

Chapter 1

The Many Faces of Complexity in Software Design

José Luiz Fiadeiro

Abstract ‘Complexity’ and ‘crisis’ have become synonyms in the (brief) history of Software Engineering. The terms ‘component’, ‘decomposition’, ‘structure’ and ‘architecture’ have been associated with methods and techniques proposed over the years to defeat the crisis, from structured programming to object/component based programming and, more recently, service-oriented architectures. This chapter discusses the nature of complexity as it arises in software design, assesses the progress that we have achieved in tackling it, and discusses some of the challenges that still remain.

1.1 Introduction

Complexity, not in the formal sense of the theory of algorithms or complexity science, but in the more current meaning of “the state or quality of being intricate or complicated”, seems to be unavoidably associated with software. A few quotes from the press over the last 10 years illustrate the point:

- *The Economist*, 12/04/2001 — In an article aptly called “The beast of complexity”, Stuart Feldman, then director of IBM’s Institute for Advanced Commerce, is quoted to say that programming was “all about suffering from ever-increasing complexity”
- *The Economist*, 08/05/2003 — A survey of the IT industry acknowledges that “computing has certainly got faster, smarter and cheaper, but it has also become much more complex”
- *Financial Times*, 27/11/2004 — The British government’s chief information officer gives the following explanation for the Child Support Agency IT project failure: “Where there’s complexity, there will, from time to time, be problems”
- *The Economist*, 06/09/2007 — In an article called “The trouble with computers”, Steven Kyffin, then senior researcher at Philips, is quote to concede that computer programmers and engineers are “compelled by complexity”

- *Financial Times*, 27/01/2009 — “It is very easy to look at the IT industry and conclude that it is fatally attracted to complexity”

But why are we so bothered about complexity? The following quote from the *Financial Times* of 27/01/2009 summarises the point quite effectively:

Complexity is the enemy of flexibility. It entangles us in unintended consequences. It blocks our attempts to change. It hides potential defects, making it impossible to be sure our systems will function correctly. Performance, transparency, security — all these highly desirable attributes leak away in the face of increasing complexity.

In this chapter, we argue that, although the public in general would readily accept that software is ‘complicated’, complexity in the sense of the quotes above has lurked under many guises since the early days of programming and software engineering, which explains why software seems to be in a permanent ‘crisis’. We also discuss the ways that we, computer scientists, have been devising to tackle “the beast of complexity”, which we classify into two main activities: abstraction and decomposition.

1.1.1 Abstraction

Abstraction is an activity that all of us perform on a daily basis without necessarily realising so. Abstraction is one of the ways we use to go around the complexity of the world we live in and simplify the way we interact with each other, organisations, systems, and so on.

Bank accounts provide a rich and mundane example of the way we use abstraction, as the following ‘story’ illustrates. In “A Visit to the Bank”, Paddington Bear goes to Floyds Bank to withdraw money for his holiday. He decides to leave the interest in for a rainy day but is horrified to learn that it only amounts to three pence. Tension mounts when he finds out that he cannot have back the very same notes that he deposited in the first place: he knew perfectly well that he had spilled marmalade over them...

We (or most of us) have learnt that an account is not a physical storage of bank notes that we manipulate through the cashier just as we would do with a safe box or a piggy bank. However, the advantages (and dangers?) of working with bank accounts as abstractions over the physical manipulation of currency are not restricted to avoiding handling sticky bank notes (or other forms of ‘laundering’...). Indeed, a bank account is not (just) a way of organising the storage of money, but of our business interactions: it solves a good part of the complexity involved in business transactions by decoupling our ability to trade from the manipulation of physical bank notes.

Much the same can be said about the way we use computers. Abstraction pervades computing and much of the history of computer science concerns precisely the development of abstractions through which humans can make full usage of the

(computational) power made available by the machines we call computers by tackling the complexity of programming them. In the words of Peter Denning [17]:

Most computing professionals do not appreciate how abstract our field appears to others. We have become so good at defining and manipulating abstractions that we hardly notice how skilfully we create abstract ‘objects’, which upon deployment in a computer perform useful actions.

Paddington’s view of his bank account may make us smile because these are abstractions that we have learnt to live with a long time ago. However when, not long ago, we tried to organise a transfer from Leicester to Lisbon, it turned out that providing the clerk with the SWIFT and IBAN codes was not sufficient and that a full postal address was indispensable. (No, this was not at Floyds Bank but a major high-street bank in the UK.) What is more, when given a post code that, for some reason, did not look credible enough to his eyes, the clerk refused to go ahead with the transfer on the grounds that “the money might get lost in the post”...

This example from ‘real life’ shows that the fact that abstraction is such a routine activity does not mean that we are all equally and well prepared to perform it in a ‘professional’ way — see [48] for a discussion on how abstraction skills in computer science require education and training. The following paragraph from the 27/01/2009 article of the Financial Times quoted above can help us understand how abstraction relates to complexity:

Most engineers are pretty bright people. They can tolerate a lot of complexity and gain a certain type of power by building systems that flaunt it. If only we could get them to focus their intellect instead on eliminating it. The problem with this message is that, for all our best efforts, we almost never eliminate complexity. Most of the time, when we create a system that appears simple, what we have actually done is shift the complexity somewhere else in the technology stack.

Indeed, operating systems, compilers and, more recently, all sorts of ‘clever’ middleware support the layers of abstraction that allow us to program software systems without manipulating directly the code that the machine actually understands (and we, nowadays, rarely do). The current emphasis on model-driven development is another example of this process of abstraction, this time in relation to programming languages, avoiding that IT specialists spread marmalade over lines of code...

Why is it then that, in spite of phenomenal progress in computer science for at least three decades, which the quote from P. Denning acknowledges, is complexity still haunting software as evidenced by the articles cited at the beginning of this section? Expanding the quote from the 08/05/2003 edition of The Economist:

Computing has certainly got faster, smarter and cheaper, but it has also become much more complex. Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. [...] In the late 1990s, the internet and the emergence of e-commerce “broke ITs back”. Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. [...]

What is the significance of the internet to the complexity of software? In this chapter, we will be arguing that the reason for the persistence of the ‘complexity

crisis' is in the change of the nature of complexity, meaning that programming and software engineering methodology often lags behind advances in more technological areas (such as the internet) and, therefore, fails to develop new abstractions that can be used for tackling the complexity of the systems that are being built.

1.1.2 Decomposition

Although we started this chapter with quotes that have appeared in the press during the last 10 years, the threat of complexity was the topic of a famous article published in the Scientific American 10 years before that, in 1994, following on the debacle of the Dallas international airport baggage handling system — glitches in the software controlling the shunting of luggage forced the airport to sit empty for nine months:

The challenge of complexity is not only large but also growing. [...] When a system becomes so complex that no one manager can comprehend the entirety, traditional development processes break down. [...] To keep up with such demand, programmers will have to change the way that they work. [...] Software parts can, if properly standardised, be reused at many different scales. [...] In April [1994], NIST announced that it was creating an Advanced Technology Program to help engender a market for component-based software.

Nothing very surprising, one could say. Indeed, another way of managing complexity that we use in our day to day is embedded in the Cartesian principle of divide and conquer — breaking a complicated problem down into parts that are easier to solve, and then build a solution to the whole by composing the solutions to the parts.

The literature on component-based software development (CBD) is vast (e.g., [10, 15]). Therefore, what happened to component-based software if, according to the sources quoted by The Economist in 08/05/2003, the challenge of complexity was still growing in 2003? A couple of years later, an article in the 26-01-2005 edition of the Financial Times reported:

“This is the industrial revolution for software,” says Toby Redshaw, vice-president of information technology strategy at Motorola, the US electronics group. He is talking about the rise of service oriented architectures (SOAs) a method of building IT systems that relies not on big, integrated programs but on small, modular components.

“Small, modular components”? How is this different from the promise reported in the Scientific American? What is even more intriguing is that the article in the Scientific American appeared almost 20 years after Frank DeRemer and Hans H. Kron wrote [18]:

We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.[...] We argue that structuring a large collection of modules to form a ‘system’ is an essentially distinct and different intellectual activity from that of constructing the individual modules.

Why didn't these modules fit the bill given that, in 1994, component-based software was being hailed as the way out of complexity? DeRemer and Kron's article

itself appeared eight years after the term ‘software crisis’ was coined at the famous 1968 NATO conference in Garmisch-Partenkirschen which Douglas McIlroy’s addressed with a talk on Mass Produced Software Components.

Given that, today, we are still talking about ‘the crisis’ and ‘components’ as a means of handling complexity, did anything change during more than 40 years? As argued in the previous subsection, our view is that it is essentially the nature of the crisis that has been changing, prompting for different forms of decomposition and, therefore, different notions of ‘component’. Whereas this seems totally uncontroversial, the problem is that it is often difficult to understand what exactly has changed and, therefore, what new abstractions and decomposition methods are required. For example, the fact that component-based development is now a well-established discipline in software engineering makes it harder to argue for different notions of component. This difficulty is well apparent in the current debate around service-oriented computing.

The purpose of this chapter is to discuss the nature of complexity as it arises in software design, review the progress that we have achieved in coping with it through abstractions and decomposition techniques, and identify some of the challenges that still remain. Parts of the chapter have already been presented at conferences or colloquia [22, 23, 24, 25]. The feedback received on those publications has been incorporated in this extended paper. Sections 1.2, 1.3 and 1.4 cover three different kinds of programming or software design — ‘programming in-the-small’, ‘programming in-the-large’ and ‘programming in-the-many’, respectively. Whereas the first two have been part of the computer science jargon for many years, the third is not so well established. We borrow it from Nenad Medvidović [54] to represent a different approach to decomposition that promotes *connectors* to the same status as *components* (which are core to programming in-the-large) as first-class elements in software architectures [66]. Section 1.5 covers service-oriented computing and contains results from our own recent research [27, 28, 30], therefore presenting a more personal view of an area that is not yet fully mature.

The chapter is not very technical and does not attempt to provide an in-depth analysis of any of the aspects that are covered — several chapters of this volume fulfil that purpose.

1.2 Programming in-the-small

The term programming in-the-small was first used by DeRemer and Kron [18] to differentiate between the activity of writing ‘small’ programs from those that, because of their size, are best decomposed into smaller ones, possibly written by different people using programming-in-the-small techniques. To use an example that relates to current programming practice, writing the code that implements a method in any object-oriented language or a web service would be considered as programming in-the-small.

Precisely because they are ‘small’, discussing such programs allows us to illustrate some of the aspects of complexity in software development that do not relate to size. For example, the earlier and more common abstractions that we use in programming relate to the need for separating the software from the machine that runs it. This need arises from the fact that programming in machine code is laborious (some would say complicated, even complex). The separation between program and code executable on a particular computer is supported by machine-independent programming languages and compilers. This separation consists of an abstraction step in which the program written by the programmer is seen as a higher-level abstraction of the code that runs on the machine.

High-level programming languages operate two important abstractions in relation to machine code: control execution and memory. Introduced by E. Dijkstra in the 70s [19], structured programming promoted abstractions for handling the complexity of controlling the flow of execution; until then, control flow was largely defined in terms of *goto* statements that transferred execution to a label in the program text, which meant that, to understand how a program executed, one had to chase *goto*’s across the text and, inevitably, would end tangled up in complex control flows (hence the term ‘spaghetti’ code). The three main abstractions are well known to all programmers today — sequence, selection, and repetition. As primitives of a (high-level) programming language, they transformed programs from line-oriented to command-oriented structures, opening the way to formal techniques for analysing program correctness.

Another crucial aspect of this abstraction process is the ability to work with data structures that do not necessarily mirror the organisation of the memory of the machine in which the code will run. This process can be taken even further by allowing the data structures to reflect the organisation of the solution to the problem. This combination of executional and data abstraction was exploited in methodologies such as JSP — Jackson Structured Programming [43] — that operate a top-down decomposition approach. The components associated with such a decomposition approach stand for blocks of code that are put together according to the executional abstractions of structured programming (sequential composition, selection and iteration). Each component is then developed in the same way, independently of the other components. The criteria for decomposition derive from the structure of the data manipulated by the program.

JSP had its own graphical notation, which we illustrate in Fig. 1.1 for a run-length encoder — a program that takes as input a stream of bytes and outputs a stream of pairs consisting of a byte along with a count of the byte’s consecutive occurrences in the input stream. This JSP-diagram includes, at the top level, a box that represents the whole program — *Encode run lengths*. The program is identified as an iteration of an operation — *Encode run length* — that encodes the length of each run as it is read from the input. The input is a stream of bytes that can be viewed as zero or more runs, each run consisting of one or more bytes of the same value. The fact that the program is an iteration is indicated by the symbol * in the right hand corner of the corresponding box. This operation is itself identified as the sequential composition of four more elementary components. This is indicated by the sequence of boxes

that decompose *Encode run length*. The second of these boxes — *Count remaining bytes* — is itself an iteration of an operation — *Count remaining byte* — that counts bytes.

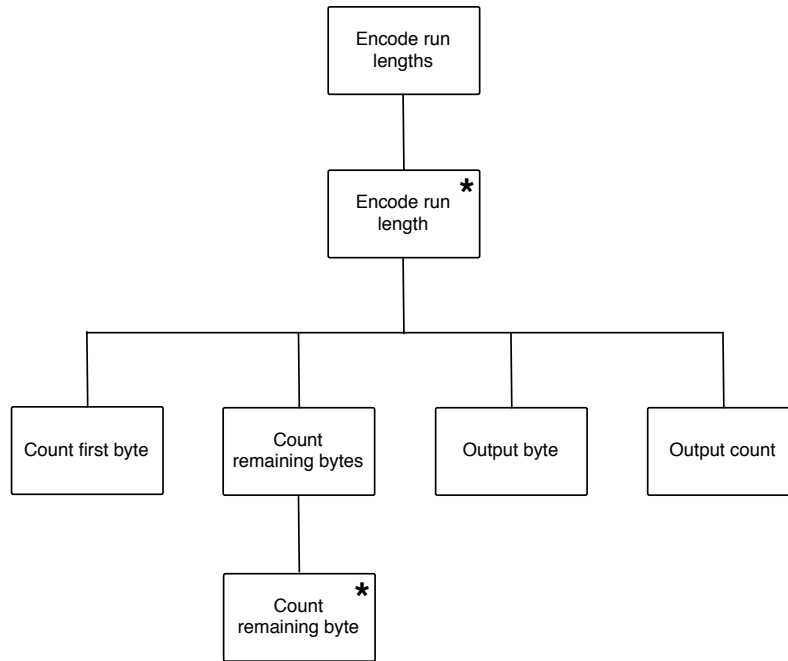


Fig. 1.1 Example of a JSP-diagram.

An advantage of structured programming is that it simplifies formal verification of program correctness from specifications, for example through what is known as the Hoare calculus [41] (see also [40, 60]). Typically, we consider a specification to be a pair $[p, q]$ of state conditions¹. A program satisfies such a specification if, whenever its execution starts in a state that satisfies p (called the ‘pre-condition’) and terminates, the final state satisfies q (called the ‘post-condition’).

In order to illustrate how, together with the Hoare calculus, we can define a notion of ‘module’ (or component) through which we can define a compositional (bottom-up) approach to program construction, we introduce another graphical notation that we will use in other sections to illustrate similar points in other contexts.

An example of what we will call a program module is given in Fig. 1.2. Its meaning is that if, in the program expression $C(c1, c2)$, we bind $c1$ to a program that satisfies the specification $[p1, q1]$ and $c2$ to a program that satisfies the specification $[p2, q2]$, then we obtain a program that satisfies the specification $[p, q]$.

¹ A frame — the set of variables whose values may change during the execution of the program — can be added as in [60]. For simplicity, we only consider partial correctness in this chapter; techniques for proving that the program terminates, leading to total correctness, also exist [40, 60].

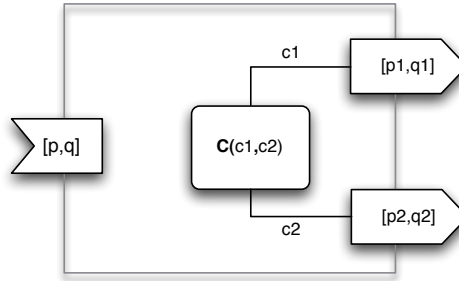


Fig. 1.2 A program module.

One can identify $[p, q]$ with the interface that is *provided* by the module, and $[p1, q1]$ and $[p2, q2]$ with those of ‘components’ that are *required* by the module so that, upon binding, the expression forms a program that meets the specification $[p, q]$. Notice that the module does not need to know the inner workings of the components that implement $[p1, q1]$ and $[p2, q2]$ in order to make use of them, thus enforcing a form of encapsulation.

Using this notation, we can define a number of module schemas that capture the rules of the Hoare calculus and, therefore, define the basic building blocks for constructing more complex programs. In the Appendix (Fig. 1.30) we give the schemas that correspond to assignments, sequence, iteration, and selection. Two instances of those schemas are presented in Fig. 1.3: one for assignment and one for iteration.

Modules can be composed by binding a requires-interface of one module with the provides-interface of another. Binding is subject to the rules of *refinement* [60]: $[p, q] \sqsubseteq [p', q']$ iff $p' \dashv p$ and $q' \vdash q$. That is, $[p', q']$ refines $[p, q]$ if its pre-condition p' is weaker than p and its post-condition q' is stronger than q . This is illustrated in Fig. 1.4.

The result of the binding is illustrated in Fig. 1.5: the body of the right-hand-side module is used to (partially) instantiate the program expression of the left-hand-side module; the resulting module has the same provides-interface as the left-hand-side module, and keeps the unused requires-interface of the left-hand-side module and the requires-interface of the right-hand-side module. A concrete example is given in Fig. 1.6 for the binding of the two modules depicted in Fig. 1.3 (notice that, x being an integer program variable, the condition $x > 0$ entails $x \geq 1$).

These notions of program module and binding are, in a sense, a reformulation of structured programming intended to bring out the building blocks or component structure that results from the executional abstractions. Notice that, through those modules, it is the program as a syntactic expression that is being structured, not the executable code: there is encapsulation with respect to the specifications as argued above — the interface (specification) provided by a module derives only from the interfaces (specifications) of the required program parts — but not with respect to

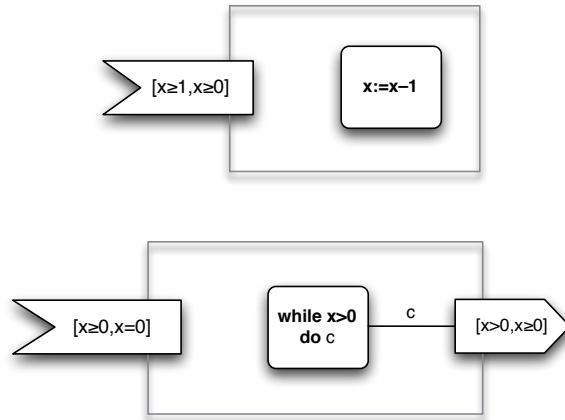


Fig. 1.3 An instance of the assignment schema and one of iteration.

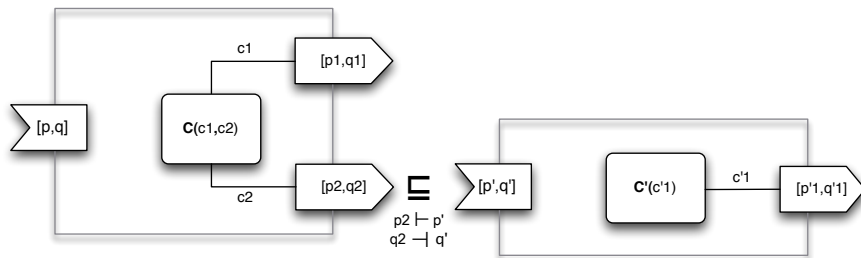


Fig. 1.4 Binding two modules.

the executable code: in the second module on Fig. 1.3, one cannot reuse code generated for c to generate code for *while $x > 0$ do c* . Other programming abstractions exist that allow for code to be reused, such as procedures.

Procedural abstractions are indeed a way of developing resources that can be reused in the process of programming an application. Resources can be added to program modules through what we would call a uses-interface. Examples are given in Fig. 1.7, which correspond to two of the schemas discussed in [60] (see also [40]): one for substitution by value and the other for substitution by result. Uses-interfaces are different from requires-interfaces in the sense that they are preserved through

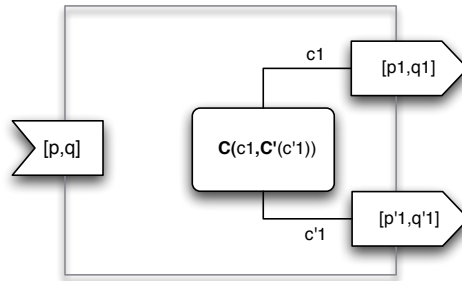


Fig. 1.5 The result of the binding in Fig. 1.4.

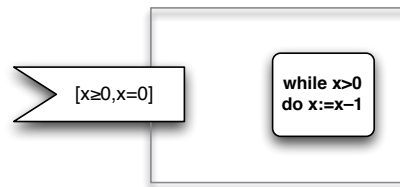


Fig. 1.6 The result of binding the modules in Fig. 1.3.

composition, i.e., there is no syntactic substitution like in binding. Like before, the module does not need to know the body of the procedure in order to make use of it, just the specification, thus enforcing a form of encapsulation.

JSP-diagrams can be viewed as providing an architectural view (*avant la lettre*, as the notion of software architecture emerged only much later) of programs. To make the connection with other architectural views reviewed in later sections of this chapter, it is interesting to notice JSP-diagrams can be combined with the notion of program module that we defined above. Essentially, we can replace the syntactic expressions inside the modules by JSP-diagrams as illustrated in Fig. 1.8. Binding expands the architecture so that, as modules are combined, the JSP-architecture of the program is built.

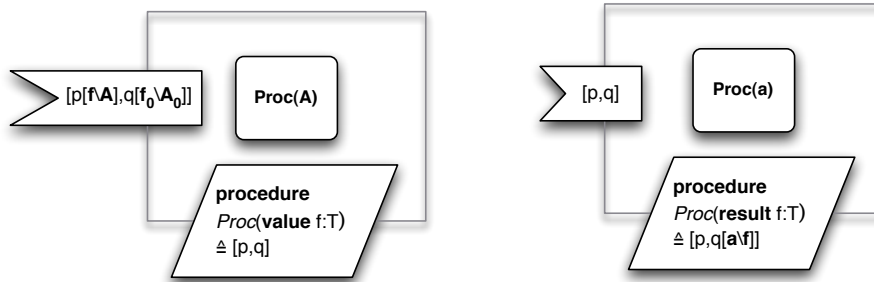


Fig. 1.7 Two schemas for procedural abstraction (see [60] for details). By A_0 we denote the value of the expression A before the execution of the command (procedure call).

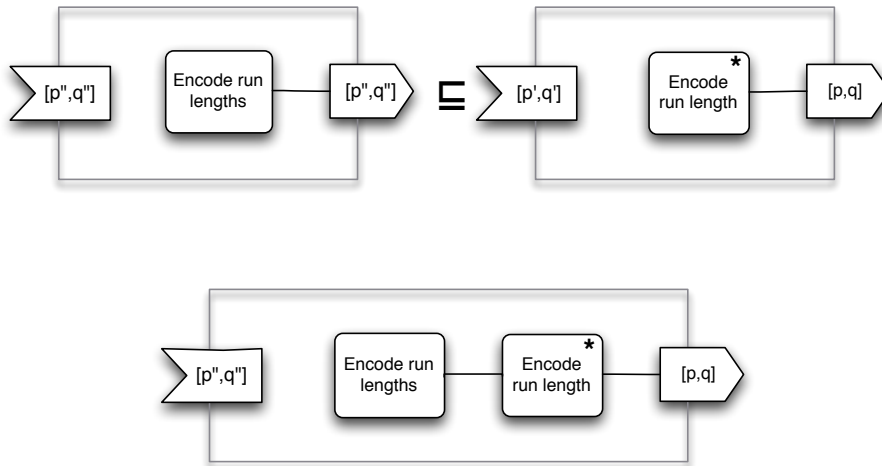


Fig. 1.8 Building JSP-diagrams through program-module composition.

1.3 Programming in-the-large

1.3.1 Modules and module interconnection languages

Whereas the program modules and JSP-diagrams discussed in the previous section address the complexity of understanding or developing (correct) executional structures, they do not address the complexity that arises from the size of programs (measured in terms of lines of code). This is why the distinction between programming in-the-small and programming in-the-large was introduced in [18]:

By large programs we mean systems consisting of many small programs (modules), possibly written by different people. We need languages for programming-in-the-small, i.e., languages not unlike the common programming languages of today, for writing modules. We also need a “module interconnection language” for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

Notice that, as made clear by the quote, the term programming in-the-small is not derogatory: ‘small’ programs whose correctness can be formally proved will always play an essential role in building ‘large’ software applications that we can trust to operate safely in mission-critical systems (from avionics to power plants to healthcare, *inter alia*). The problem arising at the time was that, as the scope and role of software in business grew, so did the size of programs: software applications were demanded to perform more and more tasks in all sorts of domains, growing very quickly into millions of lines of code. Sheer size compromised quality: delivery times started to suffer and so did performance and correctness due to the fact that applications became unmanageable for the lone programmer.

To address this problem, programming in-the-large offered a form of decomposition that addressed the global structure of a software application in terms of what its modules and resources are and how they fit together in the system. The main difference with respect to programming in-the-small is in the fact that one is interested not in structuring the *computational* process, but the *software-construction* (and evolution) process². Hence, the resulting components (modules) are interconnected not to ensure that the computation progresses towards the required *final state* (or post-condition, or output), but that, in the *final application*, all modules are provided with the resources they need (e.g., the parsing module of a compiler is connected to the symbol table). In other words, it is the flow of resources among modules, not of control, that is of concern.

The conclusions of Parnas’ landmark paper [62] are even clearer in this respect:

[...] it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed

² Procedural abstractions, as mentioned at the end of Section 1.2, do offer a way of simplifying program construction by naming given pieces of program text that would need to be repeated several times, but they are not powerful enough for the coarse-grained modularity required for programming in-the-large.

to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more sub-routines, and instead allow subroutines and programs to be assembled collections of code from various modules.

That is to say, we cannot hope and should not attempt to address the complexity of software systems as products with the mechanisms that were developed for structuring complex computations. That is why so-called module interconnection languages (MILs) were developed for programming in-the-large [63]. Indeed, the quote from [18] makes clear that the nature of the abstraction process associated with programming in-the-large is such that one can rely on a compiler to link all the modules together as intended by the programmer(s). Hence, MILs offered primitives such as *export/provide/originate* and *import/require/use* when designing individual modules at the abstract level so as to express the dependencies that would need to be taken into account at the lower level when “knitting the modules together”.

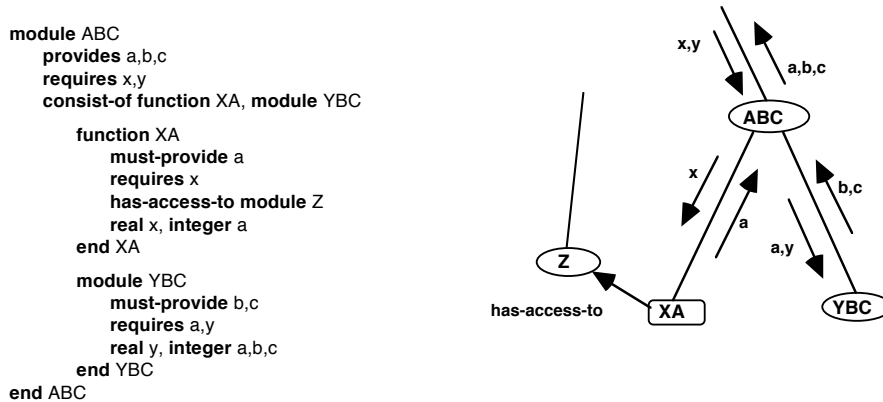


Fig. 1.9 An example of a MIL description taken from [63].

Module-interconnection structures are essential for project management, namely for testing and maintenance support: they enforce system integrity and inter-module compatibility; they support incremental modification as modules can be independently compiled and linked, and thus full recompilation of a modified system is not needed; and they enforce version control as different versions (implementations) of a module can be identified and used in the construction of a system. Figure 1.9 illustrates the kind of architecture that is described in such languages. The dependencies between components concern access to and usage of resources.

In order to illustrate how notions of module can be formalised, we use a very simple example in which modules consist of procedures, variables and variable ini-

tialisations (similar to [60]³). Procedures can be abstract in the sense that they are not provided with a fully-developed body (code). Some of those procedures or variables are exported and some are imported (imported procedures are abstract); the interface of the module consists of the specifications of exported and imported resources. An example, also borrowed from [60], is given in Fig. 1.10 where frames are added to pre-/post-condition specifications.

```

module Tag
  export Acquire, Return;
  import Choose;

  var u : set  $\mathbb{N}$ ;

  procedure Acquire (result t :  $\mathbb{N}$ )
     $\hat{=}$  Choose ( $\mathbb{N} - u, t$ );
     $u := u \cup \{t\}$ 

  procedure Return (value t :  $\mathbb{N}$ )
     $\hat{=}$   $u := u - \{t\}$ ;

  procedure Choose (value s : set  $\mathbb{N}$ ; result e :  $\mathbb{N}$ )
     $\hat{=}$   $e: [s \neq \{\}, e \in s]$ ;

  initially  $u = \{\}$ 
end

```

Fig. 1.10 Example of a module borrowed from [60].

Using a diagrammatic notation similar to that used in Section 1.2, we could represent the module *Tag* and its interface as in Fig. 1.11. We say that a module is *correct* if, assuming that resources (e.g., a procedure *Choose*) are provided that satisfy the specifications that label the import-interfaces, the body of the module (e.g., *Tag*) implements resources (e.g., procedures *Acquire* and *Return*) that satisfy the specifications that label the export-interfaces.

Binding two such modules together consists in identifying in one module some of the resources required by the other. This process of identification needs to obey certain rules, namely that the specification that labels the export-interface of one module refines the specification of the import-interface of the other module. This is illustrated in Fig. 1.12 where the specification of *Choose* is refined by that of *Pick* (*Pick* will accept a set of natural numbers and return a natural number).

Typically, in MILs, the result of the binding is a configuration as depicted in Fig. 1.13. In our case, the edge identifies the particular resource that is being imported. In

³ Notions of module were made available in the wave of programming languages that, such as Modula-2 [71], followed from structured programming.

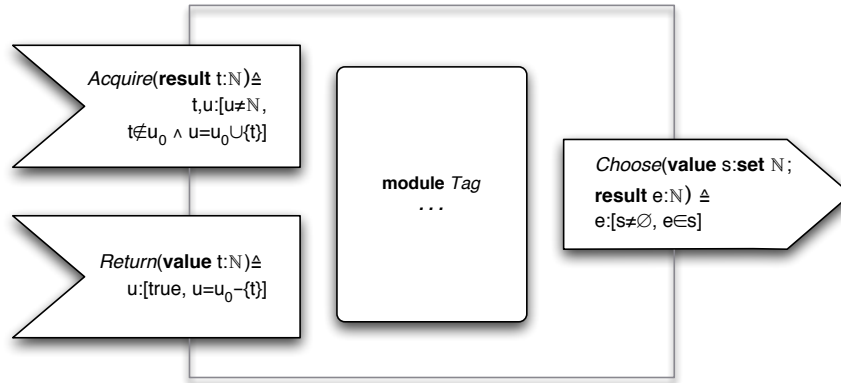


Fig. 1.11 An interface for the module in Fig.1.10.

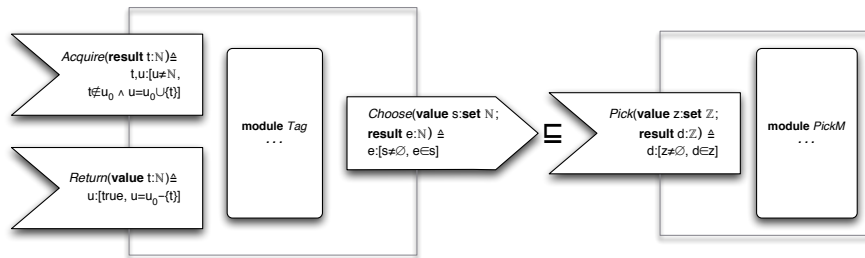


Fig. 1.12 Binding modules through refinement.

MILs, the link is represented by a direct reference made in the code inside the module (which is interpreted by the compiler as an instruction to link the corresponding implementations) and, in diagrams, the edge may be used to represent other kinds of relationships as illustrated in Fig. 1.9. Notice the similarity with the program modules defined in Section 1.2 where binding defines an operation over JSP-diagrams, which we can identify with program configurations (or architectures). The difference between the two notions is that MILs do not operate at the level of control structures (as JSP-diagrams do) but organisational ones.

Another important aspect of modules is reuse, which can be supported by a notion of refinement between modules. In the case of our example, and following [60] once again, we say that a refinement of a module $\langle Exp, Imp, Loc, Init \rangle$ — where Exp , Imp and Loc stand for the sets of exported, imported and local resources, respectively, and $Init$ is an initialisation command — by another module $\langle Exp', Imp', Loc', Init' \rangle$ consists of two injective functions $exp : Exp \rightarrow Exp'$ and $imp : Imp \rightarrow Imp' \cup Loc'$ such that, for every $e \in Exp$ (resp. $i \in Imp$), $e \sqsubseteq exp(e)$ (resp. $imp(i) \sqsubseteq i$), and $init \sqsubseteq init'$. Notice that exported interfaces of the refined module can promise more (i.e., they refine the original exported resources) but the imported interfaces of the original module cannot require less (i.e., they refine the

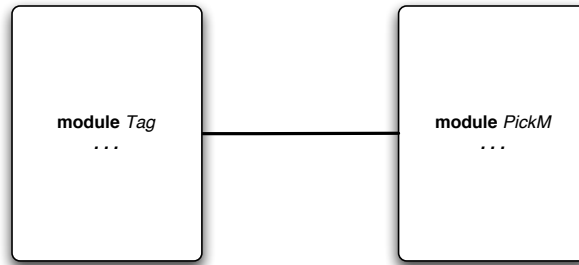


Fig. 1.13 Linking modules.

corresponding resources in the refined module). Moreover, imported resources of the original module can be mapped to local resources of the refined one.

1.3.2 Object-oriented programming

Object-oriented programming (OOP)⁴ can be seen to define a specific criterion for modularising code: objects group together around methods (variables, functions, and procedures) all the operations that are allowed on a given piece of the system state — “Object-oriented software construction is the software development method which bases the architecture of any software system on modules deduced from the types of objects it manipulates (rather than the function or functions that the system is intended to ensure)” [57].

This form of state encapsulation offers a mechanism of data abstraction in the sense that what is offered through an object interface is a collection of operations that hide the representation of the data that they manipulate. This abstraction mechanism is associated with so-called abstract data types [50] — “Object-oriented software construction is the building of software systems as structured collections of possibly partial abstract data type implementations” [57].

In OOP, modules are classes. A class interface consists of the specifications associated with the features that it provides to clients — attributes (A), functions (F), or procedures (P) — and a set of invariants (I) that apply to all the objects of the class. A class is correct with respect to its interface if the implementations of the features satisfy their specifications and the execution of the routines (functions or procedures) maintains the invariants. An example, using a diagrammatic notation similar to the one used in previous sections, is given in Fig. 1.14.

As modules, classes do not include an explicit import/require interface mechanism similar to the previous examples, which begs the question: how can modules be interconnected? OOP does provide a mechanism for interconnecting objects:

⁴ We follow Meyer [57] throughout most of this section and recommend it for further reading not just on object-oriented programming but modularity in software construction as well.

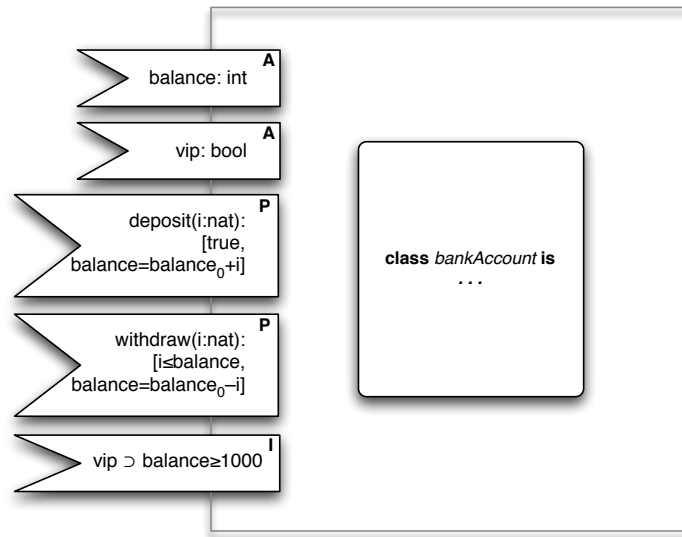


Fig. 1.14 The interface of the class *bankAccount*.

clientship — an object can be a client of another object by declaring an attribute (or function) whose type is an object class; methods of the client can then invoke the features of the server as part of their code⁵. For example, *bankAccount* could be a client of a class *customer* through an attribute *owner* and invoke *owner.addDeposit(i)* as part of the code that executes *deposit(i)* so as to store the accumulated deposits that customers make on all the accounts that they own.

The difference in relation to an import (or required) interface is that clientship is programmed in the code that implements the classes, not established through an external interconnection language. In a sense, clientship is a more sophisticated form of procedure invocation in which the code to be executed is identified by means of a pointer variable. That is, clientship is essentially an executional abstraction in the sense of programming in-the-small.

Classes do offer some ‘in-the-large’ mechanisms (and therefore behave as modules) through the mechanism of *inheritance*. Inheritance makes it possible for new classes to be defined by adding new features to, or re-defining features of, existing classes. This mechanism is controlled by two important restrictions: extension of the set of features is constrained by the need to maintain the invariants of the source class; redefinition is constrained by the need to refine the specifications of the features. An example of a class built by inheriting from *bankAccount* is given in Fig. 1.15. These restrictions are important for supporting dynamic binding and polymorphism, which are run-time architectural techniques that are typically absent from MILs (where binding is essentially static, i.e., performed at compile time).

⁵ Import statements can be found in OOP languages such as Java, but they are used in conjunction with packages in order to locate the classes of which a given class is a client.

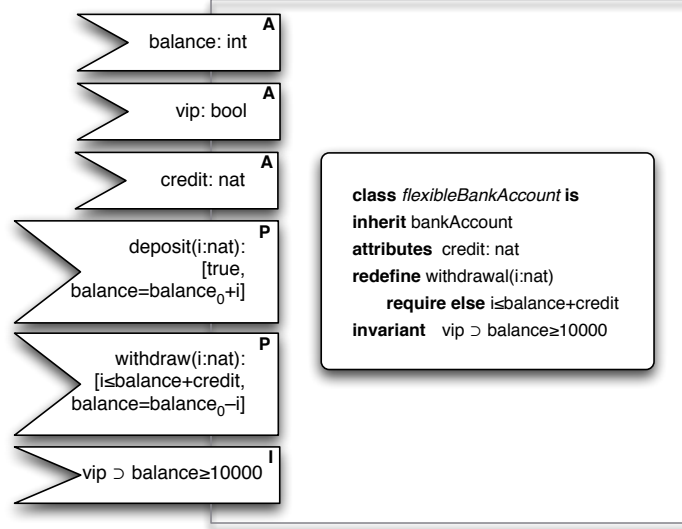


Fig. 1.15 The interface of the class *flexibleBankAccount*, which inherits from *bankAccount*.

Formally, inheritance can be defined as a mapping ρ between the interfaces of the two classes, say $\langle A, F, R, I \rangle$ and $\langle A', F', R', I' \rangle$, such that

1. for every routine $r: [p, q] \in F \cup R$, if $\rho(r): [p', q']$ then
 - a. $p' \dashv \rho(p)$
 - b. $q' \vdash \rho(p_0) \supset \rho(q)$
2. $I' \vdash \rho(I)$

Notice that the first condition is a variation on the notion of refinement used in Section 1.2 in which the post-condition of the redefined routine needs to imply the original post-condition only when the original pre-condition held before execution. On the other hand, the original invariant cannot be weakened (it needs to be implied by the new one). Together, these conditions ensure that an instance of the refined class can be used where an instance of the original class was expected. Notice the similarity between this formalisation of inheritance and that of module refinement discussed in Section 1.3.1.

Multiple and repeated inheritance offer a good example of another operation on modules: composition, not in the sense of binding as illustrated previously, but on building larger modules from simpler ones. An example of repeated inheritance (copied from [57]) is shown in Fig. 1.16: repeatedly inherited features that are not meant to be shared (for example, *address*) need to be renamed.

Formally, repeated inheritance can be defined over a pair of inclusions $C_1 \overset{I_1}{\hookrightarrow} C \overset{I_2}{\hookrightarrow} C_2$ between sets of features (inheritance arrows usually point in the reverse direction of the mappings between features) where C contains the features that are meant to be shared between C_1 and C_2 ; these inclusions give rise to another pair of mappings

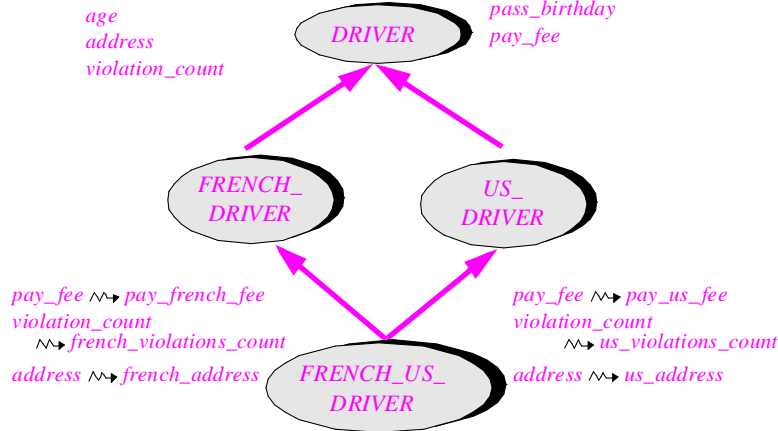


Fig. 1.16 An example of repeated inheritance borrowed from [57].

$C_1 \xrightarrow{\rho_1} C' \xrightarrow{\rho_2} C_2$ that define an amalgamated union of the original pair. An amalgamated union is an operation on sets and functions that renames the features of C_1 and C_2 that are not included in C when calculating their union. In relation to the specifications of the shared routines, i.e., routines $r':[p',q'] \in F' \cup R'$ such that there is $r \in F \cup R$ with $r' = \rho_1(t_1(r)) = \rho_2(t_2(r))$, we obtain:

1. $p' = \rho_1(p_1) \vee \rho_2(p_2)$
2. $q' = \rho_1(p_{1_0} \supset q_1) \wedge \rho_2(p_{2_0} \supset q_2)$

where $t_n(r):[p_n,q_n]$. These are the combined pre-/post-condition rules of Eiffel, which give the semantics of interface composition.

The reason we detailed these constructions is that they allow us to discuss the mathematical semantics of refinement (including inheritance) and composition. We have already seen that logic plays an essential role in the definition of specifications and refinement or inheritance. Composition (in the sense of repeated inheritance) can be supported by category theory, a branch of mathematics in which notions of structure can be easily expressed and operations such composition can be defined that preserve such structures. For instance, one can express refinement (or inheritance) as a morphism that preserves specifications (i.e., through refinement mappings), from which composition operations such as repeated inheritance result as universal constructions (e.g., pushouts in the case at hand). Amalgamated union is an example of a universal construction and so are conjunction and disjunction — composition (in the sense of repeated inheritance) operates as disjunction on pre-conditions but as conjunction on post-conditions precisely because the inheritance morphism is co-variant on post-conditions but contra-variant on pre-conditions. Several other examples are covered in [21], some of which will be discussed in later sections.

The use of category theory in software modularisation goes back many years and was pioneered by J. Goguen — see, for example, [39] for an overview of the use

of category theory in computer science and [12] for one of the first papers in which the structuring of abstract data type specifications was discussed in mathematical terms⁶. Abstract data types (ADTs) are indeed one of the pillars of object-oriented programming but it would be impossible to cover in this chapter the vast literature on ADT specification. See also [38] on how ADTs can be used in the formalisation of MILs. Finally, it is important to mention that ADTs, specifications (pre-/post-conditions and invariants) as well notions of abstraction and refinement/reification, are also at the core of languages and methods such as VDM [44], B [1] and Z [72], each of which offer their own modularisation techniques.

1.3.3 Component-based software development

The article of the Scientific American quoted in Section 1.1.2 offers component-based software explicitly as a possible way out of the ‘software crisis’. However, one problem with the term ‘component’ is that, even in computer science, it is highly ambiguous. One could say that every (de)composition method has an associated notion of component: *ça va de soi*. Therefore, one can talk of components that are used for constructing programs, or systems, or specifications, and so on. In this section, we briefly mention the specific notion of component-based software that is usually associated with the work of Szyperski [67]⁷ because, on the one hand, it does go beyond MILs and object-oriented programming as discussed in the previous two sub-sections and, on the other hand, it is supported by dedicated technology (e.g., Sun Microsystem’s Enterprise JavaBeans or Microsofts’s COM+) and languages and notations such as the UML (e.g., [15]), thus offering a layer of abstraction that is available to software designers.

Indeed, component-based development techniques are associated with another layer of abstraction that can be superposed over operating systems. So-called component frameworks make available a number of run-time layers of services that enforce properties such as persistence or transactions over which one can rely on when developing and interconnecting components to build a system. By offering interconnection standards, such frameworks also permit components to be connected without knowing who designed them, thus promoting reuse.

Components are not modules in the sense of programming in-the-large (cf. Section 1.3): a component is a software implementation that can be executed, i.e., a resource; a module is a way of hiding design decisions when organising the resources that are necessary for the construction of a large system such as the usage of components. Components also go beyond objects in the sense that, on the one hand, components can be developed using other techniques than object-oriented programming

⁶ See also [37] on the applications of category theory to general systems theory.

⁷ “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.”

and, on the other hand, the interconnection mechanisms through which components can be composed are also quite different from clientship.

More specifically, one major difference between a component model and an object-oriented one is that all connections in which a component may be involved are made explicit through provides/exports or requires/imports interfaces that are *external* to the code that implements the component — “in a component setting, providers and clients are ignorant of each other” [67]. In the case of OOP, connections are established through clientship and are only visible by inspecting the code that implements the objects — the client holds an explicit reference to and calls the client, i.e., the connections are not mediated by an interface-based mechanism that is external to the code. That is, one could say that objects offer a white-box connection model whereas components offer a black-box one.

Whereas components in the sense discussed above are essentially a way of modularising implementation (and promoting reuse), there is another important aspect that is often associated with components — their status as architectural elements and the way they modularise change, i.e., the focus is on “being able to manage the total system, as its various components evolve and its requirements change, rather than seeking to ensure that individual components are reusable by multiple component systems” [15].

From the point of view of complexity, the aspects of component-based software that interest us are the notions of interface and binding/composition of a component model. Typically, a component specification is defined in terms of the interfaces that the component provides (or realizes) and those that it requires (or uses), and any dependencies between them. An interface is, as before, a set of operations, each specified via pre-/post-conditions, and an ‘information model’ that captures abstract information on the state of the component and the way the operations relate to it. Specific notations have been proposed within the UML for supporting the definition of components or component specifications, including the ‘lollipop’ for provided interfaces and the ‘socket’ for required interfaces. An example is shown in Fig. 1.17, using the stereotype ‘specification’ to indicate that the architecture applies to component specifications, not to instances (implementations) [15].

Connections between components are expressed through ‘assembly connectors’ by fitting balls into sockets — they bind the components together but do not compose them. However, components can have an internal structure that contains sub-components wired together through assembly connectors. The rules and constraints that apply to such forms of composition are not always clear, especially in what relates to specifications.

1.4 Programming in-the-many

We borrow the term ‘programming in-the-many’ from Nenad Medvidović [54] and use it to mark the difference between the concern for size that is at the core of programming in-the-large and the complexity that arises from the fact that systems are

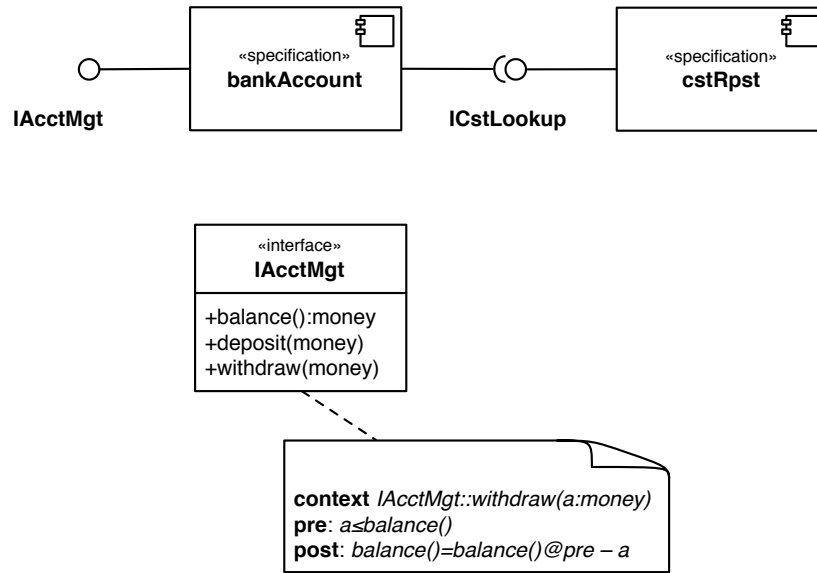


Fig. 1.17 An example of a component specification architecture using UML notation.

ever more distributed and heterogeneous, and that software development requires the integration and combination of possibly ‘incompatible’ systems. An important driver for this more modern emphasis comes from the pressures that are put on systems to be flexible and agile in the way they can respond to change. As put in [32], “[...] the ability to change is now more important than the ability to create [e-commerce] systems in the first place. Change becomes a first-class design goal and requires business and technology architectures whose components can be added, modified, replaced and reconfigured”.

This is not to say that research in component-based development has not addressed those challenges. For example, design mechanisms making use of event publishing/subscription through brokers and other well-known patterns [34] have found their way into commercially available products that support various forms of agility in the sense that they make it relatively easy to add or remove components without having to redesign the whole system. However, solutions based on the use of design patterns are not at the level of abstraction in which the need for change arises and needs to be managed. Being mechanisms that operate at the design level, there is a wide gap that separates them from the application modelling levels at which change is better perceived and managed. This conceptual gap is not easily bridged, and the process that leads from the business requirements to the identification and instantiation of the relevant design patterns is not easily documented or made otherwise explicit in a way that facilitates changes to be operated. Once instantiated, design patterns code up interactions in ways that, typically, requires evolution to be

intrusive because they were not conceived to be evolvable: most of the times, the pattern will dissolve as the system evolves.

Therefore, the need arises for semantic primitives founded on first principles through which interconnections can be externalised, modelled explicitly, and evolved directly, leading to systems that are ‘exoskeletal’ in the sense that they exhibit their configuration structure explicitly [47]. This is why, in this section, we would like to emphasise a different form of abstraction and decomposition that promotes ‘connectors’ to the same status as components as first-class elements in software architectures⁸ [66].

Connector abstractions [56] and the architectural styles that they promote are also supported by developments in middleware [58, 59], including the use of reflection [46]. An important contribution to this area comes from so-called coordination languages and models [36]. These languages promote the separation between ‘computation’ and ‘coordination’, i.e., the ability to address the computations that need to take place locally within components to implement the functionalities that they advertise through their interfaces separately from the coordination mechanisms that need to be superposed on these computations to enable the properties that are required of the global behaviour of the system to emerge. An example is *Linda* [35], implemented in *Java* through *JavaSpaces*, part of the *Jini* project (see also IBM’s *TSpaces* as another example of coordination middleware). Another example is *Manifold* [5]. Whereas, in *Linda*, components communicate over shared tuple-spaces [7], *Manifold* is based on an event-based communication paradigm — the *Idealized Worker Idealized Manager* (IWIM) model [3].

The importance of this separation in enabling change can be understood when we consider the complexity that clientship raises in understanding and managing interactions. For example, in order to understand or make changes to the way objects are interconnected, one needs to examine the code that implements the classes and follow how, at run time, objects become clients of other objects. This becomes very clear when looking at a UML collaboration diagram for a non-trivial system. In a sense, clientship brings back the complexity of ‘spaghetti’ code by using the equivalent of *goto*’s at the level on interactions.

Several architectural description languages (ADLs) have emerged since the 90s [55]. Essentially, these languages differ from the MILs discussed in Section 1.3.1 in that, where MILs put an emphasis on how modules *use* other modules, ADLs focus instead on the organisation of the *behaviour* of systems of components interconnected through protocols for communication and synchronisation. This explains why, on the semantic side, ADLs tend to be based on formalisms developed for supporting concurrency or distribution (Petri-nets, statecharts, and process calculi, *inter alia*). Two such ADLs are *Reo* [4], which is based on data streams and evolved from

⁸ As could be expected, the term ‘architecture’ is as ambiguous as ‘component’. We have argued that every discipline of decomposition leads to, or is intrinsically based on, a notion of part (component) and composition. The way we decompose a problem, or the discipline that we follow in the decomposition, further leads to an architecture, or architectural style, that identifies the way the problem is structured in terms of its sub-problems and the mechanisms through which they relate to one another.

the coordination language *Manifold* mentioned above, and *Wright* [2], based on the process algebra CSP — Communicating Sequential Programs [42].

In order to illustrate typical architectural concepts and their formalisation, we use the basic notion of connector put forward in [2]: a set of *roles*, each of which identifies a component type, and a *glue* that specifies how instances of the roles are interconnected. The example of a *pipe* is given in Fig. 1.18 using the language COMMUNITY [31] (CSP is used in [2]).

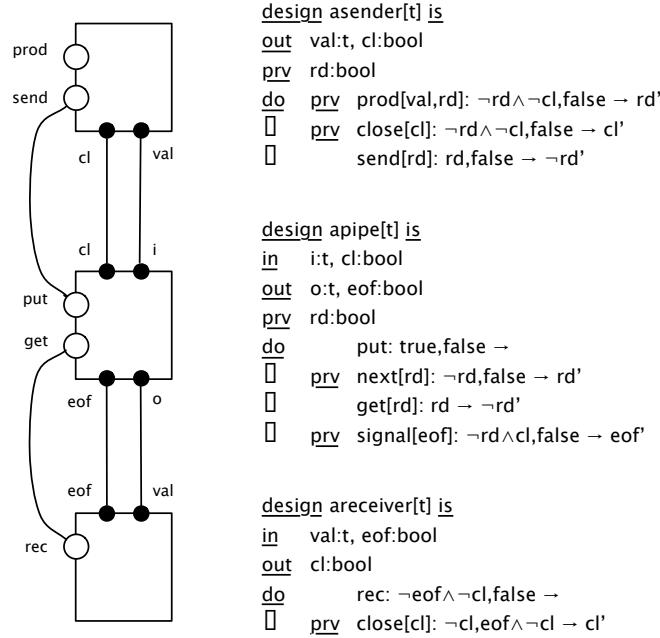


Fig. 1.18 An example of a connector (pipe) in *CommUnity*.

The roles and the glue of the connector are COMMUNITY ‘designs’, which provide specifications of component behaviour that can be observed over communication channels and actions. A COMMUNITY design consists of:

- A collection of channels, which can be output (written by the component, read by the environment), input (read by the component, written by the environment), or private (local to the component) — denoted O , I and Pc , respectively.
- A collection A of actions. Every action a is specified in terms of
 - The set W_a of output and private channels that the action can write into (its write-frame); for example, the action *prod* of *asender* can write into the channels *val* and *rd*, but not into *cl*, i.e., $W_{prod} = \{val, rd\}$.
 - A pair L_a, U_a of conditions — the lower (or safety) guard and the upper (or progress) guard — that specify a necessary (L_a) and a sufficient (U_a) condi-

tion for the action to be enabled, respectively; for example, action *close* of *areceiver* is only enabled when *cl* is false — $L_{close} \equiv \neg cl$ — and is enabled if *eof* is true and *cl* is false — $U_{close} \equiv eof \wedge \neg cl$. When the two guards are the same, we write only one condition as in the case of action *get* of *apipe*.

- A condition R_a that describes the effects of the action using primed channels to denote the value taken by channels after the action has taken; for example, action *signal* of *apipe* sets *cl* to true — $R_{signal} \equiv cl'$.

Actions can also be declared to be private, the set of which is denoted by Pa .

Each role is connected to the glue of the connector by a ‘cable’ that establishes input/output communication and synchronisation of non-private actions. Notice that all names are local, meaning that there are no implicit interconnections based on the fact that different designs happen to use the same names for channels or actions. Therefore, the cable that connects *apipe* and *areceiver* identifies *eof* of *apipe* with *eof* of *areceiver*, *o* with *val*, and *get* with *rec*. This means that *areceiver* reads *eof* from the channel *eof* of *apipe* and *val* from *o*, and that *apipe* and *areceiver* have to synchronise to execute the actions *get* and *rec*.

Designs can be abstract (as in the examples on Fig. 1.18) in the sense that they may not fully determine when actions are enabled or how they effect the data made available on channels. For example, action *prod* of *asender* has *val* in its write-frame but its effects on *val* are not specified. Making the upper (progress) guard false is another example of underspecification: the lower guard defines a necessary condition for the action to be enabled but no sufficient condition is given. Such abstract designs can be refined until they are fully specified, in which case the design is called a program. A program is, essentially, a collection of guarded commands. Non-private actions are reactive in the sense that they are executed together with the environment; private actions are active because their execution is only under the control of the component.

An example of a refinement of *apipe* is given in Fig 1.19. Formally, refinement consists of two mappings — one on channels, which is co-variant, and the other on actions, which is contra-variant. In the example, the refinement mapping introduces a new private channel — a queue — and is the identity on actions. The mappings need to preserve the nature of the channels (input, output, or private) and of actions (private or non-private). Private actions do not need to be refined but non-private ones do, in which case their effects need to be preserved (not weakened), lower guards can be weakened (but not strengthened) and upper can be strengthened (but not weakened), i.e., the interval defined by the two guards must be preserved or shrunk. For example, for all actions of *rpipe*, the lower and upper guards coincide. A given action can also be refined by a set of actions, each of which needs to satisfy the same constraints. Finally, new actions introduced during the refinement cannot include output channels of the abstract design in their write frames. A full formal definition can be found in [31].

COMMUNITY encapsulates one of the principles that have been put forward for modularising parallel and distributed programs — superposition or superimposition [14, 33, 45]. Indeed, programming in-the-many arose in the context of the advent of

```

design rpipe[t] is
in   i:t, cl:bool
out  o:t, eof:bool
prv  q:queue(t); rd:bool
do   put[q]:  $\neg \text{full}(q) \rightarrow q' = \text{enqueue}(i, q)$ 
□     prv next[rd, q]:  $\neg \text{empty}(q) \wedge \neg rd \rightarrow o' = \text{head}(q) \wedge q' = \text{tail}(q) \wedge rd'$ 
□     get[rd]:  $rd \rightarrow \neg rd'$ 
□     prv signal[eof]:  $\neg rd \wedge cl \wedge \text{empty}(q) \rightarrow \text{eof}'$ 

```

Fig. 1.19 A refinement of the design *apipe*.

concurrency and distribution, i.e., changes in the operating infrastructure that emphasise cooperation among independent processes. Programmers find concurrency ‘complicated’ and, therefore, a source of complexity in software design. For example, it seems fair to say that extensions of OOP with concurrency have failed to make a real impact in engineering or programming practice, one reason being that the abstractions available for OOP do not extend to concurrency in an intuitive way. In contrast, languages such as *Unity* [14], on which COMMUNITY is based, have put forward proper abstractions and modularisation techniques that follow on the principles of structured programming.

In Fig. 1.20 we present a COMMUNITY design for a luggage-delivery cart. The context is that of a simplified airport luggage delivery system in which carts move along a track and stop at designated locations for handling luggage. Locations in the track are modelled through natural numbers modulo the length of the circuit. Pieces of luggage are also modelled through natural numbers, zero being reserved to model the situation in which a cart is empty. According to the design *cart*, a cart is able to *move*, *load* and *unload*. It moves by incrementing *loc* while it has not reached its destination (the increment is left unspecified). The current destination is available in *dest* and is retrieved from the bag each time the cart stops to load, using a function *Dest* that we assume is provided as part of a data type (e.g., abstracting the scanning of a bar code on the luggage), or from the environment, when unloading, using the input channel *ndest*. Loading and unloading take place only when the cart has reached its destination.

```

design cart is
in   nbag, ndest:nat
out  loc, dest:nat
prv  bag:nat
do   move[loc]:  $loc \neq \text{dest} \rightarrow loc' > loc$ 
□     load[bag, dest]:  $loc = \text{dest} \wedge \text{bag} = 0 \rightarrow \text{bag}' = \text{nbag} \wedge \text{dest}' = \text{Dest}(\text{nbag})$ 
□     unload[bag, dest]:  $loc = \text{dest} \wedge \text{bag} \neq 0 \rightarrow \text{bag}' = 0 \wedge \text{dest}' = \text{Dest}(\text{nbag})$ 

```

Fig. 1.20 A COMMUNITY design of an airport luggage-delivery cart.

In Fig. 1.21 we present a superposition of *cart*: on the one hand, we distinguish between two modes of moving — slow and fast; on the other hand, we count the number of times the cart has docked since the last time the counter was reset. Notice that *controlled_cart* is not a refinement of *cart*: the actions *move_slow* and *move_fast* do not refine *move* because the enabling condition of *move* (which is fully specified) has changed. Like refinement, superposition consists of a co-variant mapping on channels and a contra-variant mapping on actions. However, unlike refinement, the upper guard of a superposed action cannot be weakened — this is because, in the superposed design, actions may occur in a more restricted context (that of a controller in the case at hand). In fact, superposition can be seen to capture a ‘component-of’ relationship, i.e., the way a component is part of a larger system. Another difference in relation to refinement is the fact that input channels may be mapped to output ones, again reflecting the fact that the part of the environment from which the input could be expected has now been identified. Other restrictions typical of superposition relations apply: new actions (such as *reset*) cannot include channels of the base design in their write-frames; however, superposed actions can extend their write-frames with new channels (e.g., *load* and *unload* now have *count* in their write-frames).

```

design controlled_cart is
in  nbag, ndest:nat
out count,loc,dest:nat
prv bag:nat
do  move_slow[loc]: 0 < |loc-dest| ≤ 2 → loc'=loc+1
    [] move_fast[loc]: |loc-dest| > 2 → loc'>loc
    [] load[bag,dest,count]: loc=dest ∧ bag=0 → bag'=nbag ∧ dest'=Dest(nbag) ∧ count'=count+1
    [] unload[bag,dest,count]: loc=dest ∧ bag≠0 → bag'=0 ∧ dest'=Dest(nbag) ∧ count'=count+1
    [] reset[count]: true,false → count'=0

```

Fig. 1.21 A superposition of the COMMUNITY design *cart* shown in Fig. 1.20.

COMMUNITY combines the modularisation principles of superposition with the externalisation of interactions promoted by coordination languages. That is, although superposition as illustrated in Fig. 1.21 allows designs to be extended (in a disciplined way), it does not externalise the mechanisms through which the extension is performed — the fact that the cart is subject to a speed controller and a counter at the docking stations. In COMMUNITY, this externalisation is supported by allowing designs to be interconnected with other designs. In Figs. 1.22 and 1.23, we show the designs of the speed controller and the counter, respectively.

Neither the speed controller nor the counter make reference to the cart (as with refinement, names of channels and actions are treated locally). Therefore, they can be reused in multiple contexts to build larger systems. For example, in Fig 1.24 we depict the architecture of the controlled cart as a system of three components inter-

```

design speed is
in  dest,loc:nat
do  slow[loc]: 0 < |loc-dest| ≤ 2 → loc'=loc+1
□    fast[loc]: |loc-dest| > 2 → loc' > loc

```

Fig. 1.22 A COMMUNITY design of a speed controller.

```

design counter is
out count:nat
do  inc[count]: true,false → count'=count+1
□    reset[count]: true,false → count'=0

```

Fig. 1.23 A COMMUNITY design of a counter.

connected through cables that, as in the case of connectors, establish input/output and action synchronisation.

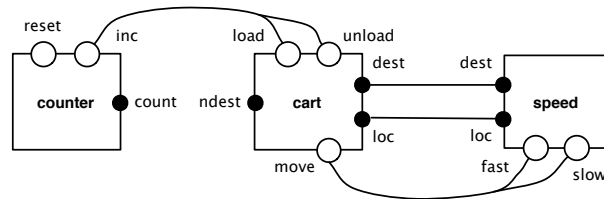


Fig. 1.24 The COMMUNITY architecture of the controlled cart.

The design *controlled_cart* depicted on Fig. 1.21 is the result of the composition of the components and connections depicted on Fig. 1.24. This operation of composition can be formalised in category theory [51], much in the same way as repeated inheritance (cf., 1.3.2) except that the morphisms in COMMUNITY capture superposition. The notion of refinement discussed above can also be formalised in category theory and refinement can be proved to be compositional with respect to composition — for example, one can refine *speed* by making precise the increment on the location; this refinement carries over to the controlled cart in the sense that the composition using the refined controller yields a refinement of the controlled cart. Full details of this categorical approach to software systems can be found in [21].

Extensions of COMMUNITY supported by the same categorical formalisations can be found in [52] for location-aware mobile systems (where location is defined as an independent architectural dimension) and in [26] for event-based architectures. Notions of higher-order architectural connectors were developed in [53] and dynamic reconfiguration was addressed in [69].

Finally, notice that, as most ADLs, COMMUNITY does not offer a notion of module in the sense of programming in-the-large, i.e., it does not provide coarser structures of designs (though the notion of higher-order architectural connectors presented in [53] goes in that direction by offering a mechanism for constructing connectors). Through channels and actions, COMMUNITY offers an explicit notion of interface through which designs can be connected, but neither channels nor actions can be seen as provided or required interfaces.

1.5 Programming in-the-universe

Given the tall order that the terms ‘small’, ‘large’ and ‘many’ have created, we were left with ‘universe’ to designate yet another face of complexity in software design, one that is more modern and sits at the core of the recent quotes with which we opened this chapter. The term ‘universe’ is also not too far from the designation ‘global (ubiquitous) computing’ that is often used for characterising the development of software applications that can run on ‘global computers’, i.e., “computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms” (see the Global Computing Initiative at cordis.europa.eu/ist/fet/gc.htm). It is in this context that we place service-oriented architectures (SOA) and service-oriented computing (SOC).

1.5.1 Services vs components

SOC is a new paradigm in which interactions are no longer based on the exchange of products with specific parties — clientship as in object-oriented programming — but on the provisioning of services by external providers that can be procured on the fly subject to a negotiation of service level agreements (SLAs). A question that, in this context, cannot be avoided, concerns the difference between component-based and service-oriented design. Indeed, the debate on CBD vs. SOC is still out there, which in our opinion reflects that there is something fundamental about SOC that is not yet clearly understood.

A basic difference concerns the run-time environment that supports both approaches. Component models rely on a homogeneous framework in which components can be plugged in and connected to each other. Services, like components, hide their implementations but, in addition to components, they do not reveal any implementation-platform or infrastructure requirements. Therefore, as put in [67], services are more self-contained than typical components. However, as a consequence, interactions with services are not as efficient as with objects or components, a point that is very nicely put in [65]: where, in OO, clientship operates through a direct mapping of method invocation to actual code and, in CBD, invocation is per-

formed via proxys in a slower way but still within a communication environment that is native to the specific component framework, SOC needs to bridge between different environment boundaries and rely on transport protocols that are not necessarily as performant.

Indeed, where we identify a real paradigm shift in SOC — one that justifies new abstractions and decomposition techniques — is in the fact that SOAs provide a layer of middleware in which the interaction between client and provider is mediated by a broker, which makes it possible to abstract from the identity of the server or of the broker when programming applications that need to rely on an external service. Design patterns or other component-oriented solutions can be used for mediating interactions but abstraction from identity is a key feature of SOC: as put in [20], services respond to the necessity for separating “need from the need-fulfilment mechanism”.⁹

Another difference between components and services, as we see it, can be explained in terms of two different notions of ‘composition’. In CBD, composition is integration-oriented — “the idea of component-based development is to industrialise the software development process by producing software applications by assembling prefabricated software components” [20]; “component-based software engineering is concerned with the rapid assembly of systems from components” [6]. The key aspect here is the idea of assembling systems from (reusable) components, which derives from the principle of divide-and-conquer.

Our basic stance is that what we are calling programming in-the-universe goes beyond this assembly view and abandons the idea that the purpose of programming or design is to build a software system that is going to be delivered to a customer; the way we see this new paradigm is that (smaller) applications are developed to run on global computers (like the Web) and respond to business needs by engaging, dynamically, with services and resources that are globally available at the time they are needed. Because those services may in turn require other services, each such application will create, as it executes, a system of sub-systems, each of which implements a session of one of the services that will have been procured.

For example, a typical business system may rely on an external service to supply goods; in order to take advantage of the best deal available at the time the goods are needed, the system may resort to different suppliers at different times. Each of those suppliers may in turn rely on services that they will need to procure. For instance, some suppliers may have their own delivery system but others may prefer to outsource the delivery of the goods; some delivery companies may have their own transport system but prefer to use an external company to provide the drivers; and so on. In summary, the structure of an application running on a global computer, understood as the components and connectors that determine its configuration, is intrinsically dynamic.

⁹ Notice that mechanisms that, as SOAP, support interconnections in SOAs, do not use URLs (universal resource locators) as identities: “there is no built-in guarantee that the URL will indeed refer back to an object actually live at the sending process, the sending machine, or even the sending site. There is also no guarantee that two successive resolution requests for the same URL will yield the same object” [67].

Therefore, the role of architecture in the construction of a service-oriented system needs to go beyond that of identifying, at design time, components that developers will need to implement or reuse. Because these activities are now performed by the SOA middleware, what is required from software architects is that they identify and model the high-level business activities and the dependencies that they have on external services to fulfil their goals. A consequence of this is that, whereas the notion of a ‘whole’ is intrinsic to CBD — whether in managing construction (through reuse) or change (through architecture) — SOC is not driven by the need to build or manage such a whole but to allow applications to take advantage of a (dynamic) universe of services. The purpose of services is not to support reuse in construction or manage change of a system as requirements evolve, but to allow applications to compute in an open-ended and evolving universe of resources. In this setting, there is much more scope for flexibility in the way business is supported than in a conventional component-based scenario: business processes need not be confined to fixed organisational contexts; they can be viewed in more global contexts as emerging from a varying collection of loosely coupled applications that can take advantage of the availability of services procured on the fly when they are needed.

1.5.2 Modules for service-oriented computing

A number of ‘standards’ have emerged in the last few years in the area of Web Services promoted by organisations such as OASIS¹⁰ and W3C¹¹. These include languages such as WSDL (an XML format for describing service interfaces), WS-BPEL (an XML-based programming language for business process orchestration based on web services) and WS-CDL (an XML-based language for describing choreographies, i.e., peer-to-peer collaborations of parties with a common business goal).

A number of research initiatives (among them the FET-GC2 integrated project SENSORIA [70]) have been proposing formal approaches that address different aspects of the paradigm independently of the specific languages that are available today for Web Services or Grid Computing. For example, recent proposals for service calculi (e.g., [9, 13, 49, 68]) address operational foundations of SOC (in the sense of how services compute) by providing a mathematical semantics for the mechanisms that support choreography or orchestration — sessions, message/event correlation, compensation, *inter alia*.

Whereas such calculi address the need for specialised language primitives for *programming* in this new paradigm, they are not abstract enough to address those aspects (both technical and methodological) that concern the way applications can be developed to provide business solutions independently of the languages in which services are programmed and, therefore, control complexity by raising the level of

¹⁰ www.oasis-open.org

¹¹ www.w3.org

abstraction and adopting coarser-grained decomposition techniques. The Open Service Oriented Architecture collaboration¹² has been proposing a number of specifications, namely the Service Component Architecture (SCA), that address this challenge:

SCA is a model designed for SOA, unlike existing systems that have been adapted to SOA. SCA enables encapsulating or adapting existing applications and data using an SOA abstraction. SCA builds on service encapsulation to take into account the unique needs associated with the assembly of networks of heterogeneous services. SCA provides the means to compose assets, which have been implemented using a variety of technologies using SOA. The SCA composition becomes a service, which can be accessed and reused in a uniform manner. In addition, the composite service itself can be composed with other services [...] SCA service components can be built with a variety of technologies such as EJBs, Spring beans and CORBA components, and with programming languages including Java, PHP and C++ [...] SCA components can also be connected by a variety of bindings such as WSDL/SOAP web services, Java™ Message Service (JMS) for message-oriented middleware systems and J2EE™ Connector Architecture (JCA) [61].

In Fig. 1.25 we present an example of an SCA component and, in Fig. 1.26, an example of an SCA composite (called ‘module’ in earlier versions). This composite has two components, each of which provides a service and has a reference to a service it depends on. The service provided by component *A* is made available for use by clients outside the composite. The service required by component *A* is provided by component *B*. The service required by component *B* exists outside the composite.

Although, through composites, SCA offers coarser primitives for decomposing and organising systems in logical groupings, it does not raise the level of abstraction. SCA addresses low-level design in the sense that it provides an assembly model and binding mechanisms for service components and clients programmed in specific languages, e.g., Java, C++, BPEL, or PHP. So far, SOC has been short of support for high-level modelling. Indeed, languages and models that have been proposed for service modelling and design (e.g., [11, 64]) do not address the higher level of abstraction that is associated with business solutions, in particular the key characteristic aspects of SOC that relate to the way those solutions are put together dynamically in reaction to the execution of business processes — run-time discovery, instantiation and binding of services.

The SENSORIA Reference Modelling Language (SRML) [30] started to be developed within the SENSORIA project as a prototype domain-specific language for modelling service-oriented systems at a high level of abstraction that is closer to business concerns. Although SRML is inspired by SCA, it focuses on providing a formal framework with a mathematical semantics for modelling and analysing the business logic of services independently not only of the hosting middleware but also of the languages in which the business logic is programmed.

In SRML, services are characterised by the conversations that they support and the properties of those conversations. In particular:

¹² www.osoa.org

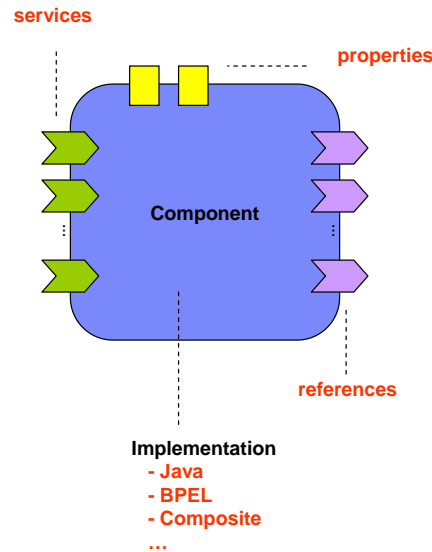


Fig. 1.25 An example of an SCA component. A component consists of a configured instance of an implementation, where an implementation is the piece of program code providing business functions. The business function is offered for use by other components as services. Implementations may depend on services provided by other components – these dependencies are called references. Implementations can have settable properties, which are data values which influence the operation of the business function. The component configures the implementation by providing values for the properties and by wiring the references to services provided by other components [61].

- messages are exchanged, asynchronously, through ‘wires’ and are typed by their business function (requests, commitments, cancelations, and so on);
- service interface behaviour is specified using message correlation patterns that are typical of business conversations; and
- the parties engaged in business applications need to follow pre-defined conversation protocols — requester and provider protocols.

On the other hand, the difference between SRML and more generic modelling languages is precisely in the fact that the mechanisms that, like message correlation, support these conversation protocols do not need to be modelled explicitly: they are assumed to be provided by the underlying SOA middleware. This is why SRML can be considered to be a domain-specific language: it frees the modeller from the need to specify aspects that should be left to lower levels of abstraction and concentrate instead on the business logic.

The design of composite services in SRML adopts the SCA assembly model according to which new services can be created by interconnecting a set of elementary components to a set of external services; the new service is provided through an interface to the resulting system. The business logic of such a service involves a number of interactions among those components and external services, but is independent of the internal configurations of the external services — the external ser-

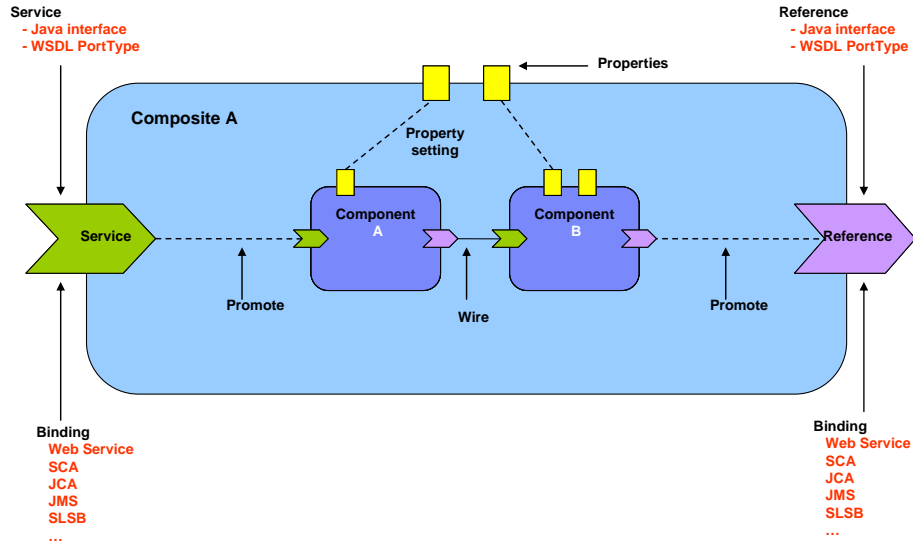


Fig. 1.26 An example of an SCA simple composite. Composites can contain components, services, references, property declarations, plus the wiring that describes the connections between these elements. Composites can group and link components built from different implementation technologies, allowing appropriate technologies to be used for each business task. [61].

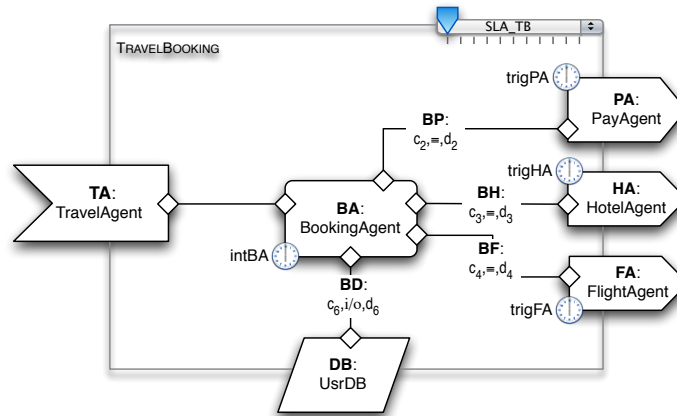




Fig. 1.27 The structure of the module *TravelBooking*. The service is assembled by connecting a component *BA* of type *BookingAgent* to three external service instances *PA*, *HA* and *FA* with interface types *PayAgent*, *HotelAgent* and *FlightAgent* (respectively) and the persistent component (a database of users) *DB* of type *UsrDB*. The wires that interconnect the several parties are *BP*, *BH*, *BF*, and *BD*. The interface through which service requesters interact with the *TravelBooking* service is *TA* of type *TravelAgent*. Internal configuration policies (indicated by the symbol ) are specified, which include the conditions that trigger the discovery of the external services. An external configuration policy (indicated by the symbol ) specifies the constraints according to which service-level agreements are negotiated (through constraint optimisation [8]).

vices need only be described by their interfaces. The actual external services are discovered at run time by matching these interfaces with those that are advertised by service providers (and optimising the satisfaction of service level agreement constraints).

The elementary unit for specifying service assembly and composition in SRML is the *service module* (or just *module* for short), which is the SRML equivalent to the SCA notion of composite. A module specifies how a set of internal components and external required services interact to provide the behaviour of a new service. Fig. 1.27 shows the structure of the module *TravelBooking*, which models a service that manages the booking of a flight, a hotel and the associated payment. The service is assembled by connecting an internal component *BA* (that orchestrates the service) to three external services (for booking a flight, booking a hotel and processing the payment) and the persistent component *DB* (a database of users). The difference between the three kinds of entities — internal components, external services and persistent components — is intrinsic to SOC: internal components are created each time the service is invoked and killed when the service terminates; external services are procured and bound to the other parties at run time; persistent components are part of the business environment in which the service operates — they are not created nor destroyed by the service, and they are not discovered but directly invoked as in component-based systems. By *TA* we denote the interface through which service requesters interact with *TravelBooking*. In SRML, interactions are peer-to-peer between pairs of entities connected through wires — *BP*, *BH*, *BF* and *BD* in the case at hand.

Each party (component or external service) is specified through a declaration of the interactions the party can be involved in and the properties that can be observed of these interactions during a session of the service. Wires are specified by the way they coordinate the interactions between the parties.

If the party is an internal component of the service (like *BA* in Fig. 1.27), its specification is an orchestration given in terms of state transitions — using the language of *business roles* [30]. An orchestration is defined independently of the language in which the component is programmed and the platform in which it is deployed; the actual component may be a BPEL process, a C++ or a Java program, or a wrapped up legacy system, *inter alia*. An orchestration is also independent of the parties that are interconnected with the component at run time; this is because the orchestration does not define invocations of operations provided by specific co-parties (components or external services); it simply defines the properties of the interactions in which the component can participate.

If the party is an external service, the specification is what we call a *requires-interface* and consists of a set of temporal properties that correlate the interactions in which the service can engage with its client. The language of *business protocols* [30] is used for specifying the behaviour required of external services not in terms of their internal workflow but of the properties that characterise the interactions in which the service can engage with its client, i.e., their interface behaviour. Fig. 1.28 shows the specification of the business protocol that the *HotelAgent* service is expected to follow.

```

BUSINESS PROTOCOL HotelAgent is


---


INTERACTIONS
  r&s lockHotel
    Ⓐ checkin,checkout:date,
      name:usrData
    ☒ hconf:hcode
BEHAVIOUR
  initiallyEnabled lockHotelⒶ?
  lockHotel✓? enables
    lockHotel‡? until date(time)≥lockHotel.checkin

```

Fig. 1.28 The specification of the service interface of a *HotelAgent* written in the language of business protocols. A *HotelAgent* can be involved in one interaction named *lockHotel* that models the booking of a room in a hotel. Some properties of this interaction are specified: a booking request can be made once the service is instantiated and a booking can be revoked up until the check-in date. The specification language makes use of the events associated with the declared interactions: the initiation event (Ⓐ), the reply event (☒), the commit event (✓), the cancellation event (✕) and the revoke event (‡).

```

BUSINESS PROTOCOL TravelAgent is


---


INTERACTIONS
  r&s login
    Ⓐ usr:usrName, pwd:password
  r&s bookTrip
    Ⓐ from,to:airport,
      out,in:date
    ☒ fconf:fcode,
      hconf:hcode,
      amount:moneyValue
  snd payNotify
    Ⓐ status:bool
  snd refund
    Ⓐ amount:moneyValue
BEHAVIOUR
  initiallyEnabled loginⒶ?
  login☒! ^ login.reply enables
    bookTripⒶ? until time≥login.useBy
  bookTrip✓? ensures payNotifyⒶ!
  payNotifyⒶ! ^ payNotify.status enables
    bookTrip‡? until date(time)≥dayBefore(bookTrip.out)
  bookTrip‡? ensures refundⒶ!

```

Fig. 1.29 The specification of the provides-interface of the service module *TravelBooking* written in the language of business protocols. The service can be involved in four interactions (*login*, *bookTrip*, *payNotify* and *refund*) that model the login into the system, the booking of a trip, the sending of a receipt and refunding the client of the service (in case a booking is returned). Five properties are specified for these interactions.

The specification of the interactions provided by the module (at its interface level) is what we call the *provides-interface*, which also uses the language of business protocols. Fig. 1.29 shows the specification of the business protocol that the composite service declares to follow, i.e., the service that is offered by the service module *TravelBooking*. A service module is said to be correct if the properties offered through the provides-interface can be guaranteed by the (distributed) orchestration performed by components that implement the business roles assuming that they are interconnected to external services that ensure the properties specified in the requires-interfaces.

Persistent components can interact with the other parties synchronously, i.e., they can block while waiting for a reply. The properties of synchronous interactions are in the style of pre/post condition specification of methods as discussed in Section 1.3.2.

The specifications of the wires consist of *connectors* (in the sense of Section 1.4) that are responsible for binding and coordinating the interactions that are declared locally in the specifications of the two parties that each wire connects. In a sense, SRML modules are a way of organising interconnected systems in the sense of programming in-the-many, i.e., of offering coarser-grained abstractions (in the sense of programming in-the-large) that can respond to the need for addressing the complexity that arises from the number of interactions involved in the distributed systems that, today, operate at the larger scale of global computers like the Web. This matches the view that services offer a layer of organisation that can be superposed over a component infrastructure (what is sometimes referred to as a service overlay), i.e., that services are, at a certain level of abstraction, a way of using software components and not so much a way of constructing software. We have explored this view in [28] by proposing a formalisation of services as interfaces for an algebra of asynchronous components understood as configurations of components and connectors.

Through this notion of service-overlay, such configurations of components and connectors expose conversational, stateful interfaces through which they can discover and bind, on the fly, to external services or expose services that can be discovered by business applications. That is, services offer an abstraction for coping with the run-time complexity of evolving configurations. A mathematical semantics for this dynamic process of discovery, binding and reconfiguration has been defined in [29], again using the tools of category theory: modules are used for typing configurations understood as graphs; such graphs evolve as the activities that they implement discover and bind to required services.

An example of this process is shown in the Appendix. Fig. 1.31 depicts a run-time configuration (graph) where a number of components execute business roles and interact via wires with other components. The sub-configuration encircled corresponds to a user-interface *AUI* interacting with a component *ant*. This sub-configuration is typed by the activity module *A_ANTO* (an activity module is similar to a service module but offering a user-interface instead of a service-interface). Because the activity module has a requires-interface, the sub-configuration will change if the trigger associated with *TA* occurs. This activity module can bind to the service

module *TravelBooking* (depicted in Fig. 1.27) by matching its requires-interface with the provides-interface of *TravelBooking* and resolving the SLA constraints of both modules (see Fig. 1.32). Therefore, if the trigger happens and *TravelBooking* is selected, the configuration will evolve to the one depicted on Fig. 1.33: an instance *AntBA* of *BookingAgent* is added to the configuration and wired to *Ant* and *DB* (no new instances of persistent components are created). Notice that the type of the sub-configuration has changed: it now consists of the composition of *A_ANTO* and *TravelBooking*. Because the new type has several requires-interfaces, the configuration will again change when their triggers occur.

Typing configurations with activity modules is a form of reflection, a technique that has been explored at the level of middleware to account for the evolution of systems [46]. In summary, we can see SOC as providing a layer of abstraction in which the dynamic reconfiguration of systems can be understood in terms of the business functions that they implement and the dependencies that those functions have on external services. This, we claim, is another step towards coping with the complexity of the systems that operate in the global infrastructures of today.

1.6 Concluding remarks

This chapter is an attempt to make sense of the persistent claim that, in spite of the advances that we make on the way we program or engineer software systems, software is haunted by the beast of complexity and doomed to live in a permanent crisis. Given the complexity of the task (pun intended), we resorted to abstraction — we did our best to distill what seemed to us to have been key contributions to the handling of complexity — and decomposition by organising these contributions in four kinds of ‘programming’: in-the-small (structured programming), in-the-large (modules, objects, and components), in-the-many (connectors and software architectures), and in-the-universe (services). The fact that, to a large extent, these forms of programming are organised chronologically, is not an accident: it reflects the fact that, as progress has been made in computer science and software engineering, new kinds of complexity have arisen. We started by having to cope with the complexity of controlling execution, then the size of programs, then change and, more recently, ‘globalisation’.

What remains constant in this process is the way we attempt to address complexity: abstraction and decomposition. This is why we insisted in imposing some degree of uniformity in terminology and notation, highlighting the fact that notions of module, interface, component, or architecture have appeared in different guises to support different abstraction or decomposition techniques. Although we chose not to go too deep into mathematical concepts and techniques, there is also some degree of uniformity (or universality) in the way they support notions of refinement or composition — for example, through the use of categorical methods — even if they are defined over different notions of specification — for example, pre/post-conditions for OO/CBD and temporal logic for SOC.

As could be expected, we had to use a rather broad brush when painting the landscape and, therefore, we were not exhaustive and left out many other faces of complexity. For example, as put in the 27/01/2009 edition of the Financial Times, cloud computing is, today, contributing to equally ‘complex’ aspects such as management or maintenance:

Cloud computing doesn’t work because it’s simpler than client-server or mainframe computing. It works because we shift the additional complexity to a place where it can be managed more effectively. Companies such as Amazon and Google are simply a lot better at managing servers and operating systems than most other organisations could ever hope to be. By letting Google manage this complexity, an enterprise can then focus more of its own resources on growth and innovation within its core business.

To us, this quote nails down quite accurately the process through which complexity has been handled during the last fifty years or so: “we shift the additional complexity to a place where it can be managed more effectively”. That is, we address complexity by making the infrastructure (or middleware) more ‘clever’ or by building tools that translate between levels of abstraction (e.g., through compilation or model-driven development techniques). For example, the move from objects to components to services is essentially the result of devising ways of handling interactions (or clientship): from direct invocation of code within a process (OO), to mediation via proxys across processes but within a single component framework (CBD), and across frameworks through brokers and transport protocols (SOA) [65].

Unfortunately (or inevitably), progress on the side of science and methodology has been slower, meaning that abstractions have not always been forthcoming as quickly as they would be needed to take advantage of new layers of infrastructure, which justifies that new levels of complexity arise for humans (programmers, designers, or analysts) when faced with new technology: notions of module tend to come when the need arises for managing the complexity of developing software over new computation or communication infrastructures. The answer to the mystery of why, in spite of all these advances, software seems to live in a permanent crisis, is that the beast of complexity keeps changing its form and we, scientists, do take our time to understand the nature of each new form of complexity and come up with right abstractions. In other words, like Paddington Bear, we take our time to abstract business functions from the handling of bank notes (with or without marmalade).

Acknowledgements Section 1.4 contains material extracted from papers co-authored with Antónia Lopes and Michel Wermelinger, and Section 1.5 from papers co-authored with Antónia Lopes, Laura Bocchi and João Abreu. I would like to thank them all also Mike Hinchey for giving me the opportunity (and encouraging me) to contribute this chapter.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)

2. Allen, R., Garland, D.: A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.* **6**(3), 213–249 (1998)
3. Arbab, F.: The IWIM model for coordination of concurrent activities. In: P. Ciancarini, C. Hankin (eds.) *COORDINATION, LNCS*, vol. 1061, pp. 34–56. Springer (1996)
4. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* **14**(3), 329–366 (2004)
5. Arbab, F., Herman, I., Spilling, P.: An overview of manifold and its implementation. *Concurrency - Practice and Experience* **5**(1), 23–70 (1993)
6. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008 ESC-TR-2000-007 (2000)
7. Banâtre, J.P., Métayer, D.L.: Programming by multiset transformation. *Commun. ACM* **36**(1), 98–111 (1993)
8. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. *J. ACM* **44**(2), 201–236 (1997)
9. Boreale, M., et al.: SCC: A service centered calculus. In: M. Bravetti, M. Núñez, G. Zavattaro (eds.) *WS-FM, LNCS*, vol. 4184, pp. 38–57. Springer (2006)
10. Brown, A.W.: Large-Scale, Component Based Development. Prentice Hall PTR, Upper Saddle River, NJ, USA (2000)
11. Broy, M., Krüger, I.H., Meisinger, M.: A formal model of services. *ACM Trans. Softw. Eng. Methodol.* **16**(1) (2007)
12. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: *IJCAI*, pp. 1045–1058 (1977)
13. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola [16], pp. 2–17
14. Chandy, K.M., Misra, J.: *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1988)
15. Cheesman, J., Daniels, J.: *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
16. De Nicola, R. (ed.): *Programming Languages and Systems, LNCS*, vol. 4421. Springer (2007)
17. Denning, P.J.: The field of programmers myth. *Commun. ACM* **47**(7), 15–20 (2004)
18. DeRemer, F., Kron, H.H.: Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Software Eng.* **2**(2), 80–86 (1976)
19. Dijkstra, E.W.: *A Discipline of Programming*, 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1976)
20. Elfatraty, A.: Dealing with change: components versus services. *Commun. ACM* **50**(8), 35–39 (2007)
21. Fiadeiro, J.L.: *Categories for Software Engineering*. Springer (2004)
22. Fiadeiro, J.L.: Software services: Scientific challenge or industrial hype? In: Z. Liu, K. Araki (eds.) *ICTAC, LNCS*, vol. 3407, pp. 1–13. Springer (2004)
23. Fiadeiro, J.L.: Physiological vs. social complexity in software design. In: *ICECCS*, p. 3. IEEE Computer Society (2006)
24. Fiadeiro, J.L.: Designing for software’s social complexity. *IEEE Computer* **40**(1), 34–39 (2007)
25. Fiadeiro, J.L.: On the challenge of engineering socio-technical systems. In: M. Wirsing, J.P. Banâtre, M.M. Hözl, A. Rauschmayer (eds.) *Software-Intensive Systems and New Computing Paradigms, LNCS*, vol. 5380, pp. 80–91. Springer (2008)
26. Fiadeiro, J.L., Lopes, A.: An algebraic semantics of event-based architectures. *Mathematical Structures in Computer Science* **17**(5), 1029–1073 (2007)
27. Fiadeiro, J.L., Lopes, A.: A model for dynamic reconfiguration in service-oriented architectures. In: M.A. Babar, I. Gorton (eds.) *ECSA, LNCS*, vol. 6285, pp. 70–85. Springer (2010)
28. Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. In: D. Gianakopoulou, F. Orejas (eds.) *FASE, LNCS*, vol. 6603, pp. 18–33. Springer (2011)
29. Fiadeiro, J.L., Lopes, A., Bocchi, L.: An abstract model of service discovery and binding. *Formal Asp. Comput.* (to appear)

30. Fiadeiro, J.L., Lopes, A., Bocchi, L., Abreu, J.: The *SENSORIA* reference modelling language. In: Wirsing and Hölzl [70], pp. 61–114
31. Fiadeiro, J.L., Lopes, A., Wermelinger, M.: A mathematical semantics for architectural connectors. In: R.C. Backhouse, J. Gibbons (eds.) *Generic Programming, LNCS*, vol. 2793, pp. 178–221. Springer (2003)
32. Fingar, P.: Component-based frameworks for e-commerce. *Commun. ACM* **43**(10), 61–67 (2000)
33. Francez, N., Forman, I.R.: Superimposition for interacting processes. In: J.C.M. Baeten, J.W. Klop (eds.) *CONCUR, LNCS*, vol. 458, pp. 230–245. Springer (1990)
34. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
35. Gelernter, D.: Generative communication in linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985)
36. Gelernter, D., Carriero, N.: Coordination languages and their significance. *Commun. ACM* **35**(2), 96–107 (1992)
37. Goguen, J.A.: Categorical foundations for general systems theory. In: F. Pichler, R. Trappl (eds.) *Advances in Cybernetics and Systems Research*, pp. 121–130. Transcripta Books (1973)
38. Goguen, J.A.: Reusing and interconnecting software components. *IEEE Computer* **19**(2), 16–28 (1986)
39. Goguen, J.A.: A categorical manifesto. *Mathematical Structures in Computer Science* **1**(1), 49–67 (1991)
40. Gries, D.: *The Science of Programming*, 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1981)
41. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**, 576–580 (1969)
42. Hoare, C.A.R.: *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
43. Jackson, M.A.: *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA (1975)
44. Jones, C.B.: *Systematic software development using VDM* (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
45. Katz, S.: A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.* **15**(2), 337–356 (1993)
46. Kon, F., Costa, F.M., Blair, G.S., Campbell, R.H.: The case for reflective middleware. *Commun. ACM* **45**(6), 33–38 (2002)
47. Kramer, J.: Exoskeletal software. In: *ICSE*, p. 366 (1994)
48. Kramer, J.: Is abstraction the key to computing? *Commun. ACM* **50**(4), 36–42 (2007)
49. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola [16], pp. 33–47
50. Liskov, B., Zilles, S.: Programming with abstract data types. In: *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pp. 50–59. ACM, New York, NY, USA (1974)
51. Lopes, A., Fiadeiro, J.L.: Superposition: composition vs refinement of non-deterministic, action-based systems. *Formal Asp. Comput.* **16**(1), 5–18 (2004)
52. Lopes, A., Fiadeiro, J.L.: Adding mobility to software architectures. *Sci. Comput. Program.* **61**(2), 114–135 (2006)
53. Lopes, A., Wermelinger, M., Fiadeiro, J.L.: High-order architectural connectors. *ACM Trans. Softw. Eng. Methodol.* **12**(1), 64–104 (2003)
54. Medvidović, N., Mikic-Rakic, M.: *Programming-in-the-many: A software engineering paradigm for the 21st century*
55. Medvidović, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.* **26**(1), 70–93 (2000)
56. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: *ICSE*, pp. 178–187 (2000)
57. Meyer, B.: *Object-oriented software construction* (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)

58. Mikic-Rakic, M., Medvidović, N.: Adaptable architectural middleware for programming-in-the-small-and-many. In: M. Endler, D.C. Schmidt (eds.) *Middleware, LNCS*, vol. 2672, pp. 162–181. Springer (2003)
59. Mikic-Rakic, M., Medvidović, N.: A connector-aware middleware for distributed deployment and mobility. In: *ICDCS Workshops*, pp. 388–393. IEEE Computer Society (2003)
60. Morgan, C.: *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
61. OSOA: Service component architecture (2007). Version 1.00
62. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972)
63. Prieto-Diaz, R., Neighbors, J.M.: Module interconnection languages. *J. Syst. Softw.* **6**, 307–334 (1986)
64. Reisig, W.: Modeling- and analysis techniques for web services and business processes. In: M. Steffen, G. Zavattaro (eds.) *FMOODS, LNCS*, vol. 3535, pp. 243–258. Springer (2005)
65. Sessions, R.: Fuzzy boundaries: Objects, components, and web services. *Queue* **2**, 40–47 (2004)
66. Shaw, M., Garlan, D.: *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
67. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
68. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service-oriented computation. In: S. Drossopoulou (ed.) *ESOP, LNCS*, vol. 4960, pp. 269–283. Springer (2008)
69. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.* **44**(2), 133–155 (2002)
70. Wirsing, M., Hölzl (Eds), M.: *Rigorous Software Engineering for Service-Oriented Systems, LNCS*, vol. 6582. Springer (2011)
71. Wirth, N.: *Programming in MODULA-2* (3rd corrected ed.). Springer-Verlag New York, Inc., New York, NY, USA (1985)
72. Woodcock, J., Davies, J.: *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)

Appendix

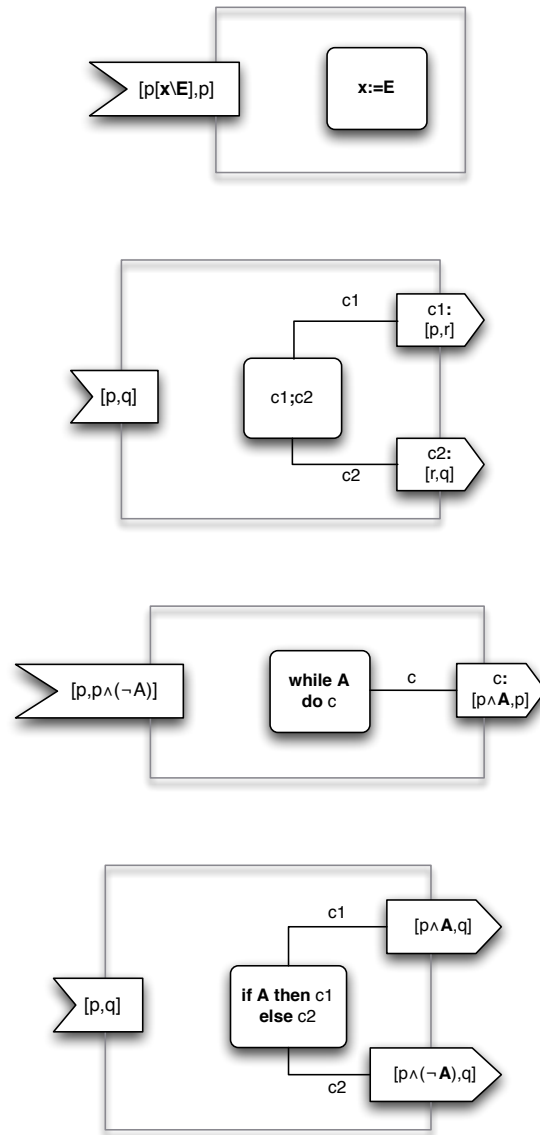


Fig. 1.30 Module schemas for assignment, sequence, iteration, and selection.

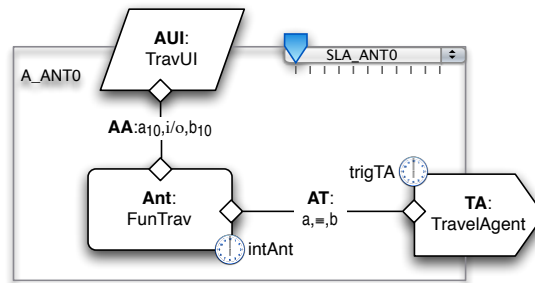
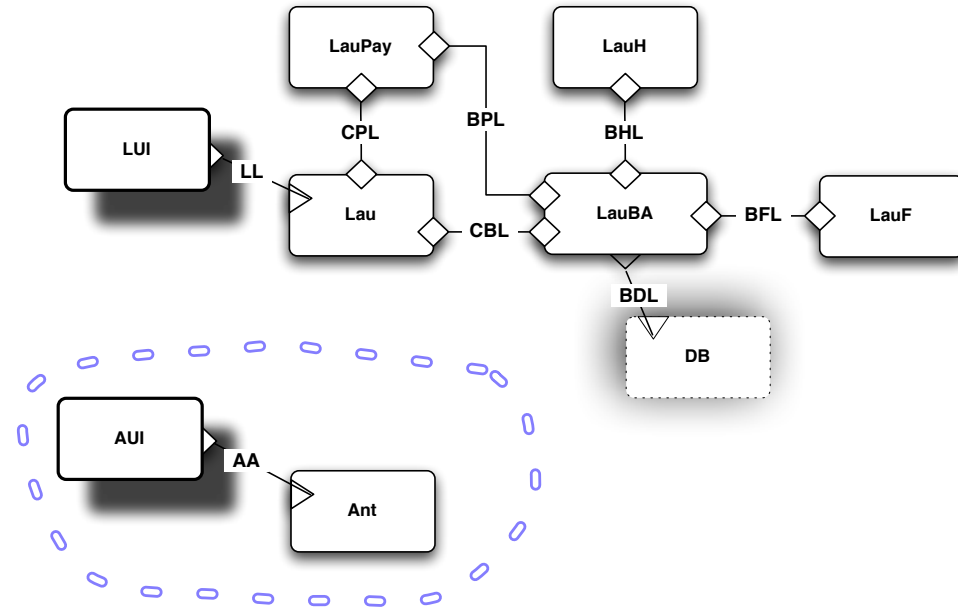


Fig. 1.31 A configuration, a sub-configuration of which is typed by an activity module.

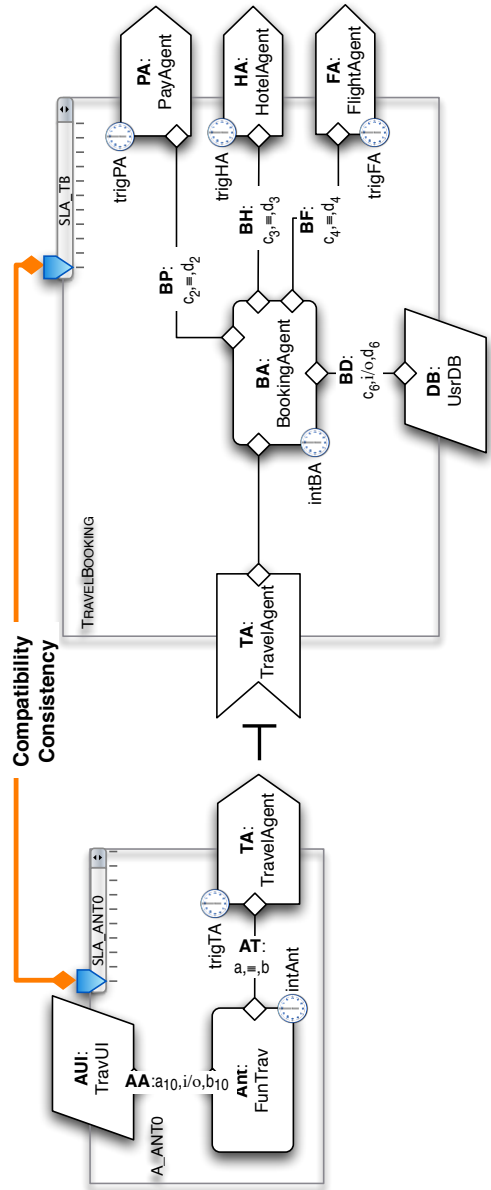


Fig. 1.32 Matching the activity module of Fig. 1.31 with the service module *TravelBooking*.

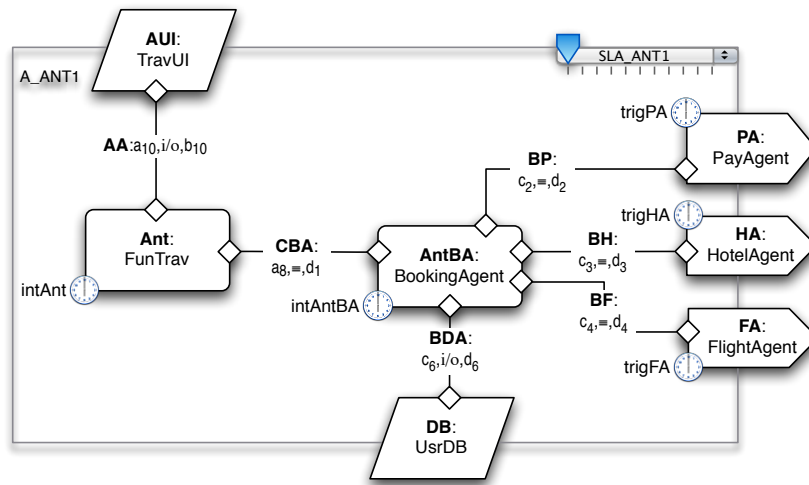
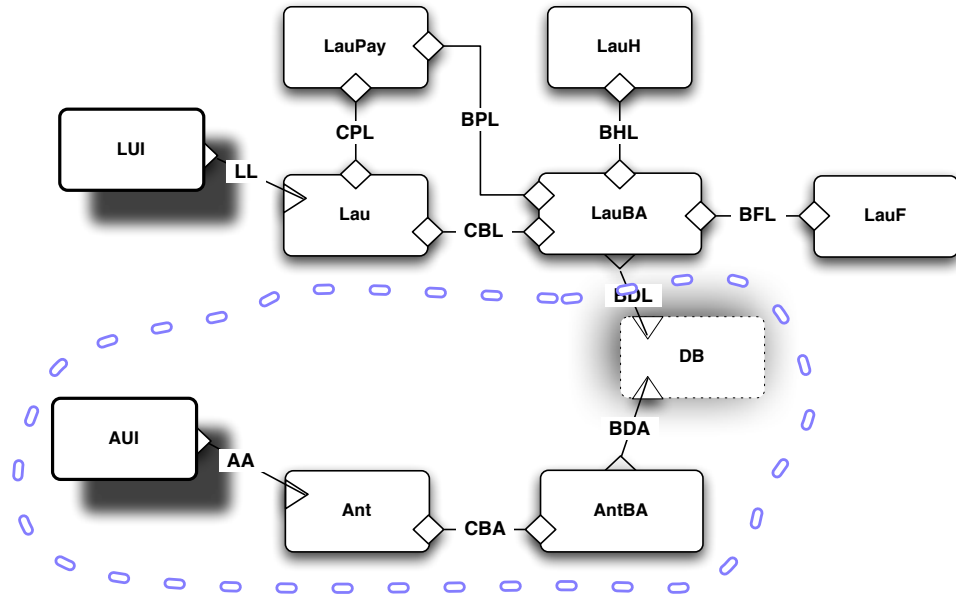


Fig. 1.33 The reconfiguration resulting from the binding in Fig. 1.32.

Index

- SENSORIA, 31
 - SENSORIA Reference Modelling Language (SRML), 32, 33, 35, 37
- 1968 NATO Conference on Software Engineering, 5
- Architectural Description Languages
 - COMMUNITY, 24–29
 - Reo, 23
 - Wright, 24
- Bear, Paddington, 2, 3, 39
- Complexity
 - and Cloud Computing, 39
- Complexity Crisis, 4
- Composition, 7, 10, 11, 18–21, 23, 28, 30, 35, 38
- Coordination Languages
 - Linda, 23
 - Manifold, 23, 24
- Dallas International Airport Baggage Handling System
 - Software Problems, 4
- Denning, Peter, 3
- DeRemer, Frank, 4, 5
- Dijkstra, Edsger W., 6
- Exports Interface, 14, 21
- Feldman, Stuart, 1
- Goguen, Joseph A., 19
- Hoare calculus, 8
- Imports Interface, 14, 21
- Inheritance, 17–19, 28
- Jackson Structured Programming (JSP), 6
- Kron, Hans H., 4, 5
- Kyffin, Steven, 1
- Marmalade Stains
 - Bank Note, 2
 - Source Code, 3
- McIlroy, Douglas, 5
- Medvidović, Nenad, 5, 21
- Meyer, Bertrand, 16
- Parnas, David Lorge, 12
- Pre/Post Conditions, 7, 8, 14, 18–21, 37, 38
- Programming Languages
 - Eiffel, 19
 - Modula-2, 14
 - Unity, 26
- Provided Interface, *see* Provides Interface
- Provides Interface, 8, 21, 37, 38
- Redshaw, Toby, 4
- Refinement, 8, 15, 18–20, 25–28, 38
- Required Interface, *see* Requires Interface
- Requires Interface, 8, 9, 21, 37
- Service Component Architecture (SCA), 32–35
- Software Crisis, 1, 2, 5, 20
- Superposition, 25, 27, 28
- Szyperski, Clemens Alden, 20
- Uses Interface, 9