

1

How Can Invasive Software Composition Help You

Contents

1.1	A Short Overview of the Book.....	4
1.2	The Component Dream.....	6
1.3	CoSy: A Component System for Adaptation and Extension	9
1.4	Aspect Separation: New Dimensions of Modularity ...	12
1.5	Requirements for Composition	18
1.6	Epilogue	20
1.6.1	Remarks	20
1.6.2	Goals of Invasive Software Composition	20

This introduction explains briefly what invasive software composition is about and why it specializes a more general trend in software engineering, composition technology.

Do you want to know the level of reuse in our company?

Null, absolutely null. And do you know why?

Well, it's much better to produce software from scratch, to be able to charge for it again and again....

Anonymous software engineer from well-known IT company

It takes time to build software. Unfortunately, the competition in the software industry forces companies to reduce their time to market more and more. To cut this Gordian knot, many of them try to build software from prefabricated components, components off-the-shelf (COTS). COTS should be reused as many times as possible so that the software development costs can be reduced and products can be shipped faster (*component-based development*) [Sun Microsystems, 1997, Siegel, 1998, Box, 1998, JavaSoft, 2000]. However, major questions of this approach are still open. How can components be prepared for reusability? How can they be adapted flexibly during composition? How can the component-based software process be organized for large systems that consist of thousands of versions and variants?

This book answers several of these questions. It presents a new, reuse-centered way to construct software systems, *invasive software composition (ISC)*. The method is based on components but focuses on *composition*, i.e., on the methodology of *how* components are composed. To achieve better reuse, the method adapts and integrates components *invasively*. Because everything is centered around a standard language, Java, this method provides a wealth of material for the system architect.

Invasive software composition is one methodology of the growing field of *composition systems*, systems that concentrate on the composition of components. They generalize many of the approaches to component-based engineering we have seen the last 40 years. To show this, we present a *tower* of component-based systems (Fig. 1.1). Component-based systems can be compared in terms of three major aspects. First of all, it is important how components appear, i.e., which kind of *component model* is employed. This determines when a component can be exchanged for another. Next, we need a flexible *composition technique* that offers a wide range of composition operations. Thirdly, a *composition language* is required, in which *composition recipes (composition specifications)* can be written. They describe how systems should be built from components and contain information about their architecture. If all three requirements are met, we speak of a *composition system*. Such systems form the top level in Fig. 1.1. Readers will gain insight into this approach, which is, so far, the most general approach to component-based engineering.

Invasive software composition is a specific composition technology, based on a flexible component model. Therefore, it provides a basis for unifying several software engineering techniques, such as generic programming, architecture systems, inheritance, view-based programming,

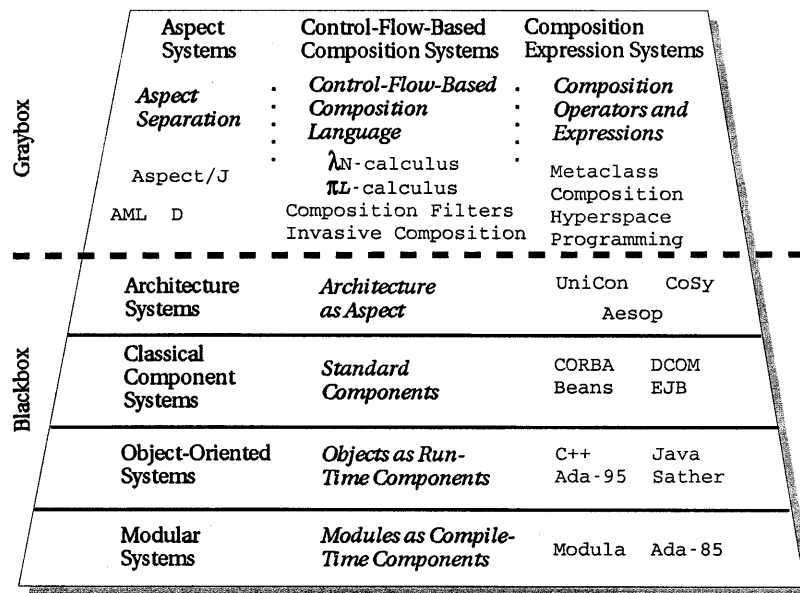


Fig. 1.1. The historical and conceptual tower of component systems. Some of them are full-fledged composition systems. The central technical concept is denoted by italic shape. Examples are denoted by typewriter font.

and aspect systems. In the last decade, these new programming styles have been developed more or less in isolation. Based on well-known and new results, this book shows that all approaches can be modeled as different variants of invasive composition techniques. Mainly, there are three reasons for this. Firstly, invasive composition defines a graybox component model that allows for invasive adaptations of components. A *fragment box* may consist of a set of arbitrary program fragments¹ that can be adapted at variation points, so-called *hooks*. Fragment boxes can be used to model generic components, views, and aspects. Secondly, *composition operators* (*composers*) transform the hooks to other program elements (*invasive composition*). In this way, they parameterize, adapt, connect, and extend components and realize the basic composition operations for the above technologies. Thirdly, we employ an object-oriented language, Java, for composition. Such a language facilitates configuration management and can be employed to describe generic expanders, view mergers, and aspect weavers. The demonstrator library of invasive composition, Compost, only requires standard Java tooling and can be integrated easily into the object-oriented software development process.

¹ A *fragment* is a snippet from a program or a specification. In BETA, it is a sentinel derivable from a nonterminal of the grammar [Lehrmann Madsen, 1994].

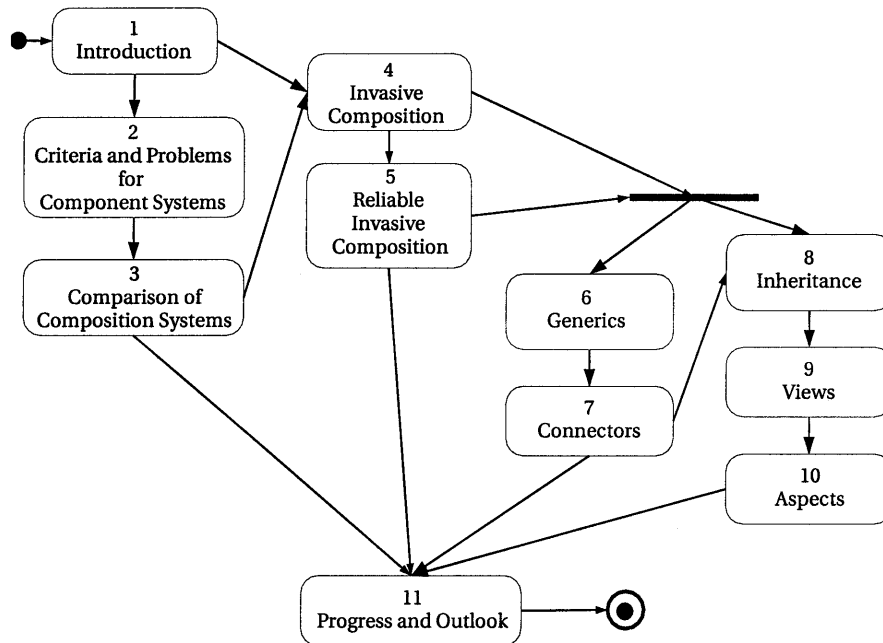


Fig. 1.2. The roadmap for this book. Readers may skip the chapters that review the literature and start with Chap. 4. After Chap. 5, several applications are presented, some on declared hooks, others on implicit hooks.

Invasive composition has more advantages. Although it uses a graybox model, several soundness criteria can be defined that ensure the reliability of the compositions. For instance, *sound view merging* and *sound weaving* can be defined, operations that enable reliable views and aspects. Then, software variant and build management can be simplified by composition programs. Since compositions are expressed in the same language as the components, the same composition mechanisms can be applied to the components as well as to composition recipes. This opens a way for *meta-composition*, a technology to simplify the production of composition programs for large systems.

1.1 A Short Overview of the Book

The three basic mechanisms of a composition system – *component model*, *composition technique*, and *composition language* – motivate a list of requirements for composition (Sect. 1.5). To introduce this, the rest of this chapter investigates three case studies of component systems. Then,

Chap. 2 outlines the requirements. Chapter 3 evaluates several approaches to component systems according to these criteria and introduces terminology. This reveals that, over time, the component models, the composition techniques, and the composition languages have been considerably generalized. Chapter 3 also presents several modern approaches to software composition. By reading both chapters, the reader should get an overview of what a future composition system should look like.

Chapter 4 presents the component model of invasive composition and its demonstrator library Compost. Compost contains composers for Java fragment boxes and can be used to write composition recipes as object-oriented programs. Chapter 5 presents some formal features of the composition-based software process and defines basic criteria for reliable compositions, i.e., compositions which do not invalidate component uses.

Parts III and IV of the book present applications of invasive composition. Chapters 6 and 7 show applications of *declared* hooks that have been defined by component developers in *composition interfaces*. Chapter 6 discusses *generic programming*. Hooks generalize *generic type parameters* to arbitrary generic program elements. Chapter 7 discusses applications in *software architecture systems*. A composer can model a *connector*, and a hook can model a *port* (both are concepts from architecture languages). Hence, composition programs are architecture descriptions but rely on a standard language.

Chapters 8–10 show applications of *implicit* hooks, hooks that are always available in a component. Chapters 8 and 9 discuss concepts from object-oriented programming and view-based programming. Briefly, composers can extend hooks of fragment box components. This can be used to model inheritance and views. Finally, Chap. 10 discusses *aspect-oriented programming*. This engineering method separates specifications of aspects from the algorithmic components and weaves them together with *aspect weavers*. Fragment boxes model *aspects*, hooks model *join points*, and aspect weavers can be regarded as complex composition operators.

To begin, the rest of this introduction presents several case studies on different aspects of system construction. Firstly, we look at the origins of UNIX, one of the oldest component systems. Secondly, the CoSy system, a modern component system for compiler construction and repository-based architectures, is presented. CoSy supports view-based engineering, and due to this flexibility, it is a commercial success. Next, aspect-oriented programming (AOP) is investigated. We end this chapter with a list of important requirements for system composition.

1.2 The Component Dream

My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry.

D. McIlroy [McIlroy, 1969]

Back in 1968, a group of leading computer scientists met in Garmisch-Partenkirchen, Germany, to look at an important question encountered in the software development of the 1960s: how can software be produced systematically. A draft of the conference was recorded [Buxton et al., 1969], showing that the conference invented three terms that shaped the field of software construction for at least 30 years.

Firstly, the conference coined the term *software crisis*, stating that the size and complexity of software systems had grown so enormously that planning, implementation, and maintenance could no longer be managed. Secondly, the conference proposed considering this black art of programming as a well-defined engineering science, the science of *software engineering* [Thayer and McGettrick, 1993]. Thirdly, a visionary engineer named Douglas McIlroy presented a paper, *Massproduced Software Components*, in which he claimed that any ripe industry is based on *component technology* [McIlroy, 1969]. Several phases of industrial maturity can be distinguished: the phase of manufacturing by skilled individuals, the chaotic phase at the beginning of industrial rationalization, and the mature phase of well-defined production processes in which all steps of a product's assemblage are standardized so that the products can be massproduced. In the 1960s, software construction was undoubtedly in the first phase. But still today, we see many products produced in the way our forefathers produced their clothes: by individual handcrafting. McIlroy clearly stated that what was needed to overcome this phase of infancy was a mature technology that would provide parameterizations, assemble components in well-defined procedures, and configure complete systems by pressing some buttons in a configuration tool.

Despite enormous progress in software construction, McIlroy's vision has not yet become a reality. Both research and industry have developed a large number of approaches for component technology, but none of them has solved all of the problems. McIlroy himself has realized at least part of his dream. In the 1970s, he influenced UNIX. One of his major contributions was *pipes* [Kernigham and McIlroy, 1990]. The idea is to have a set of components (called *processes*), which communicate via byte streams (*pipes*) and build up *pipe-and-filter* graphs. The UNIX shell has been built around this paradigm and is still one of the most popular component environments [Bourne, 1978].

Why has McIlroy's component model been successful? In this early work, we can already identify the three basic concepts of composition

systems. Firstly, the *component model* of UNIX is very simple but, nevertheless, flexible. Every component has a *standardized interface* with simple connection points, three standard byte streams `stdio`, `stdout`, and `stderr`. Due to this strict standardization, every component can be coupled to every other component while it can interpret the byte stream in its own way, keeping the information about what the data means as its secret.

Secondly, the *composition technique* of UNIX is also simple: components are attached to byte streams, no matter where they come from or go to. The technique abstracts from the byte stream's location and transfer method, i.e., it ignores whether a component is reading from a shared buffer, from an Internet socket, from a named pipe, or from a disk; all techniques are hidden behind the same interface. Also, UNIX provides simple filter components (such as `cat`, `tr`, or `sed`). These filters modify data in the byte streams between components and can be used to *adapt* components to each other if they do not fit. Without these little languages, UNIX would not have had that much success.

Thirdly, UNIX contains at least three *composition languages* for constructing composition recipes, the *shell*, the *C programmers workbench* [Dolotta et al., 1978], and *makefiles* [Feldman, 1978, Feldman, 1988]. Both shell scripts and C programs connect sets of components and build larger systems. Shells focus on flexibility and ease, while C provides type checking² and efficiency. Makefiles provide a simple mechanism to rebuild a system. Makefile-like tools are the most widely used system configuration mechanisms, in particular, for large systems with thousands of variants and versions.

What Can We Learn for Software Composition? Future component systems should support *standard interfaces*. They should provide languages or other mechanisms to *adapt* components to each other, as well as a bunch of *composition languages*. On the other hand, UNIX pipes-and-filters are not the only way towards component systems that you can imagine. But how do we go beyond them? What about other application domains where pipe-and-filters are not sufficient? The next section presents a system that goes beyond UNIX, in so far as data that is exchanged can be typed and extended easily.

² Of course, C has a weak typing concept, but it is stronger than that of the shell, which only knows about strings.

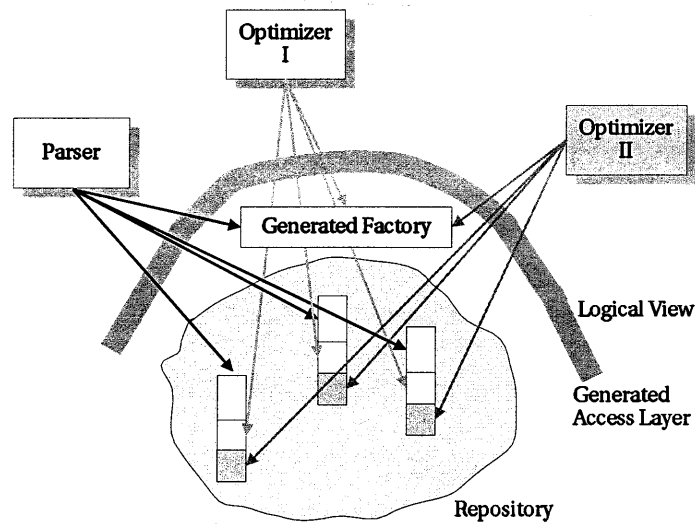


Fig. 1.3. A CoSy compiler's architecture: a repository system. Adding a new optimizer leads to extension of shared data structures. The logical view of an engine on the repository is mapped to the physical form with the help of a generated factory class and access layer.

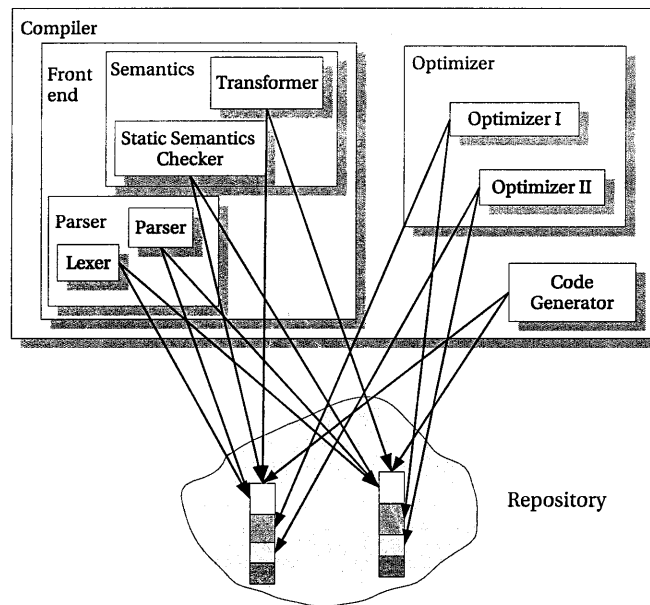


Fig. 1.4. The CoSy compiler architecture can group engines into compound engines. Appropriate glue code is generated to mediate between engine protocols.

1.3 CoSy: A Component System for Adaptation and Extension

As far back as 1988, ACE could see the limitations of the classical approach to compiler construction and joined forces with Europe's best compiler researchers to create a revolution in compiler production technology. Today, compiler developers use the CoSy Compilation System to construct production quality compilers for a variety of programming languages, processor architectures, and software development markets...

The modular approach, covering isolated compiler component development, reuse of components and the specialization and focus of compiler development groups, leads in turn to lower development and maintenance costs...

Within two days of installing the CoSy software, Ericsson Radio Systems' engineers had modified a sample CoSy compiler and were generating executable code for their DSP cores. Within 12 months, validated compilers were available ...

Excerpts from advertisements on CoSy (ACE b.V., Amsterdam)

From 1990 to 1995, a group of several European companies and research institutes executed a research project on compilers and compiler frameworks, COMPARE (COMPilers for PARallel architectures). Its goal was to develop compilers for efficient code on novel architectures and to provide a compiler component system that would facilitate their construction. It was foreseen that novel architectures would need extensive optimizations; and since developing optimizers would cost a lot, the component system should guarantee a maximal amount of reuse of components. Somewhat later, the project resulted in a commercial product, the CoSy compiler framework. This framework is now successfully marketed by ACE b.V., Amsterdam, and has already been used to build many industrial-strength compilers [ACE b.V, 2000a, ACE b.V, 2000b].

The motivations for developing the CoSy framework were the following [Alt et al., 1994]. Firstly, it should be easy to assemble new compilers from a set of prefabricated components. Secondly, extending compilers with new parts should be easy (Fig. 1.3). Different phases of the compiler share common data in a repository, and that data has to be extended appropriately. Since a compiler company ships binary components to its clients, extension of the data structures should not require recompilation of the components. Thirdly, despite the necessary modular structure, compilers should still run fast. Having too many interfaces within the system should be avoided since it slows down the compiler.

Most of these goals have been reached in CoSy. First of all, CoSy permits configuring new compilers from a prefabricated set of components (also called *engines*) within an hour. CoSy contains a composition language, the

engine description language (EDL), which is used to specify the architecture of the compiler. Since sets of engines can be encapsulated into larger subsystems, a compiler is hierarchically structured (Fig. 1.4). Engines may be grouped sequentially, in a data parallel or pipelined fashion, or in a client-server manner. From these specifications, the EDL compiler generates coordination code [Alt et al., 1993, Alt, 1997]. EDL offers several standard interaction protocols that the EDL compiler maps to each other. For engines that do not fit directly with each other, it generates adaptation code. Hence, CoSy is one of the few commercially available frameworks for repository-based architectures.

Secondly, CoSy-made compilers can be extended easily, since it provides a flexible access mechanism to the shared-data repository. Each engine accesses the common data with a specific *view*. Depending on which and how many engines are configured in the compiler, the common data is extended appropriately. Due to the extension mechanism, even binary engines can be reused without recompilation, although their underlying data structures are extended.

Conceptually, this provides two advances over UNIX. Firstly, components no longer read and write streams of bytes. Instead, they communicate complex data structures in a repository. This saves the overhead of externalizing and reparsing data structures. It is the reason why repository-based systems are more efficient than their UNIX counterparts. Secondly, components can be extended, together with the data they communicate in the repository. In UNIX, components are blackboxes.

However, such an extensible component system is confronted with a severe technical problem. What happens if we extend a compiler with a new optimizer phase *Optimizer-II* (Fig. 1.3)? In a class-based system, this means that base classes common to the parser and the optimizers have to be extended (*base-class extension*). However, then all their subclasses must be recompiled, as well as all client components that use them. In a commercial setting in which binary components are sold and distributed, this poses a problem: whenever a compiler is extended with a new component, all other components will be invalidated. Hence, when a company extends the framework's base classes it should ship new binaries of all components and force the client to recompile his applications. Normally, a company cannot afford such a procedure. This extensibility problem is called the *syntactic fragile base-class problem (syntactic FBCP)* [Szyperski, 1998]. First identified in [Forman et al., 1995, Hamilton, 1996], it is an obstacle to the commercial component-based development of large object-oriented systems or libraries.

As a solution to this extensibility problem, CoSy employs the view concept. An engine's view describes which data is required from the repository, which data is shared with others, and how the data is accessed. Views are specified in a data description language, fSDL, which supports multi-

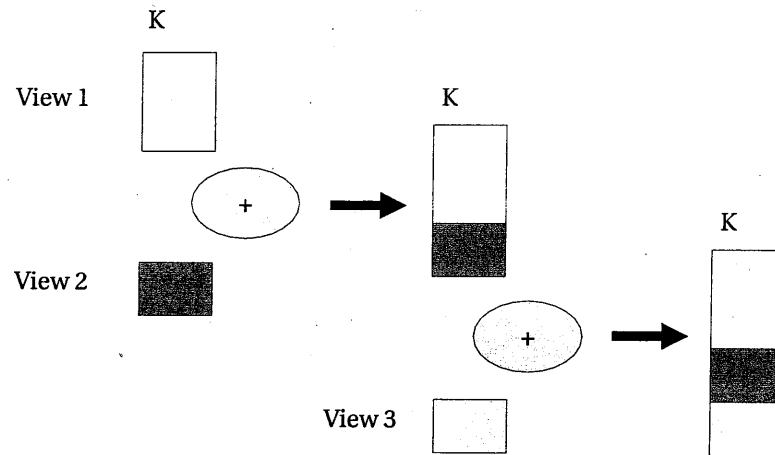


Fig. 1.5. A view merge operator in CoSy.

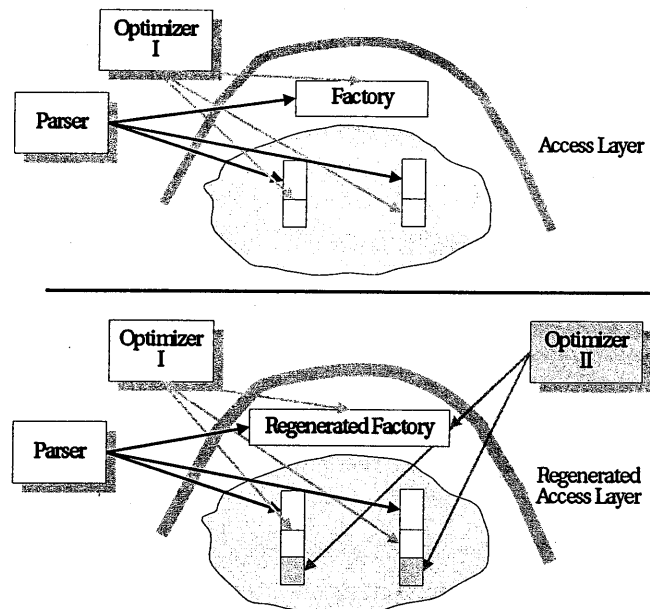


Fig. 1.6. When a new component is added, factory and access layers are regenerated, to hide the extension of the common data structures.

ple and mixin-based inheritance [Walters et al., 1994, Buhl, 1995].³ Based on this, fSDL defines *view merge operators* (Fig. 1.5). With these operators, CoSy merges all view specifications of engines. The operators check that no fields are contradictorily defined and calculate the final layout of every class in the repository.

Then, this layout is used to calculate an engine-specific *view mapping* that maps the logical view of every component to the access of the physical structures. From the view mapping, the implementation of two system layers is generated: a *factory* to create repository data objects [Gamma et al., 1994] and an *access layer* to access the objects (Fig. 1.3). (A *factory* is a class whose methods create objects [Gamma et al., 1994]. If it substitutes standard object constructors, its implementations can be exchanged, parameterizing the behavior of the system.) At runtime, the factory is used to allocate data in the repository; the data is accessed by the access layer.

Equipped with this generation power, extending a compiler becomes easy. When a component with a new view is added to the system, the view mapping is recalculated and the factory and the access are regenerated and recompiled (Fig. 1.6). However, the rest of the system, also binary components, need not be recompiled and can be reused as is.

What Can We Learn for Software Composition? CoSy indicates that extensibility is important for future component-based development. It demonstrates that views and view mappings permit systems to be extended easily. CoSy solves the syntactic fragile base-class problem: even if a compiler has been built from binary components, it can be evolved easily since all accesses to the repository are generated.

1.4 Aspect Separation: New Dimensions of Modularity

Modularity helps when developing systems. Modules can be replaced by other modules while the rest of the system stays intact. However, designers would like to modularly exchange in several dimensions: different features should be exchanged independently of each other. For instance, in the architecture of buildings, plans for rooms, water, gas, and electricity are specified separately (Fig. 1.7). When architects want to exchange parts of the electricity support for a room, they never exchange the complete room. Instead, they only modify the electricity plan of the room, which does not affect the other plans. After all plans are finished, the construction process integrates them into the physical layout of the building and eliminates remaining conflicts.

³ *Mixin-based inheritance* is a variant of inheritance. During the inheritance step, class fragments, so-called *mixins*, are integrated into the superclass to form the subclass.

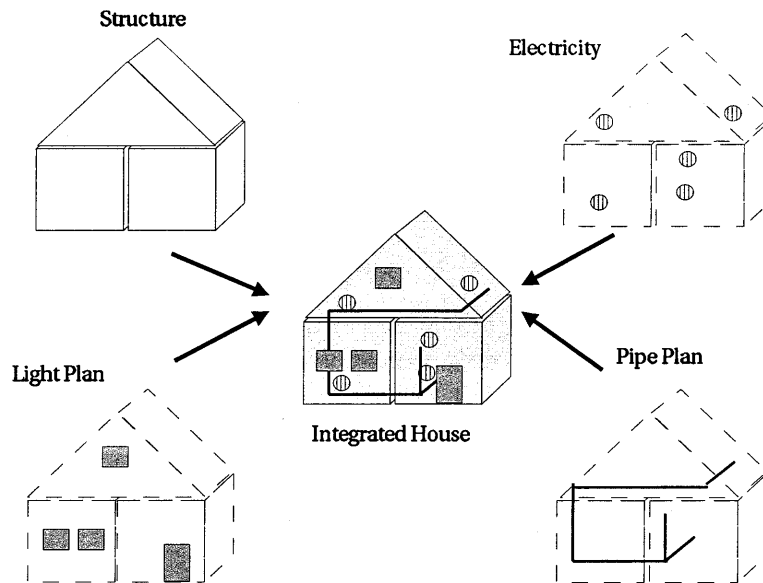


Fig. 1.7. Aspect separation in architecture. The construction process weaves separate plans into an integrated building.

This principle of *aspect separation* can also be found in software engineering [Kiczales et al., 1997]. Aspects of software, such as persistence, debugging, or animation, should be described separately and exchanged independently without disturbing the modular structure of the system.

Example 1.1 For the examples in this book, we use a simple scenario, the *production cell* case study [Lewerentz and Lindner, 1995]. The case study models part of a production cell from a company in Karlsruhe, Germany. The cell contains several machines that process a pipeline of metal blanks. A conveyor belt, a rotary table, and a robot transport the metal blanks to a press. After giving the workpieces a new form, the robot takes the forged metal plates out of the press and a second conveyor belt transports them into a repository. Appendix B explains in more detail how the cell works.

As an example for aspects, consider the class *Robot* in Fig. 1.8. If we want to debug an implementation of the production cell, we might want to insert print statements at the entry and exit of the procedures. Usually, this is done by hand, by preprocessor macros, or by delegation to a debugging class; none of the methods supports specifying the algorithm and its debugging aspect separately.

Separating specifications of *aspects* from the *core* components is the main idea of aspect-oriented programming (AOP) [Kiczales et al., 1997].

<pre> 1 public class Robot { 2 public void spin() { 3 while (true) { lifeCycle (); } 4 } 5 6 7 public void lifeCycle () { 8 WorkPiece p; 9 rotateToTable(); 10 p = takeUp(); 11 rotateToPress(); 12 layDown(p); 13 } 14 15 16 } </pre>	<pre> 1 public class Robot { 2 public void spin() { 3 <u>System.out.println("enter spin");</u> 4 while (true) { lifeCycle (); } 5 <u>System.out.println("exit spin");</u> 6 } 7 public void lifeCycle () { 8 <u>System.out.println("enter lifeCycle");</u> 9 WorkPiece p; 10 rotateToTable(); 11 p = takeUp(); 12 rotateToPress(); 13 layDown(p); 14 <u>System.out.println("exit lifeCycle");</u> 15 } 16 } </pre>
--	--

Fig. 1.8. A method with algorithm and debugging aspect. On the right side, the debugging aspect is woven in (denoted by underlined font).

While the essence of an algorithm, i.e., the application-specific functionality, still resides in the core, all other aspects are segregated out to aspect specifications. A special compiler, a so-called *weaver*, mixes the specifications and translates them to the final form (Fig. 1.9). Most often, the weaving process relies on common names in the specifications which relate specification items to each other. Those names are called *join points*. They must occur in the core components so that the aspect specifications can refer to them. Hence, in AOP, systems have a core part to which several aspects relate.

Example 1.2 Figure 1.9 mentions two aspects, debugging and persistence. A production cell might be programmed for persistent behavior. When the cell must be stopped the machines should not forget which workpieces they carried such that the cell can be revived. Let us suppose that we define the following two aspect specifications for the class Robot.

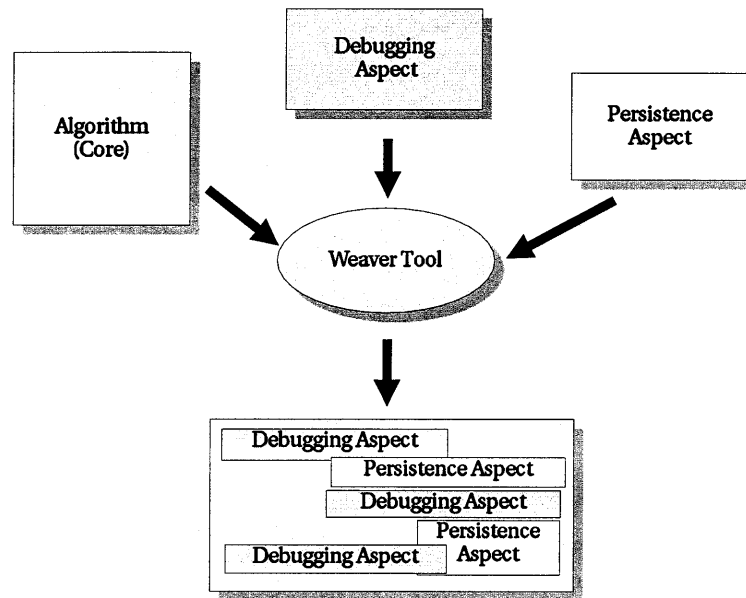


Fig. 1.9. Aspect-oriented specification of software. Aspects are separated, but woven together with a specific compiler, the weaver.

```

1 aspect Persistence {
2   Robot is persistent;
3 }
4 aspect Debugging {
5   Robot.methods have prologue {
6     System.out.println("enter <method>");
7   }
8   Robot.methods have epilogue {
9     System.out.println("exit <method>");
10  }
11 }
  
```

A weaver could be specified with the following simple expression in a weaving language:

```

1 Package robots = weave Robot with Persistence, Debugging;
  
```

It evaluates the aspect specifications and generates the code of List. 1.1 for Robot, distributing the aspect code over the system: all methods of Robot are extended with debugging code, and the constructor is made to load objects from a database.

```

1 public class Robot {
2   public Robot() {
3     System.out.println("enter Robot");
4     this = <<load Robot object from database>>;
5     System.out.println("exit Robot");
6     return this;
7   }
8   public void spin() {
9     System.out.println("enter spin");
10    while (true) { lifeCycle (); }
11    <<store Robot object to database>>;
12    System.out.println("exit spin");
13  }
14  public void lifeCycle () {
15    System.out.println("enter lifeCycle");
16    WorkPiece p;
17    rotateToTable();
18    p = takeUp();
19    rotateToPress();
20    layDown(p);
21    System.out.println("exit lifeCycle");
22  }
23 }

```

List. 1.1. A first version of the robot of the production cell.

This separation of concerns simplifies the structure of the software. Since aspects are specified independently, the algorithms can be described more clearly, without being intermingled with unnecessary details. Also, the aspect specifications become rather simple, clear, and concise.

Hence, aspect weaving offers a new form of configuration management. Aspect specifications can be exchanged independently of the core and independently of each other. Each configuration creates a system that may have different behavior or nonfunctional qualities. A simple gedanken experiment shows why this increases reuse. If a system consists of n modules with v variants per module, $n \times v$ variants can be built. If it consists additionally of k aspects and each aspect has l variations, $nvkl$ variants can be built without ever editing one of the $n \times v$ modules. In other words, $nvkl$ variants can be built by developing $nv + kl$ module and aspect variants. Since the same core components can be reused in entirely different reuse contexts, aspect-oriented programming improves reuse.

What Can We Learn for Software Composition? Future component systems should support aspect separation and should integrate a mechanism to compose the aspects and the components.

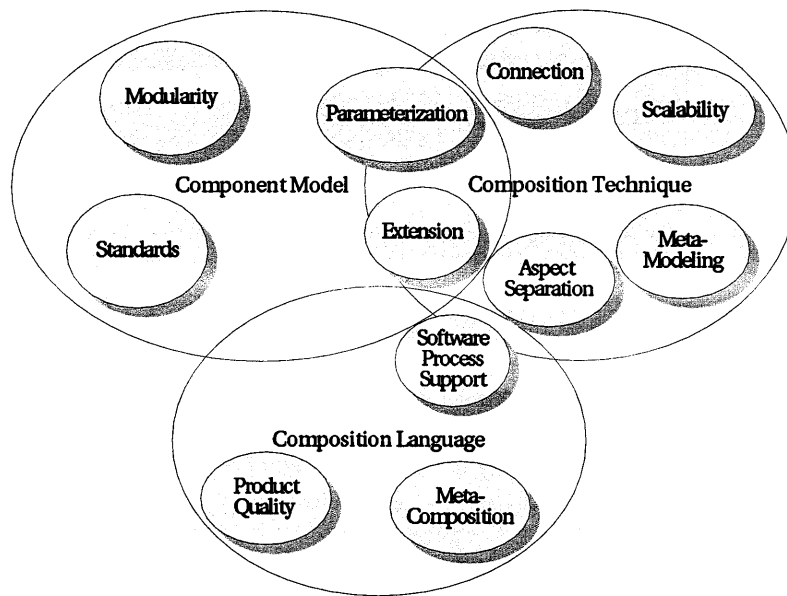


Fig. 1.10. Requirements for software composition.

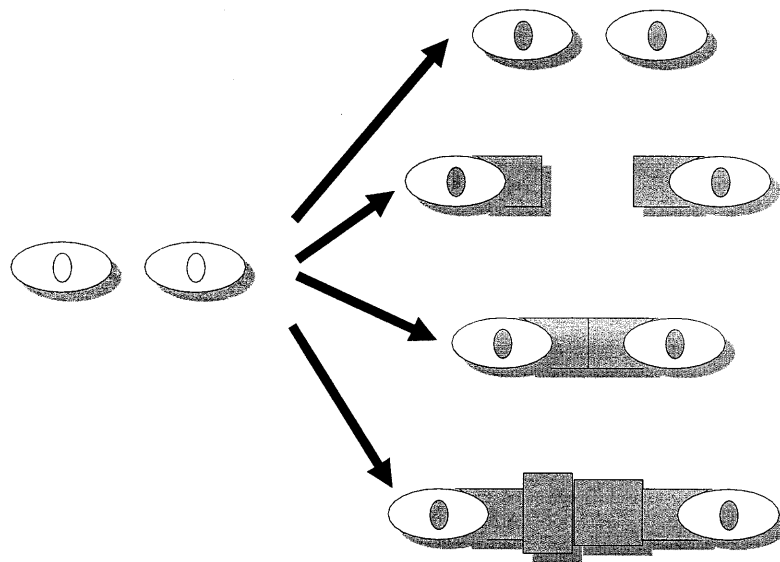


Fig. 1.11. Tasks during composition from top to bottom: 1) parameterization, 2) parameterization and adaptation to interfaces, 3) parameterization and connection with adaptation, and 4) parameterization, connection, adaptation, and gluing.

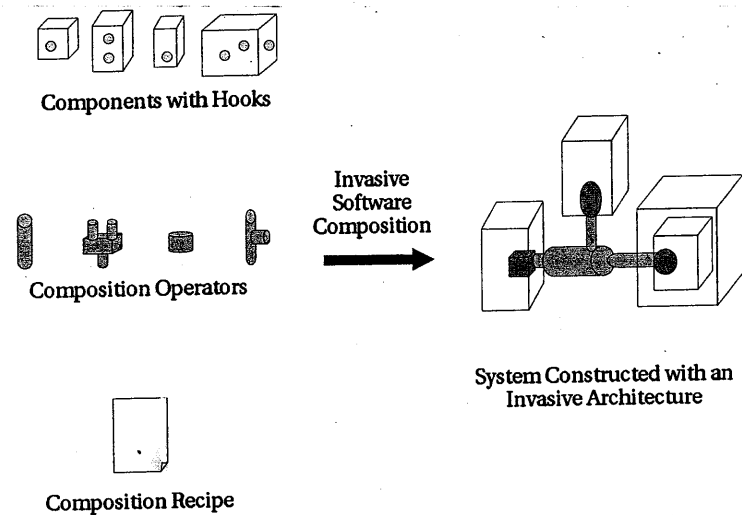


Fig. 1.12. The global picture of invasive software composition. Components are grayboxes. Composers glue components, modify them at hooks, and integrate them into each other.

1.5 Requirements for Composition

The previous sections raised some questions that are important to answer for future composition systems. These questions lead to the following composition requirements (Fig. 1.10), to be detailed in Chap. 2. The figure groups them into three categories, requirements of the component model (*what should a component look like?*), requirements of the composition technique (*how should components be composed?*), and requirements of the composition language (*how is a system composed?*).

Component model The following requirements should hold for a component model of a composition system.

Modularity. Building software is expensive. We would like to reduce the costs and shorten the time to market. Building software from prefabricated components off-the-shelf is attractive, since components can be reused in several, if not many, systems. But what should the interfaces of a module look like?

Parameterizability. Often components need to be parameterized before they can be reused in an application. However, standard parameterization mechanisms are restricted to type parameters in languages with generic classes. Can we generalize these?

Conformance to Standards. Components should conform to standards. They guarantee a better fit of components during composition and lower the learning curve for component-based development.

Composition Technique The following should hold for the basic composition technique of a composition system.

Connection. Types of parameters, protocols, and assertions should be adapted to each other. For this, a component system should provide *adaptation* and *gluing* (Fig. 1.11). *Adaptation* makes a component fit to an interface. Then the component is better prepared for reuse and can be plugged together with other components. *Gluing* mediates between specific components. Often, a component system generates *glue code* that maps component protocols specifically to each other. And this increases reuse.

Extensibility. Often, software is not built for change. Instead, it should be easy to extend existing systems with new functionality, new parts, and new nonfunctional qualities. Such updates should be done automatically, i.e., without editing old parts of the system.

Aspect Separation. In analogy to building architectures, components should not only be composed in a blackbox manner. Instead, different functional and nonfunctional aspects should be distinguished.

Scalability. Connections and other compositions should scale in binding time and technique.

Metamodeling. For components to be adapted and modified during composition, the composition needs to have a model of the components. If composition language and component language coincide, such a model is a *metamodel*.

Composition Language Besides a component model and a composition technique, a component system should provide a composition language.

Product-Consistency Support. A composition language should help to ensure quality features of software systems.

Software-Process Support. The language should support the composition-based software construction process. The language should be expressive enough to express variants and versions of product lines, and should be powerful enough to describe large systems. Additionally, the language should be easy to understand.

Metacomposition Support. As the composition recipes can grow with the system, the language itself should be based on composition.

1.6 Epilogue

1.6.1 Remarks

What should be understood as a component is pretty much debated in the literature. Components appear on different granularity levels, deal with different stakeholder requirements, or are simply design concepts. In this book, we will assume that a component is a software part that must be *composed* with other components in a *system composition* to form a final system. Hence, a *software component* is simply a software item that is subject to software composition. It may appear on different levels of granularity, may be a design or implementation item, and may be in source or binary form. And a *system composition* is a software build process by which components are composed for software products. On top of this pretty general definition, specific *component models* can be defined. A *component model* summarizes requirements, features, and interfaces of a set of components off-the-shelf.

In some systems, a component is understood to be a runtime object. Although we refer to components as static items, invasive composition can also be defined similarly for runtime components (Sect. 4.5.1). Focusing on the static scenario has advantages: firstly, it permits a static type check of the system composition, and secondly, it produces more efficient systems.

1.6.2 Goals of Invasive Software Composition

The quality of life is just like that: it cannot be made, but only generated. In our time we have come to think of works of art as "creations", conceived in the minds of their creators.

The quality without a name cannot be made like this.

C. Alexander [Alexander, 1979]

The following chapters intend to show that *invasive software composition* is a new technology for improving reuse in the construction of software systems (Fig. 1.12). Invasive composition fulfills many of the requirements from the previous section. Hence, it provides a basis for second-generation component systems, composition systems. Along with the technique, the Compost system will be described. It supports a gray-box component model, an invasive composition technique, and Java as the composition language.

Was [Alexander, 1979] right in claiming that beauty cannot be invented, but only be generated, i.e., composed from basic patterns? For the field of software, judge yourself, after you have seen the concepts of invasive software composition.



<http://www.springer.com/978-3-540-44385-8>

Invasive Software Composition

Aßmann, U.

2003, XII, 334 p., Hardcover

ISBN: 978-3-540-44385-8