specific application, and how to hide representation details.

4. *Advice on the usage and applications of existing domain abstractions.* It is always difficult for the reuser to understand how a particular abstraction can be reused and what are the possible applications. For example, it is not always clear to component reusers what are the possible applications of a selected abstraction. Therefore, the domain analyser must know to advise on how to reuse and what are the possible applications of an abstraction.

5. *Reuse assessment and improvement.* The domain system should analyse and assess components against reuse guidelines that are represented and should report the percentage of matching guidelines, so that the designer is aware of his component's potential for reuse. Also it should provide suggestions on how that abstraction can be improved for reuse automatically. Assessment reports can be produced based on the grading system introduced earlier, a component is weakly/ limitedly/ strongly/ immediately reusable.

4.1 Domain classification

Domain classification is an important and difficult part of modern domain engineering. It helps to identify effective reusable abstractions and model the problem domain. Booch (1987) has proposed a classification scheme, known as Booch's components. In his scheme, components are classified into *structures, tools, and subsystems*. He has characterised a structure as an ADT (abstract data type) or ASM (abstract state machine). Most of the ADS are considered as monolithic or polylithic components. Monolithic components are stacks, strings, queues, dequeues, rings, maps, sets, and bags. Polylithic components are lists, trees, and graphs. Tools are utilities, filters, pipes, sorting, searching, and pattern matching. Again these are further classified into various *forms of a component*, which represent variations on the theme of components for differences on time and space requirements. The forms are sequential, guarded, concurrent, and multiple.

Booch's work has been used as a starting point for constructing reusable components. However, his notion of forms represents only minor variations in implementation and is cumbersome for the reuser to choose a particular implementation because there are too many variants. For example there are more than twenty-six variant forms of stack components.

Our objective is to formulate realisable domain reuse guidelines to represent the design of reusable components of abstract data structures (ADS). These reuse guidelines are kept as general as possible, and not specific to any particular language, but specific to this domain of ADS. The main purposes of these guidelines are firstly, to support development for reuse in the application domain of ADS. Secondly, to estimate the reuse potential of a program automatically, and thirdly, to improve components for reuse by representing these guidelines within this domain. Domain reuse guidelines are based on a proposed classification scheme.

In our work, we have proposed a classification scheme for the domain of abstract data structures (ADS) as shown in Figure 3. In this scheme, ADS have been classified into sequential and concurrent structures. The sequential structure is further classified into linear, and non-linear structures. An important further sub-classification is static and dynamic abstractions which can be kept together as a single abstraction. This classification is important for the following reasons.

- Guidelines that have been formulated refer to specific parts of the classification structure, mainly sequential structures.
- Sub-classification is limited to static and dynamic structures which are single, generalised, and easy to reuse.
- A single and generalised abstraction is more reusable than an abstraction with several versions, which are called forms in Booch's components (Booch 1987).
- The domain boundary is clearly defined which is important to do domain analysis effectively.

Booch's sub-taxonomy needs further refinement and his classification scheme is far too general (structures, tools, and subsystems) which makes the domain boundary and scope undefined and divergent. Also there are good reasons for keeping abstractions together rather than having several versions (or forms) for each minor variation. It may be difficult for the reuser to understand each of these minor variations before reusing a component. For example, Booch's notion of bounded and unbounded components should always be designed as manageable. In our work on domain analysis, support is provided in identifying frequently reusable abstractions.

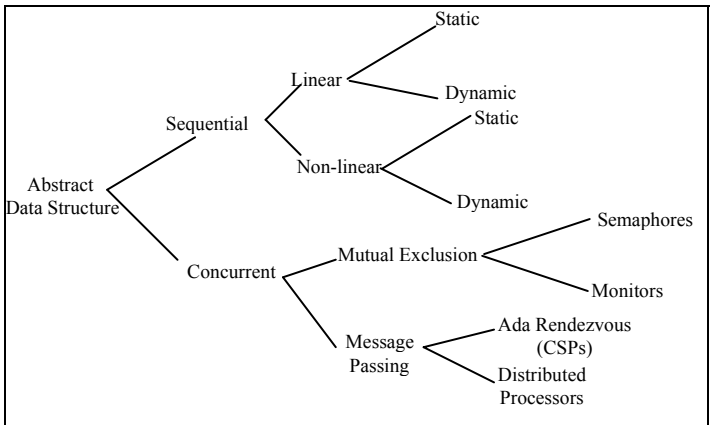

Figure 3: Classification of Abstract Data Structures

4.2 Domain reuse guidelines

As mentioned earlier, our objective is to produce a set of objective and practical domain reuse guidelines which can be

applied systematically to improve reusability. Ideally, we would like their expression and their application to be so systematic that it can be completely automated.

Compared to some of the existing guidelines (Hollingsworth 1992; Weide et al. 1991; Gautier and Wallis 1990; Booch 1987; Braun and Goodenough 1985) discussed in the earlier section on reuse guidelines, our guidelines are domain specific, classified, and objective. Our domain reuse guidelines fall into a number of classes based on the domain classification:

- Design of abstract data types
- Design of interfaces
- Design of static structures
- Design of dynamic structures
- Design of concurrent structures
- Design of space management

1. *Design of abstract data types*. The notion of an abstract data type allows you to express real world entities of an application domain. It allows you to separate a specification from an internal representation of a structure (principle of information hiding). It means that we are able to specify an abstraction of a component in terms of its actual interface descriptions together which is useful to generalise that abstraction for reuse. It allows the designer to view a system at a more abstract level and to change the representation of ADS without affecting their use in other parts of the system.

One of our guidelines on ADS states that,

```
For all complex structures, provide two
representations such as static and dynamic
structures for each domain abstraction.
```

This guideline says, for each structure, provide two abstractions such as static which is represented using an array structure and dynamic which is represented using dynamic structure (access/pointer). This provides a choice and maximum flexibility for the reuser with improved reuse potential. For example, in Ada, we can design two packages for each structure implemented statically and dynamically. If an abstraction is to be represented in Ada then we can apply various Ada reuse guidelines. For example, one on the rationale for choosing private types. That is, choose limited private for complex and dynamic structures, and choose private type for static structures. However, the Ada library mechanism is inadequate in that it rises naming conflict when there are two library units with similar names which means that the implementation of similar components must have different names.

Another important guideline (Braun and Goodenough 1985) on the design of abstract data structures emphasises the need for providing methods for a list of operations such as object creation, object termination, state change, state inquiry, and input and output. They have not considered operations on exceptions that deal with error conditions. We believe that the operations on exceptions and handling are significant for reusable and reliable components. In our work we have extended this guideline to include operations on exceptions handling.

Our extended guideline on ADS states that a reusable component should be provided with the following functions.

- Creation
- Termination
- Conversion
- State inquiry
- State change
- Input/ output representation, and
- Exceptions

Creation involves both creating and initialising an object, termination is a means of making the object inaccessible for the remainder of its scope, conversion allows for the change of representation from one type to another, state inquiry functions allow the user to determine the state of the object and boundary conditions, state change functions allow modifying or changing the contents of the object, input/ output representations are primarily useful for debugging purposes, and exceptions deal with error conditions and exception handling procedures. Each operation emphasises one or more functionality so that the services offered by the component are increased thus leading to improved reusability. Sometimes components which do not provide all these operations may well be reused. In such cases, the component has to be measured based on the degree of reusability.

2. *Other guidelines*. Our guidelines on the design of reusable static and dynamic structures, and on space management are essential, objective and realisable. Some of our important domain guidelines are,

- Always, define a constrained array structure to represent a component of static structure.
- Always select dynamic object representation for all complex structures and hide detailed structural information.
- If the abstract structure is complex and all operations are independent of the type of the structure element then that component should be implemented as a generic package with the element type as a generic parameter.
- Always provide a procedure to record the maximum size of the free list with a counter so that the user may increase or decrease the size of the free list. when decreasing the free list size, space in excess of the new size is returned to the system.
- Always provide a procedure to release the free list, so that all space in the free list is returned to the system completely
- For each exception, provide an exception handler.

5. Automation

The guidelines discussed in this paper have been partially or completely automated in our system for which a prototype has been developed as shown in Figure 4. Some of them involve straightforward transformation and others might need user interaction and domain knowledge. This system takes an Ada component, checks through various reuse guidelines that are applicable, provides reuse advice and analysis to the reuser, and generates that component which is improved for reuse. Ada components are modelled using component templates and reuse guidelines are checked objectively against that template. Some

of these domain reuse guidelines have been represented and analysed using component templates. For most of these guidelines, automation depends on some user interactions and domain knowledge.

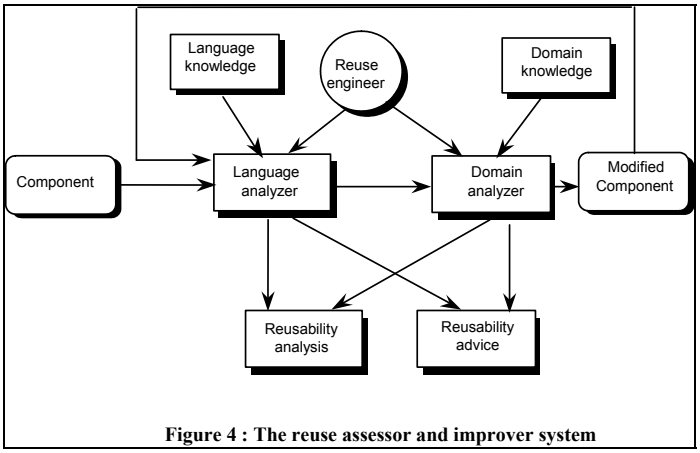

Figure 4 : The reuse assessor and improver system

One of the major objective of this system is to demonstrate, how well-defined reuse guidelines can be used to automate the process of reuse assessment by providing support for language analysis and domain analysis. For example, this system takes an Ada component specification, assesses it through two analysis phases, estimates its reusability according to how well it satisfies a set of reuse guidelines and generates a component which is improved for reuse.

The system interacts with the engineer to discover information that can't be determined automatically. The conclusion of this first pass is an estimate of how many guidelines are applicable to the component and how many of these have been breached. The report generator produces a report with all the information that has been extracted about that component and changes that have been made for reuse.

The second pass involves applying domain knowledge to the system. The component templates have been modelled representing static and dynamic structures. Their reusability is assessed by comparing the component with that template. The support provided by the system ensures that the reuse engineer carries out a systematic analysis of the component according to the suggested guidelines. He or she need not be a domain expert. Again, an analysis is produced which allows the engineer to assess how much work is required to improve system reusability.

For example, a scheme for automating one of our domain guideline is shown algorithmically in Figure 5. This scheme involves identification of procedures and domain related information against a component template, and adds operations automatically to those components with perhaps some human assistance.

```
For each ADS do
  Analyse abstractions and advice;
  Identify attributes for static and dynamic
        structures with user interactions;
  Identify procedures against Braun and
        Goodenough's list;
  Add operations that are missing;
  Follow further schemes for Static and/ or Dynamic
        Structures;
  Produce reports and potentially reusable
        components;
End loop
```

Figure 5: Scheme for automating domain guidelines

Guidelines for automation are represented in two distinct ways:

- Wherever possible, a rule-based representation is used so that it is clear when a guideline should be applied. We have found that rule-based representations are mostly applicable for language-oriented guidelines.
- For domain-oriented guidelines, we are mostly concerned with checking that a component fits a model of a reusable domain abstraction. In this case, we have developed templates of these abstractions which represent the reuse guidelines.

However, it remains to see how many numbers of guidelines are significant for reuse, and further investigation is underway to improve its limitations. The system has demonstrated that it is possible to formulate and automate practical and objective reuse guidelines supporting the development of potentially reusable software components.

6. Conclusion

Reusable components can be produced and re-engineered effectively in a large scale if we can formulate objective and realisable guidelines and apply them systematically. We took the existing work on reuse guidelines as a starting point and made possible to use it for automation. Domain analysis can play a major role in supporting development for reuse. we have proposed a classification scheme for guidelines and for the domain of ADS components. We also believe that our approach is applicable to other languages, methods, tools, and application systems.

Acknowledgement

The author wish to thank Prof Ian Sommerville, Lancaster University for providing valuable comments on this paper during this work was carried out.

7. References

Biggerstaff, T.J. and Perlis, A.J. (1984), "Foreword to the special issue on software reusability", IEEE trans. on software engineering, September.

Biggerstaff, T.J. and Perlis, A.J. (Editors) (1989), "Software Reusability: Concepts and Models", Vol.I & II, ACM Press, Addison-Wesley.

Booch, G. (1987), "Software Components with Ada", Benjamin/Cummings.

Bott, M.F. and Wallis, P.J.L. (1988), "Ada and software reuse", Software Engineering Journal, September.

Braun, C.L. and Goodenough, J.B. (1985), "Ada Reusability Guidelines", Report 3285-2-208/2, USAF.

CAMP (1987), "Common Ada Missile Packages", Final Technical Report, Vols. 1, 2 and 3. AD-B-102 654, 655, 656, Airforce Armament Laboratory, FL.

Carter, J.R. (1990), " The Form of reusable Ada Components for Concurrent Use", Ada Letters, vol.X, No.1, Jan/Feb.

Dennis, R.J.St. (1987), "Reusable Ada (R) software guidelines", proc. of the 12th annual Hawaii International conference on system sciences, pp.513-520.

Gargaro, A. and Pappas, T.L. (1987), "Reusability issues and Ada", IEEE software, pp.43-51, July.

Gautier, R.J. and Wallis, P.J.L. (Editors) (1990), "Software Reuse with Ada", Peter Peregrinus Ltd for IEE/BCS.

Genillard, C., Ebel, N., and Strohmeier, A. (1989), "Rational for the design of reusable abstract data types implemented in Ada", Ada letters, vol.IX, No.2, March/April.

Hall, P. A. V., (1993) Domain analysis, Walton, P and Maiden, N (Editors) Íntegrated Software Reuse: Management and Techniques, Ashgate Publishers.

Hollingsworth, J (1992). Software components design for reuse: a language independent discipline applied to Ada, PhD thesis, Dept. of computing and Information, Ohio State Univ., Columbus, December.

Hooper, J. W. and Chester, R. O. (1991). Software Reuse: Guidelines and Methods, Plenum Press.

Keenan, P. (1987), "Reuse of Designs as a First Step Towards the Introduction of Ada Component Reuse", IEE Colloquium on Reusable Software Components, May.

Krueger, C (1992) Software Reuse, ACM Surveys, Vol. 24, No. 2, June 1992.

Latour, L. (1991), " A methodology for the design of reuse engineered Ada components", Ada Letters, spring.

Lubars, M. (1991), Domain analysis and domain engineering in IDeA, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

Maiden, N A M and Sutcliffe, A G (1992) Exploiting reusable specifications through analogy, Communications of the ACM 34(5), May, 1992.

McCain, R. (1985), "Reusable Software Component Construction: A Product Oriented Paradigm", In Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference, Long Beach, CA, 125-135, October 21-23.

Moore, J M and Bailin, S C 1991. Domain Analysis: Framework for reuse, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

Meyers, S (2004) The Most Important Design Guideline? IEEE Software, July/August 2004

Neighbors, J.M. (1984), "The Draco Approach to constructing Software from reusable components", IEEE Trans. on Software Engineering, vol.SE-10, No.5, pp.564-574, September.

Prieto-Diaz, R and Frakes, W. B (1993) Advances in software reuse, Proc. of the second international workshop on software reusability (IWSR-II Lucca, Italy, March 1993) IEEE Computer Society Press, March 1993.

Prieto-Diaz, R. (1990), "Domain Analysis: An Introduction", ACM SIGSOFT, Software Engineering Notes, vol 15, no.2, Page 47, April.

Prieto-Diaz, R. and Arango, G (1991) Software Modelling and Domain Analysis, IEEE Computer Society Press Tutorial.

Ramachandran, M. (1994) Knowledge-based support for reuse, Proceedings of Intl. conf. on software engineering and knowledge engineering (SEKE94), Latvia, July.

Ramachandran, M. and Sommerville, I. (1995) A framework on automating reuse guidelines, Proceedings of Intl. conf. on software engineering and knowledge engineering (SEKE95), USA.

Schafer, W., Prieto-Diaz, R., and Matsumoto, M. (1994). Software Reusability, Ellis Horwood.

Simos, M. (1991), The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse, Prieto-Diaz, R and Arango, G (ed) Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial.

Soloway, E and Ehrlich, K. (1984), "Emprical studies of programming knowledge", IEEE Transactions on Software Engineering, Vol. SE-10, No.5, September.

Sommerville, I. and Morrison, R. (1987), "Software Development with Ada", Addison-Wesley.

Sommerville, I. and Ramachandran, M. (1991), "Reuse Assessment", First International Workshop on Software Reuse, Dortmund, Germany, July.

Tracz, W. (1990), "The 3 Cons of Software Reuse," in the proceedings of the *Third Annual Workshop on Software Reuse*, July, Syracuse, NY.

Wartik S and Prieto-Diaz, R. (1992), Criteria for comparing reuse-oriented domain analysis approaches, Intl. J. of Soft. Eng. and knowledge Eng., Vol 2, No. 3.

Weide, B.W et al. (1991) Reusable software components, Advances in Computers, Yovits, M. C (ed.), Vol. 33, Academic Press.