

Essay for DIF 8901 Object Oriented Systems
May 2003
Kirsti Elisabeth Berntsen stud.nr. 014079

Software development and reuse, an issue of both technology and people

The widespread focus and adoption of Object Orientation in the late 1980'ies was thought to be the necessary prerequisite for achieving reuse on a large scale in both the design and implementation of software. However, a quick survey of publications on the subject of reuse since this breakthrough tells of varying problems and challenges to achieving this goal.

This essay gives an overview of some of the described issues which are grounded in technological as well as organizational issues. Finally an attempt is done to compare the programming process or experience at an individual level to findings in the research of groupware adoption.

Why reuse?

Given the impression that to establish firm traditions of reuse isn't easy, why bother? Offhand a few reasons come to mind. Large software systems consists of so many parts, that many of the part-solutions must have been met before in other systems. A simple example of this is the drop down menus of graphical user interfaces (GUI). Complex problems do well with all possible contributions to simplify matters, so that available resources can deal with the interesting new issues. Modularity and reuse could contribute to reduce complexity and thus contribute to better quality in terms of easier testing and easier maintenance. A reused component will be subject to revised scrutiny and thus gain better quality. Competition in the market requires that systems be produced for the least possible cost, even for in-house products which increasingly compete with commercial alternatives.

On the other hand, is it easier to reuse than to make anew? The Not-invented-here-syndrome is a common term used to bag some of the difficulties. The experiences vary, but some later reports of broad approaches seem to be more successful.

Effects of reuse – an example

Metrics collected in two case studies of a reuse program at Hewlett-Packard describes the effect of reuse on quality assurance, costs and production time. Lim, Wayne C. (1994) demonstrate better quality, shorter time to market and indicates a return on investment through a quantitative and qualitative evaluation. The first case was a project on Productivity Software for HP with reuse of code (C and system language), applications, architecture utilities and files. The second case was Firmware for plotters and printers with reuse of C code. Their assessments of the results were:

- 2- 4 times reduction in defects (defects found in each reuse are accumulated, provides incentives for better defect prevention)
- 40 – 60 % increased productivity (fewer products made from scratch)
- time to market (up to 42% improvement on critical path of development)
- Costs (initial cost for tools, process, integration) for making reusable firmware was 111% compared to non-reusable versions and the integration costs were an additional 19%)
- Break-even (recovery costs) was estimated to be two years for the one case and six years for the other.

As part of its program for reuse HP has *producers* which make the *work-products* which in turn the *consumers* use to make other software. The work products are the by-products or products of the software development process such as code, design and test plans.

Technological background

Given that the invention of OO in computing is a logical prerequisite for the concept of component based systems building and reuse, the following gives a cursory description of some of the basic concepts of OO and later developments.

What is Object Orientation (OO)?

Starting from the original concepts of encapsulated objects which interact with their surroundings through messages, the topic of OO has evolved into a large topic with many variations to the basic theme. The concept of OO holds different meaning from different angles, be they the analysis of the problem domain, design, implementation or maintenance/extension of software systems.

A model of object-oriented computing as a framework for discussion proposed by Gordon Blair et.al. (1991, ch. 5.3) describes four dimensions, namely encapsulation, classification, polymorphism and interpretation:

Encapsulation is defined as the grouping together of various properties associated with an identifiable entity in the system in the lexical and logical unit, i.e. the object.

Furthermore, access to the object should be restricted to a well-defined interface.

Classification is the ability to group associated objects according to common properties.

Various classifications can be formed representing different groupings in the system. All objects within a particular grouping will share all the common properties for that grouping but may have other differences.

Polymorphism implies that objects can belong to more than one classification.

Classifications can therefore overlap and intersect. Thus it is possible for two different classifications to share common behaviour.

Interpretation is defined as the resolution of polymorphism. In polymorphic environments, it is possible for a particular item of behaviour to have several different meanings depending on the context. It is therefore the task of interpretation to resolve this ambiguity and to determine the precise interpretation of an item of behaviour.

Major areas for the use of OO are in Blairs book said to be:

- Programming languages
- Software engineering
- Databases
- Artificial intelligence
- Human Computer Interfaces
- Operating systems
- Distributed systems

The level of adoption of OO into these fields today seem to differ somewhat. Procedural languages (i.e. basic, Fortran and C) are still used both together with OO-based languages and alone, both for new systems or in the maintenance/extension of legacy systems.

From the modelling point of view, the essence of the OO paradigm is the ability to model as a structure of interacting objects. This makes the OO approach better for modelling real world problems than the traditional approach used with procedural programming languages. Patterns of interacting objects can further be abstracted into a corresponding pattern of interacting roles. The role model abstraction belongs to the realm of modelling whilst the class abstraction belongs to the realm of implementation, much like the concept of overloading which also describes behaviour sharing in terms of specification. (Renskaug 1996).

Tools and technology to better Software development and reuse

The OO concept therefore holds concepts of abstraction which should help to bridge the gap of understanding between real word problems with non computer literate users and the implementers of programs and code. Further the modular aspect of objects should make them reusable in later programming tasks within the same company or even in entirely different contexts and places. The class libraries of some programming tools are an example of one kind of reuse. Commercial Off-the-shelf (COTS) software is an example of software for sale to be used in the building of other systems. In later years one speaks of CBS (Component Based Systems), though this does not necessarily apply only to the OO type of programming. Schemas, Patterns and Frameworks are new concepts in the continually diversifying language of the computing communities. Some of these approaches overlap and have different definitions.

In order to create components for reuse there has been developed several approaches ranging from reengineering old code by *wrapping* or applying *middleware* or *glueware* such as CORBA or COM.

Patterns and Frameworks

Coplien (2001) describes what a good pattern should do:

- It solves a problem: patterns capture a solutions, not just abstract principles or strategies

- It is a proven concept: patterns capture solutions with a track record, not theories or speculation.
- The solution isn't obvious: many problem-solving techniques (such as software design paradigms or methods) derive solutions from first-principles. The best patterns generate a solution to a problem indirectly – a necessary approach for the most difficult problems of design.
- It describes a relationship: patterns don't just describe modules, but describe deeper system structures and mechanisms.
- The pattern has a significant human component (minimize human intervention). All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

A pattern language defines a collection of patterns and the rules to combine them into an architectural style. Pattern languages describe software frameworks or families of related systems.

Some tools have class libraries with classes with interdependencies. These interdependent classes are in effect patterns.

Frameworks are described as the combination of components and patterns.

Described problems and challenges

Architectural mismatch

There has in the last decade been real progress and increased support for compositional approaches to software. Industry standards such as CORBA, domain-specific architecture, toolkits, application generators etc. support reuse. However Garlan, Allen and Ockerbloom (1994) claim that two main problems with reuse still cause problems. The first of these they term as “Low level problems of interoperability” which include incompatibilities in programming languages, platforms and database schemas. The most important one is the “Architectural mismatch” where the assumptions on the architecture of the system have been misunderstood. They base their conclusions on the ABLE-project (Architecture-Based Languages and Environments) carried out over five years. They decided that they wanted the following reusable parts

- An OO Database -> OBST
- A toolkit for constructing GUIs -> InterViews from Stanford University
- An event-based tool-integration mechanism -> SoftBench from HP
- An RPC mechanism -> Mach RPC Interface Generator
- All in C++ or C, with available source code.

In their effort they discovered integration problems due to excessive code, poor performance (i.e. save took several minutes), need to modify external packages to make them work together, need to reinvent existing functions, unnecessarily complicated tools and an error-prone construction process where recompilation took much time and the code had dependencies.

To deal with this they define a system to be a configuration of *components* and *connectors*. The components are Entities of the system as tools, Databases, servers, filters, etc. The connectors determine the interaction between components as client-server protocols, RPC links, pipes etc.

In order to avoid architectural mismatch they say that Architectural assumptions must be made explicit. This includes assumptions on documents, developing description vocabulary and languages for software architecture. Orthogonal subcomponents should be used with clear dependencies so that they are easy to replace with other modules and easy to reconfigure.

Bridging Techniques comprise *Mediators components* that take over some of the tasks and *smart connectors* that translate data in multiple protocols. *Wrappers* around a component or connector can be used to offer a convenient interface. An alternative is termed *Negotiated interfaces* which are components and connectors that handle several styles and can configure themselves dynamically.

Finally they suggest developing architectural design guidelines.

Reusable Software Library

A Reusable Software Library(RSL) is for some organizations a core element in a strategy for reuse. Poulin (1995) describes IBM's experiences with an RSL in an enterprise-wide initiative for reuse. Due to a perceived lack of code in the library as a cause for little reuse the company launched a campaign, with incentives, to populate the library. Unfortunately this was a narrow focus evaluated by the amount of code in the library. In evaluating the following actual reuse Poulin concludes that it in a corporate RSL it is necessary to distinguish between domain specific software and domain-independent software. The latter combined with taking the lead on providing standards, resolution of common issues and technical consulting should be the focus and responsibility for the corporation. The domain specific should be the responsibility of the local organizations.

In evaluating IBM's RSL project Poulin (1995) states that "programmers cannot reuse software unless it is useful. One of the challenges were the lack of standard methods to classify software.

16 questions on Reuse

A survey carried out on 29 different organizations in the software developing industry in 1991-92 by Frakes and Fox (1995) gives answers to what they state to be 16 central and common questions regarding reuse. They define reuse as: The use of existing software knowledge and artefacts as an asset to build new software artefacts. This is not to be confused with porting which is moving a system across environments and platforms.

The answers were given by software engineers, managers, educators, and others in the software development and research community. Some of the conclusions that are drawn are related below.

- There are significant differences in reuse between different industries : i.e. higher level for telecommunications and lower level for aerospace industries
- A defined software process that promotes reuse does affect software reuse levels
- Education in school and at work improves reuse and is a necessary part of a reuse program
- Perceived economic feasibility influences reuse
- Most developers prefer to reuse rather than to build from scratch
- The majority (75%) does not agree that CASE tools have promoted reuse. This may be because CASE tools are not being used, used correctly or may actually not promote reuse
- Choice of programming language does not affect code reuse levels. To increase reuse the focus should be on other factors

Answers where the response statistics show no significant correlation lead to conclusions such as:

- Satisfaction with quality does not influence reuse levels
- Many organizations(>50%) do not measure reuse levels, quality, or productivity
- Company and project size are not predictive of reuse levels
- Organizations of any size may succeed to introduce systematic reuse
- Having a reuse repository does not improve reuse
- No significant correlation between reuse-levels and reported inhibition by legal issues
- Comparing levels of organizational code reuse between the two groups "rec. rewards" and "no rewards" show no significant differences. Similar findings for individual reuse
- Software engineering experience has no effect on reuse of lifecycle objects. This may be caused by historical lack of training in reuse

The authors suggest that to improve reuse one should concentrate on educating developers about reuse and gaining a common understanding on the economic feasibility of reuse. It is also necessary to introduce a common development process that promotes reuse and make high quality assets available to developers.

Classification of issues

A classification of issues in respect to component bases systems (CBS) is described by Brereton and Budgen (2000). CBS are defined as : Software components as units of independent production, acquisition and deployment that interact to form a functional system. The idea is to use components as building blocks to enhance software-based system development and use. Components can take a wide range of forms and sizes, and should be independent of specific architectural style. While objects can be components, all components are not objects.

With components as a commodity for sale the different actors of transaction have different perspectives on the software product:

The Component providers consider:

- Granularity: size and shape of components which affect the choice of tools and level of investment.
- Portability: the ability to transfer between systems and to operate within a range of environments. The use of “glueware”/middleware such as the Common Object request Broker architecture (CORBA) or Microsoft’s Distributed-/Component Object Model (DCOM/COM) partially address portability as does the use of platform independent languages such as Java.

Component Integrators consider:

- Component selection: find, characterize and evaluate against requirements.
- Interoperability: the ability to integrate which might involve “gluing”, ”wrapping” or adaptation of components. Potential difficulties include architecture mismatch, functional deficiencies and quality maintenance.
- Combining quality attributes when the total system inherits properties for its separate components.
- Maintenance of components which come from different parts of distributed or multiple organizations.

Common needs for the providers and integrators consist of the need to predict limit-values of operation (i.e. 32 bits problem) and to describe components in order to locate, understand and evaluate them, sometimes by an automated tool.

Improving the Commercial Off-The-Shelf Software Development process

Morisio, Seaman, Parra and Basili set up a project on COTS SW D as follows:

- COTS-based processes differ considerably from traditional software development
 - New activities
 - Reduced activities
 - Modified activities
- Disadvantage of COTS projects:
 - Dependence on vendors
 - Flexibility in requirements
- The new process needs to be defined
 - Requirements and design phases are most challenging

Organizational issues

Fafchamps (1994) has done an extensive qualitative study on the organizational issues of reuse practices at Hewlett-Packard at 10 engineering sites. Working on the assumption that more than technological issues affect reuse, the relationship between the different roles of *producer* and *consumer* of reusable components was analyzed in terms of different organizational structure models. Which of these support reuse and which don’t?

The four models of organization was identified :

- Lone producer : an individual provides reuse service to at least two consumer teams
- Nested producer : each product team has a member dedicated to providing reuse services and expertise

- Pool producer : two or more teams collaborate to produce and share components
- Team producer : a team is dedicated to produce reusable components

Fafchamps concludes that the Team producer model is the one that appears to have the most potential for the successful implementation of a long-term reuse strategy based on three dimensions. These are termed as “membership in a legitimate organizational niche”, “a clear career path” and “a home functional area”. When individuals function as a team and personal career issues do not conflict with the teams agenda, there can be a transition in frame of mind of the participants from a “project” as in temporary state to an “organization” as a stable institution. Fafchamps also suggests role rotation between teams to nurture a culture of collaboration as well as the educational aspect of learning.

The successful redefined producer-consumer roles in this study, was as follows:

Producers:

- Focus on providing reusable code modules
- Are responsible for total instrument design or architecture
- Conduct interviews with end-users
- Are the official marketing contact for software that spans the product line.
- Achieve job-satisfaction by “seeing technical solutions applied across multiple instruments”

Consumers:

- Take marketing inputs from the producers, and thus have more time for examining design issues
- Use the code and solutions from the producers
- Achieve job-satisfaction by producing higher quality instruments.

This study gives new content to what is often summarily called “Not-invented-here”. Resistance to reuse can be a symptom of weak collaboration inherent in the organizational structure. On a personal level the more a reuse program disrupts a persons professional expectations, the more resistance it will encounter. Eagerness to implement reuse should follow a careful analysis of the reuse scope.

Win-win, the balancing of efforts and rewards due to basic human nature.

In studies of the use of Groupware, (Grudin, 1989) meaning systems that are meant to aid and abet interaction and communication between members of a group, it has been found that a number of social and other mechanisms come to the fore. For example: Who gains and who loses (or gains less), with deployment of the Information System is an important issue. The most successful functions in terms of usage will be those that have an even balance of gains between different users in the different roles of the organization. E-mail is an example of this. Where there is an imbalance, for example in the case of some types of meeting schedulers: the ones who usually call the meeting get less work but those who have to keep an updated electronic calendar for no other purpose, suddenly acquire a new task with little perceived gain. Many of those who acquire the new task fail to do it on a regular basis, which means in fact that the system fails.

Later studies (Grudin and Palen, 1995) show that an adaptable user interface with good functionality improves user interest and adoption. Another result seems to be that management interest and focus also improve motivation in employees. Peer group pressure will also over time influence and motivate use.

Poulin (1995) describes IBM's incentive program for employees to providing reusable parts into a library (RSL). The focus in the beginning was on increasing the size of the library, and so providing code was rewarded. However they found it would be necessary to focus also on rewarding actual reuse, both on an individual and on a team level basis.

I believe these results are applicable also in the setting of achieving reuse and collaboration in software production. The balancing of gains and efforts on a personal level will help secure motivation. An environment where reflection on ways and means will also contribute to learning and creativity in the organization and thus a continued focus on doing a good and better job to achieve good and better products in terms of developing software. This has to go hand in hand with the technological skills, tools and knowledge to actually do the job at hand.

Summing up

In summing this quick survey of different experiences and research on reuse I will quote from the paper of Sindre, Conradi, and Karlson (1995) on the REBOOT project (Reuse Based on Object-Oriented Techniques), which I think adequately describes the overall situation for reuse and what it takes to accomplish. It is an issue which requires continuous effort and focus within all organizations dealing with Software Development in that it requires learning and development of both technology and organizations.

“The most important point about REBOOT may be that it has dealt thoroughly with both with the technical and organizational aspects of reuse, in a comprehensive manner. Thus an approach to reuse introduction and organization has been provided together with an accompanying technology to fill the various needs an organization will have as its reuse capabilities increase.

Another important contribution of REBOOT is the high emphasis on applications, yielding important practical experience. The most prominent experience made is that organizational aspects can hardly be underestimated. It was found much easier to apply the organizational parts of the methodology than the advanced technology. This was because most application organizations did not have a sufficient reuse maturity to immediately take advantage of the reuse tools.

References

Blair, Gordon et.al.: *Object-Oriented Languages, systems and applications*. Pitman Publishing, 1991

Brereton, Pearl and Budgen, David: *Component-Based Systems: A Classification of Issues* IEEE Computer, Nov.2000, p. 54-62.

Coplien, James. *Software Patterns*. Bell Laboratories, Naperville, Illinois. (05/2001)
<http://hillside.net/patterns/>

Fafchamps, Danielle: *Organizational Factors and Reuse*. IEEE Software, Sept. 1994, p. 31-41.

Frakes, William B. and Fox, Christopher J.: *Sixteen Questions About Software Reuse* CACM, Vol. 38, No. 6 (June 1995), p. 75-87.

Garlan, David, Allen, Robert and Ockerbloom, John: *Architectural Technology: Why Reuse is So Hard*. IEEE Software, Nov. 1995, pp. 17-26.

Grudin, J.: "*Why groupware applications fail: problems in design and evaluation*" Office and technology, 4:3 (1989), s. 245-264.

Grudin, J. and Palen, L., 1995. [Why Groupware Succeeds: Discretion or Mandate?](#) Proc. ECSCW'95, 263-278. Dordrecht, The Netherlands: Kluwer.

Lim, Wayne C.: "*Effect of Reuse on Quality, Productivity and Economics*". IEEE Software, Sept. 1994, pp. 23-30.

Maurizio Morisio, Carolyn Seaman, Amy Parra, Victor Basili, Steve Kraft, and Steve Condon: *Investigating and Improving a COTS-Based Software Development Process* Proc. 22nd International Conference on Software Engineering (ICSE'2000), p. 31-40, ACM Order No. 592000.

Poulin, Jeffrey S.: *Populating Software Repositories: Incentives and Domain-Specific Software* Journal of Systems and Software, 1995:30, p. 187-199.

Sindre, Guttorm, Conradi, Reidar and Karlson, Even-Andre: *The REBOOT Approach to Software Reuse* Journal of Systems and Software, Sept. 1995 (Vol. 30, No. 3), p. 201-212.

Renskaug, Trygve: *Working with objects: the OORAM software engineering method*. Manning Hill Publications (1996), preface