PART I

SOFTWARE ARCHITECTURES

1

EVOLUTION OF SOFTWARE COMPOSITION MECHANISMS: A SURVEY

Carlo Ghezzi and Filippo Pacifici

1.1. INTRODUCTION

In engineering, developing a system always implies designing its structure, by providing a proper decomposition into separate parts and relationships among them. This approach allows engineers to address the complexity of a system by applying a divide-and-conquer approach—that is, by recursively focusing on limited subsystems and on the interaction between them.

In this setting, it is possible to identify two phases: the definition of the behavior of single parts and the composition of parts. Composition is the way the whole system is constructed by binding components together. In this phase, engineers focus on how the relations among parts are established, rather than on the specific internal structure or behavior of the components.

Software systems evolved from small and monolithic systems to increasingly large and distributed systems. Accordingly, composition gained greater importance because soon software engineers realized that software development could not be considered as a one-person, monolithic task. Instead, software systems had to be described as complex structures, obtained through careful application of specific composition mechanisms.

Software composition can be analyzed along two directions: process and product architecture. From a process viewpoint, composition has to do with the way software

Emerging Methods, Technologies, and Process Management in Software Engineering. Edited by De Lucia, Ferrucci, Tortora, and Tucci

development is structured in terms of work units and organizations. From a product architecture viewpoint, it refers to the way products are structured in terms of components and their connections.

This chapter focuses on the evolution of software composition principles and mechanisms over the past decades. The evolution has been consistently in the direction of increasing flexibility: from static to dynamic and from centralized to decentralized software compositions. For example, at the code level, there has been an evolution from monolithic program structures to functional and then object-oriented program decompositions. At a more abstract and higher level, software architectures evolved from tightly coupled and structured composites, as in the case of multi-tier architectures, to decentralized, peer-to-peer composites. In a similar way, software processes evolved from fixed, sequential, and monolithic to agile, iterative, and decentralized workflows (9).

This chapter surveys several concepts from software engineering and puts them in a historical perspective. It can be considered as a tutorial on software composition.

This chapter is organized as follows. Section 1.2 provides some basic foundational concepts for software composition. Sections 1.3 and 1.4 set the stage by discussing when the problem of software composition originated and how it was initially tackled. Section 1.5 focuses on some milestone contributions to concepts, methods, and techniques supporting design of evolvable software. Section 1.6 is about the current stage. It argues that our main challenge is to write evolvable software that lives in an open world. It identifies where the main challenges are and outlines some possible research directions.

1.2. BASIC CONCEPTS

The concept of *binding* (21) is a recurring key to understanding software composition at the code and architecture level. Binding is the establishment of a relationship among elements that form a *structure*. *Binding time* is the time at which such a relationship is established. For example, these concepts may be used to describe the semantic properties of programming languages, which may differ in the policy adopted to bind variables to their type, subclasses to their superclasses, function invocations to function definitions, or executable code to virtual machines. At an architectural level, a server may be bound to several clients. Typical binding times are: *design time*, *translation time*, *deployment time*, and *run time*. A design-time binding is, for example, an association that links a class in a UML class diagram to its parent class. A translation-time binding is established when the source description is processed.[1] For example, there are languages that bind variables to their type and function calls to function definitions at translation time. Object-oriented languages perform the binding at run time. Likewise, there are cases where the binding between a certain executable program unit and the node of distributed system on which it is

[1]We use this term instead of the most common, but more restricted, term "compile-time" to also include other languages processing aspects like link/load and preprocessing.

executed is set at deployment time; in other cases, it is set at run time to support system reconfiguration.

Another important orthogonal concept is *binding stability*. A binding is *static* if it cannot vary after being established; otherwise, it is *dynamic*. In most cases, design-time, translation-time, and deployment-time bindings are static and run-time bindings are dynamic, although there are exceptions. The key difference is thus between pre-run-time binding and run-time binding: Run-time binding adds much flexibility to the system with respect to pre-run-time binding because of the change policies it can be based upon. In the case of dynamic binding, one may further distinguish between *explicit* and *automatic* binding. In the former case, for example, the user asks the system to change a binding and specifies the new component that has to be bound, while in the second case the system has some autonomy in deciding when the binding has to be changed and is provided with mechanisms to find the new element to be bound. In practice, there is a spectrum of possible solutions that stay in between fully dynamic run-time binding and static pre-run-time binding.

These concepts may also be applied at process level, to describe or specify how software development is organized. For example, one may bind people to specific roles (such as *tester*) in an almost static way, or this can change as process development proceeds. Similarly, a development phase (such as *detail design* may be required to be completed before the next phase (*coding*) can be started. In this case, the binding between a phase and the next is established through a predefined, sequential control flow. Alternatively, one may organize the two phases as finer-grain concurrent steps.

Binding characterization is useful to understand the evolution of the software, both in the way engineers develop software and in the way they structure the architecture of applications. This evolution brought software systems from static, centralized, and monolithic structures, where bindings were defined at design time and frozen during the whole system life cycle, to modular, dynamic, and decentralized designs and design process, where bindings define loosely coupled structures, can change without stopping the system, and can cross interorganizational borders.

It is worthwhile to try to identify the ultimate sources and driving forces of this evolution. Advances in software technology are of course an endogenous force. Indeed, modern languages and design methods provide support for dynamic bindings and continuous change. However, the demand for dynamism and evolution mainly originates in the environment in which software systems live. The boundaries between the software system and the external environment are subject to continuous change. We have used the term *open-world software* to describe this concept (7). In the early days of software development, a closed-world assumption was made. It assumed that the requirements specifying the border between environment and systems to develop were sufficiently stable and they could be fully elicited in advance, before starting design and development. As we will illustrate in the rest of this chapter, this assumption proved to be wrong. Software increasingly lives in an open world, where the boundaries with the real world change continuously, even as the software is executing. This demands change management policies that can support dynamic evolution through run-time bindings.

1.3. EARLY DAYS

Up to the 1960s,[2] software development was mainly a single-person task; the problem to be solved was well understood, and often the developer was also the expected user of the resulting application. Thus, often there was no distinction between developers and users. The problems to solve were mainly of a mathematical nature, and the programmers had the needed mathematical background to understand the problem being solved. Programs were relatively simple, compared to today's standards, and were developed by using low-level languages, without any tool support. As for process, engineers did not follow any systematic development model, but rather proceeded through continuous *code and fix*—that is, an iteration of the activities of writing code and fixing it to eliminate errors.

Soon this approach proved to be inadequate, due to (a) the growing complexity of software systems and (b) the need for applying computing not only to scientific problems but also to other domains, like business administration and process control, where problems were less understood.

At the beginning of the 1970s, after software engineering was recognized as a crucial scientific topic (8, 31), the *waterfall development process* was developed (38) as a reference process model for software engineers. It was an attempt to bring discipline and predictability to software development. It specified a sequential, phased workflow where a requirements elicitation phase is followed by a design phase, followed by a coding phase, and finally by a validation phase. The proposed process model was fixed and static: The decomposition into phases and the binding between one phase and the next were precisely defined. The conditions for exiting a phase and entering the next were also precisely formulated, with the goal of making any rework on previously completed phases unnecessary, and even forbidden. The underlying motivation was that rework—that is, later changes in the software—was perceived as detrimental to quality, responsible for high development costs, and responsible for late time to market. This led to investing efforts in the elicitation, specification, and analysis of the needed requirements, which had to be frozen before development of the entire application could start.

This approach made very strong implicit assumptions about the world in which the software was going to be embedded. This world was assumed to be static; that is, the system's requirements were assumed not to change. The problem was just to elicit the requirements right, so that they could be fully specified prior to development. Completely and precisely specified requirements would ensure that that no need for software changes would arise during development and after delivery.

Another assumption was that the organizations in which the software had to be used had a monolithic structure. Companies were isolated entities with centralized operations and management. Communication and coordination among different companies were limited and not crucial for their business. Centralized solutions were therefore natural in this setting.

[2]For a short history of software engineering, the reader can refer to reference 22.

Based on these assumptions and on the software technology available at the time, engineers developed monolithic software systems to be run on mainframes. Even though the size of the application required several people to share the development effort, little attention was put on the modular structure of the application. Separately developed functions were in any case bound together at translation time, and no features were available to support change in the running application. To make a change, the system had to be put in offline mode, the source code had to be modified, each module had to be recompiled, rebound, and redeployed. Because little or no attention was put on the modular structure, changes were difficult to make and their effect was often unpredictable.

1.4. ACHIEVING FLEXIBILITY

The need to accommodate change and software evolution asked for more flexible approaches. It became clear that in most practical circumstances, system requirements cannot be fully gathered upfront, because sometimes customers do not know in advance what they actually require. If they do, requirements are then likely to change, maybe before an application that satisfies their initial specification has been developed completely. This of course questions the implicit assumptions on which the waterfall model is based. Moreover, software architectures turned out to be difficult to modify, because the tight coupling among the various parts made it impossible to isolate the effect of changes to restricted portions of the application. Seemingly minor changes could have a global effect that disrupted the integrity of the whole system.

Finally, the evolution of business organizations also asked for more flexibility. The structure of monolithic and centralized organizations evolved toward more dynamic aggregates of autonomous and decentralized units. Software development processes also changed, because off-the-shelf components and frameworks became progressively available. Software development became distributed and decentralized over different organizations: component developers and system integrators.

In the sequel we discuss the major steps and the landmark contributions on the road that led to increasing levels of flexibility.

1.4.1. Design for Change

A first conceptual contribution toward supporting evolution of software systems was the principle of *design for change* and the definition of techniques supporting it. Parnas (35–37) suggested that software designers should pay much attention in the requirements phase to understand the future changes a system is likely to undergo. If changes can be anticipated, design should try to achieve a structure where the effect of change is isolated within restricted parts of the system. Examples of possible changes to be taken into account are: changes in the algorithms that are used for a specific task; changes in data representation; and changes in the specific devices used by a program, such as sensors or actuators, with which the system interacts.

Parnas elaborated a design technique, called *information hiding*, through which an application is decomposed into modules, where the major sources of anticipated requirements changes are encapsulated and hidden as module secrets and unaccessible to other modules. A clear separation between *module interface* and *module implementation* decouples the stable design decisions concerning the use of a module by other modules from the changeable parts that are hidden inside it. Module clients are unaffected by changes as long as changes do not affect the interface. This great design principle enables separation of concerns, a key principle that supports multi-person design, and software evolution.

Information hiding allows the design of a large system to be decomposed in a series of modules, each of which has an interface separated from the implementation. If the binding between interface and implementation is established statically at translation time, interfaces can be statically checked for consistency both against their implementation and against their use from client modules. Static checks prevent faults from remaining undetected in running systems.

The concepts of information hiding and interface versus implementation were incorporated in programming languages and supported separate development and separate compilation of units in large systems (39). As an example, let us consider the Ada programming language, which was designed in the late 1970s. Ada modules (21), called *packages*, support information hiding and separate compilation of interfaces and implementations. Once an interface is defined and compiled, both its implementation and its client modules' implementation can be compiled. In general, a unit can be compiled if all the modules it depends on have already been compiled. This allows the compiler to perform static type checking among the various units (see Fig. 1.1).

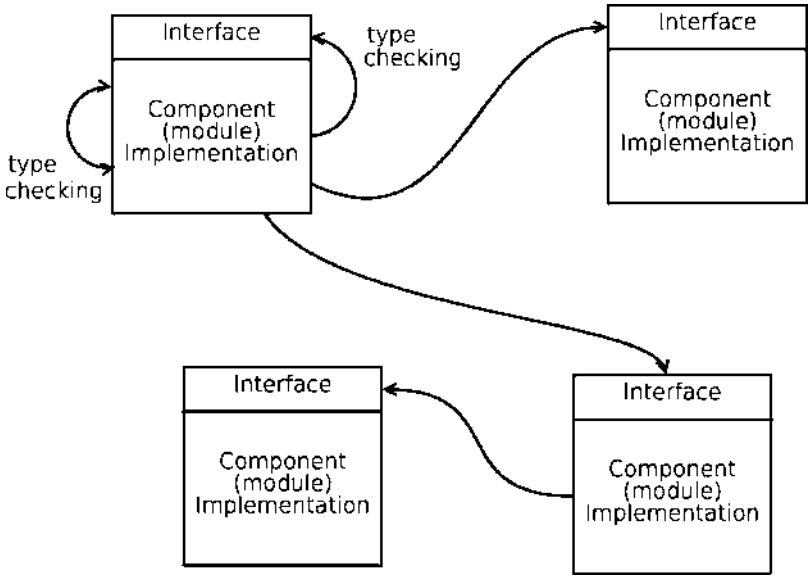

Figure 1.1. Structure of a modularized program.

There are many examples of languages that provide, at least in part, similar features. For example, in C programs it is possible, though not mandatory, to separate the declarations of function prototypes from the implementation and to place them in different files (from the implementation). In this way it is possible to define the interface exported by a module, though it is not possible to specify exactly which functions a module imports from another one.

Although changes are conceptually facilitated if they can be isolated within module implementations, the resulting implementation structure is fixed. All bindings are resolved prior to run time and cannot change dynamically. To make a change, the entire application has to be moved back to design time, changes are made and checked by the translation process, and then the system is restarted in the new configuration.

1.4.2. Object-Oriented Languages

The direct way information hiding was embedded into modular languages provided only a limited support to software evolution and flexible composition of software systems. Object-oriented design techniques and programming languages provided a further major step in this direction. Object orientation aims at providing improved support to modularization and incremental development; it also supports dynamic change.

According to object-oriented design, a software system is decomposed into *classes*, which encapsulate data and operations and define new abstract data types, whose operations are exported through the class interface. A major contribution of object-oriented languages is that the binding between an operation requested by a client module and the implementation provided by a server module can change dynamically at run time. This is a fundamental feature to support software evolution in a flexible manner. Moreover, most object-oriented languages provide it in a way that retains the safety of static type checking (21).

As an example, consider an application, in the field of logistics, that tracks containers. The operations available on a container allow its users to load an item into the container, to get the list of contained items, and to associate the container with a carrier. The carrier may be a train or a truck where the container is loaded for transportation, or it may be the parking area where the container is temporarily stored. The container provides an operation to get its geographical position, which is implemented by asking its carrier to provide it. A parking area has a fixed position, while the truck and the train have their own way of providing the position dynamically. Consider an evolution of the system where containers are equipped with a GPS antenna that can be used to provide their position. This can be implemented as a change in the implementation of the get_position( ) operation, which now uses the data provided by the GPS instead of asking its carrier. A client that uses a container would be unaffected by this new way of implementing the operation: An invocation of get_position( ) would be automatically redirected to the redefined operation (Fig. 1.2).

In general, given a class that defines an abstract data type, it is possible to define changes by means of a *subclass*. A subclass is statically bound to the originating class (its *parent class*) through the *inheritance* relation. A subclass, in turn, may have

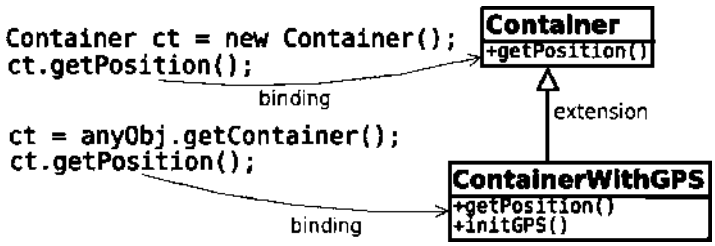

Figure 1.2. At run time we can't be sure on what version of a method will be called.

subclasses; this generates a hierarchy of classes. A subclass may add new features to its parent class (such as new operations) and/or redefine existing operations. Adding new operations does not affect preexisting clients of the class, because clients may continue to ignore them. Redefining an operation is also guaranteed not to affect clients if certain constraints are enforced by the programming language, as in the case of Java. The key composition features that support evolution are *polymorphism* and *dynamic binding*. With polymorphism, it is possible to define variables that may refer to different classes; that is, the binding between a variable and the class of the object it refers to is set at run time. These variables (called polymorphic variables) are defined as a reference to a class, and they can refer to objects of any of its subclasses. In other words, the *static type* of a variable is defined by the class it refers to in the variable's declaration. Its *dynamic type* is determined by the class of the object it currently refers to. The dynamic type can be defined by any subclass of the class that defines the static type. With dynamic binding, the operation invoked on an object is determined by the class that defines its dynamic type.

The flexibility ensured by dynamic binding may raise type safety problems: Since the invoked operation cannot be determined at compile time, how can one be sure that it will be correctly called at run time? Most object-oriented languages, including Java, retain the benefits of type safety in the context of dynamic binding by restricting the way a method may be redefined in a subclass. Because type checking is performed at compile time by considering the static type of a variable, a possible solution consists of constraining a redefinition not to change the interface of the method being redefined.[3] Software composition can thus be type checked statically even if bindings among objects may change dynamically at run time.

1.4.3. Components and Distribution

Off-the-shelf components (41) became progressively available to software developers. This asked for changes in the methods used to design applications: The traditional dominant design approaches based on top-down decomposition were at least partly replaced by bottom-up integration. With the advent of off-the-shelf components, process development became decentralized, since different organizations at different times are in

[3]References 11 and 27 provide a thorough analysis of these issues and give general criteria for type safety.

charge of different parts of a system. This has clear advantages in terms of efficiency of the development process, which can proceed at a higher speed, since large portions of the solution are supported by reusable components. Decentralized responsibilities also imply less control over the whole system, because no single organization is in charge of it. For example, the evolution of components to incorporate new features or replace existing features is not under control of the system integrator.

Components are often part of a system that is distributed over different machines. Indeed, distribution has been another major step in the evolution of software composition. With distribution, the binding between a component and the virtual machine that executes it becomes an important design decision. The requirement to distribute an application over different nodes of a network often comes from the need to reflect a distributed organization for which the application is developed. Often, however, the binding of functions to nodes can be ignored during system development and can be delayed to deployment time.

In a distributed system, the binding among components is performed by the *middleware* (18). Java RMI (28) is a well-known example of a middleware that supports client–server architectures by means of remote method invocations (Fig. 1.3). A Java object can be accessed by a remote machine as if it were executed on the same machine. The client invokes methods as if it was invoking them on a local object. The RMI middleware realizes the binding by marshaling the request and then sending it via TCP/IP. Then the server unmarshals the request, executes the method, and sends back the result in the same way. CORBA (33) is another middleware that supports distributed components written in different programming languages.

Different binding policies may be adopted to support distribution. One possibility is to perform a static deployment-time binding between a component and a virtual machine. Alternatively, one may think of dynamically binding a component to a virtual machine at run time. This is the case of dynamic system reconfiguration,

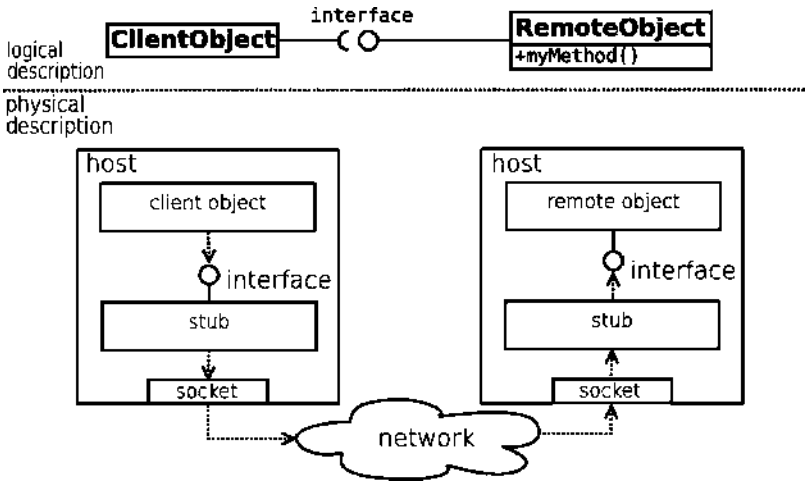

Figure 1.3. Distinction between logical and physical description with Java RMI.

whose purpose is to try to achieve better tuning of the reliability and performance of a distributed application by rebinding components to network nodes at run time.

1.5. SOFTWARE COMPOSITION IN THE OPEN WORLD

As we discussed, software development evolved toward flexible and decentralized approaches to support continuous requirements evolution. This was achieved by trying to anticipate change as far as possible, as well as by shielding large portions of an existing application from certain kinds of changes occurring in other parts. This approach continues to be a key design principle that software engineers should master. However, the speed of evolution is now reaching unprecedented levels: The boundary between the systems to develop and the real world they interact with changes continuously. In some cases, changes cannot be anticipated; they occur as the system is executing, and the running system must be able to react to them in some meaningful manner. Earlier we called this an *open-world scenario*.

Open world scenarios occur in emerging application areas, like ambient intelligence (17) and pervasive computing (40). In these frameworks, computational nodes are mobile and the physical topology of the system changes dynamically. The logical structure (i.e., software composition) is also required to change to support location-aware services. For example, if a person equipped with a PDA or a phone moves around in a building and issues a command to print a document, the binding should be dynamically redirected to the driver of the physically closest printer.

Location-aware binding is just a case of the more abstract concept of *context-aware binding*. In the case of ambient intelligence, context information may be provided by sensors. As an example, an outdoor light sensor may indicate daylight; a request for more light in a room may be bound to a software module that sends signals to an actuator that opens the shutters. Conversely, outdoor dark conditions would cause binding the request to switching the electric light on. Other examples of context-aware binding may take into account the current status of the user who interacts with the environment.

New enterprise-level applications are also subject to continuously changing requirements. Emerging enterprise models are based on the notion of networked organizations that dynamically federate their behaviors to achieve their business goals. Goal-oriented federated organizations may be supported by dynamic software compositions at the information system level. In this case, the composition of the distributed software is made out of elements owned and run by different organizations. Each individual organization exposes fragments of its information system that offer services of possible use for others; and, in turn, it may exploit the services offered by others. The binding between a requested and a provided service may change dynamically.

Moving toward these scenarios requires both new software composition mechanisms and new levels of autonomy in the behavior of the systems, which should be able to self-adapt to changes by reorganizing their internal structure. This implies that composition mechanisms should be self-healing; and, more generally, they should be based on monitoring and optimizing the overall quality of service, which may vary over time due to changes in the environment.

The rest of this section describes some examples of composition mechanisms that aim at a high degree of decoupling and dynamism to cope with open-world requirements.

1.5.1. Global Coordination Spaces

A middleware can provide a Global Coordination Space (GCS, Fig. 1.4), through which components may interact with one another and coordinate their behaviors in a highly decoupled manner. The GCS acts as a *mediator*. Each component participating in the architecture may produce data of possible interest for other components. However, components do not interact directly to coordinate and exchange the data. Coordination and data exchange are mediated by the GCS. Thus, data producers have no explicit knowledge of the target consumers, nor have they any direct binding with one another. The mediator acts as an intermediary between producers and consumers of the data.

There are two main kinds of architectures based on GCSs: publish–subscribe architectures and tuple–space architectures. Both of them allow the system to be very dynamic since components can come and go without requiring any system reconfiguration and without requiring changes that affect the components that are already part of the architecture. Communication here is intrinsically asynchronous since producers send their data to the mediator and continue to execute without waiting for the target consumers to handle them.

Publish–Subscribe Systems. Publish–Subscribe (PS) systems (19) support a GCS where components may publish *events*, which are delivered to all the components that registered an interest in them through a *subscription*. The middleware thus provides facilities for components to register their interest in certain events, via a subscription, and to be notified when events of interest for the component are generated. The GCS, here an event *dispatcher*, is a logically global facility, but of course it may be implemented as a fully distributed middleware infrastructure. Many examples of PS middlewares have been described in the literature (13, 14, 16, 25). Some of them are industrial products, others are research prototypes providing advanced features. Some have a centralized

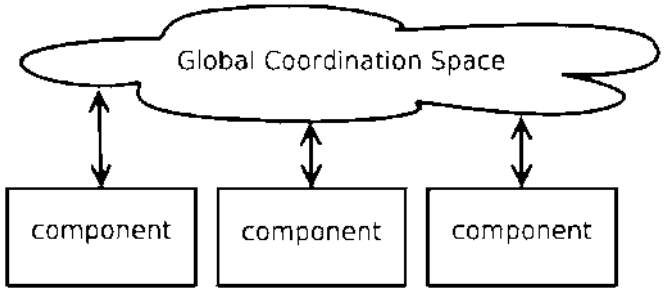

Figure 1.4. Global Coordination Spaces.

dispatcher; others have a distributed dispatcher. They may also differ both in the functionality they provide and in the quality of service they support. Regarding functionality, they may differ in the kinds of events they may generate and in the way subscriptions may be specified. For example, it is possible to distinguish between *content-based* and *subject-based* (or topic-based) PS systems. In the former case, an event is an object with certain attributes on which subscriptions may predicate. For example, an event may signal the availability of a new flight connecting two cities (Milano and Chicago) starting from a certain date (e.g., December, 1) and at a certain fare (e.g., 400 EUROs). A subscription may state an interest in flight information about flights from, say, Madrid to Dublin at a low fare (below 70 EUROs). In a topic-based PS system, instead, events are indivisible entities. For example, an event would be generated by an airline (say, for example, Alitalia) when a new flight announcement is published. Thus, for example, we might have "Alitalia" or "Continental" or "United" events. The events in this case do not carry a value.

Tuple–Space Systems. Tuple–Space (TS) systems support coordination via persistent store: The GCS provides features to store and retrieve persistent data. This coordination style was pioneered by Linda (12). Linda supports persistent data in the form of *tuples*. Tuples may be written into the tuple space; they can also be read (nondestructively) and deleted. Read and delete operations specify a tuple via a template. The template is used to select a tuple via pattern matching. If the match procedure identifies more than one tuple, one is chosen nondeterministically. If none match, the component that issued the read or the delete remains suspended until a matching tuple is inserted in the tuple space.

Many variations of the original Linda approach have been implemented by TS-based middleware, such as Javaspaces (20). There are also implementations, such as Lime (29), that fit the requirements of mobile distributed systems even more closely.

PS and TS systems have similar goals. Both aim at providing flexible support to distributed and dynamic software architectures. PS middleware is more lightweight: Persistence may be added as a service, but it is not directly supported as a native feature.

1.5.2. Service-Oriented Architectures

GCSs support decoupling and dynamism by removing any direct binding between sources and targets of a message. Service-Oriented Architectures (SOAs) (34) reach similar goals by providing facilities that allow components to register their availability and the functions they perform, and support discovery of the components that can perform the requested functions. Once it has been discovered, a component can be used remotely through direct binding. In principle, registration, discovery, and binding can be performed dynamically at run time.

SOAs are composed mainly of three types of components (Fig. 1.5): *service consumer*, *service provider*, and *service broker*. The service provider provides a service. The service is described through a specification (which we may assume to consist roughly of its interface description). The provider advertises the presence of the service to the

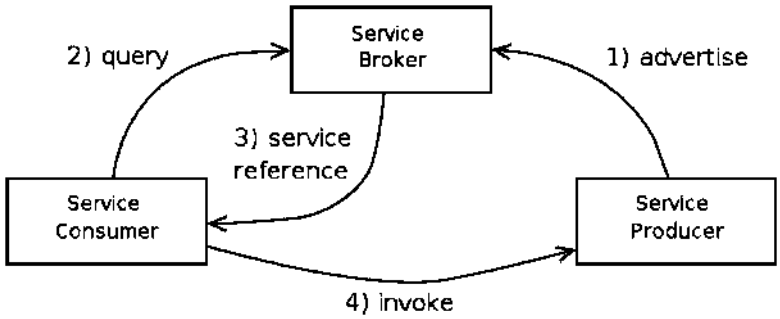

Figure 1.5. Service-oriented architecture.

broker, which maintains a link to the service and stores its interface. The consumer who needs a service must submit a request to a broker, providing a specification of the requested service. By querying the broker for a service, it gets back a link to a provider. From this point on, the consumer uses the operations offered by the provider by communicating with it directly.

SOAs provide a nice solution in the cases where complex interactions are needed between loosely coupled elements and where the requirements are subject to changes, thus requiring fast and continuous system reorganizations. Earlier we mentioned the example of ubiquitous mobile applications, where the discovery phase may be performed dynamically to identify the available local facilities to bind to, as the physical location changes during execution. We also mentioned the case of information systems for networked enterprises—that is, dynamic enterprise federations that are formed to respond to changing business opportunities. In this case, each organization participating in the federation exports certain functionalities, which grant a controlled access to internal application. In turn, it may use services offered by others and possibly choose among the different providers that provide similar services, based on some explicit notion of requested and offered quality of service.

Many middleware solutions exist which implement support for SOAs—for example, JINI network technology (43) or OSGI (3). At the enterprise level, *Web services* (4), and the rich set of standard solutions that accompany them, are emerging as a very promising approach, not only in the research area but also among practitioners.

IBM (43) defines a Web service as

> . . . a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions, which can be anything from simple requests to complicated business processes. . . . Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service . . . .

Web service (WS) technology offers a big advantage over other service-oriented technologies in that it aims at making interoperation between different platforms easy. This is particularly useful in an open-world environment, where it is impossible to

force all the actors to adopt a specific technology to develop their systems. WS technology consists of a set of standards that specify the communication protocols and the interfaces exported by services, without imposing constraints on the internal implementation of the services. Conversely, other solutions, like Jini, are based on Java both for the communication and for the implementation of services. Interoperability allows a seamless integration of the web service solutions developed by different enterprises.

Communication among Web services is based on the Simple Object Access Protocol [SOAP (10)], which describes the syntax of messages that can be exchanged in the form of function calls. It is an XML-based protocol and interaction is usually performed over the http protocol, thus allowing a less intrusive intervention in information systems to export services. After discovery, once the binding between a service consumer and a service provider is established, communication is achieved by sending a message through the SOAP protocol.

Services can be seen as an evolution of off-the-shelf components. Both implement functions of possible use and value for others; both go in the direction decentralized developments, since the organization in charge of developing the off-the-shelf component or the service is, in general, different from the organization that uses them. In both cases, this has obvious advantages (reduced development time, possible enhanced reliability), but also possible disadvantages (dependency on other parties for future evolutions) over complete in-house developments. The main difference between services and off-the-shelf components is that the latter become part of the application and are executed in the application's domain. Services, on the other hand, are owned and run by the provider. Another difference concerns the composition mechanisms. In the case of services, the choice of possible targets for a binding and the establishment of the binding can be performed dynamically at run time.

The most striking difference, however, is in the degree of control and trust. Off-the-shelf components cannot change after they become part of the application, unless they are explicitly replaced by the owner of the application. On the other hand, services are owned by the providers, who control their evolution. Thus services can change unexpectedly for the users.

Although web services are becoming a real practical approach to evolvable system architectures, many problems are still to be solved before they can be applied in open environments or in the case where systems have to meet stringent dependability requirements. The next section outlines the main challenges that must be met by architectures supporting dynamic software compositions in general, and in particular by SOAs.

1.6. CHALLENGES AND FUTURE WORK

We discussed a number of existing and promising approaches supporting software composition in the open world. As we observed, as solutions become more dynamic and decentralized, many problems have to be solved to reach the necessary degree of dependability that is required in practice. In this section, we try to identify where the main problems are and possible research directions. Some of these are new problems; many have always been with us and are just getting more difficult in the new setting.

Challenge 1: Understand the External World. Requirements have always been a crucial problem for software engineers (32, 42). Eliciting and specifying them in continuous change conditions is even harder. In the case of pervasive systems, it is often required that solutions are highly adaptable. For example, support to elderly or physically impaired people in their homes requires developing solutions that can be adapted and personalized to the specific needs of their expected users. At the business level, the requirements of federated enterprises need to be properly captured to support their interoperation. It is important to identify what should be exposed as service for others and what should be kept and protected as proprietary. Easy and dynamic integration is then necessary to capture business opportunities quickly. Although this is a research topic in business organization, software engineers should be aware of these trends. They should be able to capture the requirements of dynamic organizations and shape architectural solutions accordingly.

Challenge 2: Software Process Support. Business organizations are increasingly agile in the way they respond to business opportunities. Software development must also proceed in an agile fashion, to provide fast solutions to achieve competitive advantages and reduced time to market. It is still an open issue as to how agile development methods can comply with the high dependability standards that may be required by the applications. Problems become even harder because development teams are increasingly distributed and autonomous (1, 9, 24). How can standardized practices be enforced? What are the key drivers of productivity? How can process quality be defined and measured?

Challenge 3: Service Specification. The continuous evolution toward services that can be developed and then published for use by other parties makes it necessary to provide a specification for the service that possible users may understand. Software specification has been and still is an active research area. The need for precise module specifications that client modules could use dates back to reference 35. The problem, however, is crucial in the case of services, because of the sharp separation between provider and user and because specifications need to be a stronger and enforceable contractual document. Being a contract, the specification must provide a precise statement that goes beyond a purely syntactic interface definition and even beyond a mere definition of functionality. It must also state the quality of service (QoS) assured by the service. Specifications should be searchable. They should be stated in terms of shared ontologies (2). Finally, they should be composable; that is, it should be possible to derive the specification of a composition of services whose specifications are known.

Challenge 4: Programming and Composition. We described different paradigms for dynamic service composition, such as PS, TS, and SOAs. Coordination among components range from *orchestrated* compositions, where a workflow coordinates the invocation of services, once they have been discovered and bound, to *choreographed* compositions, like in a PS or TS style, where services must comply with a common protocol of exchanged messages and proceed in a peer-to-peer fashion (26, 30). How can different composition mechanisms be integrated in programming

language? How can such language support a full range of flexible binding mechanisms, from static and pre-run time to fully dynamic? How can binding policies include negotiation and even rebinding strategies in case of deviations from a stated target for service quality? How can self-healing or even self-organizing behaviors arise through proper compositions?

Challenge 5: Trust and Verification. Services are developed and run in their own domains. Service users have no control over them; they do not have access to their internals. If services can only be bound at translation or at deployment time, and no change occurs afterwards, then applications can undergo traditional verification and validation procedure. But in the case of dynamic binding, and because service implementations may change in an unannounced manner, traditional approaches fail. Verification and validation must extend to run time. Continuous service monitoring (6) must ensure that deviations from the expected QoS is detected, and proper reactions are put in place. In general, one must be able to find ways to make the service world as dependable and secure as required by the requirements. Building sufficiently dependable and secure systems out of low-trust services is a major challenge.

Challenge 6: Consistency. In a closed-world environment, a correct application is always in a consistent state with the environment. In an open-world setting, environment changes trigger changes in the software and generate inconsistencies. Inconsistency management has been recognized as problematic by many researchers in the past (5, 15, 23). It is necessary to deal with it in the case of dynamically evolvable systems.

This is just a short list of important challenges that should be part of the future agenda of software engineering research, By no means should the list be viewed as exhaustive. It just focuses on some of the problems our group is currently working on. Progress in these and other related areas is needed to continue to improve the quality of software applications in the new emerging domains.

ACKNOWLEDGMENTS

This work has been influenced by the experience gained in the National project ART DECO and the EU funded project SeCSE.

Elisabetta Di Nitto and Luciano Baresi helped develop our view of software composition.

REFERENCES

1. Flexible and distributed software processes: Old petunias in new bowls?
2. Owl specification. http://www.w3.org/TR/owl-ref/.
3. OSGI Alliance. *OSGI Service Platform: Release 3, March 2003*. IOS Press, Amsterdam, 2003.

4. G. Alonso, H. Kuno, F. Casati, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, Berlin, 2004.
5. R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, 1991.
6. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 193–202, 2004.
7. L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer* **39**(10):36–43, 2006.
8. F. L. Bauer, L. Bolliet, and H. J. Helms. Report on a Conference Sponsored by the NATO Science Committee. In *NATO Software Engineering Conference 1968*, page 8.
9. B. W. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, Reading, MA, 2004.
10. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, HF Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1.
11. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* **17**(4):471–523, 1985.
12. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM* **32**(4):444–458, 1989.
13. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, 2000.
14. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering* **27**(9):827–850, 2001.
15. G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Transactions on Software Engineering and Methodology* **5**(3):191–230, 1996.
16. G. Cugola and G. P. Picco. REDS: A Reconfigurable Dispatching System. Technical report, Politecnico di Milano, 2005.
17. K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J. C. Burgelma. Scenarios for ambient intelligence in 2010 (ISTAG 2001 Final Report). *IPTS, Seville*, 2000.
18. W. Emmerich *Engineering Distributed Objects*. John Wiley & Sons, New York, 2000.
19. P. T. H. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys* **35**(2):114–131, 2003.
20. E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, 1999.
21. C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, New York, 1997.
22. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 2003.
23. C. Ghezzi and B. A. Nuseibeh. Special issue on managing inconsistency in software development. *IEEE Transactions on Software Engineering* **24**(11):906–907, 1998.
24. J. D. Herbsleb and D. Moitra. Global software development. *Software, IEEE* **18**(2):16–20, 2001.
25. Tibco Inc. TIB/Rendezvous White Paper, 1999.

26. N. Kavantzas, D. Burdett, and G. Ritzinger. WSCDL: Web Service Choreography Description Language, 2004.
27. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**(6):1811–1841, 1994.
28. Sun Microsystems. Java Remote Method Invocation Specification, 2002.
29. A. L. Murphy, G. P. Picco, and G. C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 524–533, 2001.
30. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proceedings of 8th International Conference on Coordination Models and Languages (COORDINATION06) LCNS 4038*, pages 63–81, 2006.
31. P. Naur, B. Randell, and J. N. Buxton. *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*. Petrocelli/Charter, New York, 1976.
32. B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, 2000.
33. R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, New York, 1998.
34. M. Papazoglou and D. Georgakopoulos. Service-oriented computing: Introduction. *Communications of ACM* **46**(10):24–28, 2003.
35. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* **15**(12):1053–1058, 1972.
36. D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering* **2**(1):1–9, 1976.
37. D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 264–277, 1978.
38. W. W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON*, pages 1–9, 1970.
39. B. G. Ryder, M. L. Soffa, and M. Burnett. The impact of software engineering research on modern programming languages. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) **14**(4):431–477, 2005.
40. M. Satyanarayanan. Pervasive computing: Vision and challenges. *Personal Communications, IEEE* [*see also IEEE Wireless Communications*] **8**(4):10–17, 2001.
41. C. Szyperski. *Component Oriented Programming*. Springer, Berlin, 1998.
42. A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. *In Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19, 2000.
43. J. Waldo. Jini architecture for network-centric computing. *Communications of the ACM* **42**(7):76–82, 1999.