Adaptability

The enabler of reuse

Pentti Virtanen
Pentti.Virtanen@sci.fi
University of Turku
Turku Centre for Computer Science

Abstract

The purpose of this study is to use reusability metrics in finding the necessary attributes of reusable software. The most important of them are understandability, adaptability and trustworthiness. The next step is to analyze understandability and adaptability to develop better programming methods. The solution is to develop cohesive components from clear abstractions. The abstractions should be selected in a way, which enables the combinatorial explosion when software products are produced.

Keywords: IRIS, Process, Reuse, Adaptability

BRT Keywords: FB0503

Introduction

This study is for every developer and researcher, who is interested in improving reusability of software components. Adaptability is especially important because it leads to the mechanisms of producing very reusable software.

The research question is how to produce more reusable software. At the beginning the approach is conceptual. The concepts of understandability, adaptability and trustworthiness are found from reuse metrics. The use of metrics is necessary because it is the only way to scale the reusability of the components. The next step is to analyze understandability and adaptability to develop better reusability methods. When the concepts are analyzed, the research continues by using the constructive approach. The components should be constructed in a way, which enables the combinatorial explosion when software products are produced. The result is a number of guidelines for the construction of reusable components. Finally, the problems in current programming languages are presented with implications to their solutions.

The structure of the study is the following. At first, the definition of reuse is presented with a broader view of reusability. In metrics section, the components of reusability are revealed. Reuse the process is studied to find out the problems in reuse. The problem of understandability is analyzed first; after that an example of a typical method is used to introduce to the problem of adaptability. The currently used adaptability mechanisms are described to show their versatility. The guidelines of producing reusable software present the mechanisms, which are necessary in every project where reuse is important. At last the problems of current programming languages are described with conclusions of the future development.

Reuse

Definitions

Every software product has some fixed features. These determine what the software product is. *Reuse* considers producing these fixed properties during the development of the software product. In *use* the adaptation of the component is done at run time, which reduces the performance. The influence of both use and reuse to the software development is almost the same. The user gets the features of the component without the need to develop them separately. For example, an application can use a word processor as a component. This word processor is hardly considered to be part of the application.

Boxes of reuse

A reusable component can be seen as a box, which contains the code it's. In *the black box reuse* only the outside can be seen (Goldberg 95, page 208). The reuser sees the interface, not the implementation of the component. The interface contains the public methods of the component. If the black box code changes, the change will propagate into the applications that reuse it. The propagation is done in compilation or link phases of the development.

In *the glass box reuse* the inside of the box can be seen as well as the outside, but it is not possible to touch the inside (Goldberg 95, page 208). This solution has the advantage compared to the black box reuse that reuser can understand the box and its use better. The disadvantage is that it is possible that the reuser will rely on a particular way of implementation. That can be hazardous when the implementation changes.

In *white box reuse* it is possible to see and change the inside of the box as well as its interface (Goldberg 95, page 208). A white box can share its internal structure or implementation with another box through inheritance or delegation. The new box can retain the reused box as-is or by adaptation. It is necessary to test anything new that is created.

A *transformational reuse* is an approach where the developer provides the description of what is wanted and a black box program generates the implementation details (Goldberg 95, page 209). This approach is used in the case of application generators.

Cloning is easy to do. Just copy a chunk of code and paste it to some other place. It is not strategic reuse because the copy can change without reflecting the change back to the original.

Reuse is not black and white; there are tones of gray. The boxes of reuse are a paradigm, which describes the independence of the component when reused.

Levels of reuse

It is a possible to reuse every software artifact. Because software artifacts are mental models, they can be reused as such. Reuse of an idea is easy in theory but hard work is needed in the implementation.

Application reuse is a large-scale reuse. An application can send commands to another application. There are many different more and less sophisticated ways to do it. Using commercial off the self-software instead of producing the software is also a kind of

reuse. Packed software has typically a large number of options, which are used to adapt the software to varying user needs.

A Software framework is a set of related objects, which provide a well-defined set of services for the reuser. It can be seen as a frame to which a developer can add functionality. The difference between a program library and a framework is in that the framework calls the added code. The newly developed code is always the initiator in the case of program libraries. One of most known application frameworks is Microsoft Foundation Class, which is used in building software to Windows platforms. Another important group of application frameworks is business objects. They implement a basic functionality of a business area and they are used in building more sophisticated applications in that business area.

A *design pattern* is a description of communicating objects and classes that are customized to solve a general design problem in a particular context (Gamma 95, page 3). Each pattern solves one design problem. The problem and its solution are described as a cognitive model, which can be adapted to solve similar problems. Automation has been developed to support the use of design patterns (Florijn 97).

Typical class specification reuse includes adapting the class to another context. The specialization of the class is used to adapt the class to a new use. The inheriting class can have more member variables and member functions than the original class. It is also possible to change the member functions in the inherited class.

Measuring reusability

Introduction

> Before you can reuse something, you need to find it, know what it does and know how to reuse it (Tracz 95, page 93).

There is another view to reuse (Tracz 95, page 94).

> Firstly, before you can reuse something, there has to be something to reuse and secondly before you can reuse something, it needs to be useful.

Taxonomy

Several approaches to reuse metrics have been proposed. Poulin has made a taxonomy, which classifies these approaches (Poulin 97, page 110). There are two main groups of approaches to reuse measurement. There are empirical methods and qualitative methods. Empirical methods depend on objective data. An analyst can calculate them easily and cheaply. That is a very desirable property of a metric. The qualitative methods rely on a subjective assessment on the software's adherence to method's attributes. The use of an assessment makes qualitative methods more expensive than empirical methods.

Each of these methods focuses on a few areas. Module oriented methods address unique attributes of the code. These attributes are familiar from methods used to estimate effort. Typical attributes are numbers of lines of code, complexity and successfully passed tests. Component oriented methods consider the module and all supporting information, which can be used to assist reuse.

Reusability is related to *portability* because they both assess the ability to use a component with another component. *Maintainability* is related to white-box reuse because they both assess the ease of adaptation of a component. *Understandability* is an attribute, which assesses the ease of comprehension of a component.

In empirical studies it is important to look for *causality* which makes an attribute to contribute reusability. Comparing attributes of reused components and that of the components, which are not reused, we can find attributes, which are typical to reused components. That does not mean that changing that attribute will contribute reuse. For example, there are more reused small components than large ones. Making all components small is not a good practice because benefits of making large reusable components are larger.

Literature overview

The model of Prieto-Diaz and Freeman encourages white-box reuse and evaluates which components the programmer can modify the easiest (Prieto-Diaz 87, Poulin 97, pp. 114-116). Their metrics use traditional program size, complexity metrics applied to particular programming languages. Documentation is evaluated subjectively. Reuse experience is the most important human factor in their metrics.

Selby looked at instances where reuse succeeded in NASA environment and tried to determine why (Poulin 97, page 116). They found that reusable software module is small and its interface is simple. It has few calls to other modules. Good documentation is also one of its properties. The reuse of the low-level system and utility functions is more common than the reuse of human interface functions. Selby validated these results statistically.

The ESPRIT-2 project REBOOT (Reuse Based on Object-Oriented Techniques) developed the taxonomy of reusability attributes (Poulin 97, page 116). It has four reusability factors, which are specified using a few criteria. Each criterion has at least one metrics. Reusability is a number from 0 to 1, which is calculated by normalizing the metrics. REBOOT allows an analyst to change the weights of the metrics in calculating reusability because the importance of a metric may change from site to site.

The four reusability factors are portability, flexibility, understandability and confidence of the reuser. Portability expresses the ease of reuse in another environment. The criteria of flexibility are generality and modularity. Understandability's criteria include code complexity, self-descriptiveness, documentation quality and component complexity. *Confidence* is the probability of the error-free reuse as assessed by the reuser.

The NATO Standard for Software Reuse Procedures is more concerned about the statistics of the reuse process (Poulin 97, page 123). This metric is helpful to eliminate unsuitable candidates for reuse. Modules, which have been considered many times without actual reuse, can be suspected not being useful. The same applies to modules of high complexity and large number of defects.

The Army Reuse Center inspects all software submitted to Defense Software Repository System (Poulin 97, page 123-124). The preliminary inspection estimates the efforts to reuse the component without modification to develop a new component instead of reuse to maintain the module and finally the expected number of reuses.

The thing to notice here is that the estimated effort to reuse the component without modification, which is needed in Object Component Process Metrics, is recorded as part of the reuse library administration.

IBM's method stresses that the developer needs to have access to other sources of information than the code alone (Poulin 97, page 126). Basically that information is somewhere in the development project documentation of the module.

There are several common points in the previous metrics though the views are different. Firstly, there is an issue of understandability. The documentation, simplicity and connections to the other modules are its factors. Then there is an issue of adaptability. Its factors are the programming environment, modularity and generality. The third issue is confidence, the factors of which are the numbers of actual reuse and the number of remaining defects.

In order to improve the reusability of software we must increase its adaptability and understandability and decrease the number of defects.

Reuse process

In this chapter the idea of the reuse process at the developer level is used to get a better understanding of reusability. The reuse process consists of four major steps (Goldberg 95, pp. 223-246):

1. Define reuse
2. Set up a process of populating a library of reusable assets
3. Set up a process of sharing reusable assets
4. Set up a process of maintaining reusable assets

The process of interest here is the process of sharing reusable assets. It involves three steps (Goldberg 95, pp. 241-242):

1. Communicate the availability of reusable assets
2. Locate and retrieve reusable assets
3. Understand and use reusable assets

Though reusability includes the suitability of the component to each of these phases, the most important of reusability attributes are those of them which contribute to finding them, knowing what they do and knowing how to reuse them (Tracz 95, page 93). That proposes defining reusability as a compound of ability to find, understand and apply for a component. If we prefer money as the unit of reusability, the *total cost benefit of reuse* is identical to reusability. In equation form

Reusability = cost to reproduce – number of reuses * (cost to find + cost to understand + cost to apply)

Poulin presented a more detailed cost benefit analysis from IBM (Poulin 97, pages 77-83), which also takes indirect consequences of reuse in the equation. Though it is right to do so, it does not contribute to the understanding of what a reusable component really is. The effort can be used instead of prices without a large error. If it cannot be done, the licence costs or some other costs of reusable components are too large. In the case of application frameworks and business objects and parameterized applications the licence costs are large, but in that case they present the largest part of the functionality of the final application.

What are the properties of a component, which is easy to find? Clearly, it is not primarily a property of the component itself, but mostly the property of the reuse library and the reuse organisation. The property of the component, which has impact on reusability is its *proper classification.* A properly classified component can be found from the place where its existence is expected. In order to assist the potential user in searches, there can be keywords attached to components. Classification is one of the

qualification criteria in REBOOT project (Poulin 97, pages 129-130). In Smalltalk type browsers the components can be found using inheritance references and cross-reference lists of senders and implementers of the methods. Integrated development environments and configuration management tools contain facilities to assist the component search. They all require that a component contains proper keywords or it is located in a logical place. The search of commercial components from the Web can be done by search engines. A search engine finds the seller's web site using the keywords, which the potentential buyer gives to the engine. It is also a great help, if the component is sold by a seller, who have a large number of connections to the buyers.

The effort of understanding is basically done by chunking the interface and support documentation. The understanding of a chunk is based on the mental models, which the one studying the component already has. If the abstraction of the component is already known, it can be understood readily. If that is not the case, all unknown concepts must be studied. That includes tracing to their origins and chunking and tracing until every necessary part of the component is understood. The properties of the component, which decrease the effort are thorough documentation, including self-documenting code and in-line comments. If the we consider white-box reuse also the effort to understand the code must be included. A component with smaller size, simple interfaces, fewer parameters, high cohesion and low coupling is likely to be easier to understand.

The effort of applying a component to a new place has the prerequisite that it is well understood enough. Several attributes must be considered before that. Firstly, it must be fairly portable. The effort to port the module to its new environment must be reasonable. Reuse of a component is more probable, if it is independent of the environment and it is written using the same programming language as the reusing application. Modularity is also a very important property. It can be measured by fan-out and fan-in-metrics . The problem is the following.

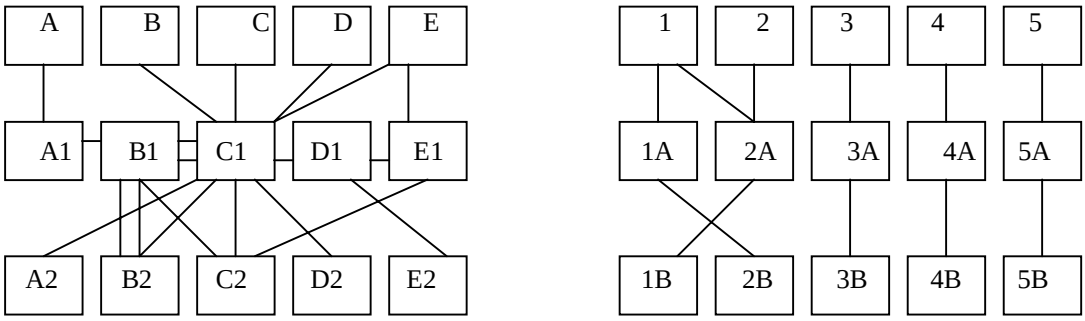

Figure 1. Fan-ins and fan-outs for reuse.

In Figure 1, there are two groups of modules connected by lines, which indicates a fan-in or fan-out. Now consider the reusing module C1 or 3A. Clearly, it is easier to reuse module 3A. The idea in the figure is not to simply calculate the number of fan-ins and fan-outs. The point is in seeing that it is easy to disconnect 3A from the web of dependencies. Just take 3A and 3B and reuse 3A as it is done in 3. There are not only called modules and their callers, there are also common variables which create two-sided dependencies between modules. A package contains several components, which are used together. The question is which of the modules must be within the package, which is to be reused. Reuse packaging is essential to successful reuse because proper packages will generate confidence on the reuse of the component. The package includes good documentation and it is easy to locate reusable assets in it. The packages of multiple

components can be organized as object libraries or frameworks. Jacobson uses packages in both use case components and object components (Jacobson 97, pp. 129-132 and pp. 165-168). That is new compared to the traditional way to use only classes as components.

The *confidence* factor describes developer's confidence on successful reuse. If the source code is available and the developer understands it readily enough confidence for reuse exists. Without source code the developer will assess the interface and its *understandability*. If there is some data about subsequent tries in reusing the component, it can help. Poor *error tolerance* and errors shown in the first tests will decrease the confidence. Before large-scale reuse, the components can be tested in pilot projects. That will increase *observed reliability*. Commercial component manufacturers will use their reputation to increase the confidence of their components. Here the REBOOT taxonomy of reusability attributes (Poulin 97, page 117) is used. There is still one major factor of applicability, the generality, which must be taken into account. *Generality* is the property of object-orientation, which has made it almost as a synonym of reuse. The increased generality means that a component can be used (without change) in more varying situations than before. For example, consider reusing a sort algorithm, which can handle only arrays of integers. A more general sort algorithm, which can handle any kinds of collections, is preferable. Generality defines how suitable the component is for black-box reuse. When white-box reuse and variations of glass-box reuse are considered, the adaptability of the component is the dominant factor. Object-oriented software has two mechanisms to adapt its components: *extensibility* and *inheritance* (Carroll 95, pp. 48). Extensibility includes other kinds of flexibility than inheritance. The generic classes, which are called templates, are most important of these. Different kinds of patterns are important extensibility mechanics in higher levels of abstractions.

Object Component Process Metrics can be used to measure the reuse process (Virtanen 98). In fact, that is an essential part of it. Object Component Process Metrics sees an application as an aggregation of reused components. The work effort needed to produce an application consists mainly of the work needed to reuse standard components. Large granularity parts are aggregated from smaller granularity parts by coupling these in a new way. Coupling of components is also considered as a way to add flexibility to software production.

Understandability

Concepts

Understandability is a property of a component but it depends largely on the individual who is supposed to understand it. It helps largely if the existing mental models and the new ones conform to each other. When we consider understandability of a component, *familiarity* is the best way to increase understandability. A component is familiar, if the developer has used it one or more times. Cognitive Complexity model estimates that the effort of chunking and tracing will be two-thirds of that previous time (Cant 94), which means that using the same components repeatedly will be effective. In the user interface production Microsoft ActiveX-components can be considered *standard components* which should be familiar to most developers using Microsoft tools.

When a component is not familiar, the user must study it more closely. This is the other part of understandability. The user must understand the components of the component. The study will be accomplished by using the interface, possibly code and all

the documentation, which is included in the component packaging. Clearly a small component is easier to comprehend than a large one because it is easier to trace. A small component contains a smaller interface, smaller documentation and smaller feature set. Serious difficulties will also occur if the component is not properly packaged. That means tracing large areas of code, documentation and interfaces before useful information is found.

The problem of a component, which has a large number of connections, shown in the figure (Figure 1) is not only in that such a component cannot be used because it cannot be disconnected from the current use. Studying such a component requires large amounts of tracing and chunking and finally there is no confidence that there are no undiscovered surprises. Suppose that the component can be outlined. Because the component is not familiar, background information must be used. Understanding will depend on conformance of the component and previous experiences and knowledge that the user remembers. The last choice is to begin to study previously unknown areas. Tracing and chunking is used again. The user tries to understand one chunk and after that another. A common problem is that a writer of documentation assumes that the user already knows the other concepts, which are needed to explain the concept number one. Clearly, the coupling of the parts of a component is not only a problem of outlining. It is also a problem of the human's ability to understand only a chunk at a time. Because this is a kind of a chicken-and-egg problem, tutorials and other documentation, which first reveals the basic concepts more widely and approach the details, are useful. For example, Microsoft's application framework MFC is a very large collection of reusable objects (Microsoft 93). If a developer studies it using a reference manual, it is very difficult to understand it because most of text expects that he/she understands the other parts of the reference.

Understandability analysis

Understandability is one of the main arguments for object oriented analysis. It is assumed that if concepts familiar in the application area can be used also in the analysis phase of developing an application to that area, it would produce a better application with less effort. Firstly, no conversions are needed between the analysis concepts and the programming concepts. Secondly, less work is needed to understand the software. The business area concepts are familiar to the end users of an application if no major business process re-engineering is done. Experienced software analysts are also familiar with general business concepts and concepts which they know from their experience.

Experienced users have gradually achieved a working knowledge of traditional analysis methods and the use of personal computers and currently used software. Analysts are typically novices in object oriented methods. These methods are in rapid change though attempts of standardization have and will be seen. The skill of an average analyst is on a conceptual rather than basic level. That is not enough to conduct an analysis project teaching the methods to the users and fellow developers. Old mental models from old ways to develop software are clear in their minds. Forgetting and overriding them is time consuming. Applying old mental models to new paradigms will bring out numerous mixed solutions and confusions between them.

It is easy to notice that the artifacts that object- oriented methods produce are not easy to understand. Firstly, even the basic concepts have not been defined unambiguously. There are industry groups, which have different concepts, and solutions to all phases of object oriented software development. The literature is full of varying

proposals in the name of object orientation. In an organization confusion can be avoided by standardization. The success of a local standard depends on the solution, even the words used. For example, the word inheritance has a metaphor of parents and descendants, which can mislead to speak about change history, not about generalization and specialization of concepts. The latter ones are the new words used in UML (Booch 97, page 51).

One part of understandability is the pure volume of diagrams, methods and processes, which are required. Structured methods also required a large number of diagrams, but in those days the CASE-technology was not so good that it could have been used to produce all of them. In the analysis paralysis a development project cannot end the analysis phase because there are always inconsistencies to correct and new ideas to include (Basset 97, page 207).

The problem of adaptability

Basically the question of code reuse and adaptability is about how to use the functionality of a chunk of code without copying and changing it slightly. The copy and change strategy will lead to large programs that mostly contain the same code. It is very difficult to maintain the integrity of the copy and the original code if a developer must accomplish it.

In the next example, one typical chunk of code is presented (Figure 2). Then its reusability is evaluated in both the black box reuse and the white box reuse. It is important to notice the objections against possible reuse. Firstly, taxonomy problems prevent reuse, which is implemented by inheritance. Secondly, the misuse of concepts is a considerable reuse inhibitor.

```
Class AddressBook;
//.member variable m_addressBook is a collection of bookItems
m_addressBook BookItemCollection;

Public Address  GetAddress ( FullName parameterName)
{
// get the address of a person whose fullname is given as a parameter.
// However only surname is used in the search
BookItem auxItem;
Do
 {
        auxItem = this.GetNextItem ();
        if ( auxItem.Compare(parameterName.SurnameOf))
        {
                break;
        }
} while not this.EndOfItems;
return auxItem.AddressOf;
}
```

Figure 2. The question of adaptability.

Consider the method getAddress in Figure 2. Is it reusable or not? What prevents us from making a copy of it and pasting the chunk to another place? Clearly, there is not enough information to make precise conclusions. The syntax is not important here. The programming language is carefree Java. The purpose of this method is to make search on an address book using a surname and return the address. It could have been taken directly from some application where such functionality is needed.

If we consider black box reuse, only the interface is used. The call is of the form Address=GetAddress (parameterName). This method is a method of an address book object. That object must be within the scope of a possible user. The same applies also to class of the parameter, FullName, and to the class of the return value, Address. Because an object-oriented programming language is used, also their descendants can be used. It is possible to construct the parameter object parameterName and set a value to it, because the class FullName is available. An empty variable can be constructed for the return object. The values in the address book object must be stored somewhere before making the search is reasonable.

The knowledge of member variable m_addressBook and its item class BookItem can be obtained tracing to their class definitions. The member functions of these classes can also be read from their class definitions. This knowledge is not necessarily public. In that case it does not belong to the interface which is available in a black box reuse. Without looking at the source code or documentation it is not clear that the method in question uses GetNextItem and EndOfItems methods. If the reuse is put into practice using a descendant class, it is essential that these methods are used for the same purpose. Compile time errors are prevented if it is not possible to remove member functions in inheritance. The member functions from BookItem class, which are used here, Compare and AddressOf should also be known if this class had been overridden. The member function SurnameOf in parameter class FullName is also interesting because then its name, SurnameOf , reveals that only surname is used as a parameter to function Compare.

Nothing in the code reveals whether any side effects exist. It is not a good practice to change the value of parameterName during such a call. Keyword final, const in C++, should be used to make sure that it is the case. Confidence is a substantial promoter of reuse as deduced earlier.

If effective code is needed and the address book has a large number of items, the search algorithm is an essential property of the method. If the only member function resembling a search function is GetNextItem, it is clear that black box reuse is not reasonable if the address book is large. If a hash algorithm is also available it is not clear that it is also used in the member function GetAddress.

The lesson from the previous was that good documentation is required to support black box reuse. In a glass box reuse, it is possible to find out what the developer of this code had in his/her mind provided that cues from variable names and substantial amount of tracing and chunking are used. It is not clear that the cost of reuse is smaller than that of rewrite. Reuse based on scavenging old code is typically possible only to the original developer (Carroll 95, page 2).

In white-box reuse all the code is available and it is allowed to change the code. Suppose that the customer’s addresses are available using the business object class Customer and that the customer’s names can be taken from a user interface class. The search algorithm must be changed, if the number of customers is not small. Because that would prevent reuse and favor rewrite, such an assumption is made here. The

addressBook class has no conceptual connection to the customer class though they both contain names and addresses. A customer is not a special case of an address Book. Because the address book class is a singleton class (Gamma 95, pp. 127-134), it is not either a good idea to attach it to the customer class as a member variable. Besides, the customer object already contains a customer name and address. The same kinds of problems occur when the screen variable is used as a parameter to the function. A text box control is not a FullName and a FullName is not a text box control. The name of the member function SurnameOf should also be changed to the new use. Depending on the solution, the BookItem class must be rewritten. It could be connected to class Customer.

The lesson from the white-box reuse example is that it is difficult to get anything useful from the old code, if the context changes considerably though the general purpose of the code remains the same. Taxonomy problems prevent reuse using inheritance. Though the multiple inheritance was used, a reasonable object model would not result. If the rewrite were not as easy as in this example, various kinds of implementation inheritance solutions would be introduced. A common thing to these solutions is that the pureness of object taxonomy is sacrificed. That will lead to problems later because it is not any more justified to trust on conformance between business concepts and concepts used in the application.

The conclusion is not that reuse is impossible. On the contrary, all of these problems can be avoided. Basically the problems are due to misuse of the member function. The algorithm for a search from a collection is needed. It is found from an application class, this time from class AddressBook, which is not that kind of utility class that is needed. A larger re-engineering of the class structure should be done. The lesson remains the same: a considerable additional effort is needed to produce reusable components.

One note has still its place. If the language were some Smalltalk variant, a direct sending of a message would be possible if appropriate methods had been defined. There were no compile time checks that prevent reuse. This topic will be discussed more widely later.

Reuse mechanisms

Inheritance

Inheritance and aggregation are the only original object oriented ways of adapting code to a new context. In inheritance the new context should conceptually specialize the original context. For example an address book is a special case of a collection because it contains only names and addresses. Specialization is an additive process, new data members and member functions can be added. It is also possible to change the member functions of the original by overriding them.

It is possible to use member functions, which have the same names in different classes. The decision, which of them is used, can be deferred to run-time. It is also possible to create a class, which is a specialization of two or more classes.

The advantages of inheritance are quite well known. Code and data from the parent class can be reused in the descendant class. Only the changes are needed to handle. Inheritance hierarchies using business concepts will make understanding of application concepts also easier. The use of abstract data types will increase the level of abstraction. The ease of understanding and reuse will increase the productivity.

The disadvantages of inheritance have also got increasing attention. In the early implementations of object orientation pure object oriented languages such as Smalltalk emerged. The introduction of C++ made it gradually the most important object oriented language. One of the most important factors in that was the lack of performance in pure object oriented languages. Every method invocation was solved at run time.

> *Dynamic binding* is "the guarantee that every execution of an operation will select the correct version of the operation, based on the type of the operation's target" (Meyer 97,page 1195).

Dynamic binding has also a serious drawback in that it was not known before a particular run that every method invocation would be solved at all. If there were not a method that would correspond to the invocation an error message "message not understood" was given. That is intolerable in mission critical applications. C++ solved member function invocation most at compile time. For dynamic binding a mechanism of virtual functions was introduced. The details of the type systems and binding in object orientation will be discussed more widely later.

The second problem comes from the generalization and specialization structures. Reality cannot be modeled using single inheritance taxonomy. A bird is a flying animal. An ostrich is a bird, but ostriches do not fly. Also if a customer is an external agent and an employee is an internal agent, an object which is both the customer and the employee cannot be defined in single inheritance taxonomy (Bassett 97, page 144). The solutions for the latter would be duplicating their properties or fragmenting them into clusters of smaller classes. The multiple inheritance solution requires that conflicts between the duplicate properties are solved. Both the customer and the employee might have name and address. Having two copies of these, as object attributes, would not be a good solution because it reduces cohesion and manageability.

A problem mentioned before was the disability to remove member variables and member functions from the inherited classes. That disability guarantees that an heir invoking a member function of a parent will succeed. It prevents the removing ostrich's ability to fly. Aggregating all member variables will also result in unnecessary variables. Their existence will lead to more error prone code and added difficulty in understanding the code.

The fragmentation of member functions is also a property of object oriented code. Small method size is a desired property (Lorentz 94, page 43). That is because reusing such methods by inheritance is easier than reusing larger methods. Larger methods must be split to override a part of code. Figure 3 is a pictorial presentation of this kind of fragmentation?

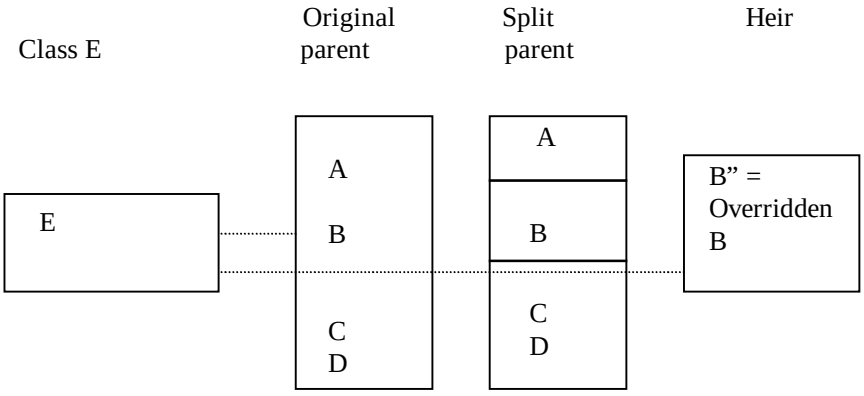

Figure 3. Reuse of a method where one part is overridden.

Suppose that the original method contains chunks A, B, C and D. A little change is now required in chunk B. In object oriented programming a new class must be created. That new class contains a method for the version of chunk B, let us call it B". The method of the parent class is split so that the code executed in the case of an heir object is A, B", C and D. The chunks B and B" include an invocation of the method E of another class. Resulting web of calls between layers of code is sometimes called lasagna code (Bassett 97, page 153).

Another problem is in that the only way to adapt code for reuse is to define a new class where the method can be overridden. However every reuse is not due to a new specialized concept. For example a new context of invocation is found and the adaptation is better suited to the class in question. In C++ member function overloading can be used for that purpose. Overloaded member functions have the same name but their parameter lists are different. However, there may be a new context, which would like to use the earlier parameter list but to adapt the code. Scattered fragments of code add the tracing time needed to understand and change the code and thus reduce productivity.

It is also a problem that a polymorphic call is implemented in different classes. Their interface is the same but the implementation can evolve to different directions, which can lead to inconsistent behavior later. Some of these problems can be avoided by status checking at run time.

The encapsulation of the code only within abstract data types (and their classes) can lead to explosion of the number of classes and methods. In the electronic data interchange (EDI) a supplier's customers have different product ordering formats though only the supplier's generic order processes should be adapted to each format. There are, for example, 100 data formats and 10 processes. The traditional object-oriented solution would be 100 classes, which contain a method for each process. The total number of methods is 1000. Any change in the process must be made in all of these 100 classes. A better solution would be having 10 classes for each process or one class, which contains methods for all 10 processes. Each method must be capable of handling every data format. If there is a method to adapt the customer's ordering format to the supplier's generic format only 100 methods and 10 methods to support the processes are required. In that solution only 110 methods are required. A solution, which is based on the similar idea, has been implemented successfully in Noma Industries (Bassett 97, pp. 188-195).

Extensibility by templates

Templates are a way to increase flexibility in object oriented class hierarchies. Templates are used to define parameterized classes. For example it would not be reasonable to define an own class for each type of items in a collection. Figure 4 is a picture of these two kinds of flexibility.

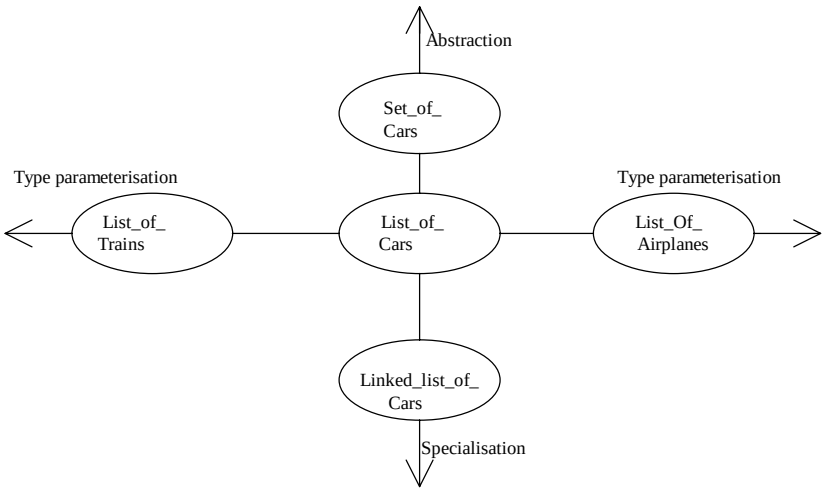

Figure 4. Horizontal and vertical generalization (adapted from Meyer 97, pp. 317-318).

Parameterized classes are also added to Unified Modeling Language (Booch 97, pp. 26-27) to be used in static structure diagrams. UML also includes another mechanism for the classification of types and use cases: stereotypes. A stereotype is a classifier marked by <<stereo type name >> near by the name it is classifying.

The previously described problem in getAddress-function can be solved using C++ templates. A generic find algorithm was needed (Koenig 96). In a template the class to be used is parameterized. The requirement that the classes used implement compare-operation still remains. The support to operation ++ corresponds support to function GetNextItem.

Templates are useful in producing collections, which contain many types of items. These items are iterated and polymorphic calls are done during the iterations. The use is not constrained to that kind of use.

The lessons from templates are their versatile use in classifying abstractions and in producing generic reusable components. Templates add a second way to produce reusable components besides the traditional inheritance.

Bassett Frames

Frames are a specific way to implement reuse (Bassett 97, pp. 70-195). It is based on a frame processor, which modifies source code using specific directives. These directives direct normal copy-paste and find-replace modifications, which the programmers usually make in white-box reuse.

It is stressed that in construction, properties of reuse such as generality and adaptability are most important. In use it is important that the component's functionality, efficiency and ease of use are appropriate.

Bassett's frames are a variation of a macro language. Many current programming languages include a macro language but it has remained in the background. Copying code is familiar from COBOL copy-statements and C 's #include directives. Variables can be assigned by #define directives and they can be tested by #ifdef and #ifndef directives. Macros are indeed a mechanism for reuse, which has been used for a long time.

The lesson from Basset comes from the better utilization of the characters of a macro. Adding a programming language level support to direct the generation of the source code adds flexibility to it. In macros there were multiple level copies but the intelligent direction was usually missing. Macros are also considered difficult to

understand and as tools of experts. That is because the outcome of realizing the macro was typically left inside the compile process. The code construction process contains only recompilation. Very simple directives typically direct code generators. The outcome of a generation is typically defined by the hard-coded inside the generator.

The second lesson comes from seeing reuse as a construction time process ruled by same-as-except principle. The traditional way of programming, taking a chunk of code and pasting it to a new place and making little editing is extremely effective in a settled environment. Maintenance is the problem because keeping copies consistent remains as the programmer's responsibility.

The third lesson comes from the principle of maximum diversity with the minimum number of components. Frames capture processes in a way that it is possible to reuse the process implementation with any objects needed.

Guidelines

Interfaces

Interfaces are used to reduce complexity by structuring components. The goal is to reduce coupling and increase cohesion. In reuse, it is essential that a component can be removed from its original context and used in another.

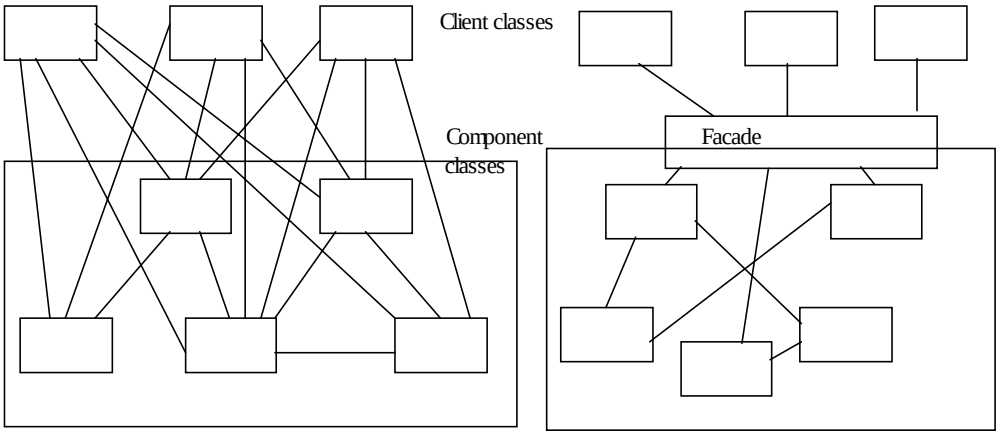

Figure 5. A facade design pattern (Gamma 95,pp. 185-193).

A facade design pattern is simple interface. Figure 5 is an example of the use of the façade where the coupling between the client classes and the component classes has decreased. The application programmer needs less tracing to find the services of the subsystem. He/she does not need to understand the implementation issues of it. That is another advantage because the implementation can be changed provided that the interface remains the same.

UML has new diagrams to support the use of interfaces, packages and components. Jacobson proposes to use a facade and a package in reusable software components (Jacobson 97, page 93).

The packages are the way to describe the scope of the variables in a diagram. In C++ namespaces are introduced to prevent the conflicts of variable names in packages from different sources. For example many diagrams of GUI-applications contain classes from Microsoft Foundation Classes. This utility package uses a large number of names,

which must be imported to the diagrams of the application. Name conflicts must be prevented also in new versions of the utility package and the application.

The placement of classes in source files is also a way to package the application programs. The concept module is used to denote one or more (typically source and header) source files. Programming languages typically introduce scope restrictions using these.

In distributed objects the interface is described separately using a special type of definition language. This created another kind of an interface between the client application and the objects that create the services to them.

Interfaces are however a kind of an agreement between the components. It is possible to misuse an interface. For example, adding a call interface to dynamic SQL in a database interface creates a fast path from the client to the internal structure of the database. It is then easy to create a client application, which has high coupling with the database.

An interface introduces one additional level of indirection between the components. Adding levels of indirection give flexibility but it has a performance penalty. Orfali has presented two quotations of generalizations of that (Orfali 96, page 505):

> Maurice Wilkes: There is no problem in computer programming that cannot be solved by an added level of indirection.
>
> Jim Gray: There is no performance problem that cannot be solved by eliminating a level of indirection.

The architecture of an application describes the most important interfaces within the application. These interfaces are used to reduce dependencies between the components.

> Well-defined object-oriented architecture consists of (Booch 96, page 43) a set of classes, typically organized into multiple hierarchies and a set of collaborations between those classes.

Booch (Booch 96, page 44) also emphasizes that a single class hierarchy is suitable for only the simplest application. Every other system should have one hierarchy of classes for every fundamental abstraction in the model. These groupings are called class categories or stereotypes. The latter is used in UML (Booch 97, pp. 16-17). A system with for example 50 - 100 classes should be divided to stereotypes, which contain about a dozen classes. It is also emphasized that common behavior should be handled through common mechanisms.

The overall view of the architecture can be seen as layered (Jacobson 97, pp. 170-212). A tree-tiered architecture is also common.

The lecture of this chapter is that the object models are not simple one-hierarchy based collections of classes. The abstractions must be classified to a suitable level. The object oriented methods to handle abstractions: classification, generalization and information hiding must be applied at all levels of abstractions. Also the behavior must be included. That suggests an approach, which is more like the natural language way of handling abstractions.

The performance drawbacks due to interfaces suggest handling interfaces before run time. The programmers need interfaces to comprehend the application more easily. At run time, memory and execution time requirements need another solution.

Combinations

Let us consider why a list box component is so useful. A list box component is a chunk of code that draws a list box in a window. That kind of component is available in all popular GUI programming environments. There are differences in their particular implementation but here it is sufficient to think about the most popular of these: the ActiveX-component in Microsoft Windows.

The application program adds text strings to a list box. The list box is shown in a window and a user selects one or more items from the list box. The application program acts based on the selection event of the user. Code can be attached to other list box events like event got-focus. The selection of a user is available in the other parts of the application program. The outlook of the list box can be modified both while programming and in run time.

The secret of the usability of a list box is in that it is so generic. It can be used in any window and in any application. It produces a well-defined contribution to the application, but it depends on few other components.

A generic component can be coupled with many other components. Figure 6 shows an example of a component list box of projects is shown in a project maintenance window. The projects are fetched using a query of projects and their names are added to the list box. In a window for handling tasks a list box is filled with task names using a query of tasks of a selected project. In the task window, new tasks can be added to a database using a list box of projects. From that list box the name of the project is picked to use in the code chunk, which realizes the insertion. A query fetches a collection of text strings, which are then added one by one to the list box. The order number of the selected text string indicates the user's selection.

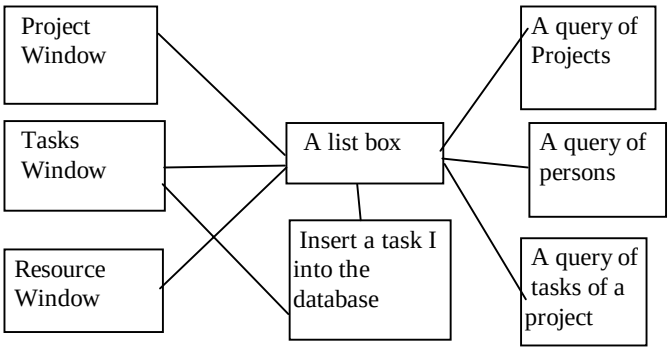

Figure 6. List box in project management application.

Part of the usability of a list box comes from generic interfaces. The message parameters are strings and integers that do not restrict the usability. Any component can send a message of a type myListBox.addItem (textStringToAdd) . If the interface were in form myListBox.addItem (queryToGiveTheItem), the list box could be used only with specific queries. In the same way it is essential that a window is a container, which can hold any types of objects. It is not restricted to list boxes only, also combo boxes and grids can be used. The only restriction is that the objects in a window must be able to support the responsibilities of a window control. In practice they are inherited from the control class.

Generic interfaces enable *the combinatorial explosion* of the use of a component. The list box can be used with any data source and in any window. Though the component itself is a standard component, there are millions of ways to use it. In some environments it is possible to make specialized list boxes: a project list box, a task list box and a person

name list box. That could be useful if each of them can be used in many occasions and there is a large amount of new code in each specialization. Normally it is sufficient to apply to the combinatorial explosion in a call of a function. It should be noticed also that the specialization should make a special kind of a list box. The code to fetch task data is not in a proper place when placed a list box class. Applying inheritance when no new code is introduced is called taxomania (Meyer 97, pp. 820-821).

In three-tiered architecture the business object classes are used to couple the user interfaces to the data storage. They fix the combinations used in a certain business case. It should be possible to change the user interface classes and data storage classes without major maintenance work in business classes. It is possible if the details of the user interface and storage management are properly hidden from the business objects.

In some cases using static combinations is not flexible enough. The requirement of recompilation each time that any combination changes will result to too much maintenance and installation work. In distributed object environments it is possible to make the matching of components together at run time. The performance overhead can sometimes be tolerated.

The problem in current programming languages - typing

The basic construct of typing is the execution of a message send x.f (arg), which executes the operation f on the object attached to x using argument arg (Meyer 97, page 611). It is also possible to have no arguments or more than one argument. A type violation occurs, if there is no function f within the inheritance tree of the object attached to x or arg is not an acceptable argument of function f.

In dynamic typing the checking of type violations is done at run time during the execution of the function call. Static typing uses rules that determine that type violations will not occur. These are checked at compile time. Statically typed languages require that each variable is declared to be of a certain type. They check every assignment and function call and assure that no type violations occur.

In an inheritance tree, there can be more than one function, which realizes the typing rules. The difference between typing and binding is in that typing is considering whether there is at least one applicable function. Binding considers which one of these should be used. So it is possible and even desirable to use static typing and dynamic binding.

The benefits of static typing come from better reliability, readability and efficiency. Static typing is used to detect type violation errors at compile time. Dynamic typing could detect them only at run time and in certain runs. If high reliability is required, for example in patient monitor programs, dynamic typing is not appropriate at all. Type definitions make programs more readable because the programmer can find out the types of the variables from the definitions. Variables, which change type during the program run, are especially difficult to trace. Better efficiency is due to better binding algorithms. It is quite normal that the name of a function is unambiguous. In the context of static typing the algorithm searches only for the polymorphic functions instead of all functions in the application (Meyer 97, pages 615-616).

The drawback of static typing comes from its restrictions to reuse. It is restrictive that the code of function f can be reused only when the types of x and arg are each within the same inheritance trees. Those, who are used to program using Smalltalk, use this as a main argument to the choice of that programming language. In some applications it is even desirable to be able to add new functions and attributes to a class in run time.

Static typing has also a problem when it is needed to override a function. These are named as covariance and descendant hiding (Meyer 97, pages 621-628). Covariance is concerned about the arguments of f when the class of x is redefined. In a descendant hiding, the function f does not exist in the descendant class. Both of these features are very desirable (Bassett 97, page 141). Fortunately, it is possible to check both of them in compile time (Meyer 97 pages 621-628). The solution is in checking all the assignments and function calls and rejecting those, which could result to a type error. The fact that it prevents also some possible solutions, which would not result to a type error in any practical program run, is not a serious problem.

Possibilities to added reuse can be made better if the static check is based on checking valid calls. The idea of type checking is to prevent invalid function calls. It is sufficient that the arg is attached to an object that supports all the function calls of arg within f. Recall the previous example of getAddress function. Its parameter parameterName is of the type FullName. The parameter is used in making function call parameterName.SurnameOf(). It would not be necessary that parameterName is of the type FullName or some of its descendants. It would suffice that its type contains the function SurnameOf. That would increase reusability at the cost of decreased readability.

Conclusions

Reusable software components are not a by-product of normal development. Typically, software consists of specific parts, which have a large number of specific dependencies. It is necessary to have clear and generic abstractions to increase reuse. The abstractions should be selected in a way, which enables the combinatorial explosion when they are mixed together. That can be accomplished by using generic code, which can handle a large number of data types. Besides increasing cohesion, decreasing coupling is also necessary. That can be accomplished by adding interfaces to break the dependencies between the components. The package of a component must contain proper documentation to assist the understanding of its intended use.

It is possible to extend the traditional methods of programming to increase reuse. There is a trade-off between adaptability and type safety. More research is needed in producing programming languages, which are more expressive than current languages, but which are still safe to use. The combination of functional and object-oriented programming languages is a way to improved expressiveness. The verb inheritance is an approach in which the combinatorial explosion of expressions as seen in natural languages can be used in a programming language (Virtanen 99).

Bibliography

Paul Bassett: Framing Software Reuse, Prentice.HalL, 1997.

Grady Booch: Object Solutions, Managing the Object Oriented Project, Addison-Wesley Publishing, 1996.

Grady Booch, Ivar Jacobson, James Rumbaugh: UML 1.0 reference, Rational Corporation, 1997.

S. N. Cant, B. Henderson-Sellers, D. R. Jeffery: Application of cognitive complexity metrics to object oriented programs, Journal Of Object Oriented Programming, 7(4),52-63,1994.

Martin Carroll, Margaret Ellis: Designing and coding reusable C++, Addison-Wesley Publishing Company, 1995.

Gert Florijn, Marco Meyers,Pieter vanWinsen: Tool Support for Object-Oriented Patterns, Lecture Notes in Computer Science Vol 1241: ECOOP'97, Springer-Verlag 1997, 472-495.
Erich Gamma, Richard Helm, Ralph Johnsson, John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing, 1995.
Adele Goldberg, Kenneth Rubin: Succeeding with Objects, Decision Frameworks for Project Management, Addison-Wesley Publishing, 1995.
Ivar Jacobson, Martin Griss, Patrik Jonsson.: Software Reuse, ACM Press, 1997.
Andwe Koenig, Barbara Moo: Ruminations on C++, Addison-Wesley Longman inc., 1996, page 213.
Mark Lorenz, Jeff Kidd: Object oriented software metrics, Prentice-Hall, 1994.
Bertrand Meyer: Object Oriented Software Construction 2nd Ed., Prentice-Hall, 1997.
Microsoft: Microsoft Visual C++ Class Library Reference, Volume I, Microsoft Corporation, 1993.
Robert Orfali, Dan Harkey, Jeri Edwards: The essential Distributed Objects Survival Guide, John Wiley & Sons inc, 1996.
Jeffrey Poulin: Measuring Software Reuse, Principles, Practices and Economic Models, Addison-Wesley Publishing, 1997.
Ruben Prieto-Diaz, Peter Freeman: Classifying Software for reusability, IEEE Software, Vol 4 No 1, January 1987, pp. 6-16.
Will Tracz: Confessions of a Used Program Salesman, Addison-Wesley Publishing, 1995.
Pentti Virtanen: Object Oriented Process Metrics, Proceedings of the 21st Information Systems Research seminar in Scandinavia, page 919, Department of Computer Science, Aalborg University, 1998, 919-937.
Pentti Virtanen: Extended Reuse with Verb Inheritance - A New Approach to Software Construction and Reuse, In F. Gerhardt , L. Benedicenti and E. Ernst (eds), Position Papers from the 8th Workshop for PhD Students in Object-Oriented Systems, page(s) 147--154, Department of Computer Science, University of Åårhus, Denmark, 1999.