Maintainable Systems with a Business Object Approach[1)]

Wiebe Hordijk[2)]
Division of Computer Science
Vrije Universiteit
1081 HV Amsterdam, NL

Sascha Molterer, Chris Salzmann
Institut für Informatik
Technische Universität München
80290 München, F.R.G.
molterer@in.tum.de

Barbara Paech[3)]
Fraunhofer Institute for
Experimental Software Engineering
67661 Kaiserslautern, F.R.G.

Panagiotis K. Linos
Department of Computer Science
Tennessee Technological University
Cookeville, TN 38505, USA
linos@csc.tntech.edu

[1)]The work for this paper was partially supported by the Bayerische Forschungsstiftung via FORSOFT project A3

[2)]Most of this work was carried out while the author was at the Technische Universität München.

[3)]Most of this work was carried out while the author was at the Technische Universität München.

Abstract

The concept of Business Objects (BOs) has been recently promoted as a new way of exploiting object-orientation for achieving large-grain reuse. In this paper, we address the issue of how to effectively re-engineer business software applications using BOs as a reuse technique. To this end, we first identify the reuse features of business objects and then compare them with other reuse techniques. In addition, we show that software re-engineering can be more economical when business objects are used. Our work also provides guidance on how to develop and use a Business Object Architecture (BOA), which is shared by a group of interrelated and interdependent software applications. We argue that such architecture allows for more efficient reuse and better maintainability and it is illustrated by means of a case study in a realistic manufacturing environment.

Keywords:*Software Maintenance, Software Reuse, Business Objects, Software Architecture, Case Study Software Re-engineering, Reverse Engineering*

1 INTRODUCTION

The success of international business enterprises is contingent upon the abundance of already existing legacy software. In order for these companies to remain competitive and effective, they have to allow for a systematic evolution of their business processes. A key issue in such an evolution is the change and maintenance of their software. Such maintenance can be sometimes radical and very costly but in most cases the required change is gradual and incremental investment of resources is necessary.

It is commonly accepted today, that software reuse is a promising approach for facilitating the evolution of software-intensive systems [Biggerstaff and Perlis 1989, Dunn and Knight 1991, Krueger 1992, Sametinger 1997, Jacobson *et al.* 1997]. In particular, systematic reuse aims to produce software from a portfolio of reusable artifacts, so that architectural similarities and common requirements among applications can be exploited to accomplish significant improvement of the productivity, quality and performance for software companies.

According to Jacobson, there are several organizations, which have demonstrated reuse levels up to 95% [Jacobson *et al.* 1997, p. 6]. Such organizations include, among others, AT&T, which achieved a reuse level of 92%, Ericson AXE, with 90% reuse, Motorola 85%, and Brooklyn Union Gas with a reuse level of up to 95%.

Basically, there are three ways of achieving effective software reusability [Canfora *et al.* 1994]. The first way is to purchase reusable software artifacts from the market (known as COTS), the second is to develop them from scratch, and the third way is to re-engineer them from existing software. The third approach is expected to bring short-term results and it is the focus of this paper.

In particular, we focus on the re-engineering of existing applications based on business objects. Business objects have recently been put forward as a new way of exploiting object-orientation for large-grain reuse [Shelton 1998]. They serve as a storage place for business policy and data. Academic and industrial activities have focused mainly on the component technology associated with business objects, namely business object facilities [Emmerich and Ellmer 1998]. However, there is little guidance on exactly how to develop a business object architecture for a set of applications.

In this paper we propose a rationale for developing a shared business object architecture (SBOAs) to allow for more efficient reuse and better maintainability. It is tailored to the application of business objects for re-engineering of an interrelated set of applications.

In order to illustrate the proposed SBOA, we discuss a case study using an existing business application which we call P in the following. P is an in-house developed information system, owned and currently used by a big manufacturing company in Germany. It is primarily used for managing data, and related technical drawings for parts. P is implemented using the SMALLTALK object-oriented programming language.

The rest of this paper is organized as follows: First we identify the reuse features of components - which are the basis of business objects - by comparing them with other reuse techniques known from the literature. Then we define the main concepts in the realm of business objects and collect their maintenance requirements. Based on that we present the SBOA and the main issues in the design of an SBOA. We discuss how to find the balance between shared and application-specific features of business objects as well as how to evolve dependencies between business objects over time. These issues, as well as the concept of an SBOA are illustrated with the system P. We close with future work.

2 REUSE

In this section we will recapitulate the essential reuse facets from the literature and typical mechanisms for code reuse. The aim is to provide the foundation for the assessment of the reuse features of business objects in the next section.

2.1 Reuse Facets

[Krueger 1992] is an extensive survey of reuse techniques. It distinguishes the following facets of reuse techniques:

- abstraction,
- specialization,
- integration mechanisms and
- interface description.

The interface description makes explicit the external facets of a reusable unit to be used by the environment, the integration mechanism provides the glue between the reuse units. It is typically based on a particular model of allowed interactions between the units. The mechanism can be applied at development, build-time or run-time [Karlsson 1995]. Typical interface descriptions are import/export, include, provided properties or provided capabilities.

Specialization (often also called customization) is achieved by changing the variable part of the reuse unit. The variability can be due to parameterization, configurability and to underspecification. The latter requires the addition of details, either declaratively (constraints) or constructively (refinement, in particular inheritance).

The abstraction facet deals with the granularity and the description level. In [Krueger 1992], however, this distinction is not made. The granularity ranges from single code elements (statements) which are meaningless without the environment to a complete architecture which is self-contained. In between, there are on the one hand patterns which describe projections of complete designs wrt. specific aims and on the other hand components which are parts of the complete design delivering a specific functionality, but usually depend on one another in the delivery. The description level typically ranges from machine code to domain specific specifications. In between, there are programming languages and general specification languages. Of course, there are some dependencies between the granularity of the unit and the description level: for example code elements are not described with specification languages. However, for the more modern reuse

Granularity	**Description**	**Specialization Mechanisms**	**Integration Mechanisms**	**Interface Descriptions**
Code Elements	Machine Code	Parameter	Development time	Import/Export
Pattern	Programming Languages	Configuration	Building time	Provided Capabilities
Component	Specification	Refinement	Run-time	Include
Architecture	Domain specific Specification	Constraints	Ad hoc	Provided Properties

Table 1: The facets of reuse techniques

techniques like components, different languages can be used in parallel. In particular, interface specification languages become very important.

Table 1 summarizes the different facets of reuse techniques and their typical values.

2.2 Reuse Technique Classification

Table 2 classifies the major techniques for code reuse according to the facets identified in the last section. Besides the facets we have also characterized the aims of the techniques, since, often, reuse is not the only aim for the technique.

According to the level of granularity, there are four major classes of code reuse techniques

1. **High-level programming constructs** aim at platform independence and abstraction, they typically apply to code elements and use parameterization for specialization. Integration is done at build time or development time. Interface description is not used.

2. **Components** provide the most precise notion of interface. [Krueger 1992] discusses code and execution components. In addition, we look at classes, which are reused through inheritance. This is the major contribution of object-oriented programming languages to reusability.

3. **Patterns** aim at reuse of experience. On the code level, one can distinguish design patterns from idioms. The latter are specific to the programming language. The description is very comprehensive. Specialization is done by refinement, integration is done at development time and the interface is only described in terms of the provided capabilities.

4. **Frameworks** also try to capture experience. However, in contrast to patterns they are self-contained. The specialization mechanisms vary. Since they are self-contained, they don't need to deal with integration and interface description.

Technique	**Granularity**	**Specialization**	**Integration**	**Interface**	**Aim**
High-level Programming Language Constructs	Code Elements	Parameterized slots	Build-time (compiler) or development time	None	Platform independence, understandability
Classes	Component	Refinement (Inheritance)	Build-time	Provided Capabilities	Reusable, data-centered
Code Components	Component	Configuration	Build-time, programming language specific	Include	Facilitate installation
Execution Component	Component (Threads, Tasks)	None	Run-time system of the programming language.e.g. schedulers	None	Describe dynamic structures making use of the runtime environment, for analyzing run-time properties
Design Patterns	Pattern	Configuration Refinement	Development-time (combination)	Provided capabilities	Capture design experience
Idioms	Pattern	None	Development-time	Provided capabilities	Capture implementation experience
Framework	Architecture	Inheritance or Configuration	None	None	Reuse of architectures

Table 2: Classification of some Major Reuse Techniques

3 THE BUSINESS OBJECT APPROACH

This section introduces business objects and discusses their reuse features. This discussion will make apparent the need for the shared business object architecture (SBOA) . The design rationale for an SBOA is the main contribution of this paper and is introduced in the next section.

3.1 Business Objects and Reusability

According to [Shelton 1998] business objects describe a thing, concept, process or event in operation, management, planning or accounting of a business organization. This distinguishes them from technological objects like GUI-objects. They are enabled by business object facilities which provide transaction and persistence [Emmerich and Ellmer 1998]. Business objects are components and therefore focus on large-scale reuse. As sketched in the last section, they can be reused on the level of classes as well as in the form of code components. The main difference is the specialization mechanism. Classes are specialized through inheritance and code components through configuration.

Clearly, configuration is the more simple mechanism. However, it is not sufficient in a large organization, where it is not possible to foresee all possible new applications which would like to reuse the business objects. Thus, there will always be the need to adapt the given business objects to different contexts. This can be achieved through inheritance, but requires a careful selection of the business features which are incorporated in the general classes. We will discuss this topic in detail in the next section.

3.2 Maintainability and the Shared Business Object Architecture

While in the literature, the focus is on business objects common to several businesses -as a basis for COTS -, we focus on the business objects common to a set of interrelated applications within an organization. In the following, we discuss the maintenance requirements resulting from this context.

Within most large organization applications are developed locally to individual departments. Thus, the requirements engineering, design and implementation of the application is tuned for that department. However, business processes typically affect several departments. Thus, parts of the business process are implemented redundantly in several applications. While this often gives rise to inconsistencies hampering the daily-work of the employees, it also affects severely the maintenance efforts needed for the applications. If the business process changes, all the applications have to be maintained.

To remedy this situation, re-engineering clearly has to strive for factoring out the business logic common to several applications. This common business logic should be reused through the different applications. The reuse could take place at development time, where the business logic is copied to a new application. However, while this reduces the development effort for the new application, it does not reduce the mainte-

nance efforts. Therefore, from a maintenance point of view, the common business logic must be part of an architecture which is shared between the applications.

3.2.1 Business Object Architecture

The typical **business object architecture (BOA)** separates the business logic from the application specific parts as well as from the persistence mechanism [Hung *et al.* 1997].

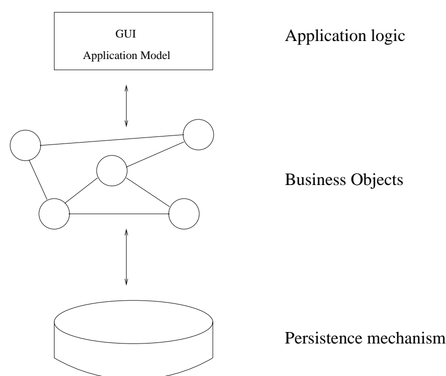

Figure 1: business object Architecture

As depicted in figure 1 the BOA consists of the

Data model tier: this basic layer holds and manages the data model of the system. This could be realized i.e. by a database or a repository system.

Business logic tier: this layer holds the encapsulated business logic which is realized as business objects.

Application logic tier: here the parts of the application specific logic are positioned. These might be GUI or application specific data or processes.

It is important to note that this separation is the ideal case and can hardly be practiced in existing system. The borders of the layers are in real life system not sharp.

The BOA is clearly sufficient to allow the change of the business logic for a single application. However, as discussed above, for a set of interrelated applications the business logic needs to be shared between all the applications. Also, this sharing should incorporate sharing of classes as well as of their instances. Therefore, we introduce in the next section the concept of a shared business architecture.

3.2.2 Shared Business Object Architecture

As depicted in figure 2 a **shared business object architecture** is a BOA which allows the business objects to be shared between several applications.

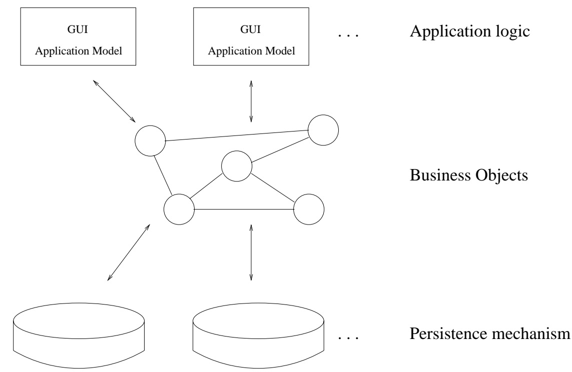

Figure 2: Shared business object Architecture

While this concept straightforwardly fulfills the maintenance requirements identified above, it introduces additional complexity in the design of the business logic tier, since it needs to fulfill the needs of several applications without introducing unnecessary dependencies between the applications. The next section will propose a solution as to how to handle this complexity.

4 THE DESIGN OF A SHARED BUSINESS OBJECT ARCHITECTURE

This section introduces a rationale for the design of an SBOA. In particular it treats the design of shared business logic and the management of dependencies between shared business objects.

4.1 Design of the Shared Business Logic

The main question in the design of the shared business logic is how to deal with features relevant for more than one application. Typically, these features are incorporated in a general class in the business logic layer so that application-specific classes inherit from the general class. However, this might induce unnecessary complexity on the specific classes for one application, when the general classes include features shared between other applications. Separating these features into general classes only shared by a subset of the applications, on the other hand, reduces the possible sharing between applications, since only instances of shared classes can be shared. In the following we will discuss the possible solutions based on a highly simplified model consisting of three classes as shown in figure 3. Classes are sets of features (attributes or methods). Features are independent of each other. Classes are named A, B, C. In some of the solutions discussed below intersection classes like $A \cap B$ (written AB for short) are introduced. The small letters a, b, etc. are used to indicate the features: $A = a \cup d \cup e \cup g$, but $a \cap d = \emptyset$.

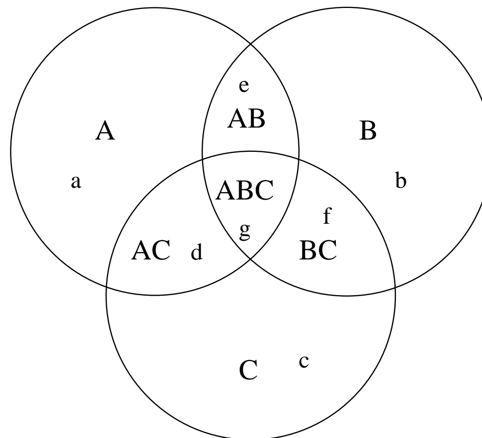

Figure 3: Simplified scenario with three classes

Before we start with the discussion of the different solutions, we introduce the implementation cost as the criteria for how to evaluate them. To emphasize the sharing of a business object by several applications we call the applications *reuse contexts* of the business object.

4.1.1 Implementation Costs

The key issue on software development and reuse are the factors of cost and quality – as in every business domain. For the term of software there is a special relation between the factor of cost and the issue of reuse and sharing: in the general software development process it is – in the short term – cheaper to build software that is application specific. In the long term, however it is cheaper to build generalized, not application specific software, which can be reused in other applications.

Formally this means, that implementing a feature φ has a cost K_φ. The cost K_S of implementing a set of features S is the sum of the costs of implementing the elements:

$$K_S = \sum_{\varphi \in S} K_\varphi \tag{1}$$

Because higher quality requirements are imposed on classes which are meant for reuse, the cost of implementing a feature in a general class is usually higher than that of implementation in an application-specific class. These additional costs are due to e.g. special quality assurance measures like inspections in the early phases or testing. Another cost driver might be that features of general classes are designed to be more general. Coming up with this generality usually requires further effort. In our view, it is safe to assume that these additional costs are proportional to the cost of implementing the feature in the application-specific class.. Thus, in the following we use the factor x for *generalization costs* and the cost of implementing a feature in a general class is xK_φ.

4.1.2 Big class

In this approach, the designer tries to foresee as many as possible future reuse contexts for the business object. All features that might be needed, are added to the class. All anticipated reuse contexts are documented. For the features, the reuse contexts for which they have been anticipated are documented.

This approach can only be applied in situations, where all or most of the reuse contexts can be foreseen. The cost during initial development is increased. The abundance of unused features may hinder performance.

Because the same class is used in every reuse context, the instances can be shared easily. During development of a new reuse context, none of the code has to be changed. Thus, reuse is very cheap.

Looking at our example, the big class would contain all features of classes A, B and C. Since all features are implemented with reuse quality, the total cost becomes:

$$K_{tot} = x(K_a + K_b + K_c + K_d + K_e + K_f + K_g)$$

4.1.3 Subclassing

In subclassing approaches, certain general features are implemented in a general class. For use in applications, this general class is subclassed and application-specific features are added. Since there are several policies for deciding which features go in the superclass and which go in the subclasses, three approaches using subclassing are given.

The fact that a class in a certain reuse context inherits the general class, might cause problems when we want the class in the reuse context to inherit another class. Some languages, like Java and SMALLTALK, do not support multiple inheritance. In such cases, the designer must choose which of the possible parent classes to inherit; for the other inheritance relationship, another construct, like delegation, can be used.

Because all classes have the same superclass, instances can be shared, but only the features that have been defined in the superclass are generally available. Moreover, there is an instantiation problem: when two applications A and B use their own subclasses XA and XB of superclass X, then an instance of XB created in B cannot be used by A as an instance of XA. In the Big Classes approach, they are all instances of the same class.

4.1.3.1 Big superclass: In this approach, the designer tries to identify those features that occur in at least two reuse contexts. Those features are implemented in a general class. The application-specific features can be added in one subclass for each application.

Like the "Big class" approach, for this approach the designer needs to be well aware of the possible reuse contexts. Unused features may hinder performance, but not as much as in the "Big class" approach.

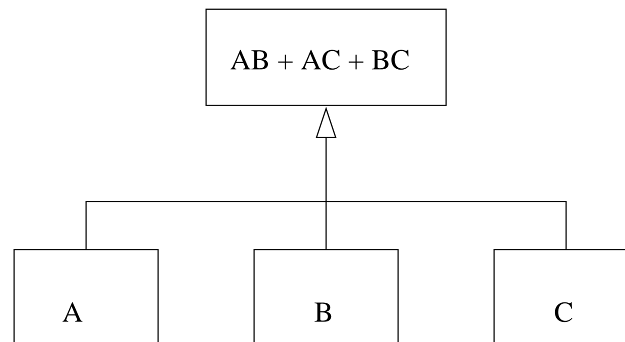

Figure 4: Class diagram for our example with a big superclass

For our example, a class diagram illustrating this approach is shown in figure 4. The superclass comprises the features in parts d, e, f and g in figure 3. The cost of the example becomes:

$$K_{tot} = K_a + K_b + K_c + x(K_d + K_e + K_f + K_g)$$

4.1.3.2 Small superclass: In this approach, the designer tries to identify those features that will be needed in every reuse context. These core features are implemented in a general class. When other features are needed, the designer adds those in a subclass.

Compared to the "Big superclass", this approach has lower initial cost but higher reuse cost. Features that occur in more than one reuse context, but not in all of them, are implemented more than once.

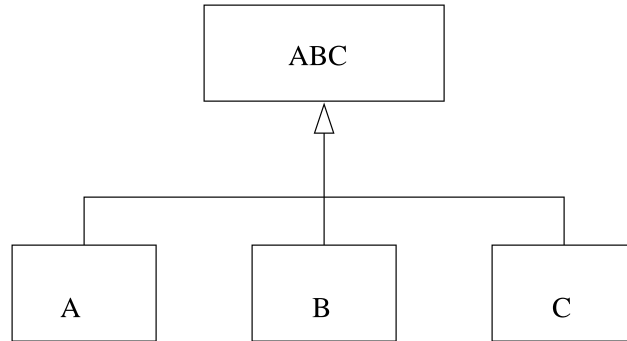

Figure 5: Class diagram for our example with a small superclass

Figure 5 illustrates this approach for our example. The superclass corresponds to part g of figure 3. The cost is:

$$K_{tot} = K_a + K_b + K_c + 2(K_d + K_e + K_f) + xKg$$

4.1.3.3 Multiple inheritance layers: In this approach, inheritance is maximized. Those features that occur in every reuse context are implemented in a general class, just like in the "Small superclass" approach. This class is inherited by classes that contain features that are needed in some, but not all, reuse contexts. Finally, the application-specific classes are derived from general classes that are as specific to that application as possible.

This approach is illustrated in figure 6. Assuming that the same quality measures are taken in the middle layer, the cost is the same as for the "Big superclass" approach:

$$K_{tot} = K_a + K_b + K_c + x(K_d + K_e + K_f + K_g)$$

This approach becomes problematic when more classes are added due to more reuse contexts. With four reuse contexts, if there are overlaps between all combinations of them, we end up with four layers and a total of 15 classes. This is clearly not feasible. In languages that do not support multiple inheritance, this approach is not possible at all.

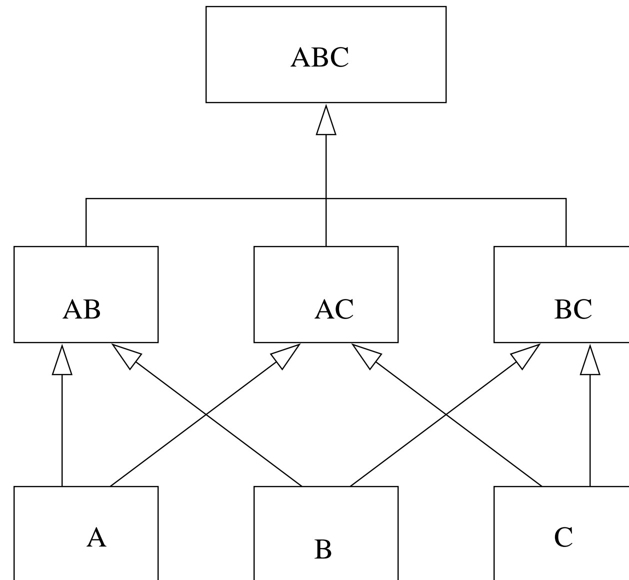

Figure 6: Class diagram for our example with multiple inheritance layers

4.1.4 Extensions

This approach is available in certain environments, like IBM San Francisco [IBM 1998] and the SMALLTALK language. One class with the core features is developed, like the superclass in the Subclassing approach. Features needed in specific contexts can be added in extensions. Extensions are added at runtime to each instance separately. This offers a bulk of possibilities for other properties, but for reuse, we only look at the case where the same extension is added to each instance of a class in a certain application.

Like for subclassing, there are a number of policies for whether to add a feature to the class or to an extension. These approaches are analogous to the subclassing approaches, with the only difference that features are not implemented in subclasses but in extensions. The cost formulas are the same, too. Therefore, we do not repeat the individual approaches here.

A disadvantage of extensions is that they can only be used in certain environments. An advantage is that all instances are of the same class, so instances can be shared.

4.1.5 Reimplement

If only a small amount of reuse contexts is envisioned for a class, or if the class is completely application specific, it may not pay off to design a class for reuse. If a class is reused that was not designed for it, one often has to reimplement most of the features. For simplicity we assume that all features have to be reimplemented.

Advantages of this approach, if it can be called one, are that the initial cost is not raised by reuse issues, and that no anticipation of possible reuse contexts is required. Disadvantages are that the reuse cost

is high and sharing of instances is impossible.

For our example, the cost is:

$$K_{tot} = K_a + K_b + K_c + 2(K_d + K_e + K_f) + 3K_g$$

4.1.6 Comparing the costs

We can generalize the cost formulas of the most important approaches from our three-classes example to the general case with n classes. When we have n classes that each (partly) overlap with each other, features can occur in any number of those classes, from 1 (application-specific) to n (core). We regard all features that occur in only one reuse context as a set, all features that occur in exactly two reuse contexts as another set, and so on. Note that these sets are mutually exclusive.

We can extend our cost function as follows:

K_i is the cost of implementing all features that occur in exactly i reuse contexts; $1 \leq i \leq n$.

Now we can express the cost of implementing all reuse contexts of a class as the costs of all features in all reuse contexts of that class, using the "Big class" approach, as follows:

$$K_{BigClass} = x \sum_{i=1}^{n} K_i, \text{ for } x \geq 1 \text{ and } n \geq 3$$

The same can be done for the "Big superclass", "Small superclass" and "Reimplement" approaches:

$$K_{BigSuper} = K_1 + x \sum_{i=2}^{n} K_i$$

$$K_{SmallSuper} = xK_n + \sum_{i=1}^{n-1} iK_i$$

$$K_{Reimplement} = \sum_{i=1}^{n} iK_i$$

If $x = 1$, that is, if no extra costs are invested in higher quality for reusable features, then the "Big class" and the "Big superclass" approaches have the same cost and are cheapest. When x increases, however, the "Big class" approach will quickly become more expensive than other approaches.

On the other end of the scale, if $x \geq n$, the "Reimplement" approach is the cheapest. Since we require n to be at least 3, this is not very likely to happen.

The interesting area is where $1 < x < n$. In this area, either "Big super" or "Small super" is cheapest:

$$K_{BigSuper} = K_{SmallSuper} \Leftrightarrow K_1 + x \sum_{i=2}^{n} K_i = xK_n + \sum_{i=1}^{n-1} iK_i$$

$$\Leftrightarrow \sum_{i=2}^{n-1} xK_i = \sum_{i=2}^{n-1} iK_i \qquad (2)$$

This implies that if $x < 2$, "Big super" will always be cheaper, and if $x > n - 1$, "Small super" will always be cheaper. When $2 \leq x \leq n - 1$, it depends on the costs of the individual features and the amount of features in every group. If the greater part of the features are used in many reuse contexts, "Big super" will be cheaper; if the greater part of the features are used by only a few applications, "Small super" will be cheaper. The greater x is compared to n, the more favorable "Small super" will be compared to "Big super". For fixed x, "Big super" gets more favorable as n increases.

These results are summarized in table 3 depending on the factor of generalization costs x.

Generalization Costs	Cheapest Reuse Strategy
$x = 1$	*big class* and *big super*
$1 < x < 2$	*big super*
$2 \leq x \leq n - 1$	*big super* or *small super*
$n - 1 < x < n$	*small super*
$x = n$	*small super* and *reimplement*
$x > n$	*reimplement*

Table 3: Reuse strategies compared on Reimplementation Costs

If an estimation of the generalization costs x somewhere between 1 and 2 is reasonable, and this is the case for a business object approach, then putting all features that are needed more than once into the superclass seems to be the best general approach. Therefore this is the approach that is taken in redesigning the case study in section 5.2.

4.2 Managing the dependencies between shared business objects

The sharing of classes and instances between different applications increases the interdependence of applications. While the business logic of one application is changed, all applications which share code with this application have to be stopped. Then the references to the shared objects and classes have to be updated according to the changes. In figure 7, all applications i, j, k and l depend on business object b. Changes of business object b caused by changes of the business logic of one of these applications affect all other applications. Therefore, applications i, j, k and l interdepend.

It is obvious, that the benefit of the reuse would be worthless, if for every modification of a business object that is needed by one application all applications have to be stopped and modified. Therefore, it is important to encapsulate the access instances and to instantiate a protocol between the objects which allow for the dynamic discovery of object features. Both cases will shortly be discussed in the following, for a more thorough discussion see [Salzmann 1999].

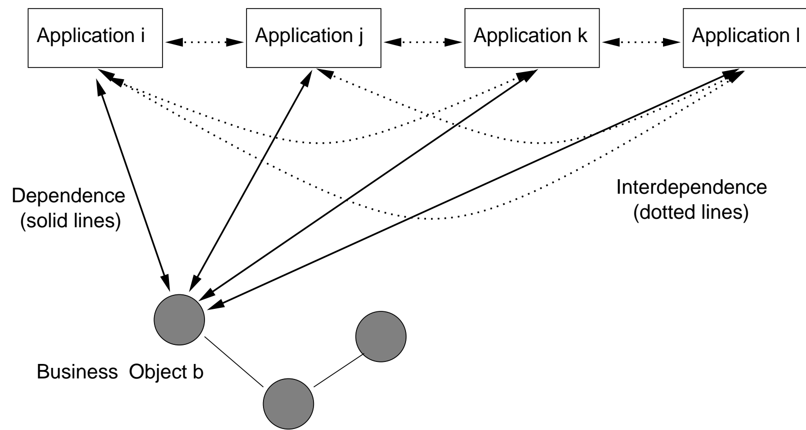

Figure 7: Raising the Level of Dependence by Sharing Business Objects

4.2.1 Semi-Dynamic Object Configuration

Semi-dynamic configuration management supports the encapsulation of object instantiation. It is already realized in frameworks like IBM's SanFrancisco through the use of *factories.*

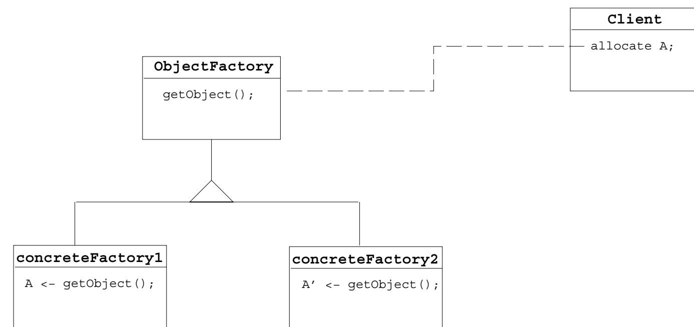

Figure 8: Semi-Dynamic Configuration: Instantiation Factories

Factories – a pattern, well known from [Gamma *et al.* 1994] – encapsulate the instantiation objects (see figure 8).

So for the case, that a class is changed such that instead of an object A an object A' is instantiated every time an object A is allocated in the source code, one needs only change one line of code in the factory instead of tracking all allocation occurences. The change can even take place at runtime, if the system supports dynamic linking facilities. This saves a lot of effort and raises the flexibility. For almost every business object factory San Francisco uses structures to ease the configuration.

However, semi-dynamic configuration changes only the instance, allocated in the *future.* The already existing instances remain the same and must be handled separately.

4.2.2 Dynamic Object Configuration

Dynamic configuration means the change of business objects and references at any time – including the runtime. One possible approach is to allow for dynamic discovery of object features. This means, that all current instances and all future allocated instances of a certain class A should be exchanged with an instance of another class A', by keeping the structure of connections – i.e. associations and aggregations. This change should happen during the productive phase – i.e. when the system is running and used – without any constraints. However, this is not yet – or only under special constraints – realized in current SBOAs.

5 A CASE STUDY: THE P SYSTEM

To illustrate our approach we discuss a case study where an existing application interrelated with several other applications is re-engineered using the SBOA. The purpose of this case study is to understand better the issues involved in re-engineering of applications with SBOA. It is not used to confirm the assumptions on generalization costs made in section 4.1, since this would require to compare the solutions with and without generalization using the same implementation technology. In the case study, only the former solution has been implemented.

In this case study, P is re-engineered by considering the commonly accepted re-engineering process model, namely reverse engineering followed by forward engineering [Byrne 1992]. During the reverse engineering phase, the design model is recovered from the existing source code [Tilley 1998]. It is worth mentioning that the typical assumption of no existing consistent documentation is also the case with P. Therefore the only reliable documentation of the system is the source code itself.

During the second phase, the recovered design model is modified and improved by keeping the functionality of the system the same. The aim is to create an easy-to-change and maintainable design for P while preserving its original functionality. The modification of the design is based upon the SBOA. We have formalized the design using CDL, a language defined by the Object Management Group (OMG) for describing object business architectures [OMG 1998]. Finally, during forward engineering, we used IBM's San Francisco [IBM 1998] as the underlining implementation framework.

Because of space limits, here we only present a small part of the new model. The detailed re-engineering products can be found in [Hordijk 1999].

First we sketch the current implementation. Then we discuss the design of the SBOA and its implementation.

5.1 Current implementation

P is an information system that manages engineering data on product parts. This includes part data, CAD/CAM documents, construction materials, norms and projects. The system communicates with another application, an information system that manages part lists, as well as with a digital drawings archive. P has been implemented in SMALLTALK, an object-oriented language, using VisualWorks, a SMALLTALK IDE from ParcPlace-Digitalk (now ObjectShare, Inc. — http://www.objectshare.com). It consists of 238 SMALLTALK classes from which 78 classes representing business entities, the remaining classes are for technical purposes (GUI, database connectivity, printing and so on).

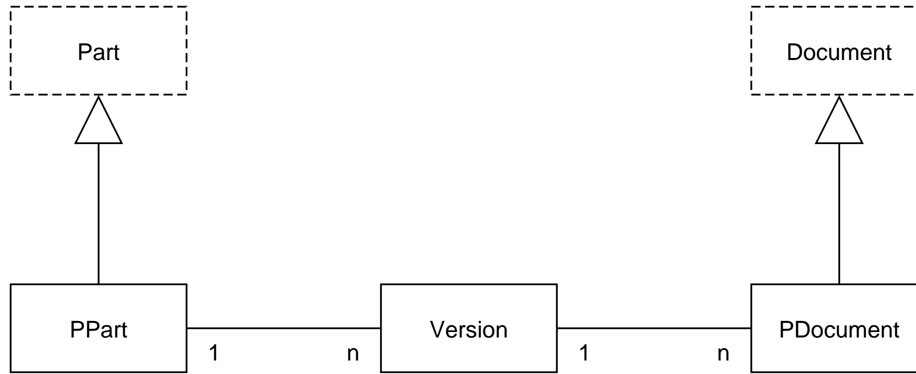

Figure 9: Most important classes

5.2 New model

The central issue in remodeling P for an SBOA is, which classes to regard as general classes and which as P-specific classes. Following the guidelines from section 4.1 we first have to determine the reuse features and their generalization costs. In the case study, generalization costs are 1, since no additional quality measures are carried out and also no effort is invested to make the features of the general classes more general. As pointed out, in that section, we believe that in general the generalization costs will be between 1 and 2. Thus, in the case study we followed the "Big super" approach from section 4.1.

So, we want to regard each feature that is probably going to be reused as a general feature. Classes which contain (almost) only such general features are general classes. Classes that contain no general features are P-specific classes. The general classes should be independent of the application-specific classes, that is, general classes cannot contain references to P-specific classes. Following these guidelines, we identified 19 general classes out of the above mentioned 78 classes which are representing business entities in P.

In the following, general classes are depicted as dotted rectangles, to distinguish them from P-specific classes, which are depicted as normal rectangles.

The classes shown in figure 9 form the backbone of P. P administers data about production parts and manufacturing aids, called Parts. Of these parts, several versions exist. Each version is described in several documents. P administers the life cycles of the versions and documents.

The classes Document and Part are split up into a general part, containing the general features, and a P-specific part, containing the P-specific features. The classes Document and Part are the general classes. The classes PPart and PDocument inherit those classes. The Version class is entirely P-specific.

5.3 New implementation

Because the entire P system would be too big to fully reimplement in San Francisco, a subset of 15 classes of the classes mentioned in the last section was selected for implementation. In most cases, these classes were not fully implemented either; instead, a subset of their attributes was selected.

The selected classes were implemented according to the San Francisco Programming Model. This implies that getter and setter methods have been implemented for each attribute, while the attributes themselves are `protected`. For each relationship, quite a few methods were implemented, depending on the kind of relationship; these include methods for iterating over the partners in a one-many relationship, adding and removing partners, etc. Apart from these, some methods were implemented in each class which are needed by the framework to make them persistent.

A lot of methods had to be implemented just because we used San Francisco. However, these are highly routine and can probably be generated by a code generator. Programmers using San Francisco business objects will still need to know the programming model, so there will always be an initial investment. This is worthwhile when business objects can be reused in other applications.

As a rough estimate, 140 classes (mostly GUI classes) of the original 238 classes will remain for the P system. 19 general classes are implemented and maintained as shared business objects apart from the P system. The remaining 59 classes which represented P specific business entities can be combined to 43 classes. Through the fact that San Francisco is a framework which offers a lot of common object services like persistence, many of the more technical classes like database connectivity classes can also be dropped.

5.4 Evaluation

The case study has enabled us to draw some conclusions on the advantages and disadvantages of the SBOA approach. Experiences with CDL are collected in [Hordijk *et al.* 1998].

5.4.1 Reusability

The system has been designed according to the “Big super” approach based on assumptions on the generalization cost and the reusability of features in P. If these assumptions are correct, the reusability of P in the new implementation is enhanced. The correctness of the reusability assumptions can only be judged after implementing the applications related to P.

It should be noted also, that reusability of the business objects is restricted to applications written in San Francisco.

5.4.2 Modifiability and configurability

The existing P implementation cannot be modified at runtime. When new system versions are distributed (which happens quite often), a small tool that resides on the users' computers downloads and installs the new version. This is done when the user starts P.

In the new application, since all business objects run in a server process in San Francisco, application programs can be started and stopped any time without affecting the server or the BO's. This means that application logic can be modified while other applications are still running. We could regard this as a restricted kind of dynamic configuration. BO classes can be added at runtime. When a BO class is modified however, San Francisco has to be restarted for the changes to take effect.

5.4.3 Integrity

An added advantage of SBOA is increased integrity of business data. In the existing P implementation, the model objects serve as what Lauesen calls 'wrapper objects' [Lauesen 1998, p. 78], which are the access points to the data for the application logic. Because all updates are performed on the model objects instead of directly on the database, old data in windows cannot corrupt the database.

The same is true for all programs written with the IBM San Francisco framework. Here, the business objects serve as wrapper objects. The SBOA provides another kind of integrity, namely integrity of data between multiple applications. Since each entity in the subject domain is represented by at most one business object, the data are the same for all applications. Without shared business objects, inconsistencies between applications can easily occur.

6 CONCLUSION

Nowadays, business enterprises need to renovate and maintain their software applications regularly so that they can remain competitive in today's market. In this paper, we have addressed an important related software maintenance problem, which results from the fact that business applications are usually developed locally for individual departments (and thus producing redundant or overlapping implementations) within an organization. In order to remedy this problem, we first considered the concept of Business Objects (BOs) as a reuse technique during re-engineering of such applications. By comparing BOs with other existing reuse techniques, we eventually justified their ability to support large-grain reuse.

Secondly, we studied the concept of a Business Object Architecture (BOA), which separates the business logic from the application specific parts as well as from the persistence mechanism. We then extended such architecture to allow for sharing of business objects between several applications and named it Shared Business Object Architecture (SBOA). We also have made an attempt to provide a rationale for designing SBOA effectively by keeping the cost low. More specifically, we have proposed ways of how the business logic can be shared economically and how the dependencies between objects can be managed efficiently within the SBOA framework.

Finally, we have validated our research findings by means of a case study. More specifically, we launched a project where a group of business applications, which belong to a manufacturing company, are re-engineered using SBOA. The new design was evaluated based on criteria such as its reusability, modifiability, configurability and integrity. Based on our experiences from this study, we conclude that the SBOA, proposed in this paper, appears to be a promising approach toward the renovation and maintenance of business software applications.

REFERENCES

Biggerstaff, T. and A. Perlis, Eds. (1989), *Software Reusability*, Addison-Wesley Pub Co.

Byrne, E. J. (1992), "A Conceptual Foundation for Software Re-engineering," In *IEEE Conference on Software Maintenance*, pp. 226–235.

Canfora, G., A. Cimitile, and M. Munro (1994), "RE2: Reverse-engineering and Reuse Re-engineering," *Software Maintenance: Research and Practice 6*, 53–72.

Dunn, M. F. and J. C. Knight (1991), "Software Reuse in an Industrial Setting: A Case Study," In *13th ICSE*, IEEE, pp. 329–338.

Emmerich, W. and E. Ellmer (1998), "Business Objects: The Next Step in Component Technology?" In *Proceedings of CBISE'98 CaiSE*98 Workshop on Component Based Information Systems, Pisa, Italy*, J. Grundy, Ed., technical report 98/12, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand, pp. 21–26.

Gamma, E. *et al.* (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Hordijk, W. (1999), "Re-engineering an existing object-oriented system with business objects," Master thesis at the Vrije Universiteit Amsterdam.

Hordijk, W., S. Molterer, B. Paech, and C. Salzmann (1998), "Working with Business Objects - A Case Study," In *Business Object Design and Implementation II*, D. Patel, J. Sutherland, and J. Miller, Eds., Springer Verlag.

Hung, K., Y. Sun, and T. Rose (1997), "A Dynamic Business Object Architecture for an Insurance Industrial Project," In *4th Intl. Object-Oriented Information System Conference*, pp. 145–156.

IBM (1998), "IBM San Francisco Extension Guide," http://www.software.ibm.com/ad/sanfrancisco/.

Jacobson, I., M. Griss, and P. Johnson (1997), *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley.

Karlsson, E.-A., Ed. (1995), *Software Reuse – A Holistic Approach*, John Wiley & Son.

Krueger, C. W. (1992), "Software Reuse," *ACM Computing Surveys 24*, 2, 131–183.

Lauesen, S. (1998), "Real-Life Object-Oriented Systems," *IEEE Software March/April*, 76–83.

OMG (1998), "Combined Business Object Facility: Business Object Component Architecture," Technical Report OMG Document bom/98-01-07, OMG.

Salzmann, C. (1999), "Managing Shared Business Objects," In *Proceedings of ICSE workshop Engineering Distributed Objects EDO'99*, W. Emmerich and V. Gruhn, Eds., Departement of Computer Science, University College London, http://www.cs.ucl.ac.uk/EDO99/.

Sametinger, J. (1997), *Software Engineering with Reusable components*, Springer Verlag.

Shelton, R. (1998), "Enterprise Re-Use," *Distributed Computing Monitor 10*, 3, 3–21.

Tilley, S. (1998), "A Reverse-engineering Environment Framework," Technical Report CMU/SEI-98-TR-005, Carnigee Mellon University.