SYNOPSIS

This document investigates and presents a framework for understanding software composition. This is used to analyze the object-oriented model, and define some requirements for software composition in the context of middleware systems.
Document History

<table>
<thead>
<tr>
<th>DATE</th>
<th>VERSION</th>
<th>MODIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 1999</td>
<td>01</td>
<td>Initial draft (by B. Tekinerdogan, has evolved into another document on 'the nature of design')</td>
</tr>
<tr>
<td>August 1999</td>
<td>02</td>
<td>Major rework of initial draft. Focus on component composition.</td>
</tr>
<tr>
<td>November 1999</td>
<td>06</td>
<td>Intermediate version, reduced scope by excluding a rigorous formal analysis of composition models.</td>
</tr>
<tr>
<td>December 1999</td>
<td>09</td>
<td>Final draft, further reduction of scope by removing analysis of composition models other than the object-oriented model.</td>
</tr>
</tbody>
</table>

Abstract

In this report, we investigate component-based software construction with a focus on composition. In particular we try to analyze the requirements and issues for components and software composition. As a means to understand this research area, we introduce a canonical model for representing software. This model is used to analyze the object-oriented model of software composition.
Table of Contents

1. SOFTWARE CONSTRUCTION = PROBLEM DECOMPOSITION + COMPONENT COMPOSITION
   1.1 PROBLEM DECOMPOSITION 1
   1.2 COMPONENT COMPOSITION 2
   1.3 SCOPE AND STATE-OF-THE-ART 2
   1.4 CONCLUSION AND CONTENTS OF THIS REPORT 4

2. A MODEL OF SOFTWARE 5
   2.1 FROM USE CASE TO PRODUCT 5
   2.2 PRODUCT DECOMPOSITIONS 6

3. SEPARATION OF CONCERNS 8
   3.1 INTRODUCTION 8
   3.2 EXAMPLE APPLICATION 8
      3.2.1 Requirements 8
      3.2.2 Separating Concerns 9
      3.2.3 How the Concerns are Related 10
   3.3 SYSTEMIC SOFTWARE ENGINEERING CONCERNS 11
   3.4 SYSTEMIC APPLICATION DOMAIN CONCERNS 12

4. COMPOSABILITY 13
   4.1 THE NEED FOR COMPOSABILITY 13
      4.1.1 Example: control flow specifications 13
      4.1.2 Example: 'arbitrary inheritance' 14
      4.1.3 Example: composing objects and synchronization 14
   4.2 DEFINING COMPOSABILITY 14
   4.3 COMPOSITION ANOMALIES 16
   4.4 REQUIREMENTS ON COMPOSITION SCHEMES 18
      4.4.1 Separable concerns 18
      4.4.2 Combining Concerns 19
      4.4.3 Sufficient Expressiveness to Define Concerns 19
      4.4.4 Spreading/distributing concerns 19
   4.5 SUMMARY 20

5. THE OBJECT-ORIENTED MODEL 21
   5.1 SCOPE AND STATE-OF-THE-ART 21
   5.2 DECOMPOSITION IN THE OO MODEL 22
      5.2.1 Decomposing the Problem into Classes 23
      5.2.2 Decomposing Classes into Methods 24
      5.2.3 Decomposing Methods into Code Fragments 25
      5.2.4 Decomposing the Problem Space using Part-of and Inheritance Relations 26
      5.2.5 Decomposing from the Problem to Code Fragments 28
   5.3 AN OBJECT-ORIENTED MODEL OF THE EXAMPLE 28

6. CONCLUSIONS: A MODEL AND REQUIREMENTS 31
   6.1 FROM PROBLEM TO FRAGMENTS TO PRODUCT 31
   6.2 REQUIREMENTS ON COMPOSITION SCHEMES 33
      6.2.1 Czarnecki 33
      6.2.2 Mezini & Lieberherr 33
1. Software Construction =
Problem Decomposition +
Component Composition

In this chapter, we introduce models for components and software composition. We approach these from the perspective that software construction is characterized by two significant activities: The first activity is to decompose a problem (in the sense of issue, system or module to be realized) into smaller sub-problems. This is typically referred to as software analysis, which partially covers design as well. The second activity is to construct new software through the composition of both newly created and existing building blocks.

1.1 Problem Decomposition

In order to optimize the design, construction and maintenance process of software, we need to apply the divide-and-conquer principle by decomposing our systems and problems into smaller parts, which can be decomposed again, recursively. This decomposition process must continue until a level is reached where each building block (a) can be understood and constructed effectively, and (b) deals only with a single concern (we will discuss the motivation for this later). Note that the word ‘problem’ in ‘problem decomposition’ is not restricted to end-user requirements, but applies to anything from given requirements to the implementation of a simple task or algorithm.

The decomposition process is a way to analyze and manage complexity—in other words, it is a problem solving technique—but at the same time, it may provide a basis for system construction and maintenance. This is because the same the result of the decomposition process—as it applies to the design phase—determines the structure and the building blocks for constructing the system.

We make the following important assumptions about software development:

• The method of decomposition determines what the building blocks are and how they are related.

• We can always identify useful and appropriate abstractions and structures for a particular application by analyzing the related problem domain.

• A software development method should be structure-preserving: this means essentially that traceability between the ‘input’-artifacts (e.g. requirements) and ‘output’-artifacts (esp. code) is such that an iterative rather than a waterfall-style development process is supported.

Concluding, the decomposition process should take domain knowledge as an input, and result in a structured set of building blocks that offer a clear mapping to the structure and abstractions of the problem domain. The motivation behind these requirements is best summarized by the need to retain conceptual integrity (as coined by Brooks in [Brooks 95]).

1 We will use the term building block to designate the general notion of pieces of software that can be combined in some (undefined) way for the construction of software.

2 This assumes that the same modeling paradigm is used in all development phases.
We claim that the ability to achieve this, is largely determined by the expression power of the component and (de-)composition models.

1.2 Component Composition

From a software-engineering point of view, a key requirement for software composition is to be able to construct a system from (new or existing) building blocks that have minimal dependencies between each other. However, the crucial issues lie exactly in those dependencies: a system without any dependencies between the building blocks is hardly interesting (since it is a mere collection of independent parts, in which case the whole is exactly the sum of the parts, not more). In the words of Rechtin [Rechtin 92]:

"All systems have subsystems and all systems are parts of larger systems... The value added by a system must come from the relationships between the parts, not from the parts per se"

From this, we can conclude that the ways in which to glue building blocks together to create systems is at least as important as the contents of the building blocks themselves. In fact, in this report, our interest in building blocks is mostly limited to those characteristics of building blocks that are visible from the outside. These visible characteristics are preferably described explicitly by the interface of a building block.

We will focus our discussion of components upon building blocks at the source-code level (cf. the traditional notion of modules). This is sufficiently close to run-time components, because there is a natural tendency to have a direct mapping between source-code components and run-time components. Such a direct mapping avoids a large conceptual gap between specification artifacts and run-time artifacts. Also, many of the challenges in software development are related to the construction, adaptation and reuse of specifications (i.e. source code).

1.3 Scope and State-of-the-art

Currently, Component-Based Software Engineering is gaining significant popularity in both the research and the industrial community. The following definition shows the typical context of the much-used term 'component':

"Component-based development represents the 'industrialization' of software development. When any manufacturing process evolves to the point where it can be based on pre-built components and subassemblies, product quality, quantity, and speed of delivery soar." (Compuware)

This is a more recent, if not trendy, usage of the term 'component'. Previously, the term component was used to denote entities that participate in a composition. It is now more and more being reserved to denote a packaging of software that (adopting [Szyperski 98]):

a) is a unit of independent deployment

b) is a unit of third-party deployment

c) has no persistent state

A more theoretical definition has been coined in the Workshop on Component-Oriented Programming during ECOOP'96 [Szyperski 97]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."
The first sentence of this definition highlights the modeling aspects of components, whereas the second sentence emphasizes the more pragmatic issues related to packaging and deployment of software independent of the languages and platforms used for development or deployment.

Typically, component-based approaches tend to be technology-focused: how to construct and package components so that they can be delivered, installed, executed, and migrated, in such a way that they can still interact with other components that have similar, but completely independent, life-cycles. Typical state-of-the-art component models are COM/DCOM, CORBA and JavaBeans (briefly explained in [AMIDST 99] and more elaborately in [Szyperski 98]). One striking feature of these models is that they support object mobility and certain degrees of platform independence. However, in this report we will largely ignore such technology issues, and focus on the modeling characteristics of component and language models.

In order to avoid this confusion between the technology-centric view of components and the modeling-centric view, we present the following definition for component technology:

**Component technology offers components as the unit of development, composition, reuse, packaging, deployment, distribution and execution, where each component (may) consist of:**

- behavior: each component has one or more responsibilities (this is an abstract property).
- implementation: the realization of (newly defined) behavior.
- composition: both the behavior of a component and the implementation itself can be defined by reusing (a combination of) existing components. A typical example is aggregation, connecting interfaces of sub-components to the interface of the aggregate component.
- interface: contractually specified interfaces that hide the details of the implementation. Typically specified with Interface Definition Language (IDL).
- dependencies: in the implementation of its behavior, a component may need to refer to one or more other components (sometimes called 'collaborators'). The external dependencies could be specified as a part of the interface.
- packaging: to be able to distribute and deploy a component, a certain packaging technology is required, such as DLLs or Java applets.
- run-time support: for the execution of a component, run-time facilities (in particular for interacting with other components in a distributed environment, but also support for transactions and events) can be provided.
- development support: several component models (e.g. JavaBeans) include functionality to support the development of applications, e.g. for visual assembly of components.

Most component models adopt the object-oriented model [Wegner 90] to a certain extent, but add no further support for modeling and construction of large systems. As we will see in this report, this means they are all limited (and very similar) with respect to their composability: the ability to merge pre-defined pieces in such a way that the desired greater whole is realized. For this reason, e.g. in the report on the International Workshop on Large-Scale Software Composition [ref], the observation is made that:

---

3 Although such dependencies should be defined explicitly, as in the WCOP'96 definition, we feel this constraints rules out a large category of component models.

4 E.g. in CORBA, inheritance is supported, but only for interface inheritance, not for behavior inheritance.
Yet, reality shows that Component-Based Software Development has proven mainly effective for systems implementation in well-understood application domains, such as graphic user interfaces, but is still insufficient for the creation of reusable and changeable architectures of large-scale software, such as telephone switches.

In this report, we will adopt the term component in a very broad sense, albeit with a focus on the functional properties: behavior, implementation, composition, interface and dependencies. This definition focuses on software construction entities such as objects c.q. classes.

1.4 Conclusion and contents of this report

These two main issues related to composing software will be discussed in this report:

- The method of decomposing the problem into separate units; this is discussed in chapter 3, “Separation of Concerns”. This chapter first introduces the notion of separation of concerns and the concept of cross-cutting. Then an example application is introduced to demonstrate these concepts. Then we address systemic concerns; both the generic software engineering concerns and application domain-specific concerns.

- The key requirement for the software components is that they are composable: chapter 4, “Composability” is dealing with this issue. In this chapter, the motivation for composability is discussed and a definition of composability is given. Then the problems in composing, the so-called composition anomalies, are briefly discussed. This results in a set of requirements on software composition models.

Chapter 2 provides a context for the above-mentioned discussions by introducing a number of definitions and a notion of what software is, and what it is composed of. In chapter 5, the object-oriented model is closely investigated to see how it deals with decomposition and composition, and how it can address the example that was introduced earlier. Chapter 6 finally concludes the report with a summary of the requirements and a model that connects the conceptual decomposition tree to the software composition tree.
2. A Model of Software

2.1 From Use Case to Product

As a foundation for discussing software construction in general, and more specifically software components and composition, we introduce a model for describing the basic properties of software, including a number of definitions:

- The **software product** that is the result of the development process consists of a (usually large) collection of **code fragments**; we adopt the notion of code fragments as the most fine-grained independent specifications that can be merged into a running system. Each code fragment has its own identity and –most important– deals with a single concern only.

- A **use case** is the (description of) behavior for a typical usage of a system or product in a way that is non-specific for data values. A use case can be implemented by a number of code fragments. The execution (instantiation) of a use case consists of the ordered execution of a set of code fragments.

- The **final code** of a **software product** consists of a set of code fragments that is organized in such a way that the complete set of use cases defined for that product can be executed, while satisfying the (operational) quality requirements of the use case/product.

- A **software building block** in the broadest sense is defined as a grouping of code fragments. Note that this definition does not necessarily require each code fragment to belong to (a single) component. Also, note that component models in general will support grouping or nesting of components themselves as well, in addition to possible further behavior specifications.

- The **development process** takes a set of requirements and transforms it step-wise into final code that can be compiled and executed. Each transformation uses knowledge (domain ~, solution ~, design ~, etc.) to either bring the intermediate product closer to code or to improve its quality characteristics.

According to the above definitions, it is possible to develop software systems by collecting all the code fragments that are needed to implement all the use cases that are required for a particular software product. In general, there will be overlap between use cases, which allows for sharing the same code fragment between multiple use cases.

*Note that such an approach towards software development has at least two serious deficiencies:*

*Use cases represent functional requirements only, whereas any product that is not a prototype needs to fulfil other quality (‘non-functional’) requirements as well. For example performance optimizations, persistence support, user interface and representation, etcetera require additional code (fragments) to be written.*

*Designing systems based only on the current specification of use cases is very sensitive to changes, since it tends to ignore common, stable abstractions that are valid for more than the current situation.*

---

5 Not necessarily sequential, and possibly interleaved!
We can thus visually represent a software product as follows:

![Diagram](image)

*Figure 1. Modeling software as a collection of fragments*

This figure shows a product as a collection of fragments, which together serve a number of use cases.

### 2.2 Product Decompositions

From the operational point of view, there is no problem with this model of software products. However, assuming a very large collection of code fragments, it becomes impossible to manage the complexity of such a system. The only solution is to decompose the system into smaller parts, which can be decomposed again, repeatedly, until the level of code fragments is reached: a divide and conquer strategy in order to manage system complexity.

- A decomposition is a particular—usually hierarchical—ordering upon design entities, such as code fragments. If there are several orderings upon the same group of entities, we refer to each ordering as a decomposition dimension; each decomposition dimension adopts its own decomposition criterion. Ideally, all decomposition criteria for an associated set of entities are independent, which implies that all decomposition dimensions are orthogonal.

To illustrate how this relates to every-day software engineering, we show two examples:

- **Functional decomposition**: we can model this in a straightforward way by considering use cases to be the functional requirements; functional decomposition splits up a use case into parts, which may be split up again, and so forth. This is illustrated by the following figure:

![Diagram](image)

*Figure 2. Modeling functional decomposition*

- **Object-based decomposition**: a product is decomposed into a number of classes. Classes consist of a number of methods, which in turn consist of a number of code fragments. Each use case may cover an arbitrary set of methods and classes, but usually all the code fragments in a single method are executed.

---

6 This is not an accurate description, since functional decomposition and decomposing system requirements into use cases is not done in exactly the same way.
At the level of classes, we can identify two additional kinds of decomposition: inheritance and part-of decompositions. This will be further discussed in section 5.1. It is important to notice that, as pointed out above, software products may contain many code fragments that are not directly involved in use cases.
3. Separation of Concerns

3.1 Introduction

In chapter 0, we have shown that software construction can be seen as a process of problem decomposition and subsequent component composition. The power of the decomposition strategy is not so much due to the smaller sizes of the resulting parts, but to the ability to separate concerns, so that they can be dealt with independently. Dijkstra wrote in 1976 [Dijkstra 76]:

"... This is what I mean by ‘focussing one's attention upon a certain aspect’; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. I usually refer to it as ‘separation of concerns’, because one tries to deal with the difficulties, the obligations, the desires, and the constraints one by one. When this can be achieved successfully, we have more or less partitioned the reasoning that had to be done — and this partitioning may find its reflection in the resulting partitioning of the program into ‘modules’— but I would like to point out that this partitioning of the reasoning to be done is only the result, and not the purpose. The purpose of thinking is to reduce the detailed reasoning needed to a doable amount and a separation of concerns is the way we hope to achieve this reduction. The crucial choice is, of course, what aspects to study ‘in isolation’, how to disentangle the original amorphous knot of obligations, constraints and goals into a set of ‘concerns’ that admit a reasonably effective separation. ..."

If a software engineer can achieve true separation of concerns, it becomes much easier to understand and solve one problem at a time, and changes will remain localized in many cases, because each change deals with one or a few concerns only.

For example in the (conventional) OO model, the separation of concerns principle is supported in three ways:
1. By defining objects as models of real-world concepts, which are “naturally” separated from each other.
2. By separating the concerns of providing an abstract object interface and the implementation of it.
3. By grouping functions together around objects so that functions that are less related are structurally separated from each other.

For more information about separation of concerns in the domain of computer science and object-orientation: see e.g. [Hürsch 95] and [Aksit 96] for a general discussion of separation of concerns, and [Kiczales 92] about separate interfaces for the functionality and the implementation of modules.

3.2 Example Application

3.2.1 Requirements

To understand and compare the various approaches, we will use one common example. The example is based on the following conceptual model, and could be thought of as a part of an
application that manages real-time (audio or video) streams in the context of a middleware system:

![Diagram](image)

**Figure 4. Conceptual model for our example application**

The idea is that the Source represents an information source (for example the network, or an object at the application level). The Sink in this case is anything from an object at the application level to an interface to the network. The Buffer is used to deal with the timing differences between the input an output of this little system. Such a buffer could be useful both within middleware or on top of it.

This application has to deal with the following concerns:

- **Synchronization:** the Buffer has only a limited size, which means that the Source should no longer add extra entries when the buffer is full (i.e. until the Sink has removed one or more entries). Obviously, the Sink should not remove any elements as long as the buffer is empty.

- **Logging:** for testing purposes, all relevant information is to be logged, e.g. when a packet is handled by each part of the system, how it is synchronized, and how much memory is consumed by the various parts of the application. Logging is blocked when Buffer reaches its top capacity (e.g. at 95%), since this is a sign of system overload.

- **Memory Management:** for optimizing the usage of the limited memory available, each of the three parts in the system may dynamically claim extra memory. However, the following rules apply:
  1. The maximum amount of memory claimed by these three parts together is limited (say, to M\text{max}). This amount may vary dynamically as well.
  2. The maximum amount of memory available for each part should be limited according to the following ratio: M_{Source} : M_{Buffer} : M_{Sink} = 4 : 8 : 1.

  The effect of these rules is that the possibility to claim more memory in each part depends on the variable M\text{max} and the memory usage of the other parts at that time.

We may introduce many other concerns, such as real-time constraints (typical in audio and video streams), or memory management (in an embedded device, it may be better to work in bursts, allowing for more continuous stand-by time in between).

### 3.2.2 Separating Concerns

To illustrate what the separation of concerns means for this example application, in Figure 5 below, the Buffer concept is worked out, taking into consideration the synchronization and logging concerns. For reasons of space and simplicity, other concerns are not worked out in this diagram, but e.g. the memory management concern or the data structure concern can be added in a straightforward manner.
Figure 5. Decomposition into fine-grained concerns of the Buffer concept

This figure shows how one may make a conceptual decomposition of involved concerns, without considering relations and dependencies between concerns, and without thinking about the consequences for an implementation environment (whether object-oriented or otherwise).

Our claim is that –if we have made a proper decomposition into concerns– such entities, which represent independent and well-defined concerns, make the best building blocks for constructing and reusing software. The main theme of this paper is how to be able to specify these entities including the relations and dependencies between the entities.

3.2.3 How the Concerns are Related

We now show how the concerns and the concepts in the example application are related, through the following table. An entry at row A and column B in the table should be read as "the concern in row A is applied to the concern in column B":

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>-</td>
<td>Y</td>
<td>n</td>
<td>i</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Buffer</td>
<td>Y</td>
<td>-</td>
<td>y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Sink</td>
<td>N</td>
<td>Y</td>
<td>-</td>
<td>i</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Synchr.</td>
<td>i</td>
<td>Y</td>
<td>i</td>
<td>-</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Logging</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>Mem.Man.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>n</td>
<td>Y</td>
<td>-</td>
</tr>
</tbody>
</table>

Legend: the direction of mapping onto is from row to column. "Y" is applied to, "y": is subject to, "n": not associated, "i" indirectly associated, "-" not applicable

The notion of ‘indirectly associated’ between Synchronization and the Source and Sink concepts is used to indicate that there is a dependency in the strict sense (because
synchronization takes place due to the interaction between Source respectively Sink and the Buffer concern), but this dependency need not be effectuated (since the synchronization can be achieved by applying it to the Buffer concept only).

The following figure presents the contents of the table in a more visual way; each column shows which concerns are affected by the specific concern in that column. The open bullet designates that the corresponding concerns are only indirectly affected.

![Diagram showing concerns](image)

*Figure 6. This diagram shows which concerns (on the horizontal axis) affect other concerns (on the vertical axis)*

Each of the bullets in the above figure requires a certain amount of code to be written; these can be independent or overlapping/shared code fragments. The notion of code fragments allows us to abstract from the actual programming language statements that are used to implement these concerns.

We can summarize that logging and memory management are cross-cutting concerns, whereas synchronization can be concentrated mostly at the buffer. It is also important to note the dependency between concerns: logging requires synchronization, and depends on the memory management concern.

### 3.3 Systemic Software Engineering Concerns

Next to the typical functional behavior of systems, applications or components, a software engineer also faces many concerns that have to do with the operational behavior of software. Examples include:

- **Multiple views**: not all interfaces of an object should be available in all circumstances and for all other objects. The resulting different interfaces are called *multiple views*. Obviously, these views are affecting all the methods of involved objects at least. It is very difficult to add and evolve multiple views in conventional object-oriented programming languages [Aksit 92a].

- **Synchronization**: synchronization constraints, especially those implemented with conventional mechanisms such as semaphores, are typically spread over the code, embedded within the application behavior. This makes it very difficult to specify and reuse synchronization strategies (e.g. monitor, read-write synchronization, etc.) independent from application-specifics. For example, the problem of reusing code with synchronization constraints through inheritance has been investigated thoroughly as the 'inheritance anomalies' problem. See [Matsuoka 90], [Bergmans 94b] and [Bergmans 96b] for more information about this topic.

- **Persistence**: In many applications, it is important to retain the values of many objects for later sessions. A persistence mechanism allows specifying which objects with which
instance variables are to be stored in a persistent storage, and in which form (e.g. mapping objects to entries in one or more tables).

- **Logging**: All the behavior of a system may (optionally) be logged, but with a shared target (file, console, etc.) and a shared formatting behavior (layout, level of detail, representations, etc.).

- **Failure recovery**: Exception handling mechanisms are typically crosscutting many parts of the system.

Other examples of cross-cutting concerns are real-time specifications, atomic transactions, power consumption, fault tolerance and memory management.

### 3.4 Systemic Application Domain Concerns

Obviously, every application is built up from many specific concerns. However, in a number of cases, also application-specific concerns are systemic (i.e. they crosscut a large part of the behavior of the system). Two examples from practice illustrate this point:

**Example 1:**

> In the construction of a large car dealer framework, the company involved has to ensure that their framework can be extended with respect to two concerns: the first concern is that of the country where the application will be installed; for each country, different regulations regarding the environment, taxes, vehicle registration, etcetera have to be taken care of by the application. As a second concern, for each car brand the framework is sold to, special regulations and features have to be added.

Such concerns may have influences throughout the entire framework. Yet modular extension of the framework is essential, since the base framework must be maintained independently from the various make-specific and country-specific extensions. Also, making extensions for new makes or countries should not influence each other, or a combinatorial explosion of changes may occur.

**Example 2:**

> The developers of an ERP (enterprise resource planning) package are struggling to find proper ways to model and deliver their software: ERP software may involve many issues of the automation of businesses, such as: financial administration, logistics, tax issues, quality assurance, laws & regulations, human resource management, etcetera. Although some of these issues can be decomposed into independent modules, many issues crosscut the entire application. Moreover, the company wants to sell the implementation of some issues as separate add-on modules, which forces the code of the issues to be separated, even though it may be quite entangled with other parts.

From these two examples, it can be concluded that the notion of concerns is definitely relevant from the point of view of modeling the problem domain in complex systems. Therefore, it is a challenge to consider the generalization of AOP to include concern-based decomposition of the problem domain. This would also do something about the somewhat awkward asymmetry between the model of the primary functionality and the aspects.

In the work of the TRESE project on the construction of adaptable and reusable software architectures [Aksit 96a], we have experienced that the composition of domain knowledge using the conventional object composition semantics is not always satisfactory. The reason for this is that some knowledge domains are heavily intertwined. Decomposition of the problem domain into independent concerns would at least be an intuitive paradigm for thinking about the analysis and design of complex systems.
4. Composability

4.1 The Need for Composability

The need for composable models of computation has been recognized since long. For many years, the 'conventional' object-oriented model has been considered a suitable abstraction for constructing composable software modules. However, as the model was applied to more and more domains, numerous problems in extending, reusing and composing objects appeared. We start this section by illustrating the need for composability with several examples:

4.1.1 Example: control flow specifications

Previously, we have described the problems of the allocation and embedding of control flow in object-oriented programs [Bergmans 96a]. The main motivation for addressing the composability of control flow specifications can be derived from the statement by Rubin and Goldberg in [Rubin 92]. They state the following (emphasis added):

"Our goal in handling control is to capture [in the requirements gathering phase] the true constraints on ordering. It has been our experience that over-constraining the order in which activities take place within a system is one of the principal causes of change requests in big systems".

They observe that it may be impractical to realize all possible execution paths, and therefore during design some trade-off may be required. But in the analysis & requirements capturing phase, only the true constraints must be described. In other words, domain dependent control knowledge must be added, but domain independent control knowledge should not be described early. From a maintenance perspective, this means that these two types of control information must be separated and maintained independently.

The domain dependent control flow is thus assumed to be fixed for a particular domain. Therefore, it makes less sense to separate this type of control flow specification from the application objects. The domain independent control knowledge is likely to change based on new management insights, new applications, data, roles or activities. This makes a separately maintainable specification important.

In the NEDIS project [ref], a large car dealer information system developed by Siemens-Nixdorf, a requirement came up that applies to many (information) systems. This requirement stated that for reasons of managing complexity and maintainability, the control flow aspect of an application should be specified separately and composed with the application part of objects, while there are still certain interaction points. Composable control flow specifications require the following properties:

- We must be able to extend an application object with control flow without modifying or rewriting application code.
- We must be able to modify the control flow related to an application object without modifying or rewriting application code.
- We must be able to extend or modify the application part of an object without touching the control flow aspect.
Obviously, the above requirements do not hold if there is a logical conflict that prohibits the independence between the application part and the control flow part: there is always some shared interface to make things work; if this interface is violated, the result may be useless or non-functioning.

4.1.2 Example: 'arbitrary inheritance'

In [Aksit 92b] the following observation was made, based on the experiences from a number of pilot studies:

"Class inheritance can be seen as excluding, overriding and/or extending the operations and local variables of the superclasses. This kind of inheritance mechanism, however, fails in modeling inheritance hierarchies which require semantics other than overriding or extending operations."

This kind of problem is especially apparent when the application domain entities can be added and organized into inheritance hierarchies by the users themselves. This is typically the case for the domain of application generators:

"The need for an inheritance mechanism other than class inheritance becomes very apparent when building application generators. An application generator accepts a certain specification, in our [in the same article presented] example a grammar specification, and generates executable code in its application domain. When developing such systems, especially in the analysis phase, the software engineer needs to define hierarchies that organize the specifications of the application domain."

The paper summarized this type of problem as the need for arbitrary inheritance mechanisms. In other words, inheritance semantics that are not restricted to composing methods and instance variables only.

4.1.3 Example: composing objects and synchronization

The need for separation and composition of aspects for the construction of extensible and reusable software systems can be illustrated by the area of concurrent object-oriented programming. In this area, much research has been devoted to the analysis of (synchronization specification) inheritance anomalies [Matsuoka 90, 93]. This term designates serious difficulties in extending and reusing concurrent object-oriented programs due to the forced rewriting and duplication of code.

The prime reason for the recurring problems in reusing and extending synchronization specifications is that the synchronization aspect is basically independent from e.g. the functional behavior and data representation. However, in most (early) concurrent programming languages, such a separation could not be achieved. More precisely, the problems can be accounted for by the tight coupling of synchronization to methods and instance variables [Bergmans 92, 94b].

4.2 Defining Composability

To start, we provide a –tentative– definition for composability:

"Composability allows for the modular specification of modules with multiple independent concerns in such a way that they can be integrated (as-is) into one working entity (possibly a module)."

From this definition we can immediately derive:
• Composability allows for constructing a module from multiple independent modules, each focusing on one or more concerns.

• Composability supports the construction of new modules from multiple other modules, thereby merging the related (multiple) concerns handled by distinct modules.

An obvious, but critical effect due to the separation of concerns is that we should be able to construct systems by composing various concerns into a single abstraction. We would like to point out some critical properties that the composition of concerns should adhere to:

• Firstly, it must be possible to specify a module (object, abstraction) as a composition of its aspects. Only at the level of this module specification, the precise composition semantics can be determined and specified.

As a result of this composition, the module merges all aspect specifications into an executable specification (cf. Aspect Weaver™ [AOPP 96]), and encapsulates the implementations of the various aspects.

• Secondly, every aspect can be specified as a composition of various aspect abstractions. Preferably, this may include aspect abstractions reused from another module:

Again, the resulting abstraction that is a composition of aspects should encapsulate its components and their implementations or specifications. These two points would intuitively provide us with two types of elements in a system: an aspect abstraction and a module abstraction. The module abstraction merges the aspects. However, there is a third issue that we must consider:

• We should be able to compose a module by putting together other modules.

This introduces an important problem: it turns out that the specifications in the various ‘sub-modules’ for one aspect are not always fully independent. This means that a module cannot completely encapsulate its aspects, since to compose a new module, the aspects of the sub-modules must be separated per module, then merged per aspect type for the new module, and all aspects types are merged to form the newly composed module:
In summary, we are looking for a mechanism that supports both:

- **weaving** (aspects into a module)
- **abstraction** (of aspects and modules into more high-level aspects or modules)

### 4.3 Composition Anomalies

In order to come up with the requirements for composition schemes, it is useful to learn from the analysis of specific composition anomalies in the past. The area of concern-composition that is best understood in object-oriented programming is that of synchronization: there are numerous publications about the topic of the so-called ‘inheritance anomalies’ [Matsuoka 90]. This (overly general) term designates the problem of reusing synchronization specifications through inheritance. It appears that with most synchronization schemes it becomes impossible to reuse or adapt either the synchronization or the functional behavior of an object independently from each other.

The work in [Matsuoka 90] introduced three categories of requirements that would suffer from the inheritance anomaly. In [Bergmans 94b], we built upon this work, and introduced a general model of object synchronization schemes, as shown in Figure 8:
Figure 8. Generic model for object synchronization.

For an explanation of this model we refer to [Bergmans 94b, 96b], here we just want to stress the following points:

- This model shows that the specification of synchronization is determined by the state of the object (system), and that there is a non-trivial (2-stage) mapping between this state and the synchronization of individual methods.

- It was shown in [Bergmans 94b] that synchronization schemes that do not follow this 2-stage mapping, or that do not meet certain requirements upon these mappings, suffer from the inheritance anomaly.

In [Aksit 94], the specification and reuse of real-time constraints was investigated. It was found that the specification of real-time constraints suffered from similar composition anomalies (so-called ‘real-time specification inheritance anomalies’). In [Bergmans 96b] the following model for the specification of real-time constraints upon objects was proposed:
Obviously, this model shows many similarities with the model for object synchronization in Figure 8. In addition, similar requirements for avoiding composition anomalies have been defined for real-time constraints: the mapping from object and system state to the constraint specifications for individual messages must be staged and adhere to certain requirements.

4.4 Requirements on Composition Schemes

Based on the analysis of composition anomalies in [Bergmans 94b, 96b] and following the classification coined in [Aksit 99], we define four requirements upon a component model to achieve composability:

4.4.1 Separable concerns

- It must be possible to specify different concerns such that they can be addressed and reused for composition separately. For examples concerns such as memory management and synchronization specifications cannot be removed from functional code in C++.

- In particular in an imperative programming model, the sequence of invocations and statements fully defines the behavior of a system. As a result, if the execution of two elements requires a certain time-ordering, this forces the software engineer to insert statements or invocations relate to one element within the code of another element. Consequently, the elements become strongly coupled.

- If concerns cannot be separated, then the evolution or adaptation of one concern necessarily affects other concerns as well. This may even have a cascading effect. It may well lead to the well-known problem where each change to a system causes a number of errors to that system, while fixing an error generally causes several others.
4.4.2 Combining Concerns

- It must be possible to define a new component as a combination of two or more existing concerns, without any modification of the existing concern specifications. This may require some glue code or script to be introduced by the new component.

- If concerns cannot be combined, it means in practice that one or more concerns must be modified or redefined when they are to be reused in another context. Because these are usually changes that depend on the context of the specific composition, a redefinition that incurs a lot of replicated code will occur. This severely reduces maintainability.

4.4.3 Sufficient Expressiveness to Define Concerns

- The (design/programming) model should support expressing the necessary concerns; for example, to express real-time constraints or synchronization constraints, or to identify the sender of a message, or address history information.

- Lack of expressiveness is generally due to either the (un-)ability to refer to state information of (other) concerns, or the (un-)ability to influence certain behavior of the virtual machine (i.e. non-functional behavior).

- If it is not possible to express some concern, a software engineer may be forced to use tricks or workarounds, usually by adding functionality in a number of locations, thereby violating maintainability properties of the system.

4.4.4 Spreading/distributing concerns

- This requirement occurs if one abstraction has impact upon the behavior of several other abstractions. This has two main characteristics: (a) it cannot be addressed through a dependency with reference semantics, and (b), the dependency relation is from the impacting concern to the impacted concerns. The latter is essentially different\(^7\) from traditional import and reuse relationships.

- This requires a mechanism for expressing the ‘spreading’ or distribution of the concern. An important characteristic of such a mechanism is that it supports open-endedness, for instance through wildcards as in composition filters [Aksit 92a] or through propagation patterns as in Demeter [Liebherr 95].

- A natural consequence of spreading concerns over other concerns is the need for composing the concerns (2.).

- If no mechanism for spreading concerns is available, the concern to be spread will be replicated manually many times, leading to maintenance problems when this concern must be updated.

Note that the above requirements all have graduations of support; existing composition schemes all support several of these requirements to some degree, but to our knowledge there are no composition schemes that have extensive support for all of these requirements. For example, the composition filters model is rather successful in meeting the first three requirements, but has little support for specifying spreading of concerns.

On the other hand, for example, AspectJ, as the typical representative of aspect-orientation, has much support for spreading concerns (although not so upon other concerns), but much

\(^7\) From the SE perspective, not from an operational perspective.
less for separation and composition of concerns.

4.5 Summary

Composability allows for the modular specification of entities with multiple independent concerns in such a way that they can be integrated (as-is) into one working entity. The concerns to be composed may vary from basic operational concerns such as synchronization and memory management, to high-level, complex domain-specific concerns.

Composability involves both the weaving of concerns to merge and integrate them, and the abstraction of atomic and woven concerns into higher-level concerns. A natural approach to address this issue is to make concerns first-class concepts in the modeling paradigm.

We have formulated the following four requirements for composability:
1. Be able to specify concerns separately
2. Be able to combining existing concerns into a new concern definition.
3. Sufficient Expressiveness to Define Concerns
4. Be able to spread/distribute concerns

We have not found any existing approach that can meet all these requirements. In the following chapter we will investigate the composition characteristics of the object-oriented model.
5. The Object-Oriented Model

5.1 Scope and State-of-the-Art

Currently, Component-Based Software Engineering is gaining significant popularity in both the research and the industrial community. The following definition gives an impression how to interpret this:

"Component-based development represents the 'industrialization' of software development. When any manufacturing process evolves to the point where it can be based on pre-built components and subassemblies, product quality, quantity, and speed of delivery soar." (Compuware)

Typically, component-based approaches tend to be technology-focussed: how to construct and package components so that they can be delivered, installed, executed, and migrated, in such a way that they can still interact with other components that have similar, but completely independent, life-cycles. Typical state-of-the-art component models such as COM/DCOM, Corba and JavaBeans (refer to [Szyperski 97][Cetus] for an overview) focus mainly on the technological challenges involved, but are limited (and very similar) with respect to their composability: the ability to merge pre-defined pieces in such a way that the desired greater whole is realized. For this reason, e.g. in the report on the International Workshop on Large-Scale Software Composition, the observation is made that:

"Yet, reality shows that Component-Based Software Development has proven mainly effective for systems implementation in well-understood application domains, such as graphic user interfaces, but is still insufficient for the creation of reusable and changeable architectures of large-scale software, such as telephone switches."

The following figure illustrates the conventional 'as-is' composition, which takes components as a whole and integrates them as part of a larger component or system. The integration is based purely on access to the public invocation interface, the every other characteristic of the components are hidden as the implementation of the component. This is also called 'black box' reuse:

![Figure 10. Conventional component composition](image)

This conventional approach hardly supports adapted, tailored or partial components to be composed together. Such an approach works only\(^8\) if these components (in particular their interfaces) have been defined with an excellent understanding (prediction) of how they will be used. However, a typical goal of defining components is to be able to use them in applications that are not yet defined and/or known yet. In an increasingly changing world, with varying context and changing requirements, effective and efficient software construction

---

\(^8\) It should be noted though, that this approach is very effective in creating independent components with very little coupling and implementation-dependencies between components.
requires tremendous flexibility of the components that are composed into systems or parts thereof.

The immediate result of a component and composition model that cannot meet these requirements, is that designers and programmers need to either use ‘tricks’ or modify existing specifications in order to construct a system that meets its requirements. The word ‘tricks’ refers here to the addition of specifications or code in one or more places, thereby creating a mixing of concerns and in increase of complexity. A typical example is to modify (new or self-owned) client code in many places in order to make a predefined component work with a different data format. In short, solving incompatibility problems by writing code or specifications in the ‘wrong’ place creates maintenance problems.

It is our premise that software composition can only be successful if the underlying (component) models provide:

- Sufficient flexibility as how to compose several building blocks together.
- Sufficient support for adapting and extending predefined components, so that they can be made to fit in a new environment without making any modifications to (a copy of) their original definitions.

We consider two essential issues in evaluating the composability of a component model:

1. **What are the basic building blocks?**
   We are convinced that the essential characteristics of the Object-Oriented model (see OO-FAQ and [Wegner 90]) provide a suitable model that supports many software engineering requirements. However, for proper extensibility, adaptability and composability, composition specifications require the identification of more fine-grained entities than objects as a whole; examples may be methods, instance variables, aspects, filters, and so forth.

2. **What are the decomposition dimensions?**
   In addition to the conventional object-oriented (part-of & is-a) dimensions, composition filters and Aspect-Oriented Programming have emphasized the notion of additional decomposition dimensions. This issue has been discussed in section 6.1

### 5.2 Decomposition in the OO Model

In the design of an object-oriented system, we can distinguish the following decomposition activities (not necessarily performed in the order presented here):

I. The problem domain is decomposed into independent concepts represented by classes.

I.1. The behavior of classes is decomposed into a set of methods.
   
   a) Each method consists of one or more code fragments.
   
   b) The state of –instances from– classes is represented by a set of instance variables.

II. In order to deal with the large amount of classes, the class-space is decomposed again:

   I.1 With *is-a* or inheritance relations.
   
   I.2 With part-of relations, that designate aggregation at the instance level.
5.2.1 Decomposing the Problem into Classes

The first step of OO analysis and design methods is to decompose the problem by identifying individual concepts and mapping them to classes. This is visualized in the following diagram, which shows how the various problem domain concepts $a$…$d$ are mapped to classes $A$…$E$:

![Diagram showing mapping of problem domain concepts to classes](image)

**Figure 11. Mapping problem domain concepts to classes**

The diagram shows a number of variations in the mapping from concepts to classes. We can categorize these mappings as follows:

1:1  Concepts $a$ and $b$ each correspond in a one-to-one manner to respectively classes $A$ and $B$. This is the ideal case in the sense that it provides for full traceability between the problem domain (and as such the requirements) and the classes in the design. It also means that a single class only deals with a single concept, which reduces the risk that changes to one property of the system affect –the implementation of– other properties.

1:m  Concept $c$ is mapped to both class $C$ and $D$. This may either imply that the same concept is implemented twice, or that its implementation is split over two classes. This is less attractive from the perspective of traceability; it is more difficult to find the classes that implement a concept (that is, without proper documentation or tool support). Splitting the implementation over multiple classes will generally result in classes that depend on –the implementation details of– each other and are tightly coupled. Having multiple classes implement the same concept can be very useful at the design level, if the implementations are behaviorally equivalent. Because each class deals only with one concern, the 1:m mapping is not considered harmful, and actually unavoidable for proper design.

n:1  In the above diagram, concepts $d$ and $e$ are both implemented by class $E$. This means that one class deals with multiple concerns. As a general statement, this may lead to design and maintenance problems because adapting or extending one concern becomes difficult to do without affecting the other concern(s).

n:m  An n:m mapping is defined between the set of concepts $\{f, g, h\}$ and the set of classes $\{F, G, H\}$ in the diagram above; each concept is mapped to more than one class, and each class is implementing more than one concept. This is the worst situation from the SE point-of-view: traceability from concepts to classes is difficult, and each class implements more than one concept from the problem domain. This implies that modifying one concern without affecting others is at least tricky, or impossible. The problems this causes both during the initial development phase and future maintenance are obvious.
5.2.2 Decomposing Classes into Methods

Figure 12 shows a mapping between classes and methods. In this mapping, a distinction is made between the methods that are defined by their respective class (indicated by solid bullets), and those that are available for that class due to inheritance (indicated by open bullets). A special case is the redefinition/overriding of method $m_3$ in class C (indicated by a gray bullet). Figure 13 shows the class diagram that corresponds to the class-method mapping on the left.

![Figure 12. Mapping classes into methods](image)

![Figure 13. The corresponding class diagram](image)

If we review the mapping categorization, we can make the following observations:

1:1 This means that each class has exactly one single method that is not shared. This is generally not a useful category, since each class should be able to define more than one method.

1:m It is very natural for a class to have multiple methods.

n:1 In principle, each method body is defined (i.e. the solid bullets) only once, by a single class. However, polymorphism implies that the method with the same name can be defined in multiple places. In the example above, this is shown for method $m_3$ that is inherited but redefined by class C (as designated by the gray bullet in the diagram). The example also illustrates that the same method can be available (through inheritance) at many classes (the open bullets). We can state that an n:1 mapping is fine, as long as implementations are not replicated.

n:m An n:m mapping means that a class can share (multiple) methods with other classes. As we discussed above, this is natural with respect to the methods that are available on
the interface, but each specific implementation (method definition) should be done only once.

Note that a class does not consist of only methods, but may also define other behavior, such as instance variables, class variables, constructors and destructors, and so forth (depending on the model). The decomposition of classes into these other properties is either similar or more straightforward, in this text the focus will remain on the methods.

5.2.3 Decomposing Methods into Code Fragments

The following figure (Figure 14) shows a mapping between methods and code fragments. Typically, this is a 1:m mapping, i.e. each method consists of one or more code fragments:

![Figure 14. Mapping methods to code fragments.](image)

As in the previous figures, the method $m_3$ is shown twice; it is defined in class $A$, and then redefined (overridden) in class $C$. The latter version is shown as $m_3'$. In this figure, $m_3$ consists of two code fragments ($f_4$ and $f_5$). In the subclass $C$, this method is redefined with an implementation that may look as follows in pseudo-code:

```pseudo
method m3()
begin
  super $m3();$
  $f_6$;
end
```

So, this method performs some code fragment $f_6$ before calling the super-class implementation of $m_3$, which causes code fragments $f_4$ and $f_5$ of $m_3$ to be executed. This is an example of code reuse.

The figure also shows an example of code duplication: code fragment $f_3$ is used by both method $m_2$ and method $m_4$. Since code fragment equivalence is based on semantic correspondence, this does not mean that the same source code is referred to twice, but one may be a copy of the other, or a retyped version with a different layout, or even really different source code; as long as the semantics and intention are equivalent, two pieces of code are dubbed as being one and the same code fragment.

Code duplication (or, more accurately, code replication) is considered as a prime cause for much maintenance problems. However, the underlying problem of code duplication is not the presence of equivalent code in more than one place as such: the real problem occurs when the same functionality (from the conceptual point of view) is implemented

---

9 It is rather difficult to formalize this equivalence notion in a way that satisfies all possible cases, one possible approach would be to compare parse trees. In this paper we will use an informal approach to defining equivalence.
twice. If this is the case, maintaining changes to this functionality becomes cumbersome and error-prone, because the same work has to be done more than once (or at least it must be considered more than once).

If we review the mapping categorization, we can summarize as follows:

1:1 Each method consists of one code fragment only: this is the case with very small methods that deal with a single concern only.

1:m A method may consist of multiple code fragments, which is perfectly eligible. Criteria for judging proper design of such methods are (a) whether different concerns dealt with in the same method (preferably not), and (b) whether a method is not too long and complicated.

n:1 A single code fragment appears in multiple methods: if the code fragment is defined once (solid bullet in the diagram), and reused (open bullet in the diagram) by the other methods, this is fine, otherwise this is a case of code duplication.

n:m Methods (also more than two) may share multiple code fragments between each other. This has the same objections as outlined in the previous item (n:1 mapping).

5.2.4 Decomposing the Problem Space using Part-of and Inheritance Relations

In the previous sections, we have discussed the following decompositions:

![Diagram](image)

Figure 15. The decomposition from problem domain to code fragments.

Of the above decompositions, the first decomposition—from the problem domain to classes—generally results in a large set of classes. This results in a flat structure that is very difficult to understand and navigate. Part-of and inheritance relations provide further hierarchical decomposition of the problem domain space as formed by classes.

Booch [Booch 94] introduces the object-oriented modeling paradigm through an extensive discussion of the complexity of systems. Based on the work of Simon & Ando and Courtois (see e.g. [Simon 96] and [Courtois 85]), he introduces the canonical model of complex systems. Booch claims that virtually all complex systems take on this same canonical form.

This canonical model identifies both the object dimension and the class dimension, i.e. it includes the structure of individual instances of classes and the class-instance relationships. Each dimension has its own decomposition hierarchy:

- The object dimension, where the hierarchical relationships between the units (i.e. the objects) are formed through part-of relations.

- The class dimension, where inheritance/is-a relations realize the decomposition hierarchy between the units (i.e. classes).
The two dimensions with their respective decomposition hierarchies are outlined by the following figure:\textsuperscript{10}:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure16}
\caption{The two decomposition hierarchies in the canonical model of complex systems.}
\end{figure}

In a typical object-oriented design notation (such as UML), inheritance and part-of relations (as well as associations) are drawn in the same 2-dimensional space, using different line types. Figure 17 shows an example:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure17}
\caption{Example of inheritance and part-of decompositions described within a single diagram}
\end{figure}

We can illustrate the decomposition dimension according to part-of relationships as follows:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure18}
\caption{Part-of decomposition of the class space.}
\end{figure}

\textsuperscript{10} In [Booch 94] a similar diagram illustrates with much detail how the instances in this 3-dimensional space relate to each other and to the classes.
In the above figure, for each class on the horizontal axis, a bullet on the corresponding vertical axis designates a part; open bullets designate parts that derive this role through inheritance (i.e. these are subtypes from one or more direct parts).

If we consider the mapping categorization, we can make the following observations (distinguish between instances, which can only be part of one other instance, and classes, which can be part-of multiple classes, designating alternatives for each of the instances):

1:1 Each class has only one class as a part; this is no useful category since there is no reason to be restricted to only 1:1 mappings with part-of relations.

1:m Each class may have multiple parts, but no class is part of more than one other class.

n:1 Each class may be part of multiple other classes, but may not have more than one part itself. The latter restriction is not useful at all.

n:m There are no restrictions on the amount of parts, nor the amount of part-of roles a classes participates in.

We can make a similar diagram for inheritance, but for the given diagram in Figure 17 with only two inheritance relations, this is far from interesting. The reasoning about the mapping categorization is also similar; a class may have multiple subclasses and superclasses (in some models only one superclass, however). The main difference is that inheritance relations cannot be cyclic, i.e. a class cannot inherit from itself.

5.2.5 Decomposing from the Problem to Code Fragments

Figure 19 elaborates the previous Figure 15 by showing how the is-a and part-of relations are a refinement of the Problem domain–Classes decomposition by structuring the space that contains all classes:

```
Problem domain
   \--- is-a \-- Classes
      \--- part-of \-- Methods
            \--- part-of \-- Code Fragments
```

*Figure 19. The decomposition from problem domain to code fragments, with a refined mapping from problem domain to classes.*

Note that this refinement of the mapping between problem domain concepts and classes in a strict, mathematical sense does not enhance the model or its expression power. The motivation for this additional decomposition comes purely from the practical/cognitive consideration that human software engineers need additional tools to be able to manage the overview of a complex (i.e. many dependencies) mapping.

5.3 An Object-Oriented Model of the example

When designing the example application in an OO manner one would like to retain all concerns fully separated (as first-class entities), for instance for the purpose of
maintainability or reuse. The following class diagram explicitly shows all specific concerns as independent entities, without optimizing the design for code reuse:

![Class Diagram](image-url)

**Figure 20. This class diagram shows the object-oriented relations between the concerns that are involved**

The point of this diagram is to illustrate that even with so few concerns involved, a combinatorial explosion will occur if a designer wants to keep these concerns fully separated. As shown in Figure 20, for the `Source`, `Buffer` and `Sink` concepts it is necessary to define a corresponding `Log` concept, resulting in the declaration of multiple `Log` classes, such as `SrcLog`, `BufLog`, etc. In combination with logging of the status of synchronization of the `Buffer` class also a class `SyncLog` is defined. Class `LogSync` contains the synchronization of the global log class. Further remarks about this diagram:

- Note that separating these concerns in independent classes—or even separate methods—is generally not possible to implement at all (e.g. synchronization, logging)!

- If there is a lot of overlap between all logging code or all memory management code, it could be separated into a single class. Reusing this code can be done through inheritance or part-of relations.

- Further optimization of the design will consist of replacing some of the classes with methods or even code fragments within other methods; but such steps compromise the modularity of the design!

We also show an alternative design that exploits inheritance to reduce the number of classes:
Although this design has fewer classes, it is obvious that the concerns are no longer fully separated. Note that a specifically optimized design might find a balance between the design in Figure 20 and that in Figure 21, for example by applying design patterns such as Strategy, Decorator and Observer. We can conclude that the object-oriented model cannot cope very well with the modular specification of independent concerns.\footnote{Please note that we are not claiming that applications like the example cannot be implemented: that is a necessary but not sufficient requirement for software development: the ability to manage complexity and maintain an application is just as important for anything but a throw-away prototype.}
6. Conclusions: a Model and Requirements

In this report, we have discussed the important matters of (component-based) software composition. In chapter 2 we introduced the starting definitions for reasoning about software and software composition, and introduced the notion of product decompositions. Chapters 3 and 4 continued to explore respectively the decomposition (separation) into concerns and the composition of software building blocks. In the following section, these two themes are merged to present a model that relates the conceptual modeling and separation of concerns to the construction of large products as repeated composition of software building blocks. The subsequent section will collect the requirements for composition schemes.

6.1 From Problem to Fragments to Product

In this report, we have described all the ingredients for presenting a unifying model that describes the relation between a conceptual decomposition of a problem (i.e. the system to be developed) and the composition model of software. The work presented here relies heavily on the following hypotheses:

1. The optimum composition of a software product is based on fine-grained code fragments that must each correspond to a separate concern.

2. The optimum set of separate concerns can be found through ‘conceptual modeling’, i.e. a modeling activity that does not take into consideration the characteristics and limitations of a software composition model. Note that the conceptual model may include implementation-related topics.

3. To evaluate the characteristics of a software composition scheme and determine its effectiveness, one must evaluate how the composition scheme supports operations upon the conceptual view of the system.

---

Figure 22. The software development decomposition-composition diamond
Figure 22 hierboven consists of two mirror sides which represent respectively on the left side a representation of the conceptual or knowledge representation of a certain application (or sub-system), and on the right side the software modularization model.

The representation of the conceptual (knowledge) model presents a hierarchic decomposition of the application. This does not necessarily represent a proper model of the application with all relevant dependencies, instead it assumes that we can describe any system by splitting it recursively into concerns (with either AND or OR conditions between the concerns). Additional relations between the concerns are not shown in this representation, but may include:

- **Sharing**: it is very well possible that some concerns are shared between different nodes in the tree. We assume that this is represented by an additional relation. An alternative would be to use a lattice instead of a tree for representing the knowledge.

- **Constraints**: constraints between nodes may exclude certain nodes to be active simultaneously, or without each other, or specify behavior that depends on properties of other nodes.

- **Dependencies (import)**: nodes of the decomposition tree may depend on other nodes. This can be represented by (directed) dependency relations. Typically, the dependents of a node are required to be available to make a node (fully) work. This relation could also be called the *import* relation.

- **Superimposition (export)**: this is the reverse of a dependency: it means that one node affects one or more other nodes. The main difference with a dependency relation is that the side that specifies the behavior is independent of, and usually specifies the nodes onto which the behavior applies. This could also be called the *export* relation.\(^\text{12}\)

The main goal of the decomposition tree is to provide a representation of a system that is completely conceptual, i.e. not limited by –the composition semantics of– any formal specification or language. The nodes in this tree all represent independent concerns that may change independently from each other. As described above, some concerns may also affect other concerns. The set of concerns can be represented as an n-dimensional space, where each dimension represents a single concern. This is illustrated hieronder in Figure 23:

\(^{12}\) Note that the conventional use of ‘export’ as in module languages is used to declare visibility only, in this case we include the notion of superimposition upon a set of other concepts as well.
a few simple manipulations of this conceptual model. The main issue will be how the software that implements this model can handle these manipulations.

The representation of the software model is based on a decomposition tree as well, which has two additional things in common with the conceptual model. Firstly, the root of this tree corresponds to the root of the conceptual decomposition tree (we may call this ‘application’, ‘system’ or ‘subsystem’, or ‘product’). Secondly, the leaves of this tree each correspond to exactly one leaf of the conceptual decomposition tree (i.e. each leaf of the software decomposition tree, deals with a single concern only). These leaves correspond to ‘code fragments’; pieces of code that each implement a single concern. The structure of the intermediate nodes of the tree is in principle open, and depends on both design decisions and the composition techniques that are offered by the composition scheme of the model that is used.

The software decomposition tree also shows only certain relations between software elements. In particular, it focuses on specification composition, i.e. composition of source-code/specification level elements into new, large-grained specification elements. At a certain level, these specification elements will coincide with the notion of ‘modules’ or ‘objects’, which are composed together (recursively in some cases) to form an application. In contrast, behavioral composition, i.e. the composition of various forms of behavior into more large-grained behavioral elements, may, or may not, coincide with the nodes of the software decomposition tree.

6.2 Requirements on Composition Schemes

We can state a number of requirements for mechanisms that support the composition side of the model (design-diamond) that we presented in the previous section. First we will present two pieces of related work where requirements on composition have been defined as well.

6.2.1 Czarnecki

In [Czarnecki 98] three requirements for ‘aspect composition mechanisms’ are stated:

- **Minimal coupling between aspects:** some coupling is always necessary, it takes place at specific join points.
- **Different binding times** (e.g. compile-time and run-time), and modes (dynamic or static) between aspects.
- **Noninvasive addition of aspects to existing code:** be able to make unforeseen adaptations through addition only, i.e., without modifying existing code.

The last requirement is addressed by the requirements for composability that we stated in section 4.4. The issue of coupling is covered there as well, but we did not explicitly address the requirement of minimizing coupling. An excellent example of how to minimize coupling can be seen in the work on adaptive programming, where it is possible to define certain behavior on a ‘class graph’ in a structure-shy manner; i.e. with minimal dependencies on the actual structure of the class graph.

6.2.2 Mezini & Lieberherr

In [Mezini 98], requirements for decoupled behavioral composition are defined. The goal of these requirements is to support the composition of behavioral abstractions that capture a
certain slice of behavior covering several classes (i.e. collaborations). They distinguish the following two requirements:

- **structure-generic**: the specification of the behavior should be generic with respect to the class-structure of the system that the collaboration is applied to. This supports reuse, both within a system and for different systems.

- **decoupled behavioral composition**: to reuse components for the construction of more complex collaborations the following conditions must be satisfied: (a) collaborations must not depend on a certain structure of composition, and (b) all implementation-specifics of a collaboration that is composed with others must remain hidden (i.e. encapsulation).

### 6.2.3 Our Proposal

Based on our previous work and the analysis earlier in this report, as well as the work of others, as exemplified by the previous sections, we propose the following requirements for the ‘ideal composition scheme’:

- Be able to specify elements separately (see section 4.4.1).
- Be able to combining existing elements into a new element definition (see section 4.4.2).
- Sufficient expressiveness to define elements (see section 4.4.3).
- Be able to spread/distribute concerns by applying one element onto many others in a non-invasive manner (see section 4.4.4).
- Minimize the dependencies between elements; some coupling is always necessary, this should take place at specific join points that are implicitly or explicitly defined as interfaces of the element. The mapping to these join points should allow for open-endedness. Encapsulation is a special case of this requirement; it hides implementation-specifics from the visible interface.
- Support different binding times on the connections between elements; this is especially relevant to support performance and robustness, while retaining maintainability.

Note that several of these requirements may be satisfied by concrete examples to a certain extent: they are by no means true/false type of requirements. For a detailed evaluation and comparison of composition schemes with respect to the presented requirements, they should be worked out and formalized.

However, the successful application of a composition scheme that meets the above requirements is fully dependent upon the ability to create a conceptual model in which the various concerns have been hierarchically decomposed until each relevant concern is separately defined.
6.3 Conclusions & Future Work

We would like to summarize the material presented in this report and the conclusions that can be drawn from that material as follows (the definition of terms in italics can be found in the appendix ‘Definitions’):

• Software can be modeled as a collection of code fragments that are connected with a number of structuring relations. What kind of structuring relations are available depends on the modeling paradigm and composition scheme that is adopted at the design and programming level.

• Software construction can be observed as a two-phase process: in the design phase the problem (system) is decomposed in order to understand, analyze and define it in detail. In the realization phase, the small-grained implementations are to be composed into a larger whole, both in the operational sense to achieve a running system, and in the specification sense, to obtain a well-structured and manageable specification of that system.

• The main artifacts of the decomposition process are concepts. The main challenge of the decomposition process is to identify the right set of concepts in such a way that the matters to be modeled are represented in accordance with the available knowledge of that domain and that the selected set of concepts presents a clear separation of concerns. The resulting concepts can be ordered according to a decomposition tree, where more detailed and fine-grained concepts are shown as children of the general, large-grained concepts.

• The leaves of this decomposition tree represent the most fine-grained concepts. Each leave should deal with only a single concern, or a set of concerns that can safely be assumed inseparable for the problem domain at hand. The implementation of each of these leaves will result in one code fragment: an inseparable specification of a certain behavior, implementing a single concept/concern only.

• We refer to the main artifact of the composition process as elements (components, modules, objects, etc. could be substituted for this term); elements are compositions of code fragments. The composition relations at the specification level (in contrast to the operational level) can be represented as a tree-like structure. The leaves of this tree are the code fragments, whereas the root of the tree represents the total program (or system, library, ...).

• One of the big challenges in software engineering is to find composition schemes that support the construction of these composition trees, given a set of code fragments and concepts, and given all the constraints resulting from the operational perspective. In current practice, we see that the composition graph involves many redundancies, and does not resemble the ideal tree structure. The main reason for this is the lack of composability of the elements, which forces a different structure and relations (e.g. merging two or more code fragments). We have defined a set of four requirements to achieve composability. These requirements relate to the ability to separately define behavior and than merge it afterwards according to certain mappings.

• The main requirement for a composition scheme is composability and its associated set of four requirements. Further the decoupling of elements and the variety in binding times are additional requirements.
We believe that our vision upon software construction and the requirements as presented in this report form a fruitful basis for investigating the issues of software analysis & design, the evaluation and comparison of composition schemes, and the invention of new, more powerful, composition schemes. To our knowledge, such a vision and model of software construction has not been presented before, nor could we find convincing contradicting evidence. The requirements upon composition schemes have been partially presented elsewhere, but have not been published collectively.
Definitions

*Aspect* { XE "Aspect" }
A (usually non-functional) concern that crosscuts the system in a systemic way. In Aspect-Oriented Programming (e.g. [Kiczales 97]), typically all aspects affect a base level that addresses the core functional concerns.

*Building Block* { XE "Building Block" }
A generic term for a software unit that can be used to recursively compose more large-grained software units.

*Code fragment* { XE "Code fragment" }
A piece of functionality, expressed in an executable specification of arbitrary granularity; this can be either a single statement or expression, up to a set of (partial) class specifications. Code fragments are in general collections of partial program specifications.

*Component* { XE "Component" }
This has become a rather overloaded term, that is lately more used to designate the technology (and its instantiations) for packing, deploying and distributing subsystems as represented by e.g. COM/DCOM, CORBA, or JavaBeans. A more general definition has been coined in [Szyperski 97]: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

*Composability* { XE "Composability" }
Composability allows for the modular specification of modules with multiple independent concerns in such a way that they can be integrated into one working entity (possibly a module)

*Composition scheme* { XE "Composition scheme" }
The model and mechanisms for composing software from separate entities. A particular composition scheme has a big impact upon the composability of the entities that are specified with that scheme.

*Concept*  
2: an abstract or generic idea generalized from particular instances

*Concern*  
A matter of interest (in a specific context) that can be investigated and reasoned about as a separate issue. Typically, the word concern is used distinctively from *concept* when the matter at hand refers to a more general issue, problem, or category of phenomena (i.e. a concern usually involves multiple concepts).

*Class*  
A template specification of objects. Classes may inherit behavior from superclasses through a statically defined inheritance relation.

*Crosscutting*  
[Merriam-Webster] something that cuts, goes, or moves across or through. In the software area, this refers to *concerns* (functionality or issues) that have to be dealt with in many locations in the software.
**Development process**

The series of activities that lead from requirements to a delivered product. It takes a set of requirements and transforms it step-wise into final code that can be compiled and executed. Each transformation uses knowledge (domain ~, solution ~, design ~, etc.) to either bring the intermediate product closer to code or to improve its quality characteristics.

**Object**

(gen.) An abstract data type that hides its implementation, is accessed through message passing, and supports the creation of many concrete instances with their own state from a template specification: the class.

**Product (Software ~)**

The final outcome of the software development process, resulting in a collection of code fragments, structured

**Separation of concerns**

the method of dividing a problem into smaller, relatively independent, subproblems that can be addressed separately. This assumes that it is possible to merge the solutions to these problems later. From a SE point-of-view, it is also important to keep the solutions separated during the entire life-cycle of the product, since it will be necessary to maintain each of the subproblems after the first development cycle.

**Use Case**

A use case defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.
List of Figures

FIGURE 1. MODELING SOFTWARE AS A COLLECTION OF FRAGMENTS ..........................................................6
FIGURE 2. MODELING FUNCTIONAL DECOMPOSITION .................................................................................6
FIGURE 3. MODELING OBJECT-BASED DECOMPOSITION .............................................................................7
FIGURE 4. CONCEPTUAL MODEL FOR OUR EXAMPLE APPLICATION ..........................................................9
FIGURE 5. DECOMPOSITION INTO FINE-GRAINED CONCERNS OF THE BUFFER CONCEPT ..................10
FIGURE 6. THIS DIAGRAM SHOWS WHICH CONCERNS (ON THE HORIZONTAL AXIS) AFFECT OTHER
CONCERNS (ON THE VERTICAL AXIS) ........................................................................................................11
FIGURE 7. COMPOSING MODULES THAT HAVE NESTED ASPECTS ..............................................................16
FIGURE 8. GENERIC MODEL FOR OBJECT SYNCHRONIZATION ..................................................................17
FIGURE 9. A GENERIC MODEL FOR REAL-TIME CONSTRAINTS ..................................................................18
FIGURE 10. CONVENTIONAL COMPONENT COMPOSITION .........................................................................21
FIGURE 11. MAPPING PROBLEM DOMAIN CONCEPTS TO CLASSES ..........................................................23
FIGURE 12. MAPPING CLASSES INTO METHODS ..........................................................................................24
FIGURE 13. THE CORRESPONDING CLASS DIAGRAM ..................................................................................24
FIGURE 14. MAPPING METHODS TO CODE FRAGMENTS ..........................................................................25
FIGURE 15. THE DECOMPOSITION FROM PROBLEM DOMAIN TO CODE FRAGMENTS .........................26
FIGURE 16. THE TWO DECOMPOSITION HIERARCHIES IN THE CANONICAL MODEL OF COMPLEX SYSTEMS 27
FIGURE 17. EXAMPLE OF INHERITANCE AND PART-OF DECOMPOSITIONS DESCRIBED WITHIN A SINGLE
DIAGRAM .........................................................................................................................................................27
FIGURE 18. PART-OF DECOMPOSITION OF THE CLASS SPACE ........................................................................27
FIGURE 19. THE DECOMPOSITION FROM PROBLEM DOMAIN TO CODE FRAGMENTS, WITH A Refined
MAPPING FROM PROBLEM DOMAIN TO CLASSES ..................................................................................28
FIGURE 20. THIS CLASS DIAGRAM SHOWS THE OBJECT-ORIENTED RELATIONS BETWEEN THE CONCERNS
THAT ARE INVOLVED ..................................................................................................................................29
FIGURE 21. THIS CLASS DIAGRAM EXPLOITS INHERITANCE TO REDUCE THE NUMBER OF CLASSES IN THE
DESIGN OF THE BUFFER ..............................................................................................................................30
FIGURE 22. THE SOFTWARE DEVELOPMENT DECOMPOSITION-COMPOSITION DIAMOND .................31
FIGURE 23. MULTI-DIMENSIONAL DESIGN SPACE MADE UP OF CONCERNS .............................................32
References


[Bergmans 92] L. Bergmans & M. Aksit, Reusability Problems in Object-Oriented Concurrent Programs, ECOOP '92 Workshop on Object-Based Concurrency & Reuse, June 1992, Utrecht, the Netherlands.


[Harrison 93] W. Harrison & H. Ossher. Subject-oriented programming (a critique of pure objects). In proceedings of OOPSLA ’93, September 1993.


M O D E L S  F O R  C O M P O S I N G  S O F T W A R E


