Software Component Reuse and OOAD (Object Oriented Analysis and Design)

Dinar Mutiara Kusumo Nugraheni, ST, MInfo Tech(Comp)
Program studi Teknik Informatika, Fakultas MIPA, Universitas Diponegoro
dinar.mutiara@undip.ac.id

Abstract

In relation to software component reuse and object oriented analysis design (OOAD), this paper presents an overview of software reuse, including the history of software reuse, current state, development and benefit and constraints. On the second part of the paper, a discussion of the relevance of software component reuse and Object Oriented and Analysis Design in terms of opportunities and challenge is given. After that, there is a provision of some general guidelines on when and how reusable components could be identified and developed. Toward the end, the paper discusses strategies and future directions of software engineering in relation to software component reuse.

Keywords: Software, reuse, OOAD

1. Introduction

Schach (2005) ilustrated the developing of COBOL payroll programs all over the world are doing essentially the same thing. However, most of them are built by the developers from scratch. If the developers utilize previously existed payroll programs, and make the program can run on a variety of hardware and adapted to fulfil specific needs of an individual organization, the effort to build payroll program would be less painful. This is the illustration is given to describe the role of component reuse in easing the effort to develop a new system.

According to Kruger (1992) software reuse is the process of creating software systems from existing software rather than building software systems from scratch or reuse is a process of implementing or updating software systems using pre-existing software development assets. The most common type of reuse is the reuse of software components, but other artifacts produced during the software development process can also be reused: system architectures, analysis models, design models, design patterns, database schemas, web services, etc

In relation to software component reuse and object oriented analysis design (OOAD), this paper presents an overview of software reuse, including the history of software reuse, current state, development and benefit and constraints. On the second part of the paper, a discussion of the relevance of software component reuse and Object Oriented and Analysis Design in terms of opportunities and challenge is given. After that, there is a provision of some general guidelines on when and how reusable components could be identified and developed. Toward the end, the paper discusses strategies and future directions of software engineering in relation to software component reuse.

2. Motivation of Software Reuse

According to Kruger (1992) the primary motivation to reuse software is to reduce time and effort which is required to build software systems. If we read to the history, software reuse is triggered by many problems in software development (software crisis).

First problem is occurs that many software projects ran over budget and schedule, for example is the OS/360 operating system, which was a classic example. This decade-long project from the 1960s eventually produced one of the most complex software systems at the time. D.A.Jardine (1972) noted OS/360 was one of the first large (1000 programmers' software projects).

Secondly, some projects caused property damage. This is related to security issues. Poor software security allows hackers to steal identities, costing time, money, and reputations. For example are the web 2.0 security issues. Brian Chess, chief scientist and a founder of Fortify, a security company, have reported a new wave of Internet attacks targeting Web 2.0 sites and the Ajax applications that have helped make them so dynamic. Chess said his researchers analysed the 12

most popular Ajax frameworks, including ones from Google, Microsoft, Yahoo, and the open source community. Researchers found that only Direct Web Removing 2.0, which is an open source framework, builds security around JavaScript, protecting it from attack. Chess estimates that 75% of Ajax applications are written using these frameworks and the other 25% are home brewed or simply coded from the ground up. The straight coding also is probably at risk since some programmers might not know they need to build in specific JavaScript security.

Another problem in software is can defect in human life. The most famous of these failures is the *Therac 25 incident*. Some used the term software crisis to refer to their inability to hire enough qualified programmers.

2.1 History and current situation

The history of software component reuse began in 1968 where NATO Software Engineering Conference is also considered as the birthplace of the software engineering field (Krueger, 1992). Problems associated with software began to receive wide notice in the late 1960s. Software is often late, over budget, and fails to perform as expected. This set of problems has come to be known as the "software crisis". It is explained that the conference focused on the software crisis, related to the problem of building large, reliable software systems in a controlled and cost-effective way. From this point, software reuse has been expected as a means for overcoming the software crisis.

Furthermore, in 1968, the IEEE sponsored the first International Conference on Software Engineering, at Garmisch, West Germany. It was at this conference that the need for software engineering as an autonomous discipline was first recognized (Randell, 1996). The discipline that would be imposed on the software development process by engineering software rather than just creating it would lead to solutions for the problems of the software crisis.

In addition, it is said that the key point of software reuse notion was the seminar paper invited at the conference: Mass Produced Software Components by Mclhroy (1968). The writer proposed a library of reusable components and automated techniques for customizing components to different degrees of precision and robustness. He felt that component libraries could be effectively used for numerical computation, 1/0 conversion, text processing, and dynamic storage allocation. Krueger said that until more than twenty three years later, many computer scientists still see software reuse as potentially a powerful means of improving the practice of software engineering. The advantage of paying back software development efforts through reuse continues to be widely acknowledged, even though the tools, methods, languages, and overall understanding of software engineering have changed significantly since 1968.

However, later it is found that software reuse has failed to become standard practice for software construction. This was not in accordance to its promise. The positive side of this failure is the computer science community interested to renew their understanding on how and where reuse can function effectively. Also, the scientist interests to find out why it has been difficult to bring the idea that sound simple of software reuse to the forefront of software development technologies.

The current proof that the idea of software component reuse benefits developers can be seen form the application of Java programming language that enables "reuse" in it API. This helps programmers in building their program without needing to invent the same wheel to the same problem or methods that have been done by their predecessors.

2.2 Benefits of software reuse

Moore (2000) noted that advantages of software reuse is cost saving. He admitted that reuse cost is much less than constructed from the scratch.

Anderson (2004) in his lectures noted mentions some of the benefit in reuse software:

- The first benefit is efficiency. It is means that can reduces time to spend in designing or coding. For instances, it can be seen in the pursued of software reuse of the Advanced Field Artillery Technical Data System (AFATDS) project. They used object-oriented design techniques. In this project, they managed to achieve the result of 13% (1OOK out of 770K) of their total code being reused code. Of this amount, 3/10 was reused as is, while 7/10 of the code had to be tailored upon reuse. The net cost savings to the project is estimated at 4%.(Moore, 2000)

- Secondly, it is profitable because reuse can lead to a market for component software. Another benefit is in debugging. The reason is reused design/code is often tested design/code.

Fichman (2001) divided the benefit of software reuse in two perspectives.

- First is from developer's perspective
  The benefits for the developers are added revenue due to income from selling reusable information and added revenue from fees or royalties resulting from redistribution of information. The other benefits are added revenue due to delivering product sooner to market place, reduced maintenance costs, added revenue due to improved customer satisfaction with product quality, reduced cost of tools, equipment and reduced cost to manage development and test.

- The other is from the consumer.
  As for the user benefits are reduced cost to design, to document, to implement, to unit test, to design tests, to document tests, to execute testing and reduced cost to product publications.

The reuse of components at different design levels is an important basis for a rapid, inexpensive and correct design of complex system" (Bottger, 1998). Beside that, it is proven that reusable components are easier to maintain over time and typically have a higher quality value meaning it is more robust and causes fewer errors.
The practice of component reuse supported the proof of the benefits it offers, is described by (Kuhns, 1998) in the case of GIS applications. The real benefits of the application of component reuse in this particular case are as stated as follows:

- reducing time consume to develop the system
- reducing the effort
- Lessen the cost needed in building application, and
- Improving the quality of the implementation.

2.3 The Constraints in implementing software reuse

Schach (2004, pp.274-275) discusses a number of challenges to reuse: First is some of professional programmers are prefer to rewrite an artifact from scratch than reuse an artifact that written by someone else. The reason is that an artifact can not be good unless they wrote it by themselves. This phenomenon is known as the *not invented here* (NIH) syndrome. NIH is a management issues and if management is aware of the problem. It can be solved, usually by offering financial incentives to promote reuse.

A second challenge is many professional programmers would be willing to reuse an artifact. However, they need to know the quality the artifact. This attitude is easy to understand. After all, every professional programmer has seen the faulty software that written by others. The solution here is to subject potentially reusable artifacts, especially code modulus, to exhausting testing before making them available for reuse.

The third challenge is still problematic. This issue arise with a contract (software can developed by an outside organization that specializes in developing such as information system). In terms of type of contract usually drawn up between a client and the information system developers, the information system is belonging to the client or the developers. Therefore, if the information system developers reuse artefacts of one client's information system while developing an information system for a different client, this constitutes robbery of the first client's intellectual property. There are no problems, when the developers and client are members of the same organizations, this problem does not arise.

Nevertheles, as we know large organization can have hundreds of thousands of potentially useful artefacts. The challenge is how should these artefacts be stored for effective later retrieval? For example, a reusable artefacts database might consist of 20, 00 items, 125 of which are sort routines. The data base must be organised so that the designer of new software can determine which of those 125 sort routines is appropriate

In addition, Kruger (1992) and Anderson (2004) mention some constraint in implementing software reuse is useful abstractions for large, complex, reusable software artifacts will typically be complex. In order to use these artifacts, software developers must either be familiar with the abstractions must take time to study and understand the abstractions.

3. Relevance to OOAD

Object-oriented programming is becoming a popular approach to the construction of complex software systems. The basic idea of the object oriented is to structuring software around "modules" representing "real world" (Qin, 2007).

The relevance of software component reuse can be seen from the lifecycle of object oriented software involving reusable components library as shown in the picture below

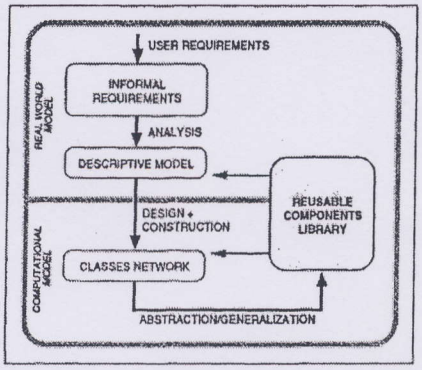

Figure 1: the Object oriented software lifecycle (nerson, 1992).

3.1 Opportunities

Danforth (1988) noted that the benefits of object orientation include support for modular design, code sharing, and extensibility.The notion of inheritance from the object-oriented paradigm, however, offers a unique contribution to component reuse.

He explains that class definitions in an object-oriented language are primarily a data abstraction mechanism. Inheritance and subclassing enhance the data abstraction mechanism by establishing a hierarchical relationship among classes and it is allow reusing within the class hierarchy. It same as Qin (2007) explained in her lectures that inheritance is defining new data type as an extension of an existing type. In addition, she also mentions that polymorphism and dynamic binding is permitting a method to be applied to objects of different classes by having system determine.

Inheritance (also known as generalization) is an easy way to implement polymorphism and has been used as the primary mechanism for reuse in modern object oriented languages. This is unfortunate, as inheritance imposes a rigid structure on the software's design that's difficult to change. Any inheritance hierarchy that shares code from parents to children will have problems when it grows to be three or more levels deep. Too many exceptions occur to maintain a pure "is-a" relationship between parents and children, where children are always considered to have all the properties and behaviours of the parents. Inheritance should only be used to share definitions (interfaces), not implementations. This practice has emerged as a first-order object oriented design principle. Whenever inheritance is used, only the last child class (leaf node) of the inheritance hierarchy should be instantiated. All parent classes should be abstract and should never be instantiated. This is because a class that tries to be both reusable and concrete to provide reusable and specific behaviour at the same time almost always fails to do either. This is a dimension of cohesiveness. One thing that makes a class cohesive is that it's dedicated to reuse or dedicated to a specific purpose, but not both.

Aggregation is a technique that collects or aggregates functionality into larger elements of functionality. It provides a structure that's far more flexible and reusable than inheritance. It's better to reuse implementation and design by aggregating small pieces of functionality together rather than trying to inherit the functionality from a parent.

In addition, Lewis (1991) explains the benefit in his research *An Empirical Study of the Object-Oriented Paradigm and Software Reuse.* The benefits are the object-oriented paradigm substantially improves productivity, although a significant part of this improvement is due to the effect of reuse. Secondly, Software reuse improves productivity no matter which language paradigm is used. Thirdly, using differences language are far more important when programmers reuse than when they do not), and the object-oriented paradigm has a particular affinity to the reuse process.

3.2 Challenges

In terms of challenges face by the application of software component reuse in object oriented design, Schmidt discuss a number of factors are responsible for the lack of widespread software reuse. They are stated as following:

- "**Organizational factors** which related to effort required to catalog, archive, and retrieve reusable components;
- **Economic factors** such as the lack of adequate chargeback schemes and monetary incentives in many development organizations;
- **Political factors** such as not wanting to share components with rival groups;
- **Psychological factors** such as perceived threat to job security and the ubiquitous ``not invented here'' syndrome".

Another challenge mentioned for developers and analysts is translating their domain expertise into reusable software components. To deal with this challenge, some approach such as transformational systems, expert systems and domain specific software architecture have been done.

In addition, traditional approaches to reuse based on automated domain analysis have often ignored fundamental challenges in large-scale software system development. These challenges include communication of architectural knowledge among developers; accommodating new design paradigms or architectural styles; resolving non-functional forces such as reusability, portability, and extensibility; and avoiding development traps and pitfalls that are usually learned only by experience.

4. General guidelines on identification and development of reusable component

According to Ramachandran (2005), reuse guidelines is used to represent characteristics that needed to create a potentially reusable components. Therefore, an objective and realisable guidelines are important, especially for:

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

He also categorizes guidelines into many classes. They are as following:

1. *Language-oriented reuse guidelines.*
   Most existing programming languages including object-oriented languages provide features that support reuse. However, it is assert that simply writing code in those languages does not promote reusability. The fact is 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.*
   This category of guidelines is relevant to a specific application domain.

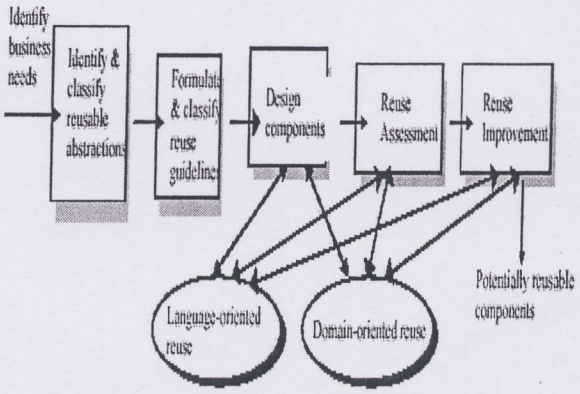

Figure 2: the process of development for reuse (Ramachandran, 2005)

The development for reuse process specify as the stages explained below:

a. Identify domain.
   This includes the identification of a specific application domain and defines its boundary. This is important to do because domain analysis has been identified as essential for effective reuse.

b. Identify and classify reusable abstractions.
   The domain abstraction is important to known by assessor as well as to identify how frequently these abstraction are used in systems develop for that domain.

c. Identify design and programming language constructs that support reuse.
   Selecting an appropriate language is an important part of development for reuse.

d. Study and formulate language reuse guidelines (rules concerning language support for reuse).
   This emphasizes the effective use of language features for reuse. This process

includes studies of existing techniques and appropriate modifications to them.

Furthermore, 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. The idea is to formulate a set of objective reuse guidelines which can assist Software Engineers when creating components for reuse.

Ramachandran points the major technical problems of development for reuse are by asking some question like, "how to identify the characteristics of a reusable component? How to assess and improve reusability attributes of a component automatically?, and, How to encode and analyse application domain knowledge?"

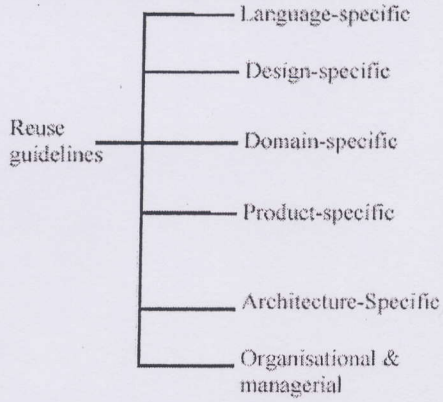

Figure 3: Classifying Reuse Guidelines

5. Reuse and the future of Software Engineering

According to Schmidt (1996), over the next few years, experts anticipate that a wealth of software design knowledge will be captured in the form of patterns and frameworks. These patterns and frameworks will span domains and disciplines such as concurrency, distribution, organizational design, software reuse, real-time systems, business and electronic commerce, and human interface design. Those expectations divided into some aspects of patterns and frameworks that will receive particular attention:

a. Integration of design patterns with frameworks and other design paradigms

It is believed that the most crucial challenge that is faced by framework developers is determining which components in a framework should be variable and which should be stable. Insufficient variation makes it hard for users to customize framework components. This results in a framework that cannot accommodate the requirements of diverse applications. Conversely, insufficient stability makes it hard for users to comprehend and reply on the framework. Inflexibility and instability can create a framework that is awkward to use and unable to satisfy other requirements, for example, such as run-time performance.

b. Integration of design patterns to form pattern languages

Pattern language can be formed as the result of the increasing integrates groups of related pattern. This pattern defines architectural styles that guide designers as they weave patterns to build entire systems. In addition, a pattern language may generate a software system based on architectural style such as real-time, business, or electronic commerce or it may guide any system endeavour, including organization and process, human interface design, and teaching. Finally, pattern languages support larger-scale reuse of software architecture and design than do individual patterns. Developing pattern languages is challenging and time-consuming, but it is believed that they will ultimately provide the greatest payoff for developing high-quality software.

c. Integration with current software development methods and software process models

Patterns can help developers navigate abstraction boundaries *across* software development phases. For instance, patterns help to bridge the abstractions in the upstream phases such as domain analysis and architectural design with the concrete realizations of these abstractions in downstream phases such as implementation and maintenance. Another points noted is that patterns and pattern languages that exist do not yet form a comprehensive software development method or complete process guide. However, they do complement

existing approaches by focusing on non-functional forces such as backwards compatibility or architectural extensibility that are often not addressed by conventional development methods and processes.

Furthermore, IEEE predicts in the future software reuse is likely to have different reaction. The next form of reuse will be the key enabler of the world trade of software via the World Wide Web. Reuse via the Web has already captured the imagination of the software industry and business community at large.

The next form of reuse centers on components and component-based development. Component is expected to be the primary driver of the dramatic changes about to take place in software development. Components lie at the very heart of the future vision of computing. Corporations expect that they soon will be running their businesses using Web-enabled enterprise business applications composed from predefined, reusable, and replaceable components distributed over networks. Although part of the application may run on a client, part on the middle-tier, and another part on a backend database server, its comprising components written in different languages and supplied from multiple sources will work together to perform the application's services.

Component-based applications offer the advantages of being both easily customized to meet current business needs and easily modified to meet changing business needs over time. Also, they leverage a corporation's investment in its legacy systems by containing valuable existing functionality wrapped into reusable components.

Thus, component-based applications are likely to be composed of an interacting mixture of pre-developed components that preserve the business' core functionality and new components that take advantage of the newest technologies, such as the Internet. Today, examples of components include objects written in languages such as Smalltalk, C++, and Java, and other software parts such as Active X controls and design frameworks.

6. Conclusion

This paper has presented an overview of software reuse and Object oriented Analysis design. Software reuse is the process of creating software systems from existing software rather than building software systems from scratch or reuse is a process of implementing or updating software systems using pre-existing software development assets.

The most common type of reuse is the reuse of software components, but other artifacts produced during the software development process can also be reused: system architectures, analysis models, design models, design patterns, database schemas, web services, etc.

7. Bibliography

[1] Anderson, K M 2004, *Software Re-Use*, viewed 25 November 2007, <http://www.cs.colorado.edu/~kena/classes/3308/f04/lectures/lecture10.pdf>

[2] Anslow, C., et al (2004): Software Visualization Tools for Component Reuse. http://www.open.org.au. Accessed 1 November 2007.

[3] Baldo, Jr J, Moore, J & Rine, D, 1997, *Software Reuse Standards*, Standard View, vol.5, no. 2 , pp. 50 - 57 ,ACM, New York, viewed 23 November 2007,<http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/270000/260559/p50-baldo.pdf?key1=260559&key2=6641095911&coll=portal&dl=ACM&CFID=62899171&CFTOKEN=32734414>

[4] Danforth, S & Tomlinson. C 1988, *Type theories and object-oriented programmimg,* ACM Computing Surveys, vol. 20, no. 1, pp. 29 - 72 ,ACM, New York, viewed 23 November 2007, < http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/70000/62060/p29-danforth.pdf?key1=62060&key2=3424632911&coll=portal&dl=ACM&CFID=39019048&CFTOKEN=54731362>

[5] Eclipse, *Guideline: Software Reuse*, viewed 23 November 2007, <http://www.epfwiki.net/wikis/openup/openup_basic/guidances/guidelines/guideline_software_reuse.html>

[6] Fichman, R G & Kemerer, C F 2001, *Incentive Compatibility and Systematic Software Reuse*, Journal of Systems and Software, Vol. 57, Iss. 1; pg. 45, New York; Apr 27, 2001, viewed 2 November 2007,

<http://www2.bc.edu/~fichman/Fichman_2000_Incentives.pdf >

[7] Henry, S M & Lewis, J A 1991, *An Empirical Study of theObject-Oriented Paradigm and Software Reuse,* Conference on Object Oriented Programming Systems Languages and Applications, pp. 184 - 196 , ACM , New York,, viewed 23 November 207,<http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/120000/117969/p184-lewis.pdf?key1=117969&key2=3102095911&coll=portal&dl=ACM&CFID=62899171&CFTOKEN=32734414>

[8] Horner, Ken 1993, *More to reuse than objects - the benefits of software reuse,* Software Magazine, viewed 23 November 2007, <http://findarticles.com/p/articles/mi_m0SMG/is_n7_v13/ai_13761674>

[9] IEEE, *Software Reuse Processes*, viewed 24 November 2007,<http://media.wiley.com/product_data/excerpt/4X/07695087/076950874X-2.pdf>

[10] Jardine, D.A. 1972 OSOS/ 360, *Operating System/360 1965-1972*, viewed 21 November 2007, <http://ldworen.net/fun/os360obit.html>

[11] Krueger, C W 1992, *Software Reuse,* ACM Computing Surveys (CSUR) ,vol. 24 , no 2, pp. 131 – 183, viewed 22 November 2007,<http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/140000/130856/p131-krueger.pdf?key1=130856&key2=5890095911&coll=portal&dl=ACM&CFID=62899171&CFTOKEN=32734414 >

[12] Larman, C 2002, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn, Prentince-Hall, New jersey, USA

[13] Mo, S, Penev,A & Stobbart, A , *Case Study of the Therac 25 Incident*, viewed 20 November 2007, <http://www.csc.uvic.ca/~csc454/notes/P4.Cases.pdf>

[14] Muncahster, P 2007, *Fortify boosts Web 2.0 protection,* viewed 1 November 2007, <http://www.itweek.co.uk/itweek/news/2189885/fortify-boosts-web-protection>

[15] Nerson, J.M (1992): Applying Object-Oriented Analysis and Design. *Communication of The ACM.* **9**: 63-73.

[16] Parnas, D. L 1994, *16th International Conference Software Engineering*, viewed 21 November 2007, <http://www.win.tue.nl/~mchaudro/cbse2007/Managing%20CBSE%20and%20Reuse.pdf>

[17] Randell, B 1996, The 1968/69 NATO Software Engineering Reports, Department of Computing Science, University of Newcastle upon Tyne, viewed 20 November 2007,<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html>

[18] Ramachandran, M (2005): Sofware Reuse Guidelines. *ACM SIGSOFT Software Engineering Notes.* **3**: 1 -7. Leeds, UK, ACM Press

[19] Schach, S R 2004, *Introduction to Object-Oriented Analysis and Design with UML and the Unified Process*, McGraw-Hill, America

[20] Schmidt, D 1996, Using design patterns to guide the development of reusable object-oriented software , ACM Computing Surveys (CSUR),vol. 28 , no. 162, ACM, New York, viewed 24 November 2007, <http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/250000/242429/a162-schmidt.html?key1=242429&key2=7834016911&coll=portal&dl=ACM&CFID=44729986&CFTOKEN=46985768>

[21] School of Informatics and Engineering, Flinders University 2007, *Object-Oriented Analysis and Design (OOAD)*, COMP 8522, semester 2, Flinders University, Adelaide

[22] Tewari, R 1995, *Software reuse and object-oriented software engineering in the undergraduate curriculum,* Technical Symposium on Computer Science Education, pp. 253 - 257 , ACM, New York, viewed 23 November 2007,<http://delivery.acm.org.ezproxy.flinders.edu.au/10.1145/200000/199808/p253-tewari.pdf?key1=199808&key2=3422095911&coll=portal&dl=ACM&CFID=62899171&CFTOKEN=32734414>