Composition Environments for Deployable Software Components

Chris Lüer and André van der Hoek
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{chl,andre}@ics.uci.edu

Abstract

Component-based software development is revolutionizing the software industry by promoting a view of software development in which applications are composed out of reusable, relatively large-grained, and mostly pre-existing components. Adoption of component-based software development leads to an important distinction in roles between those that develop and make available individual components and those that compose applications out of available components. As a result, application composition is no longer a matter of writing and combining source code, but instead of composing deployable components—components that are pre-packaged, independently distributed, easily installed and uninstalled, and self-descriptive.

A need arises for specialized environments that explicitly support the composition of an application out of deployable components. Such composition environments are starting to emerge, but an overall understanding of their nature and functionality is currently lacking.

This survey presents a comprehensive analysis of ongoing efforts in developing composition environments. Specifically, we contribute a classification of concrete features that a composition environment must exhibit and discuss the current state of the art in composition environments. Our purpose in doing so is threefold. First, we wish to establish a terminology and understanding of composition environments. Second, we wish to compare and contrast representative composition environments from a multi-disciplinary perspective that extends beyond software engineering into programming languages, visual programming environments, and component frameworks. Third, we wish to identify promising research directions that we believe will lead to the maturation, adoption, and widespread use of composition environments.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*; D.2.10 [**Software Engineering**]: Design—*Methodologies*; D.2.10 [**Software Engineering**]: Design—*Representation*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.12 [**Software Engineering**] Interoperability—*Interface definition languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages*

General Terms: Design, Languages

Additional Key Words and Phrases: Composition environment, application composition, deployable components

1 Introduction

As early as 1968, it was recognized that composing applications out of reusable and mostly pre-existing software components is more economical than what remains today's dominant practice of creating applications from the ground-up [McIlroy 1976]. While much research has ensued since then to make the vision of component-based software development a reality, only now critical technology is emerging to actually do so. Perhaps more importantly, industry, in an effort to curb ever-increasing costs, seems to be ready and has started to embrace the paradigm of component-based software development. As a result, a component market is slowly-but-surely taking foothold [Traas and van Hillegersberg 2000].

An important foundation of component-based software development is that it makes a clear distinction between those that develop and make available individual components and those that use the available components to compose whole applications [*VisualAge: Concepts and Features* 1994]. Compared to traditional software development, in which only application developers are concerned with the software development process, this introduces component developers and application composers as separate stakeholders with separate roles. As illustrated in Figure 1, component developers focus on the domain of the component, which is usually technical in nature. Application composers, on the other hand, focus on the domain of the application and are usually more oriented towards the business at hand. In other words, component-based software development encourages technical experts to develop generic components and business experts to use these components in composing specific applications.

Unfortunately, application composition in reality still requires so much technical knowledge that the desired division of labor remains elusive. With most current technology, application composers typically need to be versed in programming technologies in order to understand components and compose applications from them. While it may be a powerful way of composing applications, the typical amount of labor-intensive and sometimes complicated work involved creates a steep barrier and hinders widespread adoption of component-based application development.

A need arises for specialized composition environments that explicitly support application composers in their activities. In particular, a need arises for composition environments that support the composition of an application out of deployable components—components that are pre-packaged, independently distributed, easily installed and uninstalled, and self-descriptive. A number of environments are starting to provide functionality that can be used to do so, but the composition environments thus far are varied, their solutions are unsystematic, and they typically address the problem only partially.

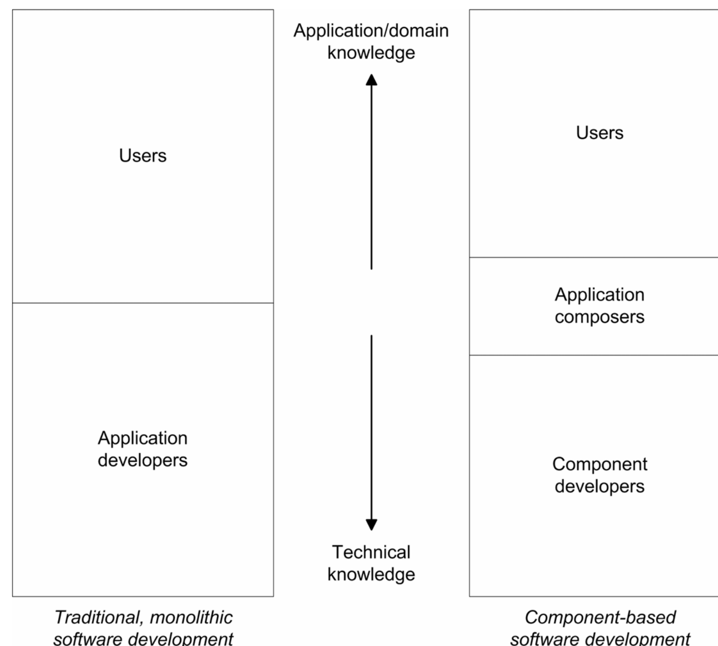

Figure 1. Different Roles in Traditional and Component-Based Software Development.

The goal of this survey, then, is to present a comprehensive analysis of ongoing efforts into developing composition environments in order to create an increased understanding of the nature of those environments. We discuss the composition environments from the perspective of both their underlying component model and their overall composition process. Both viewpoints are needed: the first forms the technical basis for deployable component composition, and the second defines the user experience when an application is being composed.

1.1 Comparison Framework

Guiding the discussion in this survey is a comprehensive comparison framework that we developed for this purpose. Table 1 introduces the high-level organization of the comparison framework that is further detailed in Section 3. The comparison framework consists of ten categories of features, four for the component model and six for the composition process as they are supported by each composition environment.

Our first comparison category, *deployable component*, examines the way in which deployable components are implemented. In some environments, a deployable component is a class or object. In others, it may be a process, thread, or procedural library. Each choice has its benefits and drawbacks, in particular in terms of typing and instantiation mechanisms that may be used. We specifically compare different component models in their ability to support typing and versioning, as well as in their mechanisms used to identify different components.

Our second category, *interfaces*, compares the features of each component model for modeling interfaces. Similar to the first category of deployable components, specific questions include whether interfaces can be typed and versioned, and what mechanism is used to uniquely identify different interfaces.

Configuration, the third category of our comparison framework, defines the basic dimensions along which we examine how each underlying component model supports the composition of deployable components into (partial) applications. Among others, we review whether connectors can be used, whether composite components can be defined, and whether a deployed component can be configured using customization parameters.

Self-description rounds out the comparison criteria along which we compare each underlying component model. We particularly examine which levels of self-description are supported, ranging from traditional syntactic and semantic self-description to quality-of-service and non-technical self-description.

The comparison of the composition processes followed by the different component environments is broken down into six categories. We determine how each composition environment supports *searching* and *selecting* deployable components, what features each environment provides for *adapting* deployable components and *composing* them into an application, what kind of support is provided for *executing* a composed application, and to what extent each environment leverages *self-description* throughout its supported composition process.

Table 1. High-Level Organization of Comparison Framework.

Component Model	***Composition Process***
Deployable component	Search
Interface	Select
Configuration	Adapt
Self-description	Compose
	Execute
	Leverage self-description

1.2 Objectives

The first objective of our survey is to establish a terminology and understanding of composition environments. Given the existing, wide range of opinions regarding the nature of deployable components and perceived tasks of composition environments, we believe it is necessary to clearly define relevant concepts and

features. Specific questions we will answer include "What is a deployable component?", "Which characteristics must a component model exhibit such that components built using that component model can be effectively deployed and composed?", and "Which basic features should a composition environment exhibit?"

Our second objective is to compare and contrast representative composition environments. In particular, we will highlight the commonalities and differences among existing composition environments and systematize the underlying technologies to gain insights into the design of such environments. The survey purposely extends beyond traditional software engineering topics such as software architecture and reuse environments to include relevant work from the areas of programming languages, visual programming environments, and component frameworks. Important contributions have been made in those areas that are of relevance to the design of composition environments.

The third and final objective is to identify promising research directions in the area of composition environments. In particular, we will highlight some serious deficiencies in the features provided by composition environments. Provision of adequate solutions to those deficiencies, we believe, will lead to the maturation, adoption, and widespread use of composition environments.

1.3 Scope

Much research in the field of software engineering concerns the composition of software components into applications. Some approaches develop mechanisms to assess whether some set of components is consistent with respect to some property [Abowd et al. 1995, Monroe 1998]. Other approaches develop languages in which to precisely specify components, their interfaces, and the overall configuration of an application [Medvidovic and Taylor 2000]. Yet other approaches concern such non-technical decisions as how to organize the process of component-based software development [Brownsword et al. 2000, Henderson-Sellers 2001] or when to buy and when to build a component [Heineman 2001]. While all of these approaches, and more, play a critical role in component-based software development, we necessarily limit ourselves in this survey to those approaches that integrally provide a composition environment through which an application can actually be constructed, instantiated, and executed.

We further limit our survey by concentrating on composition environments for deployable software components only. We particularly exclude the area of hardware composition and the area of source code reuse. Hardware composition is a well-studied subject that, although related to software composition, has some properties that make it very different in nature. Reuse of source code is also a well-studied subject that is adequately documented in existing surveys [Krueger 1992, Mili et al. 1995]. Deployable software components, however, are principally different in nature from both these areas because of their pre-packaged form, independent distribution, ease of installation and uninstallation, and self-descriptive nature. These characteristics change the nature of application composition from source to binary, from early binding to late binding, from application developers to application composers, and from technical challenge to business challenge. Our survey, therefore, studies the nature of composition environments for deployable software components only, and, as such, complements existing surveys in the field of composition.

Finally, it should be noted that many of the approaches we discuss have never been realized beyond the research prototype stage. Little empirical work has been performed to evaluate their relative strengths and weaknesses. This survey, therefore, attempts to provide a conceptual evaluation by distinguishing classes of approaches and illuminating their underlying concepts. As a result, the survey should be regarded as a guide to understanding the research literature and not as a guide to selecting an already implemented approach for use in practice.

1.4 Organization of the Survey

Section 2 of this survey defines our terminology as used throughout the remainder of the text. In Section 3, we categorize relevant features of composition environments and their supported component models. These features form the basis for our comparison framework used in Section 4 to compare and contrast a representative selection of composition environments. Section 5 contains our high-level observations regarding the surveyed composition environments and the state of deployable component composition in general. We conclude in Section 6 with an outlook at possible directions for future research.

2 Vocabulary

This section defines the terms used throughout this paper. While most of these terms are regularly used in the literature, their precise meanings often vary. We therefore state our interpretations of the terms here, with the intent of laying a common nomenclatural basis for the comparisons made in remainder of this paper. As appropriate, we reuse well-accepted definitions from the literature.

2.1 Component

Many definitions of software components exist that each characterize a component somewhat differently. Most of these definitions, however, suffer from one or more of the following drawbacks:

- *Technological bias*. Often, a definition is based on a single technology, and consequently defines a component as whatever is the convenient unit of reuse in that technology. For example, while in object-oriented programming convenient units would be either objects or classes, in distributed applications built in the C2 architectural style [Taylor et al. 1996] they are processes.
- *Domain bias*. Some definitions are geared towards a single domain and cannot easily be extended to other domains. Examples include defining components as user interface widgets or defining components as processes in distributed systems.
- *Vagueness*. A certain amount of vagueness is unavoidable if the definition is to be broad enough to cover different technologies. Nonetheless, a definition should give clear criteria that can be used to distinguish components from other software artifacts, something that is not always achieved in some of the existing definitions.

Nonetheless, some important commonalities exist among the more prevalent definitions. Szyperski defines a component as "a reusable, deployable piece of software without persistent state" [Szyperski 1997]. Councill and Heineman define a component as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [Councill and Heineman 2001] and Crnkovic et al. define a component as "an implementation of functionality that can be distributed in binary form and composed without modification according to a composition theory" [Crnkovic et al. 2002]. In combining and refining the common elements of these definitions while explicitly distinguishing a component from a deployable component as defined in Section 2.2, we arrive at the following definition:

> *A component is a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without modification, and (c) adheres to a component model.*

As compared to the definitions of Crnkovic et al. and Heineman and Councill, we intentionally do not include the criterion that a component must adhere to a composition theory (standard). Such a standard is an integral part of a component model as defined in Section 2.3. The criterion, thus, is covered through the criterion that a component must adhere to a component model and does not need to be repeated here.

2.2 Deployable Component

The composition environments we survey concern themselves with deployable components, which we define as follows:

> *A deployable component is a component that is (a) pre-packaged, (b) independently distributed, (c) easily installed and uninstalled, and (d) self-descriptive.*

This definition stipulates four criteria for a component to be deployable. First, the component must be pre-packaged, which typically means that it is distributed in binary form rather than source code. Pre-packaging has the benefit that it avoids having to rely on a user to configure and compile source code, which, to date, remains an error-prone process that typically requires significant technical skills.

The second criterion stipulates that it must be possible to distribute a deployable component independently from any application of which it may be a part. A deployable component, thus, is generally carefully designed such that it is self-contained and independent of the application context in which it will be used.

The third criterion states that a deployable component can be easily installed and uninstalled. While somewhat subjective, this is a critical aspect of deployable components. Much like source code configuration and compilation, installation and uninstallation can be a difficult process. The fewer technical skills this process requires from a user, the more deployable the component will be.

Finally, a deployable component is inherently self-descriptive. A traditional component typically only includes interfaces that define its functionality in terms of the methods it provides to, and sometimes requires from, other components. A deployable component, on the other hand, contains a variety of technical and non-technical, static and dynamic, self-descriptive meta-data that is used to assist in the composition process.

2.3 Component Model

A component model defines a standard to which a set of components must adhere in order to be composable into applications. The term is most often used for the industrial component frameworks COM [The Component Object Model Specification 1995], Java Beans [Java Beans: API Specification, Version 1.01 1997], and the CORBA Component Model (CCM) [CORBA 3.0 New Components Chapters 2001]. However, all of the composition environments surveyed in this paper support a component model of some kind, even though it may not always be explicitly specified. The reason is simple: without a component model, it would be difficult to make components interoperate. As an example, components that want to communicate with each other through remote procedure calls need to agree on a standard for passing parameters.

Generally, component models suffer from a tension between application composition and component development: the more restrictive a component model, the easier it is to compose applications. Conversely, the less restrictive a component model, the easier it is to develop components.

What exact details a component model should define is a matter of debate, and depends very much on the specific goals of the component model in question. It is generally agreed upon, however, that a component model specifies both a component standard and a composition standard [Councill and Heineman 2001]. A component standard concerns the implementation of a component and typically specifies some set of interfaces that must always be present, allowable patterns of interaction for the component, restrictions on its communication behavior, and possible context dependencies on hardware or implementation platforms. A composition standard concerns the overall composition of an application and, among others, specifies an architectural style in which components can be put together, allowable adaptation mechanisms, and constraints on global interaction behaviors. Our definition of a component model, then, is as follows:

> *A component model is a combination of (a) a component standard that governs how to construct individual components and (b) a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other.*

2.4 Connection

To compose components into an application, it is necessary to establish paths of communications among them. Connections are used for this purpose.

> *A connection is a logical link among two or more components that performs the exchange of data or control among those components.*

Because a deployable component is pre-packaged and cannot be easily changed, it is unaware of the other components with which it may interact in a composed application. Similarly, it cannot be aware of the actual communication mechanisms used to perform its interactions with the other deployable components. To turn a set of deployable components into an application, then, explicit connections must be formed that are not part of any of the deployable components. These explicit connections independently model and perform the communications among deployable components and are called connectors.

> *A connector is a component with the sole purpose of providing data or control transfer among other components.*

Ideally, all communications among deployable components should be modeled as connectors. If this is the case, the deployable components are context independent, because their whole context is created by the connectors.

2.5 Configuration

The configuration of an application is the result of the application composer's work and consists of a specification with which an application can be reproduced. Its basis is the set of components and the connections among the components. Often, however, a component is generic and must be customized before use by providing a set of parameters that determine its particular behavior. Therefore, we consider its set of customization parameters an integral part of the application configuration. Similarly, a component may not quite fit its intended purpose and must be externally adapted—using, for example, a wrapper—to make it fit better into the application. These adaptations, once again, are an integral part of a configuration.

A configuration is a specification of a reproducible application and uniquely identifies a set of components, their adaptations, their connections, and the set of customization parameters used.

2.6 Composition Environment

Deployable components are generally not executable programs of their own, yet exist in a pre-packaged form. Therefore, special composition environments must be used to build applications from deployable components. These composition environments may be very simple tools, very complex tools, or anything in between. In the simplest case, a composition environment does nothing more than to let the user define the connections among deployable components; the resulting application *is* the configuration specification and no additional external representation exists. In a more complex case, it may include a wide variety of supporting tools and languages that make application composition and related tasks easier. In particular, advanced composition environments typically provide a separate language in which to specify a configuration. In addition, such environments often provide advanced support for adapting deployable components, executing applications, and monitoring whether the behavior of an application conforms to the standards defined by the underlying component model.

A composition environment is a program that is used to construct an application out of deployable components.

A composition environment is usually accompanied by a component infrastructure (also called a component framework). The infrastructure provides basic facilities on top of which to implement components such that they adhere to the component model associated with the composition environment. Moreover, some composition environments provide a run-time infrastructure (a container [DeMichiel et al. 2001]) in which to execute the components.

2.7 Interface

To ensure that a composed application is consistent with respect to the expectations and assumptions of each of its constituent components, interfaces can be used. An interface abstracts part of the behavior of a component. The most basic type of interface only lists the services that a component provides or requires. If an interface specifies services that a component provides, the component promises to provide an implementation that fulfills that specification. If an interface specifies services that a component requires, the component cannot be executed without being connected to another component that provides an implementation to fulfill the specification. During application composition, a composition environment can analyze the interfaces to verify whether, for each component, all of its required services are connected to matching provided services on other components.

More advanced types of interfaces specify the pre- and postconditions of services, or the order in which the services can be invoked. Assuming these interfaces provide an accurate abstraction of the actual implementation of their respective components, composition environments with advanced analysis mechanisms can use the interfaces to prove certain desired properties of an application, such as whether the application is free of deadlock [Inverardi et al. 2000].

Councill and Heineman capture the nature of interfaces with their definition [Councill and Heineman 2001], which we adopt here:

An interface is an abstraction of the behavior of a component that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur.

3 Comparison Framework

In this section, we present the comparison framework with which we characterize and contrast the composition environments surveyed in Section 4. The framework consists of an organized set of desired features of composition environments that we identified by studying the literature, examining existing composition environments, and creating usage scenarios.

Table 2 introduces the features of the framework, as organized into ten high-level categories. The first four categories relate to the component model underlying a composition environment, the remaining six relate to the composition process supported by a composition environment. Both viewpoints are needed. Without a strong component model, the technical basis for component composition would be lacking. Similarly, without adequate process-level support, the user experience of composing an application out of deployable components would be less than desirable.

Below, we introduce each of the features of the comparison framework, motivate the need for that feature, and discuss its relevant issues with respect to deployable component composition.

Table 2. Comparison Framework.

Component Model	***Composition Process***
Deployable component	Search
Implementation strategy	Remote searching
Types and instances	Select
Global identity	Adapt
Versioning	Compose
Interface	Composition notation
Types and instances	Constraints
Global identity	Guaranteeing consistency
Multiplicity	Composition of distributed applications
Versioning	Execute
Interface location	Execution of partial applications
Configuration	Packaging
Composite components	Run-time changes
Connection semantics	Leverage self-description
Connection multiplicity	Leverage syntactic self-description
Connectors	Leverage semantic self-description
Connector types	Leverage quality-of-service self-description
Customization parameters	Leverage non-technical self-description
Self-description	
Syntactic self-description	
Semantic self-description	
Quality-of-service self-description	
Non-technical self-description	

3.1 Component Model

The nature of the component model used in a composition environment can have a great impact on the actual effectiveness of the environment in supporting component composition. Consider, for example, interfaces. If a component model provides extensive features for modeling interfaces, the composition environment can make use of interface specifications in automatically analyzing a composed application to verify

whether its interfaces match. As another example, if a component model has strong support for connectors, deployable components do not need to be altered in order to connect them. In both cases, the overall task of composition becomes easier and more reliable.

Below, we discuss those features of component models that we consider pertinent to deployable component composition. We first discuss the nature of deployable components, which are the cornerstone of composition. We then discuss component model features for modeling interfaces and configurations. We conclude with a discussion of self-description.

3.1.1 Deployable Component

Deployable components are the building blocks of applications. In a component market, they are usually licensed from another organization, installed and (if necessary) customized and adapted, and then connected to other components to form an application. While the definition in Section 2.2 determines what can and cannot be considered a deployable component, it intentionally leaves open the manner in which a deployable component is implemented. Hence, the detailed characteristics of a deployable component are defined by each component model in question. Substantial variation exists among those details, but at the basis of all component models surveyed in this paper is the notion that deployable components are those software elements that are composed into applications with the help of a composition environment.

Implementation Strategy

Considerable disagreement exists among component models as to how a deployable component is implemented. In some component models, a deployable component is a process. In others, a deployable component is an object or procedural library. In general, we can distinguish the following five implementation strategies:

- *Process.* In this strategy, a deployable component is a separate executable program running in an operating system. While resulting in easily deployable components, this implementation strategy lacks most of the flexibility and expressiveness present in other implementation strategies.
- *Procedural library.* As compared to processes, libraries provide a mechanism to encapsulate deployable components in a way that allows them to be easily connected. In their simplest form, for example, operating systems provide dynamic link libraries.
- *Object.* Objects, in their ability to encapsulate sets of data and their associated behavior in a fine-grained manner, provide the most flexible way of implementing a deployable component. With that flexibility, of course, comes the penalty of having to carefully manage the objects.
- *Class.* Implementing deployable components as classes provides a middle ground between processes and objects in terms of size and flexibility. In particular, it allows easy deployment of relatively fine-grained deployable components.
- *Class library.* Limiting a deployable component to be a single class places size and flexibility constraints on newly constructed deployable components. Therefore, some composition environments allow a component to be a set of classes, usually bundled together in a single archive [DeMichiel et al. 2001].

Each component model surveyed in Section 4 defines its deployable components as one of these implementation strategies. The most popular strategy is to use classes or class libraries, since they provide an acceptable balance between simplicity and manageability.

Types and Instances

Given that different component models have different implementation strategies for deployable components, it should come as no surprise that what is considered a type or an instance of a deployable component also differs significantly. This is illustrated in Table 3, which compares the realization of types and instances in C2 (see Section 4.1.1), pipe-and-filter systems (see Section 4.1.3), and Java (see Section 4.2.1). In C2, deployable components are either processes in the distributed case, or objects in the localized case. Its types are abstract specifications of deployable components as expressed in the C2SADEL architecture

description language [Medvidovic et al. 1999]. A separate concept of instances does not exist in C2, because a deployable component itself cannot be multiply instantiated and serves as a one-and-only instance.

Similar to C2, pipe-and-filter systems use processes as deployable components and have no concept of instances. Pipe-and-filter systems differ from C2 in their consideration of a type: instead of an explicit language specification, the executable program as stored on disk (from which processes are instantiated) is the type of a deployable component. Compared to language-based approaches, this is a very weak form of typing that provides little to no support for type checking.

Java as a composition environment is very different from C2 and pipe-and-filter systems, and considers classes as deployable components. Since classes can be instantiated into objects, objects are the instances of deployable components in Java. With respect to types, Java supports separate interface classes that abstractly define sets of methods. If a component declares to implement such an interface, it guarantees to provide an implementation of those methods. Two components implementing the same set of interfaces, thus, can be considered to be of the same type.

Overall, this rather simple example already illustrates the (surprising) conclusion that the concepts and implementations of types and instances differ significantly among composition environments. In our survey, then, we have two objectives for comparing their roles: (1) to determine whether the concepts are supported by a given composition environment, and (2) if they are supported, to determine how they are implemented.

Table 3. Differences in Types and Instances in Some Existing Component Models.

	C2	***Pipe-and-Filter***	***Java***
Type	C2SADEL specification	Executable program	Interface classes
Deployable Component	Process / object	Process	Class
Instance	—	—	Object

Global Identity

Deployable components are reusable and can exist in many different places. For purposes of detecting similar or distinguishing different deployable components, then, it must be possible to uniquely identify each deployable component. Because the component market is a global market, global identification becomes desirable. Such identification typically happens through a name space [Achermann and Nierstrasz 2000, Khare 1999], such as the Uniform Resource Locator name space [Uniform Resource Locators 1994], or through unique, decentralized identifiers [Box 1997].

Name spaces provide a human-readable classification of names, based on conventions about the structure of the name space. Most name spaces are logically centralized (the name space has a root), but can nonetheless be used in a distributed way through selection of appropriate conventions. For example, by using Internet domain names [Postel 1994] as the basis for a component name space, the non-distributed part of the name space is adopted from another, well-established name space, thus delegating the problem of centralization.

Decentralized identifiers are strings that are randomly generated in a way that makes it impossible or very improbable to generate the same identifier twice. Thus, decentralized identifiers are also global. They take up less space than name space identifiers and do not require naming conventions, but are usually not human-readable.

Versioning

Deployable components may exist in multiple versions [Conradi and Westfechtel 1998]. To differentiate these versions from each other, version numbers are used as an extension of the component name space with a specific meaning: components with the same name, but different version numbers are typically considered evolutions of one another. Per convention, backwards compatibility may be applied [The Component Object Model Specification 1995], but it is typically not enforced.

3.1.2 Interfaces

As defined in Section 2.7, an interface is an abstraction of the behavior of a component that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur. A primary goal of the use of interfaces is to realize information hiding. When a component specifies its provided and required functionality through interfaces, and all access to the component occurs through those interfaces, the implementation of the component is encapsulated and hidden from other components.

Component models that do not support interfaces are essentially untyped. While it still may be possible to label components with a type, without interfaces, there is no way to restrict connections among different components. All possible connections are legal, making the component model simple to use, but not very expressive. The trade-offs between typed and untyped component models are analogous to the trade-offs between typed and untyped programming languages.

Types and Instances

Different component models use the term "interface" differently. In some component models, it is used to designate an interface instance, in others it is used to designate an interface type. An interface instance, also called a port, is an integral part of a deployable component and represents the declaration of the deployable component to provide or require an implementation of the services specified by the interface instance. Each port, thus, has a direction: it is either a provision port (also called out-port), or a requirement port (in-port).

When a connection is created between two components, the connection is linked to a port of each involved component. Ports can only be connected if they are compatible according to the underlying type system, and if their directions are different. Thus, a connection is always established between a requirement port and a provision port that are instances of compatible interfaces. Ports may limit the number of connections that can be linked to them.

Interface types may either be implicit or explicit. When interface types are implicit, no external type system is present and interface compatibility is determined through the use of external analysis tools. In the simplest case, two interface instances have to be identical to be compatible, but the various subtyping relations known from programming languages can occur [Cardelli and Wegner 1985].

If interface types are explicit, external representations of the available interface types exist and each interface instance contains a reference to its type. Equality, once again, can be based on the interface types being identical or on specific subtyping relations (which will be captured explicitly in the interface type system, making type checking much simpler).

Multiplicity

Component models differ in the number of ports they allow per component. They may allow for one provision port per component, or a variable number of provision ports. Component models may either support or not support requirement ports. Environments with requirement ports have one or a variable number of requirement ports per component. Component models that do not support requirement ports suffer from insufficient component encapsulation, because component requirements cannot explicitly be specified.

Global Identity

Similar to deployable components (see Section 3.1.1), establishing a unique identity for interfaces requires a global name space or use of random unique identifiers. Component models that do not establish a unique identity for interfaces run the risk of naming conflicts when interfaces from different sources are used in one application.

Versioning

Interface name spaces may be augmented by a versioning scheme, similar to versioning of deployable components (see Section 3.1.1).

Interface Type Location

When interface types are supported, composition environments and components need to be able to access interface types in order to perform type checks. Therefore, there has to be a way to find an interface type by its name or identifier. Environments differ in the location where the interface type itself is stored.

Interface types may either be copied or referenced. In component models with interface type copying, each component that uses an interface type contains a full copy of it. This approach can cause consistency problems, because with a large number of components using the same interface type it is hard to guarantee that all copies of the interface type are equivalent. In component models with interface type referencing, a unique copy of each interface type is referenced through identifiers, but never copied. This master copy of the interface type is stored in a location that is not part of any component. Interface type referencing causes a performance overhead because remote access to the interface type may be necessary.

3.1.3 Configuration

The configuration features of a component model determine how components can be connected to form an application. Below, we introduce the configuration features along which we compare the composition environments in Section 4.

Composite Deployable Components

Applications can easily become confusing when they consist of large numbers of deployable components. Composite deployable components may alleviate this problem by providing internal structure. Composite deployable components are deployable components that are composed out of other deployable components and serve as partial applications. Deployable components that are not composite are called atomic. Without support for composite deployable components, large applications become hard to understand. Composite deployable components hide part of the complexity and thus make application composition more scalable.

Composite deployable components can be hierarchical or non-hierarchical. In a hierarchical model, every deployable component is an immediate part of at most one deployable composite component. Hierarchical models are rather inflexible, because every deployable component can be used at most once in a composite. Non-hierarchical models may become rather complex, because there may be a complex dependency graph between deployable components and the composites that they are part of.

Connection Semantics

Connections, defined in Section 2.4, have different semantics in different component models. Without specifying the semantics of connections, one cannot deduce the semantics of composed applications. A connection can mean that code is being exchanged between two components (for example, when a Java applet is transferred from web server to web browser), or that state is being exchanged (for example, through the parameters of remote procedure calls). The first case is *type-based* or code-based semantics, the second case is *instance-based* or service-based semantics [Lüer et al. 2001].

Instance-based connections are further categorized into stream-based and event-based connections, depending on the mechanism used to exchange information. Stream-based connections exchange data as a continuous stream to which new data is written by the sender, and from which the receiver can read the data. Event-based connections [Notkin et al. 1993] divide data into discrete messages that are exchanged at certain points in time.

The connection semantics employed in a given application correlate with the implementation strategy of a deployable component (Section 3.1.1). Component models in which components are processes and objects usually employ instance-oriented semantics; component models in which components are procedural or class libraries usually employ type-based connection semantics.

Connection Multiplicity

The multiplicity of a connection determines how many ports can be linked to it at each end. A basic connection has multiplicity 1-1; more complex connection multiplicities are 1-n (or n-1) and n-m (n, m being variable numbers). Connection multiplicity may depend on the type of the connection.

Connectors

Connectors, defined in Section 2.4, make composition easier because they provide a mechanism to connect two or more components without the need to modify those components. A connector, thus, has a certain functionality that does not logically belong to any of the components it connects. For example, a connector may encapsulate a specific network communication protocol.

Component models that support user-defined connectors raise the question of what the exact difference is between components and connectors. To answer this question, we observe that in distributed systems, components can easily be associated with hosts and connectors with communication connections between hosts. In localized systems, however, no obvious analogy exists, and the distinction depends on the specific architecture or architectural style used. Table 4 briefly summarizes the different characteristics of components and connectors to help in understanding the difference.

Component models may allow composite connectors [Garlan 1998]. Composite connectors are user-defined connectors that have been created by linking several existing connectors. Connectors can be introduced to programming languages in order to make classes or modules exchangeable [Flatt 1999].

Table 4. Comparison of Components and Connectors.

Component	***Connector***
application functionality	composition functionality
many different component types available	only a few different connector types available
can have any number of ports	usually has a fixed number of connection points
related to network hosts; localized	related to network wires; may be distributed

Connector Types

Some composition environments make available more than one type of connector, each having different connection semantics or different quality of service characteristics [Mehta et al. 2000]. Some environments additionally let the user define new types of connectors.

Customization Parameters

Deployable components are more reusable if they are flexible and can be customized by an application composer to fit the needs of a certain project. (Consider, as an example, a web server component that can be customized to use a different port or to use authentication.) Towards this goal, component developers can provide means to customize their deployable components through the specification of parameters that determine their behavior (also called anticipated adaptation [Bosch 1998]). Anticipated adaptation is opposed to ad-hoc adaptation (see Section 3.2.3), which is done without preparation by the component developer. Popular means of anticipated adaptation are property sheets and configuration wizards. A property sheet is a list of simple properties (such as numeric, Boolean, or string properties) that can be changed by the user and that function as parameters of component behavior. A configuration wizard is a dialog that guides the user through a sequence of customization steps.

Heineman and Ohlenbusch [Heineman and Ohlenbusch 1999] survey adaptation techniques and list requirements for them. However, many of the techniques described require some degree of source access and thus cannot be used with deployable components. Especially, inheritance (as commonly used in object-oriented languages) is not compatible with deployable components because of the well-documented fragile base class problem [Mikhajlov and Sekerinski 1998, Szyperski 1997].

3.1.4 Self-Description

Most component models support the use of self-description [Beugnard et al. 1999, Larsson and Crnkovic 2000, Lüer and Rosenblum 2001, O'Reilly 1999, Orso et al. 2000, Yacoub et al. 1999] to give a deployable component and its interfaces the ability to describe themselves at various levels of detail. As opposed to externally stored meta-data, self-description is an integral part of a component. Self-description has many advantages over externally stored meta-data. External description can get lost, may have to be updated manually, cannot easily be queried by composition environments, and is usually static.

Self-description applies to both required and provided functionalities [Ólafsson and Bryan 1997], and is categorized into four levels, namely syntactic, semantic, quality of service, and non-technical. Interfaces are one notable form of implementing self-description. Although they are typically limited to syntactic self-description, some component models supports the use of interfaces to also include semantic or even quality-of-service self-description.

Syntactic Self-Description

The first level of self-description, syntactic self-description, captures the services specified by an interface at the programming language level. It is needed to check compatibility between interfaces; if there is no syntactic compatibility, one of the deployable components will have to be adapted for both to be connected. For example, signatures of procedures are part of syntactic self-description.

Semantic Self-Description

Semantic self-description [Borgida and Devanbu 1999] specifies the behavior of the services specified by an interface. Semantic description can be formal or informal. Informal specifications usually take the form of natural language text. Since complete formal program specifications are hard to use, assertions [Findler et al. 2001, Rosenblum 1995] have been developed as a pragmatic form of incomplete formal specifications.

Quality-of-Service Self-Description

The third level of self-description is concerned with all technical issues that are not functional [Lycett and Paul 1998, Robben et al. 1998]. In essence, this level is used to specify the quality-of-service of a deployable component in terms of, among others, qualities such as performance, precision, or reliability. However, what is considered a functional requirement is not clear in many domains. For example, resizability of user interface windows may both be considered a functional requirement (because program code has to be written for it), and a non-functional requirement (because it is a usability concern). Many non-functional properties are hard to measure (for example, usability or reliability), so that it is not clear how useful description of these properties is. Additionally, there is a lack of useful notations to express non-functional properties in many cases. Nonetheless, the availability of carefully defined quality-of-service self-description can make composition much easier.

Non-Technical Self-Description

Non-technical self-description includes all other, non-technical properties that may be of interest to users. Examples include the price of a component, the author of a component or interface, and licensing information.

3.2 Composition Process

While the component model supported by a composition environment provides the technical basis for composition, the process-level features of the environment guide the user in actually composing an application. Many accepted criteria for the design of usable environments apply to composition environments, but we focus in our survey on those user-level features that are characteristic of the task of composition. We categorize the features according to the high-level application composition process as shown in Figure 2 [Lüer and Rosenblum 2001]: search, select, adapt, compose, and execute. Different composition environments support different parts of this process differently, but all generally are structured to more or less follow the steps in this process. Throughout, self-description plays an important role and hence we also survey the use of self-description in the composition process.

3.2.1 Search

Searching is an integral part of the application composition process. When requirements are determined, deployable components to fulfill these requirements have to be found. A deployable component that is of interest may depend on other deployable components, which also have to be searched for, although tools that help in this task exist [van der Hoek et al. 1997].

Remote Search

Deployable components may be developed by an organization that is different from the organizations using them. The organizations that want to use such a deployable component, then, typically have to search for it in remote repositories.

Composition environments may differ in the degree to which they integrate remote searching capabilities. At the one extreme, there may be no support at all. Application composers have to manually locate deploy-

able components and integrate them into a local repository. At the other extreme, a composition environment may be completely distributed and hide the actual location of deployable components from its user.

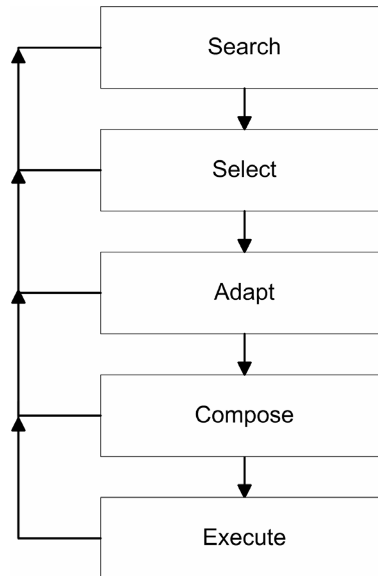

Figure 2. Composition Process Steps.

3.2.2 Select

Once candidate deployable components have been found, their suitability must be determined in order to select the right set of deployable components for the target application. Self-descriptive meta-data is critical in this endeavor. Currently, assessment of suitability is largely a human activity, although a few composition environments provide some automated analyses that can determine the consistency of a set of deployable components with respect to a certain property [Monroe 1998].

3.2.3 Adapt

Ad-hoc adaptation, as opposed to anticipated adaptation trough pre-defined customization parameters (Section 3.1.3), is the task of adapting deployable components to use them in a way that was not anticipated by their developer. Ad-hoc adaptation, also called external adaptation, is often used with legacy deployable components that are no longer supported, but must be somehow modified to function with current technology. Since deployable components are encapsulated and hide their implementations, adaptations cannot change the deployable components themselves. Instead, wrapping techniques have to be used.

A wrapper [Gamma et al. 1995, Hölzle 1993] is defined as a class that provides a new interface to another class. Clients use the wrapper and its new front-end instead of the wrapped class. In the simplest case, a wrapper just maps its own method names to the method names of the wrapped class. In the more complex case, a wrapper may also adapt the behavior of a wrapped method by adding processing steps before or after the call to the wrapped method. Transferred to deployable components, wrappers can be either other deployable components or connectors. A deployable component can wrap another deployable component simply by connecting to it and using it as a server, and providing a different interface. A connector can act as a wrapper by modifying communication between the deployable components it connects.

Shaw introduces and discusses different types of ad-hoc adaptation [Shaw 1995]. Many of these techniques require source code modification, though, and are not applicable to deployable components.

3.2.4 Compose

In order to create an application out of the set of deployable components that have been selected, the deployable components need to be connected. In particular, abstract dependencies of components as specified in their required interfaces, must be instantiated with concrete relations, typically by defining a connection or connector between two ports on different deployable components.

Composition Notation

Composition environments enable composition through scripts, diagrams, or a programming language interface. Scripts and diagrams are often intended to be usable by non-programmers; programming interfaces are more expressive, but require extensive technical knowledge and are not suitable for users without technical skills.

Scripts are text files that are written in a scripting language. A scripting language [Ousterhout 1998] is a programming language that is geared towards quick and easy programming. Development times with scripting languages are significantly lower than with system programming languages. As a trade-off, program performance in space and time is significantly worse. Most scripting languages reach their flexibility through being typeless and interpreted. For application composition, even simpler notations may be sufficient. Assuming that all complex tasks are handled by deployable components, a composition language only needs to be able to define connections and adaptations among those deployable components.

Diagrams are specifications that are expressed in a graphical notation [Shu 1988]. Typically, notations consist out of boxes and arrows between them; in the context of a composition environment, boxes generally represent deployable components and arrows connections. Diagrammatic languages are often equivalent to scripting languages; it may be possible to automatically convert diagrams into scripts and scripts into diagrams. Similar to scripting languages, there is a spectrum from Turing-complete visual programming languages to simple diagrams that express nothing more than connections between deployable components.

Constraints

Constraints are logical statements that constrain the structure of an application [Abowd et al. 1995]. Constraints can be either composition constraints or component constraints. Composition constraints regulate the way in which a set of deployable components interoperate. Topological constraints are composition constraints (for example, a constraint that forces the architecture of an application to be acyclic). Component constraints limit the kinds of deployable components that can be used in an application, for example, a constraint specifying that every component has to support a given interface.

Typically, the component model supported by a composition environment provides a base set of composition and component constraints. Some composition environments allow users to specify additional, application-specific constraints. This, in effect, is equivalent to an ad-hoc specialization of the underlying component model. Thus, constraints provide a flexible way to configure the composition environment itself.

Architectural styles [Shaw and Garlan 1996] are examples of popular sets of constraints. Other kinds of constraints might be domain-specific constraints that serve to adapt a generic composition environment to a specific domain. Medvidovic et al. [Medvidovic et al. 2002] present a case study that shows how a specific architectural style can be expressed in a constraint language.

Guaranteeing Consistency

Composition environments should guarantee the consistency of composed applications as much as possible. Inconsistency may arrive from not complying with specified constraints—either by a deployable component that is used or by the application as a whole.

Consistency checks can be performed either at design time or at run time. Design time consistency checks are performed either on-the-fly or as a separate analysis step [Robbins and Redmiles 2000]. On-the-fly checks continuously verify each design step taken. For example, the moment a user makes a connection between two deployable components, the connection is checked and if an interface type check fails, the user will be prevented from creating that connection or warned about a possible error. Design critics [Robbins and Redmiles 1998] are one technology used for this purpose.

Separate consistency checks are performed once an application is completely designed, or once the user initiates the check. If the check fails, an error message is returned and the user will have to update the application in order to make it consistent. Run time consistency checks are analogous to assertions (pre- and post-conditions) in programming languages.

Composition of Distributed Applications

Component technology has close ties to distributed technologies [Plásil and Stal 1998]. Therefore, many composition environments have integrated support for composing distributed applications (applications whose deployable components, when instantiated, are distributed over several hosts on a network).

3.2.5 Execute

The creation of an executable application is, of course, the main goal of the application process. Nonetheless, some composition environments go beyond this goal and provide mechanisms to execute a composed application. Especially since testing a composed application involves execution of the application, a flexible, usable composition environment will allow the user to build and run applications in quick iterations.

Execution of Partial Applications

Some composition environments require an application to be completed before it can be executed. Other composition environments, however, allow execution of incomplete applications or even individual deployable components for testing purposes.

Packaging

Once an application has been built out of deployable components, it needs to be packaged so it can be shipped and executed independently from the environment. Depending on the host platform on which the packaged application is to be installed, packaging may be more or less complex. In the simplest case, the deployable components and their configuration are copied into a file that can directly be executed on the host platform. In more complex cases, glue code generation may be necessary or a specific bootstrapping and run-time environment (a container) must either be packaged and distributed along with the application or already be present on the target host.

Run-time Changes

Environments may support changing the configuration of an application at run time. For example, applications for which the acceptable downtime is very small may have to be updated at run time, because updating of individual deployable components instead of updating the application as a whole may reduce downtime.

Since configuration changes at run time are still a research issue [Oreizy et al. 1999, Oreizy et al. 1998], only some composition environments support the capability, and then only in a limited manner. It is usually only possible to change configurations dynamically as long as no guarantees are needed that no program state is lost. Composition environments may provide mechanisms such as transaction support, roll-backs, and state extraction to support dynamic changes.

3.2.6 Leveraging Component Self-Description

Self-description of components (see Section 3.1.4) can be leveraged by composition environments in each step of the composition process. Composition environments differ by the amount and way in which they make use of self-description capabilities defined by the underlying component model. Some only use semantic self-description, others semantic and syntactic, and yet others will support all forms of self-description.

4 Survey

Composition environments for deployable components are still in their infancy, and no composition environment developed thus far addresses all of the concerns presented in Section 3. Nonetheless, many interesting approaches are currently under development, each based on a different paradigm to explore a different aspect of the problem. In this section, we present an overview of those approaches by discussing a representative set of composition environments.

While we primarily focus on approaches in the field of software engineering, we include work in the areas of programming languages, visual programming environments, and component frameworks. Important contributions have been made in those areas that are of relevance to the design of composition environments.

Below, we discuss each composition environment according to the comparison framework established in Section 3. For each composition environment, we first introduce the environment, then discuss unique aspects of its component model and distinguishing features of its composition process, and conclude with a summary and a brief look at similar composition environments. Table 5 gives an overview of the terminologies used in the surveyed composition environments. Appendix A contains the detailed data upon which the ensuing discussion is based.

Table 5. Terminology as Used in the Surveyed Composition Environments.

Approach	*Deployable Component*	*Connection*
C2 / ArchStudio	component	connector
Koala	component	binding
Pipe-and-Filter	filter	pipe
UniCon	component	connector
Plug-In Systems	plug-in	—
Java	class file	class path
Jiazzi	unit	connection
Visual Basic	control (VBX)	—
VisualAge	part	connection
BeanBox	bean	connection
Vista	component	link
AgentSheets	agent	—
CodeBroker	—	—
Enterprise Java Beans	enterprise bean	—
.NET	assembly	—

4.1 Software Architecture

Within the field of software engineering, composition environment have been extensively researched in the domain of software architecture. Software architecture [Di Nitto and Rosenblum 1999, Medvidovic and Taylor 2000, Perry and Wolf 1992, Shaw and Garlan 1996, Whitehead et al. 1995] is concerned with specifying and manipulating the large-grained structure of applications in terms of components and connectors. Often, the term is used for any large-grained design, including design of compile-time structure (sometimes called logical architecture). The software architecture literature, however, focuses on the design of the runtime structure of an application (also called the implementation architecture). As such, architectural components often are deployable components and a composition environment is used to assemble an application out of those deployable components.

Instance-oriented application design, the concept of connectors, and the use of architectural styles are the main contributions from the field of software architecture to the field of component composition. To illustrate these contributions, we discuss three architectural composition environments and two architectural styles that support deployable components.

4.1.1 C2 / ArchStudio

C2 is an architectural style developed in 1995 [Taylor et al. 1996]. ArchStudio is a composition environment that supports the building of applications in the C2 style [ArchStudio 3, Medvidovic et al. 2000, Oreizy et al. 1998]. While ArchStudio is based on the Java platform, the C2 style is platform- and language-independent. C2 was originally developed for adding graphical user interfaces to legacy software, but it has proven to be a suitable style for large-scale distributed applications in general. ArchStudio contains tools to display and modify architectures (e.g., Argo/C2 and ArchShell), and facilities to map changes between an architectural model and its implementation.

Component Model

The single most distinguishing feature of the C2 component model is that it constrains the structure of an application to be layered. Combined with a rule that components can only communicate by exchanging asynchronous messages via connectors, this results in applications whose structure is highly decoupled. The component model further enforces decoupling by only allowing two types of messages, namely requests (which flow upward in the architecture) and notifications (which flow downward). Components are therefore completely unaware of their surrounding context—making them deployable.

In the Java implementation of C2, deployable components are objects with their own control threads. Each deployable component has two ports, one called top and one called bottom. Both ports are bidirectional and untyped. Connectors are explicit, have unlimited cardinality, and exist in several kinds that implement different event-based communication mechanisms (e.g. distributed or local, synchronous or asynchronous). Both predefined and user-defined connector types are supported, and connectors may be distributed. Neither customization parameters nor composite deployable components are supported.

Composition Process

The ArchStudio composition environment supports the composition of C2 applications by providing a component framework (called c2.fw) with which deployable components must be implemented. The framework provides basic services such as, among others, data marshalling, transactions, and event routing.

A graphical front-end (Argo/C2, shown in Figure 3) complements the framework and allows users to compose applications directly out of deployable components by simply linking deployable components and connectors to each other. Changes to the architectural model are immediately reflected in the composed application which is modified accordingly. This is possible since deployable components in C2 are instances in their own right and a one-to-one mapping exists between the actual deployable components and the architectural components modeled in the environment. ArchStudio, thus, supports instance-oriented design and applications can directly be executed in the environment, even if the application is only partially complete.

A weakness of ArchStudio is its support for searching and selecting deployable components. It assumes all deployable components are locally available and application composers know simply by the name of a deployable component whether such a deployable component is relevant for their purpose. No self-description is available to be exploited.

Summary

ArchStudio is one of the most advanced typeless, instance-based composition environments to date. While it is restricted in its applicability by being limited to one rather complex architectural style, it provides extensive support for this type of composition paradigm. Various new connector types have been researched under C2 [Dashofy et al. 1999]. Because of the fact that C2 extensively uses connectors, it is well suited for experimentation with such novel connector types.

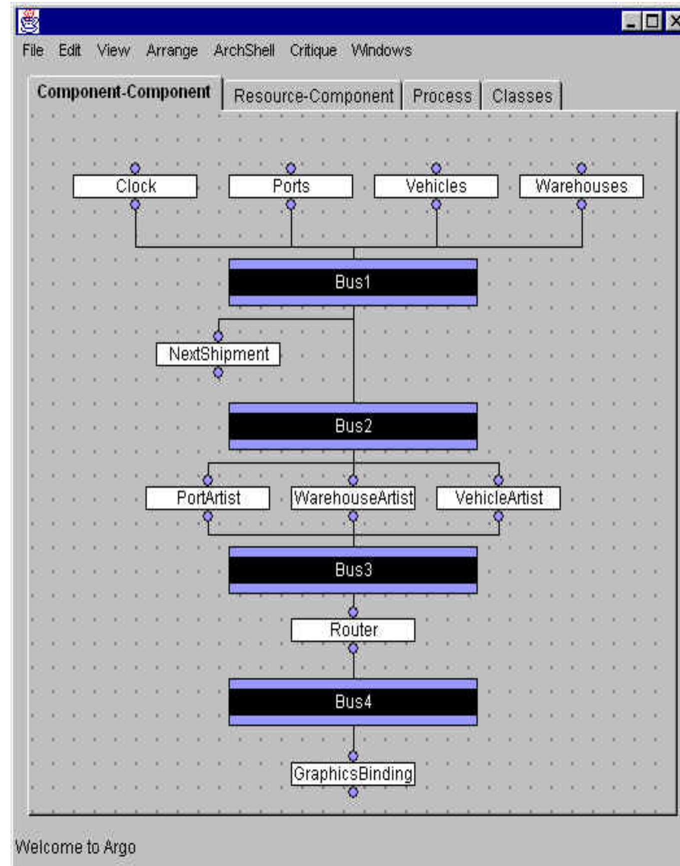

Figure 3. Composing a C2 Architecture.

Related Approaches

C2 is extended by an architecture description language, C2SADEL and a supporting tool, DRADEL [Medvidovic et al. 1999]. Both extend the ArchStudio environment to provide it with a comprehensive typing mechanism [Medvidovic et al. 1998]. Several other architecture description languages (for example, Darwin [Magee and Kramer 1996], Weaves [Gorlick and Quilici 1994], or Wright [Allen and Garlan 1997]) address similar issues as C2. Most of them, however, do not have as extensive tool support for deployable components as provided by C2 / ArchStudio.

4.1.2 Koala

Koala [van Ommering 2002, van Ommering et al. 2000] is a component model and associated tool set for embedded software. Koala is based on the Darwin architecture description language [Magee and Kramer 1996] and supports the specification of an embedded software application as a set of components and connections. Whereas components are largely implemented as regular C modules, efficiency needs of the domain demand that a specialized compiler is used to statically bind components based on the connection instructions of the architectural model.

Component Model

The Koala component model distinguishes itself in four ways. First, components are actually not deployable, since they are merely source code files. Nonetheless, the specialized compiler and the linking process supported by it leverage the available architectural description to automatically deploy the components and compose them into an application without any human intervention. Therefore, while the component model does not explicitly support our definition of a deployable component, the eventual effect of using it and its available support tools is largely the same.

Second, Koala supports diversity interfaces. In addition to regular interfaces that explicitly specify the provided and required functionality of a component in terms of sets of C functions, Koala allows the customization of component behavior through a so-called diversity interface. This is especially useful in the domain of embedded software, in which components often exhibit flexibility with respect to the specific hardware on which they operate.

Third, connectors are not supported. While explicit connections are identified in the architectural model, the aforementioned reasons of efficiency make that these connections cannot be implemented as connectors but must instead be translated into optimized, direct component-to-component procedure calls.

Finally, Koala has specific rules for component evolution, which constrain what may change between versions of a component. Similar to COM [The Component Object Model Specification 1995], a new version can only be created if backwards compatibility is maintained. If backward compatibility cannot be guaranteed, a wholly new component must be defined which starts its own line of evolution.

Composition Process

In essence, the composition process of Koala consists of creating an architecture, specifying a mapping onto implemented components, and running the specialized compiler and linker to compose the application. Both textual and graphical notations are supported in this process, but once an application has been composed it cannot be further modified. To update an application, a whole new composition cycle must be undertaken. As compared to ArchStudio, thus, Koala lacks support for dynamism.

The Koala composition environment provides no support for searching and selecting a component. Instead, it relies on the user using an existing configuration management solution. Since the Koala component model lacks support for extensive self-description, this has the advantage that relevant metadata still can be captured as comments associated with specific versions of components as they are stored in the configuration management system. Because components are composed locally, this does not pose as much of a drawback as identified in our discussion on self-description (see Section 3.1.4).

Summary

Koala is unique in combining an extensive architectural description language with an industrial-strength component model and mapping both on a particular implementation strategy. It is geared towards high-performance, low-overhead applications and successfully demonstrates how a closely integrated approach to composition can lead to significant benefits in terms of component reuse and application optimization. While the applicability of Koala's techniques is certainly wider than the very specific domain for which it is built, one should be cautioned that deployability of components was not one of Koala's design goals and that such capabilities as end-user composition or dynamism are consequently difficult to achieve.

4.1.3 Pipe and Filter

Pipe and filter [Shaw and Garlan 1996] is an architectural style that is extensively used in command shells for the quick and easy connection of text-based tools. The C Shell [Joy 1994], which is frequently the shell of choice in Unix-like operating systems, is an example of a shell that supports the pipe-and-filter architectural style. In its case, use of the style relies on the general structure of Unix programs: each has one standard input port ("stdin"), and one standard output port ("stdout"). A pipe-and-filter system, then, is a linear sequence of programs (called filters) connected via pipes. Each pipe links the output port of one program to the input port of the next and acts as a stream buffer to which the output port writes and from which the input port reads.

Program input or output is restricted to textual strings that are structured by convention. For instance, space and tab symbols are usually considered data separators. Programs that are connected with pipes are typically text-based operating system commands (such as "print directory"), whose output is modified and organized by adding a series of filters that perform such tasks as string manipulations and sorting.

Component Model

The component model underlying the pipe-and-filter architectural style is extremely simple: deployable components are filters, which are explicitly connected by generic pipes. Every deployable component has a

single input and a single output port, both of which are distinctly untyped. As a result, the validity of a connection cannot be guaranteed.

The structure of an application is restricted to be sequential: output of one program is input for the next. More complicated application structures are not supported.

Composition Process

The C Shell allows users to quickly connect and execute larger applications by using the filter symbol ("|"). Using this symbol, several deployable components can be connected at once in a command line. The actual command line specification serves as an application configuration and can be stored in a reusable shell script so that it can be used more than once.

Anticipated adaptation of components is possible through command line parameters by adding, for each parameter, the parameter symbol ("-"), the parameter name, and the parameter value after the name of the tool.

Summary

Pipe-and-filter systems are explicitly designed for quick, ad-hoc composition, which is usually all they can do. Nonetheless, the pipe-and-filter architectural style distinguishes itself in two important ways. First, it impresses through its simplicity, since it is by far the simplest composition mechanism that is commonly used. Second, it is particularly geared towards both anticipated and ad-hoc adaptation. Most filters support extensive customization parameters and the output of any textual tool can be externally modified simply by adding a post-op filter. Disadvantages are that composition is restricted to basic, sequential tasks; that the command-line interface tends to be cumbersome; and that no type system is supported.

4.1.4 UniCon

UniCon is a composition environment that is based on the use of an architecture description language of the same name [Shaw et al. 1995, Zelesnik 1996]. It was developed in 1993 based on early software architecture work by Shaw and Garlan [Shaw and Garlan 1996], and focuses on providing explicit notational support for architectural connectors. In doing so, UniCon captures typical informal descriptions of connectors in a formal way. In this respect, it constitutes an extension of module interconnection languages (MILs) [DeRemer and Kron 1976]. UniCon differs from MILs through the realization that packaging matters [Shaw 1995]. Unlike MILs, UniCon supports the composition of applications consisting of components that are provided in different formats, and that have not been designed to interoperate with each other.

Component Model

The component model of UniCon is unique in that components and connectors are of equal status. Both have interfaces, termed *roles* for components and *players* for connectors. When composing an application, roles are linked to compatible players. Compared to C2 / ArchStudio, UniCon, thus, distinguishes itself in that connectors are typed and can have bi-directional interfaces. Otherwise, the component model of UniCon is similar: there is no self-description beyond the syntax level, composite components are supported (support for composite connectors is planned, but not yet implemented), and anticipated adaptation is supported through component properties that can be set at composition time.

When application composition has concluded, the component model serves as the source for automatically generated composition files that, when executed, actually create the application. To support this process, an integral part of the component model provides facilities for specifying mappings of components onto source code. Connectors are not mapped, but chosen from a predefined set with predefined implementations. When generating a composition file, UniCon takes care of automatically inserting the right code for these connectors.

Composition Process

In terms of the composition process, UniCon exclusively focuses on the actual composition of an application; neither searching, nor selecting, nor ad-hoc adaptation is supported. UniCon supports both textual and graphical composition, and maintains consistency between the two notations. Although an application can be instantiated, there is no support for run-time changes.

Summary

UniCon is the first architecture description languages to advocate the use of connectors that have a status equivalent to components. In particular, it supports typing of connectors using interfaces (players) much like components are typed using interfaces (roles). While most of its connection mechanisms, unfortunately, are source-code based, it also provides an extension of the pipe-and-filter mechanism for the composition of deployable components.

4.1.5 Plug-In Systems

While not a generic composition environment, systems with a plug-in architecture, such as Netscape Navigator [Plug-in Guide 1998] and Adobe Photoshop [Adobe Photoshop: Application Programming Interface Guide 2000], provide an easy, low-overhead way of extending their functionality by "composing in" additional deployable components according to a strict architectural style. A complex, large application, often called the plug-in framework, defines an application programming interface that other manufacturers can use to extend its functionality through plug-ins. The extent to which a plug-in can cooperate with the framework, and the resulting level of coupling between the plug-in and original product, depends on the application programming interface in question and how much functionality it exposes. Whereas some frameworks only expose a limited set of functions and can only be extended in very specific ways, other frameworks, such as Apache [The Apache HTTP Server Project], expose almost all internals and even their most basic behaviors can be significantly modified.

All plug-ins are optional: a plug-in can extend the functionality of the framework, but is not required for use of the framework. Also, the number of plug-ins is typically unlimited and it is possible to install and use more than one plug-in at the same time.

Component Model

The component model used is defined by the framework and its specific application programming interface, and may differ for each framework. Since plug-ins provide functionality, composition is type based. Because there is only one framework in each system, the architecture of composed applications is limited to two levels (platform level and deployable component level), and there is no need for composite deployable components or deployable component adaptation. Because of their specialized nature, plug-in systems typically have only a small number of available deployable components per framework.

A strongpoint of the component model is its inherent support for anticipated adaptation. In effect, the whole style is geared towards allowing plug-ins to modify the behavior of the main components that form the application framework in certain, predefined ways.

Composition Process

Since plug-ins are normally developed for one specific framework, they are usually delivered with dedicated installation programs. Since the application architecture has a simple two-level tree topology (plug-ins usually do not have plug-ins of their own), no visualization or scripting support is needed and the composition environment reduces to simply running the install script and possibly configuring the plug-in by hand.

Summary

Plug-ins are an ad-hoc solution for platforms that want to interact with a small number of deployable components that serve the specific purpose of extending or adapting the detailed behavior of the platform. As such, they are very useful, but their scope cannot easily be extended or generalized.

Related Approaches

Compound document standards, such as Opendoc [Piersol 1994] or OLE [Brockschmidt 1994], represent a symmetric extension of plug-in technologies. While in a simple plug-in system there is exactly one program that acts as a framework, and a number of plug-in programs that act as deployable components, in compound document systems each program can act as both a framework and a deployable component. This makes it possible, for example, to embed spreadsheets into a text document and, conversely, to embed the same text document into the same spreadsheet. Compound document technologies have now largely been replaced by component models such as .NET (Section 4.5.2) and Java Beans (Section 4.3.3)

4.2 Programming Languages

Programming language research, especially in the area of object-oriented languages, has developed many mechanisms to support the design of the compile-time structure of programs. Examples of such mechanisms are inheritance, polymorphism, encapsulation, and explicit interfaces. Lately, many programming systems have started to support dynamic linking [Franz 1997] in order to make libraries independently deployable. Dynamic link libraries (DLLs) extend reusability from the source code to the compiled code. Since this has significant impact on composition, we discuss two relevant approaches from this domain. First, we discuss Java, a popular object-oriented programming platform that puts a strong focus on dynamic linking and deployability. Then, we discuss Jiazzi, a component extension of the Java platform.

4.2.1 Java

Java [Singhal and Nguyen 1998] was introduced by Sun Microsystems in 1995. Its programming language [Gosling et al. 1996] is complemented by an extensive standard class library [Java 2 Platform, Standard Edition, v 1.4.0 API Specification 2002] and a virtual machine [Yellin and Lindholm 1998]. The virtual machine and the standard class library are jointly known as the Java platform. Design goals for this platform include:

- *Portability.* Java provides a platform that allows execution of programs on a large number of different locations of the Internet independent of the host hardware;
- *Ease of programming.* The object-oriented core concepts of encapsulation, inheritance, and polymorphism are inherent to the platform. As a result, programming is made easier. Such difficult issues as memory management no longer need to be explicitly programmed. Furthermore, there are no program files. Class files, each encapsulating a single class with a well-established interface, are the only kind of executable files.
- *Internet-wide deployment*: Class files can be loaded from remote hosts on the Internet, and dynamically linked at program run time.

Deployable components for the Java platform are class files, which encapsulate a single class, or Jar files, which are sets of class files that are bundled and compressed into one file. Each class file contains either the declaration and implementation of a class or the declaration of an interface. Interfaces in Java define class skeletons in terms of public methods without including method bodies. A class may implement an interface by providing an actual implementation for each method.

To compose an application, the “class path” has to be set. The Java class path is an operating system variable (alternatively, a command line parameter of the Java run-time environment) that determines where the run-time environment looks for class files. The class path may include local directories, files, or remote locations specified by URLs. When an application is executing, the run-time environment searches for classes that are needed by using the class path. If several class files that realize the same class exist, the one that is listed first in the class path is used.

Component Model

Components (class files) are encapsulated and deployable [Drossopoulou et al. 1998, Liang and Bracha 1998]. They provide syntactic self-description of provided features through the reflection mechanism of the Java platform. Java distinguishes itself in that interfaces are entities of the same rank as deployable components, since they are also stored as class files and thus deployable themselves. This means that an extensive typing mechanism is available, allowing the Java platform to perform extensive checks on the consistency of a composed application.

Both deployable components and interfaces are identified through a naming convention that guarantees globally unique names by including the developer's Internet domain name. The class path is an explicit connector, since it facilitates changing the architecture of an application without having to change its deployable components. However, use of the class path for this purpose is tedious, because there is no mapping between the structure of the class path and the application architecture.

Composition Process

The Java platform supports composition only through its command-line interpreter which dynamically composes an application as needed. No additional process-level support is supported beyond connecting deployable components via the rudimentary mechanism of a class path and subsequently executing the resulting application using the Java virtual machine. Through external sources, however, one could consider the search process to be covered. Online resources, such as ComponentSource [ComponentSource] and other similar component archives [freshmeat.net, SourceForge.net], provide a wealth of Java components that can be downloaded for use as necessary. These archives typically provide metadata to guide a user in selecting components, but unfortunately components are not self-descriptive: the metadata is not included as part of the component that is downloaded.

Summary

While severely limited as a composition environment, Java was the first object-oriented system with a precise specification of compatibility of (deployable) components. In particular, its interface mechanism is a powerful way of typing components.

A second advantage of Java is that class paths are a simple means to compose applications. Together with binary compatibility and the wealth of online resources from which suitable components can be obtained, it becomes easy to substitute deployable components for each other.

Related Approaches

Binary Component Adaptation [Keller and Hölzle 1998] and Load-Time Adaptation [Duncan and Hölzle 1999] are add-on techniques that can be used to modify Java class files after compilation. They can be used to adapt the syntax of a class declaration (for example, by renaming methods) or to extend classes by adding semantics that do not require knowledge of the implementation (for example, adding pre- and postcondition checks). Although these approaches are dependent on implementation details of the Java platform, they demonstrate how a platform like Java can be extended to provide more advanced self-description, as well as support for unanticipated adaptation through a wrapping mechanism.

4.2.2 Jiazzi

Jiazzi [McDirmid et al. 2001] is a script-based tool that adds the notion of deployable components to the Java platform. Unlike Java Beans and similar technologies (as discussed in Section 4.3.3), Jiazzi introduces type-based deployable components and is based on research geared towards eventually introducing connectors to programming languages [Flatt 1999].

Deployable components are either atomic or composite. An atomic deployable component is simply mapped to a set of Java class files; a composite deployable component is mapped to other (atomic or composite) deployable components. Both atomic and composite deployable components have types, which are specified using special signature scripts. Figure 4 shows how Jiazzi can be used to specify parts of an application. Two example scripts are shown: “ui”, which defines the type of an atomic user interface deployable component with the help of a signature “ui_s”, and “linkui”, which defines a composite deployable component that connects “ui” with another deployable component called “applet”.

Once a complete application has been specified in scripts, the Jiazzi linker generates the actual application. The linker uses the scripts to modify the class files that implement the application. In particular, it modifies the constant names (such as names of classes) and replaces the names of signatures by the names of the classes that they are bound to in the unit definitions. The modified class files then constitute the complete application and can be executed in the usual manner of the Java platform.

Component Model

Components in Jiazzi are not as deployable as most of the other approaches discussed in this paper, since they suffer from a lack of encapsulation. In particular, there is no mechanism to prevent the implementation of a deployable component from communicating with other deployable components in ways not specified in the scripts. In addition, the component model is only active at design-time: once an application has been composed, the component model disappears into the application and cannot be easily accessed or abstracted

```
atom ui {
      export ui_out : ui_s<ui_out>;
}

compound linkui {
      export ui_out : ui_s<ui_out>,
             app_out : applet_s<ui_out>;
} {
      local u : ui, a : applet;
      link u @ ui_out to a @ ui_in, u @ ui_in to ui_out,
          a @ app_out to app_out;
}
```

Figure 4. Two Jiazza Scripts Defining the Type of an Atomic Deployable Component and Specifying the Composition of a Composite Deployable Component.

out. Since connectors are not supported, this means that deployable components are fully aware of their context, thereby violating one of the critical conditions for independent deployability.

Strong points of the Jiazzi component model are that deployable components are type-based and provide syntactic self-description of both provided and required features. Signatures serve as interface types and can consist of multiple Java classes. They are thus more abstract than Java interfaces, which specify only individual classes.

Composition Process

The composition process in Jiazzi is severely limited. An application composer only has available a simple, but not very readable scripting language. Once an application has been specified in this language, all the application composer can do is to execute a specialized linker to compose the application. The separate linking process, which involves modifying class files, has the advantages of not requiring any additional packaging and avoiding almost any performance loss, but significantly reduces interactivity. However, the configuration of an application cannot be reverse-engineered out of the generated code, nor is any self-description included. As a result, generated applications are not suitable as a distribution format when further changes to the configuration might be necessary.

Summary

Jiazzi is a simple environment with a simple component-based notation (components, in and out ports, and types) that works on top of the Java virtual machine. While problematic in some regards, it shows how the Java environment can be extended to include an external composition mechanism that explicitly recognizes deployable components and the configurations in which they are placed.

Related Approaches

ArchJava [Aldrich et al. 2002] is another extension to the Java language that, much like Jiazzi, introduces components and ports to the Java platform. While more complete in terms of its architectural model than Jiazzi (it supports connectors), and while able to enforce the use of ports and connectors through compile-time checks, ArchJava is limited to composition at compile time and consequently has the same drawback as Jiazzi in not fully supporting deployable components.

4.3 Visual Programming Environments

Visual programming [Shu 1988] is an area of programming language research that focuses on graphical, instead of textual, notations for programs. While visual programming languages are by far not as widely used as their textual counterparts, visual design notations (e.g., the various graphical representations used by the UML [OMG Unified Modeling Language Specification 2001]) are widespread. Although the underlying component model and composition process supported by visual programming environment differ from those used in composition environments, important lessons can be learned. In particular, the ease with which applications can be composed in visual programming environments is a property that composition environments attempt to achieve. We survey two of the more popular commercial environments, namely

Visual Basic and VisualAge, one prototypical environment, namely the BeanBox, and two representative academic visual programming environments, namely Vista and AgentSheets.

4.3.1 Visual Basic

Microsoft Visual Basic [Euler et al. 1991, Microsoft Visual Basic, Udell 1994] (see Figure 5) is an integrated development environment for the MS Windows operating system introduced by Microsoft in 1991. Its goal is to simplify the development of applications that make heavy use of the MS Windows application programming interface. The deployable components in Visual Basic programs are forms and controls. Forms are dialog windows, whose user interface can be created by dragging controls (window elements such as buttons) on a grid in the usual way of GUI builders. Forms and controls can be adapted through property sheets to set attributes such as color and font. Any programmatic logic to be executed upon usage of a form must be inserted into event handler procedures associated with the individual events that can happen in that form, for example the pressing of a button. A set of standard controls is predefined, but it is possible to import custom controls. Since the introduction of Visual Basic, many other integrated development environments for Microsoft Windows have been extended to support integration with Visual Basic custom controls.

Component Model

The large popularity of the Visual Basic environment has been explained with its well-defined file format to which any (custom) control must adhere [Maurer 2000]. This file format forms a critical part of the component model supported by Visual Basic and has a number of distinctive features. First and foremost, controls are fully encapsulated and can easily be exchanged, thus promoting deployability and component reuse. Furthermore, controls provide syntactic self-description that is used by the Visual Basic environment to display and adapt them visually. Finally, the file format serves as a type system: controls that are displayed and manipulated in the environment are instances of the types defined by their files.

Event handlers could be considered connectors since they link deployable components together. However, because event handlers usually contain significant operational functionality of their own, we classify them as components. Thus, there are no explicit connectors such as in C2 or Koala. Another drawback of the Visual Basic approach is that the overall architecture of an application is not visible. The visual composition environment only shows top-level deployable component instances, that is, those deployable component instances that are immediate constituents of the application. Other deployable components on which these might depend are not shown.

Composition Process

Predefined deployable components can be visually arranged and adapted in the environment. To connect deployable components, it is usually necessary to write textual event handler code in the Visual Basic programming language. Although wizards are provided to assist in this process, any kind of more complex connection has to be hand coded.

The application can be executed in the environment, but can also be packaged in a proprietary format that is closely integrated with the MS Windows operating system. As such, the environment extends into support for executing applications. It does not, however, provide any facilities for changing an application at run time.

Summary

Visual Basic introduced the masses to component-based programming. The combination of an easy-to-use and easy-to-learn programming environment with a robust file format for deployable components proved to be commercially very successful.

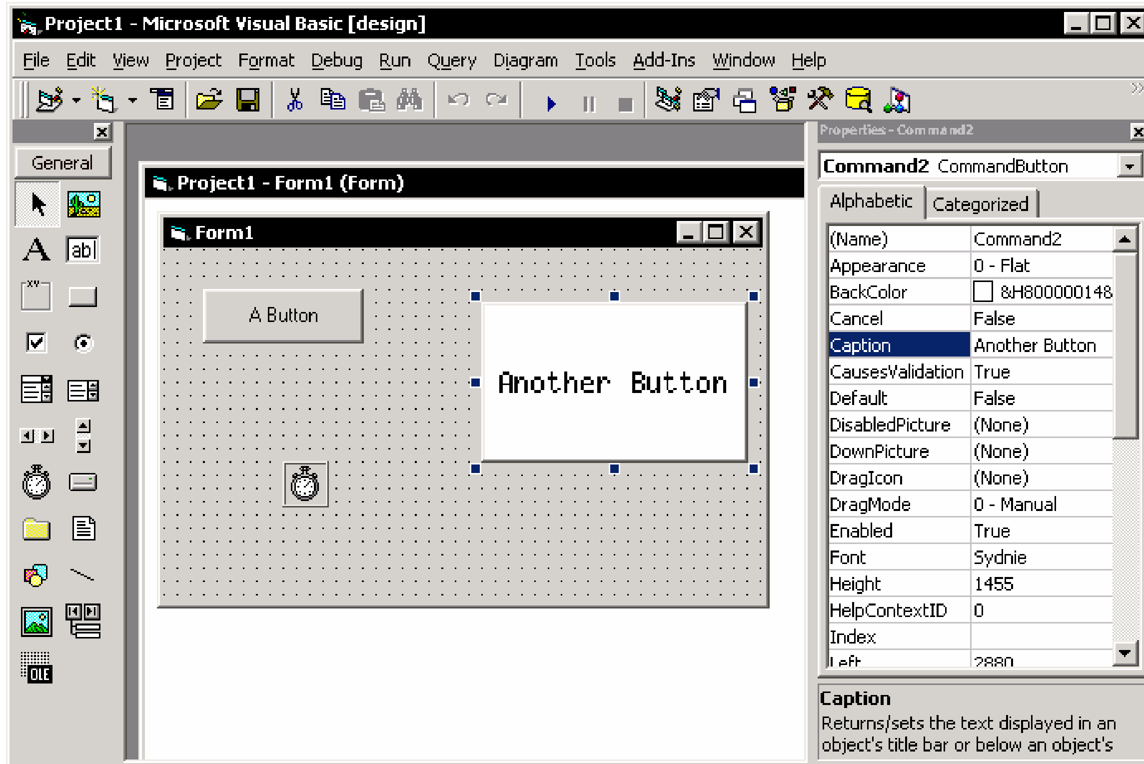

Figure 5. Composition of a Simple GUI in Visual Basic Version 6. The Form Shows Two Button Controls and a Timer; the Property Sheet for the Selected Buttons is Visible.

4.3.2 VisualAge

VisualAge [VisualAge for Java, *VisualAge: Concepts and Features* 1994] (see Figure 6) is an integrated development environment for the Smalltalk language published by IBM in 1994. Since then, versions for C++ and Java have been added. The goal of VisualAge is to increase the separation of labor among programmers by reducing the amount of technical skill needed to build an application. Apart from the traditional features of an integrated development environment, such as program editing and debugging, VisualAge therefore includes support for component-oriented visual programming. It does so by building on and extending the metaphor of graphical user interface builders (such as Visual Basic) to include support for non-visual deployable components and different kinds of connectors.

To compose an application, deployable components (which are called "parts") are dragged from a toolbar onto the part composition editor, where they can be adapted with property sheets and connected. Each deployable component has an interface consisting of attributes, actions, and events. Connections are created by linking these elements graphically. Legal types of connections include:

- *attribute-to-attribute*: whenever one attribute changes, the other one is updated to the same value;
- *event-to-action*: whenever an event occurs, the action is started;
- *event-to-attribute*: whenever the event occurs, the attribute is set to the value of the event parameter; and
- *attribute-to-action*: the action is triggered whenever the attribute changes.

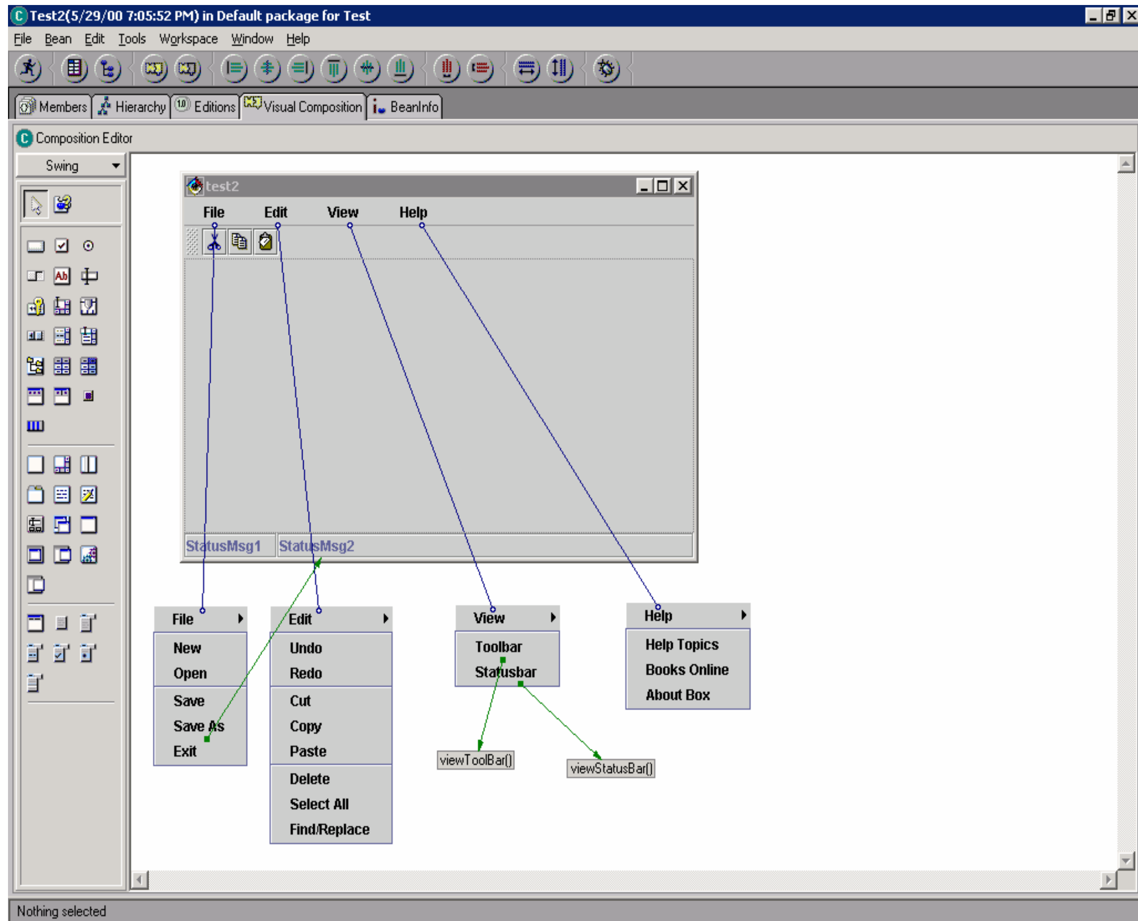

Figure 6. Screen Shot of the VisualAge Composition Editor.

New deployable components can be defined visually or through scripts (i.e., by writing code in the programming language). After a new deployable component is implemented, the interface editor is used to define its public interface. For example, if the interface includes an attribute, the component developer has to define which methods are used internally to get and set the attribute.

Component Model

Deployable components are visual entities that are realized as instances of classes. The visual environment, however, realizes features that go beyond the object-oriented class concept, such as composite deployable components and explicit connectors. Deployable components have syntactic self-description; the direction of ports depends of the types of connectors attached to it. For example, an action port has always direction "in", whereas an attribute port can have both "in" and "out" directions. Connectors are explicit and serve as communication channels among deployable components.

Composition Process

The strength of VisualAge lies in its extensive support for connecting and adapting deployable components visually. Compared to most composition environments, it is extremely easy to use yet has more compositional capabilities than simple graphical user interface builders.

Since VisualAge is a fully integrated development environment, textual representations in the underlying programming language can also be used. However, this means that the use of connectors cannot be enforced, since the environment does not maintain backward consistency from source code to the deployable components that form the application architecture. Nonetheless, some limited support for reverse engineering textual programs into visual representations exists to enable importing legacy applications.

Newer versions of VisualAge include remote repositories that can be used to store and retrieve deployable components in local area networks. The use of self-description, however, is limited so application composer generally have to know for which deployable components they are looking.

Summary

VisualAge was one of the first commercial tools to fully realize visual, component-based programming. Its strength lies in its ease of use: a meaningful application can be constructed simply by visually manipulating the deployable components that make up its configuration. Unlike many other development environments, VisualAge achieves this usability without limiting itself to graphical user interface components. Any kind of application can be developed, as long as suitable deployable components are available.

One drawback of VisualAge, however, lies in its strategy of implementing deployable components as objects. This strategy makes management of the evolution of the resulting application rather complex.

Related Approaches

Many integrated development environments now support similar mechanisms for visual composition. Examples are Microsoft Visual Studio [Microsoft Visual Studio], Borland Delphi [Borland Delphi], and JBuilder [JBuilder]. These environments use visual composition mechanisms mainly for construction of graphical user interfaces, and do not visually represent connections, which makes them unsuitable for more complex composition tasks.

4.3.3 BeanBox

The BeanBox is a prototypical environment for Java Beans [Birngruber et al. 1999, Java Beans: API Specification, Version 1.01 1997], which is a component infrastructure for the Java platform introduced by Sun Microsystems in 1996. The original purpose of the BeanBox was to provide a testbed for Java Beans, as well as to provide an orientation for people wishing to implement Java Beans environments. Several companies have since provided industrial-quality environments with technology very similar to the BeanBox, for example, JBuilder [JBuilder].

The BeanBox is based on the technologies of VisualAge (see Section 4.3.2), but simplifies them considerably. In particular, deployable component development is not supported and only three of the connection types of VisualAge are supported, namely event-to-action, attribute-to-attribute, and attribute-to-action.

Component Model

Deployable components are Java classes that are stored in Jar files. A Jar file can contain several classes, but only one of them can be a "Bean" (e.g., the main class exposing the interface of the deployable component). Any other classes in the Jar file must only be support classes used by the Bean class.

The Java Beans component model distinguishes itself by supporting syntactic self-description that goes beyond the self-description provided by the Java platform and most other composition environments. A particularly important aspect of this self-description is established through naming conventions. For instance, one of the naming conventions stipulates that methods that return the value of an attribute have to be named as "get" concatenated with the attribute name in question. As a result, the environment is able to extensively leverage introspection to uncover the supported interfaces of a deployable component. Limited semantic description is also available and is accessed in much the same way.

Simple method calls between components are allowed, but cannot be modeled by explicit connectors since there are no explicit input ports for method calls. Other types of ports, however, are available, and Ohlenbusch and Heineman present a formal specification of those types of ports [Ohlenbusch and Heineman 1998].

Composition Process

The BeanBox environment supports instance-oriented design. The classes constituting a Bean are instantiated when the Bean is visually dragged into the graphical builder. To create an application, explicit connectors are used. The connectors are automatically generated as individual classes when a connection is established.

Deployable components are connected and adapted through property sheets. The environment particularly leverages the self-description prescribed by the Java Beans standard in this process. Specific dialogs display all information about the deployable components being used. Once an application has been composed, it can be executed from within the environment. Additionally, the BeanBox can also generate an “applet”, which is a Java program that can be downloaded and executed from within a remote Web browser. The code generated for the applet encapsulates the connections that were made among the Beans that were used.

Summary

The BeanBox is an attempt to combine the technologies of visual development environments such as VisualAge with the Java language. The BeanBox distinguishes itself from other composition environments with its extensive support for self-description in its component model. It particularly shows that self-description need not be complex: simple naming conventions suffice to make deployable component composition significantly easier from a user perspective.

Related Approaches

Arabica [Rosenblum and Natarajan 2000] is an extension of the BeanBox with a focus on architectural concerns. It supports application composition according to the C2 architectural style (see Section 4.1.1) by generating wrappers that turn C2 deployable components into Java Beans and enforcing the C2 style rules as applications are composed.

The Bean Markup Language [Weerawarana et al. 2001] is a textual, XML-based notation to connect and adapt Java Beans. It comes with both an interpreter and a compiler, and shows how a new, more powerful and expressive composition notation can overlay an existing notation.

4.3.4 Vista

Vista is an environment for visual composition of applications [de Mey 1995, de Mey and Gibbs 1994]. Although originally developed for the composition and manipulation of multimedia applications only, it nonetheless supports a generic concept of deployable components and connectors. A Vista deployable component consists of an interface, a behavior, and a presentation, the latter of which is responsible for displaying the deployable component in the graphical environment. The environment is relatively independent from its underlying component model, because the component model can be explicitly specified by the application composer as a set of constraints. Data-flow-diagram-like composition and GUI-building are two different example component models supported by Vista. While this configurability is advantageous in terms of the flexibility and broad support for composition of different kinds of applications, it also brings with it the drawback that each different domain usually requires a different component model. As a result, an application composer must obtain an appropriate component model before an application can be composed. Just like the development of components, development of such a new component model is non-trivial and requires significant technical knowledge.

Component Model

Vista distinguishes itself in being able to support different, user-defined component models. Of course, these models all have to adhere to and be expressed in an underlying meta-model, which in turn explicitly defines basic representations of deployable components, ports, and connectors (called “links”). Composite deployable components are also supported, but neither customization parameters nor ad-hoc adaptation is supported.

In any component model used by Vista, it is critical that deployable components provide self-description about their ports so that they can be connected graphically. Ports are either supplier ports (out-ports), or user ports (in-ports). Deployable components are instantiated by the environment when they are being dragged into the editor; but the concrete meaning of “instantiation” is left to the underlying component model, so that both instance-oriented and type-oriented design are possible.

Composition Process

To create an application, deployable components are dragged into the visual editor and connected in the usual, graphical manner. Because of the flexibility of the system, however, the concrete user-interaction model is determined by the designer of the component model. Defining the component model, thus, par-

ticularly involves specifying a series of composition constraints that control the style (e.g., data flow, C2) in which an application can be composed.

Summary

Vista is a fully-configurable visual programming environment with support for code-based deployable components as well connectors. It represents the first attempt at combining a visual environment with a complex component model. Vista particularly distinguishes itself in the flexibility of its component model: through the specification of additional constraints, it can be configured to support significantly different types of applications in significantly different architectural styles.

4.3.5 AgentSheets

AgentSheets [AgentSheets 2002, Repenning and Sumner 1995] is a visual programming environment that allows for easy manipulation of applications represented as two-dimensional grids of agents. An agent developer defines agents by giving them a two-dimensional, rectangular depiction and an associated behavior. The behavior consists out of simple event-action sequences that allow the agent to change its depiction and to cause events in adjacent agents.

Figure 7 shows a simple AgentSheets worksheet that simulates automobile traffic on a bridge. The user's task is to modify the bridge without causing cars to crash. The simulation consists out of two kinds of interesting agents: cars and bridge elements. Cars move to the right, and if there is no bridge element underneath them, they crash down. Bridge elements simulate weight. If they are not kept in place by a sufficient number of elements below them or at their sides, they will drop. Many simple simulations like these can be quickly and easily constructed. It should come as no surprise, then, that AgentSheets has been successfully used for elementary school education in teaching elementary students many different phenomena [Repenning and Sumner 1995].

Component Model

Agents are the deployable components in AgentSheets. Deployable components can internally be complex in terms of their logic and display, but their interfaces are restricted to simple interactions with their four adjacent agents. Because the grid on which agents are placed serves as the one-and-only connector, the component model of AgentSheets is extremely simple. It does not need to support user-defined connectors, self-description, or any further configuration support.

Composition Process

AgentSheets has extensive support for the graphical creation of applications from agents. While applications are configured simply by placing deployable components on the graphical grid, the development of individual deployable components is also well supported. Many standard rules are available, and a graphical programming environment supports creating new rules or adapting existing ones. Once the application is constructed out of the desired set of deployable components, a simulation can repeatedly be executed. To facilitate rapid debugging, execution of partial applications is also supported.

Since applications are assumed to be built from a small number of predefined components, no searching or selecting procedures are provided.

Summary

AgentSheets is the most usable composition environment we surveyed. Through careful user interface design and a simple, restricted component model, the building of applications likens the touch-and-feel of moving pieces on a game board. This usability comes with a cost. Only application domains that can be represented as a two-dimensional grid with interaction rules based on adjacency rules are supported. While this is a non-trivial set, AgentSheets is not as generic a composition environment as some of the other composition environments we surveyed.

4.4 Software Reuse

Software reuse [Krueger 1992, *Software Reusability* 1994] is concerned with the problem of identifying reusable pieces of source code, storing those pieces in a repository, and enabling software developers to search for and retrieve relevant source code from the repository when they need it. Early reuse environ-

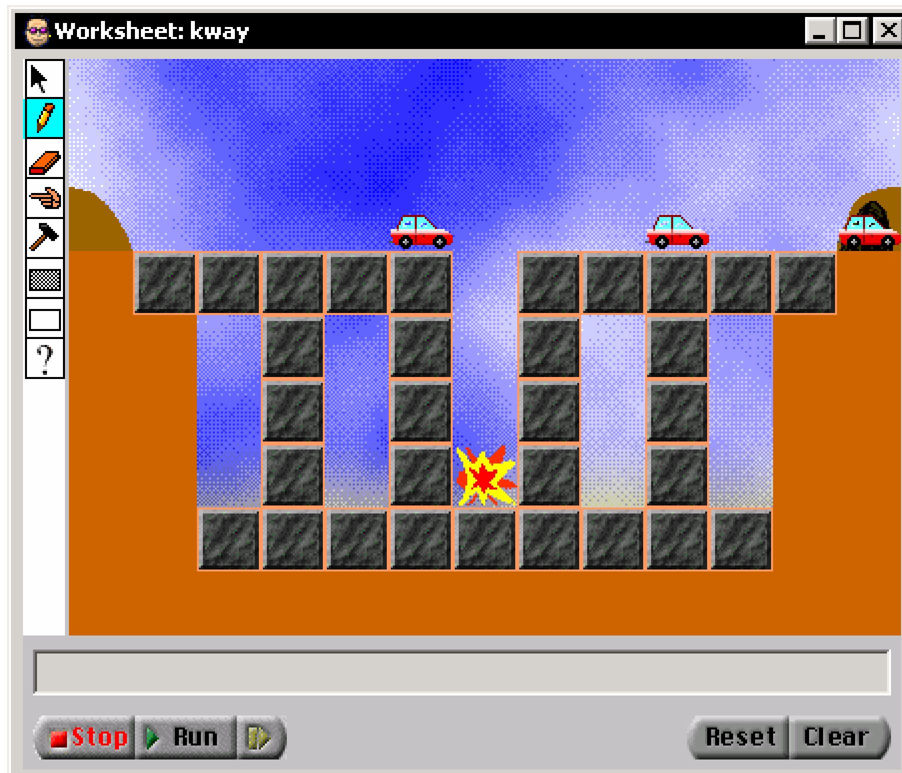

Figure 7. An AgentSheets Worksheet Simulating Cars on a Bridge.

ments primarily concerned themselves with managing small-grained, source code entities (e.g., individual procedures or small, related set of functions). Only recently, newer reuse environments have embraced the concept of component-level reuse [Aonix Select Component Manager]. While the task of actually composing applications is generally not within scope, reuse environments provide extensive support for searching and selecting components. To illustrate these capabilities from the perspective of composition, we discuss CodeBroker, a reuse environment that focuses on the task of automatically finding relevant components.

4.4.1 CodeBroker

CodeBroker [Ye and Fischer 2002] is a development environment that integrates autonomous delivery of task-relevant and personalized information from a reuse repository. When a developer sets on to write a method in the CodeBroker editor, the environment uses a machine-learning technique to check the natural-language comments and the signature of the method to look for similar methods in its reuse repository. Possible matches are presented in shorthand to the developer, but the developer can easily obtain more information about each of the alternative methods to judge their suitability.

CodeBroker adapts to the developer and task at hand by using predefined rules and machine-learning techniques to automatically adjust its behavior. Much of this automated behavior can be overridden by manual rules. For example, it is possible to specify that it should exclude from its search results those methods that the developer already uses (and thus presumably knows about).

The current prototype of CodeBroker is implemented as an extension to the Emacs text editor. As its component registry it uses the Java standard class library, and the standard Java documentation is used to search for potentially suitable methods.

Component Model

CodeBroker has an extremely limited component model. Deployable components are Java classes (once the environment identifies a suitable reusable method, it returns the whole class for inspection by the developers), but other basic concepts such as connections, interfaces, configurations, or connectors are not sup-

ported. Meta-data is available in the form of the Java documentation, but components are not self-description since the meta-data remains external to the component.

Composition Process

CodeBroker is tightly integrated with a component repository. The repository used is a very specific one (few components will have such good documentation available as the Java standard library), but the mechanisms applied are universal. Searching support is extensive and is largely performed on an automated basis. The user does not have to decide on search criteria or initiate searches, since this is done opportunistically by the environment.

No support for composition is provided, since it simply is not the focus of CodeBroker.

Summary

While not a composition environment in and of itself, CodeBroker demonstrates that the tasks of searching and selecting can be made much easier with advanced, automated support. In most composition environments, these tasks are performed manually, but it is clearly desired that advanced search and selection capabilities are tightly integrated.

Related Approaches

CodeBroker is one of the few reuse environments that bases its functionality on an integrated editor. Most other reuse environments focus on building repositories of available components that are accessed manually by an application composer. Most of these repositories nowadays reside on the Internet (e.g., Tucows [Tucows], ComponentSource [ComponentSource]). What distinguishes these repositories are their hierarchical classification schemes and search capabilities through which application composers can quickly find relevant components. Often, extensive meta-data is provided to further inform an application composer of the functionality of a component. Unfortunately, components are typically not self-descriptive; the meta-data is external and remains on the web site when a component is downloaded.

4.5 Component Frameworks

The last class of approaches we survey are component frameworks. These frameworks lay a strong basis for composition environments by not only specifying a comprehensive component model, but also providing an implementation and sometimes run-time framework with which deployable components can easily be implemented and executed.

Component frameworks are typically specified independently from any composition environment. As such, each becomes a standard and different composition environments can support the same component framework. We discuss two of the more popular component frameworks: Enterprise Java Beans and .NET.

4.5.1 Enterprise Java Beans

Enterprise Java Beans (EJB) [DeMichiel et al. 2001, Monson-Haefel 2001, Shannon 2001, Thomas 1998] is an extension of the Java Beans component model with the specific goal of supporting the creation of distributed client-server applications. In particular, EJBs are designed to represent the middle (business logic) tier of three-tier client-server applications, where the top tier consists of views on the client side, and the bottom tier consists of a database in which all application data is stored. To communicate with the top and bottom tiers, EJBs use Java Remote Method Invocations [Waldo 1998].

An Enterprise Java Beans container is a run-time environment that can execute Enterprise Java Beans. Examples of EJB containers are Bea WebLogic [BEA WebLogic Server], IBM WebSphere [IBM WebSphere Application Server], Oracle Application Server [Oracle9i Application Server], and JBoss [JBoss]. Each of these run-time containers provides an implementation of standardly specified services, such as transaction processing or object lifecycle management. This allows an EJB to be written independently of the container in which it will be run, yet benefit from the availability of these services.

Component Model

Components are deployable, because they can be used by the container without human intervention. The units of deployment are EJB Jar files, which contain one or more Enterprise Java Bean classes. However,

not EJB classes, but their instances are the units of composition by the EJB container. The container instantiates the classes and then connects them to form an application. Exactly which Enterprise Java Beans are connected to which other EJBs is determined by the application writer, and fully specified as particular dependencies at the time an EJB is deployed.

The syntactic self-description of Java Beans is extended by Enterprise Java Beans in the form of externally stored, XML-based deployment descriptors. For example, deployment descriptors exist that are used to differentiate between data types that have state and those that do not. Other deployment descriptors specify information about required data types that are not implemented in the same component.

Composite deployable components are supported, but only in a rudimentary way. In particular, to create a composite, the self-description of constituent deployable components has to be manipulated, which violates the principle of encapsulation. Neither connectors nor anticipated adaptation is supported.

Composition Process

Unlike the BeanBox (Section 4.3.3), EJB containers do not provide mechanisms for user composition. All configuration parameters are specified programmatically either inside the Enterprise Java Beans, or inside the container implementation. Nonetheless, environments similar to the BeanBox are starting to emerge [Aonix Select Component Manager], and over time it can be expected that composition of Enterprise Java Beans applications will become easier.

Summary

Enterprise Java Beans is an industrial-strength standard for composition environments for enterprise-scale client-server applications, and shows how a simpler environment (Java Beans on top of the Java platform) can be extended to include the needs of such applications. While the EJB specification is silent on the issue of usability, the aforementioned containers and additional tools help administrators deploy and manage their Enterprise Java Beans over time.

Related Approaches

Bean Bag [O'Reilly 1999] is a repository for Enterprise Java Beans that extends deployment descriptors by adding semantic and non-technical information. The CORBA Component Model (CCM) [CORBA 3.0 New Components Chapters 2001, Wang et al. 2001] is similar to the Enterprise Java Beans model. However, it is based on the CORBA platform for distributed objects instead of the Java platform, which makes it more universally employable because CORBA supports widely-used programming languages such as C++ and C.

4.5.2 .NET

.NET [Farley 2000-2001, Lauer 2001, Microsoft Dot-Net Framework FAQ 2001, Platt 2001, Pratschner 2000, Ricciuti 2001] is a new component framework that, similar to the Java platform, consists of a virtual machine (the “Common Language Runtime”) and a standard library. .NET represents an evolution of COM [Birngruber et al. 1999, Box 1997, The Component Object Model Specification 1995, Sullivan et al. 1999] and is an extension of the Win32 platform and the Microsoft Foundation Classes (MFC). It was announced in 2000, and is currently available as a beta version. One of the main design goals of the .NET framework is to extensively support interoperability among different web services.

Applications are composed out of “assemblies”, deployable components that have self-description. Unlike its predecessor Win32, .NET does not require a central registry in which all component information is stored. Instead, assemblies are autonomous and carry all necessary metadata with them. The framework interprets the metadata and composes an application as needed.

Component Model

Components are deployable, type-oriented, and include syntactic self-description. Self-description includes name and versioning information, description of the data types and resources provided by the component, a list of required components, and the security level needed to run the component. In particular, each deployable component has a globally unique identifier, in which versioning information is cryptographically encoded, so that updates can only be provided by the original manufacturer of the component. The identifiers

are organized in a global name space of interfaces, which has its roots in the domain name global name space.

No concept of connectors is present; since dependencies are specified through explicit component identities (as opposed to required interface types or ports). This has the advantage that one can assume that the dependencies are well-tested, and that type-compatibility is guaranteed. It has the disadvantage, however, that applications are very inflexible: they cannot be dynamically modified nor is it easy to statically replace a deployable component from one manufacturer with a deployable component from another. This problem is exacerbated because at this time it is unclear whether there will be any support for customization parameters or ad-hoc adaptation.

Composition Process

Composition in .NET is largely automatic. Upon loading a deployable component, its dependencies are resolved and the application is dynamically put together. In this process, it is important to distinguish private assemblies from shared assemblies. Private assemblies are used by only one application, and only need to be copied to the directory of the application in order to be used. Shared assemblies, on the other hand, can be used by more than one application, and must first be installed by using a simple command line tool ("gacutil.exe", the Global Assembly Cache Utility), which publishes its availability.

All further configuration of the application is done internally by accessing the self-description of the assemblies. While beneficial in terms of not requiring any user support, it is limited in not disclosing any information about the application to the user. Especially, there is no way to visualize the overall architecture of an application. All architectural information is encapsulated in the components and cannot be easily separated from them.

Summary

.NET is intended to be the future component framework of MS Windows. It attempts to unify the strengths of Win32, such as performance and backwards compatibility through version identification, with those of the Java Virtual Machine, such as platform independence and component deployability. In how far this will be successful still remains to be seen. From a composition point of view, however, .NET makes two important contributions: it is the first component framework with extensive and integral support for self-description, and it supports versions of distributed dependencies among deployable components.

Related Work

The Component Workbench is based on the use of a component meta-model that overlays those of .NET, CORBA, and others. Together with predefined connectors that can be used to bridge components written in accordance to those different component models (e.g., to hook up a .NET component with a CORBA component), this meta-model turns the Component Workbench into an environment for bridging different components frameworks.

5 Observations

Based on the discussions of Section 4 and the detailed survey data presented in Appendix A, we make a number of observations. First, it is clear that composition as a discipline is still in its early stages of development. While the desired features of composition environments (as discussed in Section 3) may be well-understood individually, actually putting them together to provide a comprehensive and fully-featured composition environment is a difficult and less-understood task. In fact, none of the composition environments discussed in Section 4 addresses all of the desired features at this moment in time, and each environment has limited itself to highlighting one or two capabilities (see Table 6).

Complicating the development of composition environments is the fact that different features can be implemented differently. For example, component interfaces may be supported through interface instances, interface types, or both at the same time. Similarly, a graphical or textual composition notation may be chosen (or both can be provided, introducing the need to keep them consistent). Even the notion of a deployable component is still under debate: different implementation strategies can be used that each have different characteristics and usage patterns. Although a natural tendency is to label, for each feature, one of its alternative implementations as "best" (e.g., best is to use instances and types to implement interfaces, or best is to use a graphical notation for composition), our survey shows that this cannot be done. Different choices have different ramifications. For example, forcing interfaces to be implemented using instances and types brings with it a need for application composers to understand typing mechanisms, something that may be too heavy weight for certain composition domains (consider elementary school education as supported by AgentSheets). Similarly, exclusive use of a graphical composition notation does not support the manipulation of application configurations by external tools (preventing, for example, automated installers such as InstallShield [InstallShield] to use the composition notation). Flexible composition environments are needed that can tailor their behavior to the desired features of its component model and composition process.

Table 6. Highlighted Capabilities in the Surveyed Composition Environments.

Composition Environment	***Highlighted Capability***
C2 / ArchStudio	Instance-oriented application design
Koala	Three-tiered architectural + component model + implementation approach
Pipe-and-Filter	Simple, quick composition of linear filters
UniCon	Explicit connectors
Plug-In Systems	Extreme extensibility
Java	Precisely specified rules of type compatibility among deployable components
Jiazzi	Script-based composition
Visual Basic	Robust, standard file format for distributing and reusing deployable components
VisualAge	Visual component-based programming
BeanBox	Extensible self-description
Vista	Extensible component model
AgentSheets	Extreme simplicity of composition of non-trivial applications
CodeBroker	Automated searching and selection
Enterprise Java Beans	Enterprise-scale composition with explicitly defined run-time containers
.NET	Pre-defined self-description including versioning

A second observation is that existing component environments typically fail to acknowledge the tension between usability and expressiveness. Expressiveness is the degree of choice an application composer has in developing their application. Usability determines the ease with which an application can be composed. Some composition environments are based on programming language technologies, and thus have a high degree of expressiveness, but also require a large amount of technical knowledge (for example, composition environments based on C# and Java such as .NET [Pratschner 2000] and Enterprise Java Beans

[DeMichiel et al. 2001]). Others focus on usability, but their expressiveness is severely restricted. For example, graphical user interface builders are intuitive to learn, but cannot be used to build anything except user interfaces.

The difference in focus between usability and expressiveness typically goes hand-in-hand with the focus of a composition environment on either its component model or its composition process. Those composition environments that are centered on their component model tend to be very expressive. An application composer has available a number of technological alternatives to compose an application. On the other hand, those composition environments that are centered on their composition process tend to be more usable. The task of composing an application becomes rather simple, but usually only a few restricted mechanisms are available to compose the application.

Composition environments should be as expressive as possible while being usable enough to require little technical skill. While this may be a difficult objective to achieve, environments such as AgentSheets, with its high usability yet ability to build different kinds of simulations, and VisualAge, with has maintained a high level of usability while supporting visual component-based application composition, offer much hope for a future in which high levels of both expressiveness and usability can be reached in a single composition environment. Again, we believe flexible composition environments are needed that adjust their behavior to the needs of the application composer. For example, an environment could offer simple drag-and-drop application composition as its default mode of operation, but also support, upon request, expert users in specifying wrappers, using scripts to compose applications, and leveraging other advanced features.

Our third observation is best illustrated with the help of Figure 8, which illustrates the historical relations among the various composition environments we surveyed. On the horizontal scale, composition environments are laid out by area. On the vertical scale, composition environments are laid out by approximate chronological sequence. An arrow from one composition environment to the next means that the first environment influenced the design of the second, or, somewhat stronger, that the second composition environment actually extends the first. The diagram clearly illustrates how software architecture as an area has been relatively isolated in its approach to building composition environments. With the exception of Koala, which was partially inspired by COM [The Component Object Model Specification 1995], other architectural approaches have largely been developed in isolation. Given the benefits that those architectural approaches provide (e.g., explicit connectors, instance-oriented application composition, architectural styles), combining these approaches with component platforms and visual programming environments has the potential of significantly enhancing the power of composition environments.

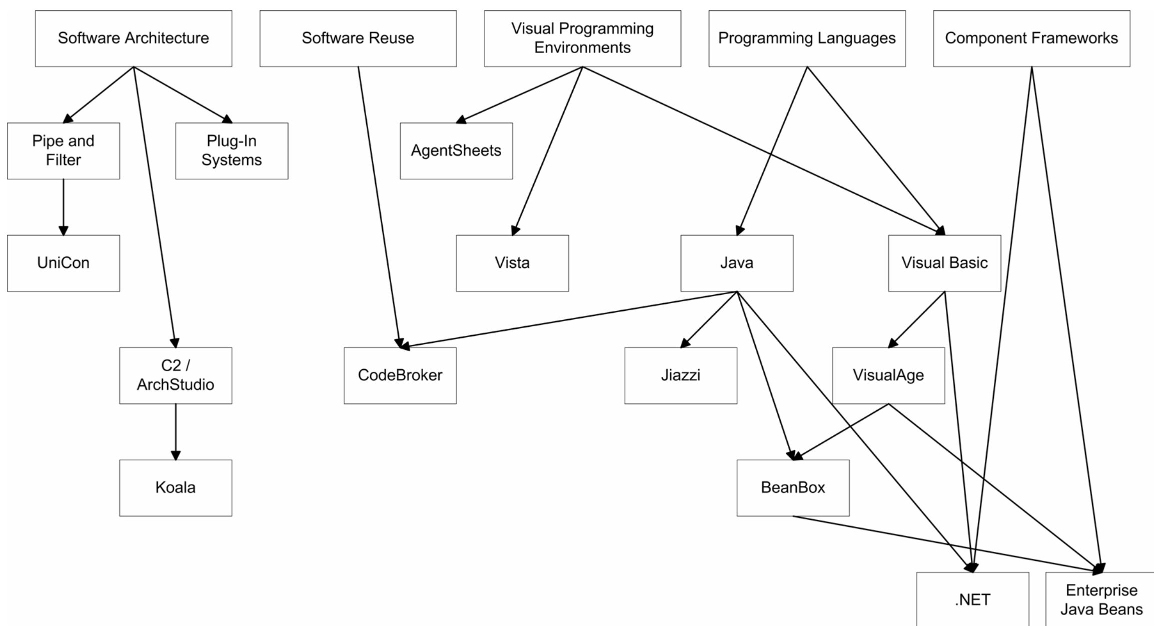

Figure 8. Historical Relations among Composition Environments.

Our fourth and final observation relates to self-description. While there is theoretical agreement on the importance of self-description for deployable components, features related to self-description are seldom found in current composition environments. We can only speculate why this is the case, but are encouraged to see newer composition environments such as Enterprise Java Beans and .NET increasingly adopt and support self-description. Given that deployable components will only be reused if the effort needed to understand their functionality is significantly less than the effort needed to reimplement that functionality, we consider further adoption of self-description essential for composition environments to succeed in their quest of fully separating component development from application composition.

6 Conclusions

Deployable components distinguish themselves by being prepackaged, independently distributed, easily installed and uninstalled, and self-descriptive. While a large amount of attention has been paid to "regular" components and their support environments (see, for example, Krueger [Krueger 1992], Mili et al. [Mili et al. 1995], Nierstrasz and Dami [Nierstrasz and Dami 1995], or Szyperski [Szyperski 1997]), this survey is a first at exclusively focusing on understanding composition environments for deployable components. Component themselves already provide many benefits from a development and reuse perspective, but we believe that deployable components have an even greater potential. In particular, similar to the way that the introduction of spreadsheets turned advanced numerical and financial calculations from a technical problem into a standard business application found on everybody's desktop, the introduction of powerful composition environments represents an important step towards turning component-based application composition from an activity that requires a solid technical background into a more mundane and business-oriented task.

For this to happen, much work remains to be done. As highlighted by the survey, composition environments are still in their relative infancy and a number of desired features are simply lacking, many are not adequately addressed, and a few are not well-understood until now. To address these deficiencies and bring composition environments into the mainstream, we propose a research agenda that simultaneously addresses each of the following four areas: developing fully-featured, integrated composition environments; comprehensively supporting each phase of composition process; supporting specification and leveraging of self-description; and scalability. We conclude this survey by discussing each of these areas below.

Integrated Composition Environments

As shown in Section 4, none of the composition environments to date integrates all desired features identified in Section 3. We consider such integration one of the greatest challenges for composition environments to succeed, especially when combined with a need to maintain a proper balance between usability and expressiveness (as identified in Section 5). Even though some of the existing approaches combine a relatively large number of features, they do so in a domain dependent way by, for example, focusing on a particular architectural style (e.g., C2 or pipe-and-filter). While these domain specific solutions are useful, there is a lack of universal, domain *independent* solutions in the area of composition environments.

The degree of universality that can be achieved remains an open question. Ideally, a composition environment covers all desired features in all of their incarnations and allows an application composer to configure the environment for their particular needs. More realistically however, we expect a small number of different classes of composition environments to emerge, each class making a different tradeoff between usability and expressiveness.

Koala [van Ommering et al. 2000] and .NET [Pratschner 2000] are perhaps the front runners in the evolution towards integrated environments. From a compositional point of view, both have succeeded in combining non-trivial feature sets while retaining a high level of usability. Koala in particular, with its architectural approach to composition, has managed to abstract many compositional details while still offering an application composer access to those details should that be necessary.

Composition Process

Compared to the large number of environments supporting the development of source code [JBuilder, Microsoft Visual Studio], the number of environments supporting deployable components is small. It should come as no surprise, then, that many composition environments for deployable components still resemble traditional development environments. The process that is followed in application composition, however, is radically different than the process that is followed for source code development. New environments are needed that integrally support the composition process as identified in Section 3.2. In particular, we expect new composition environments to integrally support searching and selecting components (for example, by maintaining a "component candidate" pool that informs a user of the suitability of each candidate component according to some measure), to provide advanced support for component adaptation (for example, by supporting many of the mechanisms laid out by Allen and Garlan [Allen and Garlan 1997]), to provide advanced composition and execution support (for example, by integrating architectural differencing and merging techniques for managing run-time patches [van der Hoek et al. 2001]), and to leverage advanced

self-description as much as possible (for example, by informing a composer of pricing policies or known component limitations).

Other process-related questions arise when one considers requirements elicitation, testing, and maintenance. All have been shown to present new problems in the context of deployable components [Brownsword et al. 2000, Orso et al. 2000, Voas 1998]. We believe a well-supported composition process with well-defined steps, such as the one introduced in Section 3.2, will provide a framework along which to tackle these difficult issues.

Wren [Lüer and Rosenblum 2001], our own composition environment, and the Component Workbench [Oberleitner and Gschwind 2002], a composition environment developed at Technische Universität Wien, are both specifically designed for deployable component composition. Both are unique in focusing on actual composition rather than development, and the process as intrinsically supported by each environment is indicative of that fact.

Self-Description

In their paper, Weinreich and Sametinger [Weinreich and Sametinger 2001] propose several criteria that a component model should fulfill: it should contain standards for interfaces, naming, metadata, interoperability, adaptation, composition, and packaging. Left out, interestingly, is self-description. Many other approaches to deployable component composition make the same mistake and only assume the availability of simple, syntactic interfaces that are automatically interpreted during the composition process. In the real world, however, automated techniques will not be able to fulfill all necessary functionality and human interpretation of rich metadata will be required. It is pertinent, therefore, that much attention be paid to all aspects of self-description: what kinds of information can be provided in a self-descriptive manner; how much information should be provided at each level; how can we not overload an application composer with too much self-description; how can self-description be leveraged in the composition process, both by automated tools and human interpretation; and what is the proper role of self-description in the composition process?

Promising approaches to self-description are provided by such composition environments as the BeanBox, Enterprise Java Beans and .NET. In particular .NET, whose identifiers encode the source IP address, a timestamp of release date and time, and versioning information, and the BeanBox, which supports the specification of arbitrary metadata that is accessed through introspection, provide promising platforms upon which to enhance support for self-description in composition environments.

Scalability

Last but not least, scalability is a key and crosscutting concern in the design of composition environments. Many surveyed composition environments work fine on a small scale, but when a large number of components are involved, they become unusable. Many scalability issues have been solved at the programming language level (for instance, through the use of namespaces, encapsulation, and hierarchical composition), but those solutions have not been ported as widespread to composition environments as they should be.

Additionally, a number of other techniques such as design patterns [Gamma et al. 1995] [Schmidt et al. 1996] and application generators (e.g., Genvoca [Batory and O'Malley 1992]) have successfully been used to address the inherent complexities of source code component composition. Their application to deployable components seems particularly attractive from the perspective of scale, especially when combined with hierarchical composition of deployable components.

Acknowledgements

We thank George Heineman for his many thoughtful suggestions that have shaped this survey.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-00-2-0599 and F30602-00-2-0607. Effort also funded by the National Science Foundation under grant number CCR-0093489. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

References

Abowd, Gregory D., Robert Allen and David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology* 4, 4 (1995), 319-364.

Achermann, Franz and Oscar Nierstrasz. Explicit Namespaces. In *Joint Modular Languages Conference 2000.* Springer, Berlin, 2000, 77-89.

Adobe Photoshop: Application Programming Interface Guide. Version 6.0 Release 1. http://partners.adobe.com/asn/developer/gapsdk/download/win/Photoshop60sdk.zip. 2000.

AgentSheets. http://agentsheets.com. 2002.

Aldrich, Jonathan, Craig Chambers and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *24th International Conference on Software Engineering.* ACM, New York, 2002, 187-197.

Allen, Robert and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology* 6, 3 (1997), 213-249.

Aonix Select Component Manager. http://www.aonix.com/content/products/select/select_compman.html.

The Apache HTTP Server Project. http://httpd.apache.org/.

ArchStudio 3. http://www.isr.uci.edu/projects/archstudio/.

Batory, Don and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology* 1, 4 (1992), 355-398.

BEA WebLogic Server. Version 6.1. http://www.bea.com/products/weblogic/server/index.shtml.

Beugnard, Antoine, Jean-Marc Jézéquel, Noël Plouzeau and Damien Watkins. Making Components Contract Aware. *Computer* 32, 7 (1999), 38-45.

Birngruber, Dietrich, Werner Kurschl and Johannes Sametinger. Comparison of JavaBeans and COM/ActiveX - A Case Study. In *5. Fachkonferenz Smalltalk und Java.* Erfurt, 1999.

Borgida, Alex and Prem Devanbu. Adding More "DL" to "IDL": Towards More Knowledgeable Component Inter-Operability. In *Proceedings of the 1999 International Conference on Software Engineering.* ACM, New York, 1999, 378-387.

Borland Delphi. Version 6. http://www.borland.com/delphi/.

Bosch, Jan. Adapting Object-Oriented Components. In *Object-Oriented Technology.* Springer, Berlin, 1998, 379-383.

Box, Don. *Essential COM.* Addison-Wesley, Reading, 1997.

Brockschmidt, Kraig. Developing Applications with OLE 2.0. http://msdn.microsoft.com/library/en-us/dnolegen/html/msdn_devwole2.asp. 1994.

Brownsword, Lisa, Tricia Oberndorf and Carol A. Sledge. Developing New Processes for COTS-Based Systems. *IEEE Software* 17, 4 (2000), 48-55.

Cardelli, Luca and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17, 4 (1985), 471-522.

The Component Object Model Specification. Version 0.9. http://www.microsoft.com/com/resources/comdocs.asp. 1995.

ComponentSource. http://componentsource.com/.

Conradi, Reidar and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys* 30, 2 (1998), 232-282.

CORBA 3.0 New Components Chapters. ftp://ftp.omg.org/pub/docs/ptc/01-11-03.pdf. 2001.

Councill, Bill and George T. Heineman. Definition of a Software Component and Its Elements. In *Component-Based Software Engineering.* Addison-Wesley, Boston, 2001, 5-19.

Crnkovic, Ivica, Heinz Schmidt, Judith Stafford and Kurt Wallnau. Anatomy of a Research Project in Predictable Assembly. (2002).

Dashofy, E. M., N. Medvidovic and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 3-12.

DeMichiel, Linda G., L. Ümit Yalçinalp and Sanjeev Krishnan. Enterprise Java Beans Specification, Version 2.0. http://java.sun.com/products/ejb/docs.html. 2001.

DeRemer, Frank and Hans H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2, 2 (1976), 80-86.

Di Nitto, Elisabetta and David Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 1999 International Conference on Software Engineering.* ACM, New York, 1999, 13-22.

Drossopoulou, Sophia, David Wragg and Susan Eisenbach. What is Java Binary Compatibility? *Sigplan Notices* 33, 10 (1998), 341-358.

Duncan, Andrew and Urs Hölzle. *Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines.* Technical Report TRCS99-09. University of California, Santa Barbara, Santa Barbara, 1999.

Euler, Laura, Eric Maffei and Adam Rauch. Create Real Windows Applications in a Graphical Environment Using Microsoft Visual Basic. *Microsoft Systems Journal* 6, 4 (1991), 57-70, 116.

Farley, Jim. Microsoft Dot-Net vs. J2EE: How Do They Stack Up? http://java.oreilly.com/news/farley_0800.html. 2000-2001.

Findler, Robert Bruce, Mario Latendresse and Matthias Felleisen. Behavioral Contracts and Behavioral Subtyping. *Software Engineering Notes* 26, 5 (2001), 229-236.

Flatt, Matthew. *Programming Languages for Reusable Software Components.* PhD Thesis. Rice University, Houston, 1999.

Franz, Michael. Dynamic Linking of Software Components. *Computer* 30, 3 (1997), 74-81.

freshmeat.net. http://freshmeat.net/.

Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, 1995.

Garlan, David. Higher-Order Connectors. In *Workshop on Compositional Software Architectures.* Monterey, 1998.

Gorlick, Michael and Alex Quilici. Visual Programming-in-the-Large versus Visual Programming-in-the-Small. In *Proceedings of 1994 IEEE Symposium on Visual Languages.* IEEE, Los Alamitos, 1994, 137-144.

Gosling, James, Bill Joy and Guy Steele. The Java Language Specification. http://java.sun.com/docs/books/jls/html/index.html. 1996.

Heineman, George T. Buidling Instead of Buying: A Rebuttal. In *Component-Based Software Engineering.* Addison-Wesley, Boston, 2001, 479-483.

Heineman, George T. and Helgo M. Ohlenbusch. An Evaluation of Component Adaptation Techniques. In *1999 International Workshop on Component-Based Software Engineering.* 1999. http://www.sei.cmu.edu/cbs/icse99/papers/.

Henderson-Sellers, Brian. An OPEN Process for Component-Based Development. In *Component-Based Software Engineering*. Addison-Wesley, Boston, 2001, 321-340.

van der Hoek, André, Richard S. Hall, Dennis Heimbigner and Alexander L. Wolf. Software Release Management. In *Proceedings of the Sixth European Software Engineering Conference*. Springer, Berlin, 1997, 159-175.

van der Hoek, André, Marija Rakic, Roshanak Roshandel and Nenad Medvidovic. Taming Architectural Evolution. *Software Engineering Notes* 26, 5 (2001), 1-10.

Hölzle, Urs. Integrating Independently-Developed Components in Object-Oriented Languages. In *ECOOP '93 - Object-Oriented Programming*. Springer, Berlin, 1993, 36-56.

IBM WebSphere Application Server. Version 4.0. http://www.ibm.com/software/webservers/appserv/.

InstallShield. http://www.installshield.com/.

Inverardi, Paola, Alexander L. Wolf and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology* 9, 3 (2000), 239-272.

Java 2 Platform, Standard Edition, v 1.4.0 API Specification. http://java.sun.com/j2se/1.4/docs/api/index.html. 2002.

Java Beans: API Specification, Version 1.01. http://java.sun.com/products/javabeans/docs/spec.html. 1997.

JBoss. http://jboss.org.

JBuilder. Version 6. http://www.borland.com/jbuilder/.

Joy, William. *An Introduction to the C Shell.* 4.3BSD User's Supplementary Documents. University of California, Berkeley, 1994.

Keller, Ralph and Urs Hölzle. *Implementing Binary Component Adaptation for Java.* Technical Report TRCS98-21. University of California, Santa Barbara, Santa Barbara, 1998.

Khare, Rohit. Internet-Scale Namespaces. In *The Workshop for Internet-Scale Technologies.* 1999. http://www.ics.uci.edu/~irus/twist/twist99/presentations/khare/ISN-Survey-Talk.pdf.

Krueger, Charles W. Software Reuse. *ACM Computing Surveys* 24, 2 (1992), 131-183.

Larsson, Magnus and Ivica Crnkovic. Component Configuration Management. In *Proceedings of 5th Workshop on Component Oriented Programming.* 2000.

Lauer, Christopher. Introducing Microsoft Dot-Net. http://www.dotnet101.com/articles/art014_dotnet.asp. 2001.

Liang, Sheng and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. *Sigplan Notices* 33, 10 (1998), 36-44.

Lüer, Chris and David S. Rosenblum. Wren—An Environment for Component-Based Development. *Software Engineering Notes* 26, 5 (2001), 207-217.

Lüer, Chris, David S. Rosenblum and André van der Hoek. The Evolution of Software Evolvability. In *International Workshop on Principles of Software Evolution (IWPSE 2001).* Vienna, 2001, 127-130.

Lycett, Mark and Ray J. Paul. Component-Based Development: Dealing with Non-Functional Aspects of Architecture. In *ECOOP '98 Workshop on Component-Oriented Programming.* 1998. http://www.abo.fi/~Wolfgang.Weck/WCOP/98/Papers/.

Magee, Jeff and Jeff Kramer. Dynamic Structure in Software Architectures. *Software Engineering Notes* 21, 6 (1996), 3-14.

Maurer, Peter M. Components: What If They Gave a Revolution and Nobody Came? *Computer* 33, 6 (2000), 28-34.

McDirmid, Sean, Matthew Flatt and Wilson C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 2001.

McIlroy, M. D. Mass Produced Software Components. In *Software Engineering.* Mason/Charter, New York, 1976.

Medvidovic, Nenad, Peyman Oreizy, Richard N. Taylor, Rohit Khare and Michael Guntersdorfer. *An Architecture-Centered Approach to Software Environment Integration.* Technical Report UCI-ICS-00-11. University of California, Irvine, Irvine, 2000.

Medvidovic, Nenad, David S. Rosenblum, David F. Redmiles and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology* 11, 1 (2002), 2-57.

Medvidovic, Nenad, David S. Rosenblum and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering.* ACM, New York, 1999, 44-53.

Medvidovic, Nenad, David S. Rosenblum and Richard N. Taylor. *A Type Theory for Software Architectures.* Technical Report UCI-ICS-98-14. University of California, Irvine, Irvine, 1998.

Medvidovic, Nenad and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26, 1 (2000), 70-93.

Mehta, Nikunj R., Nenad Medvidovic and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 2000 International Conference on Software Engineering.* ACM, New York, 2000, 178-187.

de Mey, Vicki. Visual Composition of Software Applications. In *Object-Oriented Software Composition.* Prentice Hall, London, 1995, 275-304.

de Mey, Vicky and Simon Gibbs. A Multimedia Component Kit. In *Proceedings of the 2nd ACM International Conference on Multimedia.* 1994, 299-306.

Microsoft Dot-Net Framework FAQ. http://msdn.microsoft.com/library/en-us/dndotnet/html/faq111700.asp?frame=true. 2001.

Microsoft Visual Basic. Version Dot-Net. http://msdn.microsoft.com/vbasic/.

Microsoft Visual Studio. Version Dot-Net. http://msdn.microsoft.com/vstudio/.

Mikhajlov, Leonid and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP '98.* Springer, Berlin, 1998, 355-382.

Mili, Hafedh, Fatma Mili and Ali Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering* 21, 6 (1995), 528-562.

Monroe, Robert T. *Capturing Software Architecture Design Expertise with Armani.* Technical Report CMU-CS-98-163. Carnegie Mellon University, Pittsburgh, 1998.

Monson-Haefel, Richard. *Enterprise JavaBeans.* O'Reilly, Sebastopol, 2001.

Nierstrasz, Oscar and Laurent Dami. Component-Oriented Software Technology. In *Object-Oriented Software Composition.* Prentice Hall, London, 1995, 3-28.

Notkin, David, David Garlan, William G. Griswold and Kevin Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Object Technologies for Advanced Software.* Springer, Berlin, 1993, 489-510.

Oberleitner, Johann and Thomas Gschwind. *Composing Distributed Components with the Component Workbench.* Technical Report TUV-1841-02-17. Technische Universität Wien, Wien, 2002.

Ohlenbusch, Helgo and George T. Heineman. *Complex Ports and Roles within Software Architecture.* Technical Report WPI-CS-TR-98-12. Computer Science Department, Worcester Polytechnic Institute, Worcester, 1998.

Ólafsson, Ásgeir and Doug Bryan. On the Need for "Required Interfaces" of Components. In *Special Issues in Object-Oriented Programming.* Dpunkt, Heidelberg, 1997, 159-165.

OMG Unified Modeling Language Specification. Version 1.4. http://www.omg.org/cgi-bin/doc?formal/01-09-67.pdf. 2001.

van Ommering, Rob. Building Product Populations with Software Components. In *24th International Conference on Software Engineering.* ACM, New York, 2002, 255-265.

van Ommering, Rob, Frank van der Linden, Jeff Kramer and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer* 33, 3 (2000), 78-85.

Oracle9i Application Server. Version 1.0.2.2. http://www.oracle.com/ip/deploy/ias/.

O'Reilly, Caroline. *BeanBag: An Extensible Framework for Describing, Storing and Querying Components.* MS Thesis. University of Dublin, Dublin, 1999.

Oreizy, Peyman, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

Oreizy, Peyman, Nenad Medvidovic and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 1998 International Conference on Software Engineering.* IEEE, Los Alamitos, 1998.

Orso, Alessandro, Mary Jean Harrold and David S. Rosenblum. Component Metadata for Software Engineering Tasks. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000).* Springer, Berlin, 2000, 126-140.

Ousterhout, John K. Scripting: Higher-Level Programming for the 21st Century. *Computer* 31, 3 (1998), 23-30.

Perry, Dewayne E. and Alexander L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes* 17, 4 (1992), 40-52.

Piersol, Kurt. A Close-Up of OpenDoc. *Byte* 19, 3 (1994), 183.

Plásil, František and Michael Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software - Concepts & Tools* 19, (1998), 14-28.

Platt, David S. *Introducing Microsoft Dot-Net.* Microsoft, Redmond, 2001.

Plug-in Guide. http://developer.netscape.com/docs/manuals/communicator/plugin/pgpr.pdf. 1998.

Postel, Jon. *Domain Name System Structure and Delegation.* Request for Comments 1591. Internet Engineering Task Force, 1994.

Pratschner, Steven. Simplifying Deployment and Solving DLL Hell with the Dot-Net Framework. http://msdn.microsoft.com/library/techart/dplywithnet.htm. 2000.

Repenning, Alexander and Tamara Sumner. Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *Computer* 28, 3 (1995), 17-25.

Ricciuti, Mike. Strategy: Blueprint Shrouded in Mystery. http://news.cnet.com/news/0-10003-201-7502765-0.html. 2001.

Robben, Bert, Frank Matthijs, Wouter Joosen, Bart Vanhaute and Pierre Verbaeten. Components for Non-Functional Requirements. In *Object-Oriented Technology.* Springer, Berlin, 1998, 151-152.

Robbins, Jason E. and David F. Redmiles. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology* 42, (2000), 79-89.

Robbins, Jason E. and David F. Redmiles. Software Architecture Critics in the Argo Design Environment. *Knowledge-Based Systems* 11, 1 (1998), 47-60.

Rosenblum, David S. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering* 21, 1 (1995), 19-31.

Rosenblum, David S. and Rema Natarajan. Supporting Architectural Concerns in Component-Interoperability Standards. *IEE Proceedings-Software* 147, 6 (2000), 215-223.

Schmidt, Douglas C., Mohamed Fayad and Ralph E. Johnson. Software Patterns. *Communications of the ACM* 39, 10 (1996), 36-39.

Shannon, Bill. Java 2 Platform Enterprise Edition Specification. Version 1.3. http://java.sun.com/j2ee/docs.html. 2001.

Shaw, Mary. Architectural Issues in Software Reuse: It's not Just the Functionality, It's the Packaging. In *Symposium on Software Reusability.* 1995, 3-6.

Shaw, Mary, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* 21, 4 (1995), 314-335.

Shaw, Mary and David Garlan. *Software Architecture.* Prentice Hall, Upper Saddle River, 1996.

Shu, Nan C. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.

Singhal, Sandeep and Binh Nguyen. The Java Factor. *Communications of the ACM* 41, 6 (1998), 34-37.

Software Reusability. Ellis Horwood, New York, 1994.

SourceForge.net. http://sourceforge.net/.

Sullivan, Kevin J., Mark Marchukov and John Socha. Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model. *IEEE Transactions on Software Engineering* 25, 4 (1999), 584-599.

Szyperski, Clemens. *Component Software.* ACM, New York, 1997.

Taylor, Richard N., Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22, 6 (1996), 390–406.

Thomas, Anne. Enterprise Java Beans Technology. 1998.

Traas, Vincent and Jos van Hillegersberg. The Software Component Market on the Internet: Current Status and Conditions for Growth. *Software Engineering Notes* 25, 1 (2000), 114-117.

Tucows. http://tucows.com/.

Udell, Jon. Componentware. *Byte* May (1994), 46-56.

Uniform Resource Locators. Request for Comments 1738. Internet Engineering Task Force, 1994.

VisualAge for Java. Version 4.0. http://www.ibm.com/software/ad/vajava/.

VisualAge: Concepts and Features. IBM Red Book GG24-3946-00. IBM Corporation, Boca Raton, 1994.

Voas, Jeffrey. Maintaining Component-Based Systems. *IEEE Software* 15, 4 (1998), 22-27.

Waldo, Jim. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency* 6, 3 (1998), 5-7.

Wang, Nanbor, Douglas C. Schmidt and Carlos O'Ryan. Overview of the Corba Component Model. In *Component-Based Software Engineering.* Addison-Wesley, Boston, 2001, 557-571.

Weerawarana, Sanjiva, Francisco Curbera, Matthew J. Duftler, David A. Epstein and Joseph Kesselman. Bean Markup Language: A Composition Language for JavaBeans Components. In *Proceedings of*

the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS '01). Usenix, Berkeley, 2001, 173-187.

Weinreich, Rainer and Johannes Sametinger. Component Models and Component Services: Concepts and Principles. In *Component-Based Software Engineering.* Addison-Wesley, Boston, 2001, 33-48.

Whitehead, E. James, Jason E. Robbins, Nenad Medvidovic and Richard N. Taylor. Software Architecture: Foundation of a Software Component Marketplace. In *Proc. First International Workshop on Architectures for Software Systems.* ACM, New York, 1995, 276-282.

Yacoub, Sherif, Hany Ammar and Ali Mili. Characterizing a Software Component. In *1999 International Workshop on Component-Based Software Engineering.* 1999, 133-138.

Ye, Yunwen and Gerhard Fischer. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *24th International Conference on Software Engineering.* ACM, New York, 2002, 513-523.

Yellin, Frank and Tim Lindholm. *The Java Virtual Machine Specification.* Addison Wesley, 1998.

Zelesnik, Gregory. The UniCon Language Reference Manual. http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/Reference_Manual_1.html. 1996.

Appendix A

Tables 7-9 present the detailed survey data as gathered from publications, web sites, and trial use of the composition environments we surveyed. Each row corresponds to one feature from the comparison framework defined in Section 3, and each column represents one of the composition environments discussed in Section 4.

Each table cell contains either a description of the solution, or one of the following values:

+++	feature is fully supported
++	feature is present but not fully covered in all of its aspects
+	feature is present but only supported in a limited fashion
(empty)	feature is not present

Other abbreviations used:

prov.	provided ports
req.	required ports
dia.	diagrams
prog.	programming interface
scr.	scripts
non-hier.	non-hierarchically

Table 7. Detailed Survey Data (Part I).

	C2/ArchStudio	Koala	Pipe-and-Filter	UniCon	Plug-in Systems
Deployable component	++		+	+	+
Implementation strategy	process	procedure library	process	procedure library, process	procedure library
Types and instances	+++				
Global identity		+++			
Versioning	+	+++		+	
Interface		+		+	
Types and instances	prov , req	prov, req	prov, req	prov, req	prov, req
Global identity		random id			
Multiplicity	1	any	1	any	1
Versioning	+			+	
Interface location		referenced			
Configuration	++	++		++	
Composite components		non-hier.		non-hier.	
Connection semantics	event	type	stream	various	type
Connection multiplicity	n-n	1-n	1-1	various	1-n
Connectors	+++	+++	+++	+++	
Connector types	user-def.			6	
Customization parameters				+	
Self-description					
Syntactic		+		+	
Semantic					
Quality-of-service					
Non-technical					
Search					
Remote search					
Select					
Adapt					
Compose	++	++	++	++	
Composition notation	scr, dia	scr, dia	scr, dia	scr	scr
Constraints	+				
Guaranteeing consistency	on-the-fly, analysis			on-the-fly, analysis	
Distributed applications	+				
Execute	++	++	++	++	++
Partial applications	++		+		
Packaging	+	+	++	+	
Run-time changes	++				
Leverage self-description					
Syntactic					
Semantic					
Quality-of-service					
Non-technical					

Table 8. Detailed Survey Data (Part II).

	Java	Jiazzi	Visual Basic	Visual Age	Bean Box
Deployable component	++	+	+	++	++
Implementation strategy	class	class library	procedure library	object	object
Types and instances					
Global identity					
Versioning	+		+		
Interface		++			
Types and instances	prov	prov, req			prov
Global identity	namespace	namespace			
Multiplicity	any	any			any
Versioning					
Interface location	referenced	referenced			referenced
Configuration		+			+
Composite components		non-hier.			
Connection semantics	type	type	event	event	event
Connection multiplicity	1-n	1-1	n-n	1-n, 1-1	1-n, 1-1
Connectors	+++	+++		+++	+++
Connector types	.			3	2
Customization parameters			+	+	+
Self-description				+	+
Syntactic	+		+	++	++
Semantic				+	+
Quality-of-service					
Non-technical					
Search			+	+	
Remote search					
Select			+	+	
Adapt					
Compose		+		+	+
Composition notation	prog	scr	prog, dia	prog, dia	prog, dia
Constraints					
Guaranteeing consistency	runtime	analysis	on-the-fly	on-the-fly	on-the-fly
Distributed applications	+				
Execute	++		++	++	++
Partial applications	++		++	++	++
Packaging	+	+	+	+	+
Run-time changes					+
Leverage self-description				+	+
Syntactic			+	+	+
Semantic				+	+
Quality-of-service					
Non-technical					

Table 9. Detailed Survey Data (Part III).

	Vista	AgentSheets	Code Broker	EJB	Dot-Net
Deployable component	++	+		++	++
Implementation strategy	user-defined	object		class	procedure library
Types and instances					
Global identity					+++
Versioning				+	+
Interface				++	++
Types and instances	prov, req	prov, req		prov	prov
Global identity				namespace	random ID
Multiplicity	any	4		any	any
Versioning				+	
Interface location				referenced	copied
Configuration	++	++		+	+
Composite components	non-hier.			non-hier.	non-hier.
Connection semantics	user-def.	event		type	type
Connection multiplicity	user-def.	1-1		1-n	1-n
Connectors	+++	+++			
Connector types	user-def.				
Customization parameters	+	+		+	+
Self-description	+			+	+
Syntactic	+	+		++	++
Semantic				+	+
Quality-of-service					
Non-technical				+	+
Search			++		
Remote search					
Select			++		
Adapt					
Compose	++	++			
Composition notation	scr, dia	dia		prog	prog
Constraints	+				
Guaranteeing consistency	on-the-fly			runtime	runtime
Distributed applications				+	+
Execute	++	++			
Partial applications	+	+		+	+
Packaging				+	+
Run-time changes		+			
Leverage self-description			+		
Syntactic	+	+	++		
Semantic			++		
Quality-of-service					
Non-technical					