ICS

TECHNICAL REPORT

Environments for Deployable Components

Chris Lüer

Department of Information and Computer Science
University of California, Irvine

Technical Report #02-15

May 14, 2002

Information and Computer Science
University of California, Irvine

Environments for Deployable Components

Chris Lüer

Department of Information and Computer Science
University of California, Irvine

Technical Report #02-15

May 14, 2002

Abstract

Deployable components are software components that can easily be installed and uninstalled. This usually means that they are compiled and do not consist of source code. Composition environments enable the use of deployable components by making it possible to compose applications out of deployable components and to execute those applications. A composition environment is based on a component model, and consists of a set of tools to support the composition process performed by a user. We describe the basic features of composition environments, and survey a representative selection of environments from research and industry. We conclude that the domain of composition environments is not well understood yet, and point out possible directions of future research.

Contents

ACKNOWLEDGEMENTS 4

1 INTRODUCTION 5

2 DEFINITIONS 8

2.1 COMPONENT 8
2.2 COMPONENT MODEL 8
2.3 COMPOSITION ENVIRONMENT 9
2.4 CONNECTION 9
2.5 CONFIGURATION 9
2.6 INTERFACE 10

3 FEATURES 11

3.1 COMPONENT MODEL FEATURES 12
3.1.1 Components *12*
3.1.2 Interfaces *14*
3.1.3 Self-Description *15*
3.1.4 Configuration *16*
3.2 PROCESS LEVEL 18
3.2.1 Searching *19*
3.2.2 Leveraging Component Self-Description *19*
3.2.3 Connecting and Adapting *19*
3.2.4 Executing *22*

4 APPROACHES 23

4.1 SOFTWARE ARCHITECTURE 23
4.1.1 C2 / Archstudio *23*
4.1.2 Koala *25*
4.1.3 Unicon *25*
4.2 PROGRAMMING LANGUAGES 26
4.2.1 Java Platform *26*
4.2.2 Jiazzi *27*
4.3 VISUAL PROGRAMMING ENVIRONMENTS 28
4.3.1 Visual Basic *29*
4.3.2 Visual Age *30*
4.3.3 Bean Box *32*
4.3.4 Vista *33*
4.4 SOFTWARE REUSE 33
4.4.1 Code Broker *34*
4.4.2 Agent Sheets *34*
4.5 COMPUTER NETWORKS AND INTEROPERABILITY 35
4.5.1 Enterprise Java Beans Containers *36*
4.5.2 The Worldwide Web *37*

4.6 OPERATING SYSTEMS 38
4.6.1 Pipe and Filter *38*
4.6.2 MS Windows / Win 32 ..... *39*
4.6.3 MS Windows / Dot-Net Framework *39*
4.7 PLUG-IN SYSTEMS 40
4.8 COMPARISON 41
5 CONCLUSIONS **45**
6 RESEARCH PLAN **47**
BIBLIOGRAPHY **50**

Acknowledgements

Many thanks to André van der Hoek, who provided a lot of useful advice.

1 Introduction

Component-based software engineering is a well-established discipline in industry. Components written in Java, Visual Basic, or C++ are widely available for a variety of domains [128]. However, the "component revolution" has happened without much participation by researchers [66]. As a consequence, research and practice in component-based software engineering are largely separated. This has led to a situation in which widely-used tools are not based on research results as much as they could be. For example, connectors have been recognized by researchers as a useful means for decoupling components [116], but they are rarely employed in industrial systems.

Existing approaches in component technology focus either on expressiveness or on usability, but few systems achieve both. Expressiveness is the degree of choice the developer has when writing a program. An assembler is the most expressive tool for most computers; any program that is possible in a given hardware can be implemented using an assembler. But assemblers provide a minimum level of usability. On the other hand, user applications with graphical interfaces that fulfill a single, well-defined task (such as a simple text editor) are maximally usable, but not very expressive. Component environments should be as expressive as possible while being usable enough to require little technical skill.

Existing component environments typically fail to acknowledge this tension between usability and expressiveness. Some of them 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, technologies based on C++ and Java such as COM [15] and Enterprise Java Beans [26]). 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.

One of the main benefits anticipated for component-based software engineering is its ability to support division of labor (see Fig. 1, based on [132]). While in the traditional development scenario two groups of stakeholders exist, developers and users, in the component based scenario three groups exist: component developers, application builders, and users. The component developer focuses on the domain of the component, which is usually technical in nature. The application builder, who wants to compose an application out of components manufactured by component developers, focuses on the domain of the application, which is usually more business oriented. In other words, technical experts develop components, while business experts use these components to compose business applications.

However, in reality, application composition requires so much technical knowledge that the desired division of labor remains elusive. Application builders still need to be versed in programming technologies in order to understand components and compose applications from them. Composition environments that help application builders to compose industrial quality applications from components manufactured by component developers are needed. Similar to the way that spreadsheets turned calculations involving large sets of numbers and formulas from a technical problem into a standard business application found on everybody's desktop, composition environments could turn component-based

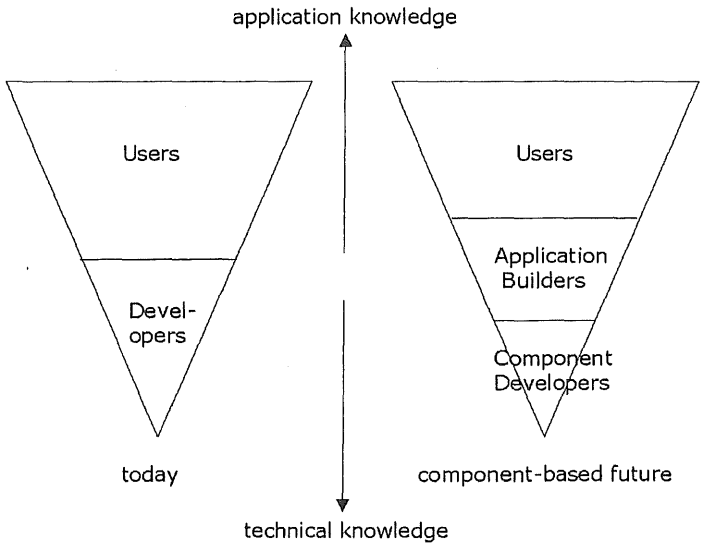

Figure 1. Traditional and component-based development scenarios.

software engineering from an activity that requires a solid technical background in software development into a business-oriented task.

In this paper, we survey technologies for composition environments. We discuss both the underlying component models, and features to support the composition process. The component model determines what kinds of components can be used and how they can be composed, and thus influences the expressiveness of the environment. Process properties determine how the user interacts with components and composed applications, and thus influence the usability of the environment.

While a large amount of research about software components and environments for them has been done before [56, 81], the bulk of this work was done on source code components. However, in order to make end-user composition possible, components should be deployable. Deployable components are software components that can be installed and uninstalled without complex technical procedures. Generally, this means that deployable components are available in compiled form, and can be used without any modifications.

A number of existing environments provide functionality that can be used to compose applications from deployable components, but the systems are varied, their solutions are unsystematic, and they address the problem only partially. Composition is typically not their main focus. In this survey, we will point out the commonalties and differences between existing composition environments, and systematize the underlying technologies in order to gain insights into the design of such environments. We will discuss selected environments from a variety of research areas, such as software architecture, visual programming, and operating systems.

Section 2 of this survey precisely defines several important terms of component technology. In Section 3, we define and categorize relevant features of composition environments. Section 4 describes a representative selection of composition environments, and

presents a detailed comparison of functionality. Section 5 contains our conclusions, and Sections 6 points out possible directions of future research.

2 Definitions

We are giving definitions for several important terms that are used throughout this paper. While most of these terms are commonly used in software engineering, their precise meanings often vary. We selected definitions that seem appropriate for the purposes of this paper, which are discussion and comparison of existing composition environments.

2.1 Component

There is a large amount of literature comparing definitions of "component" [16, 18, 24, 44, 55, 83, 122]. Many of these definitions suffer from one or more of the following drawbacks:

- *Technological bias.* The definition is based on a single technology, and defines a component as whatever is a convenient unit of reuse in that technology. For example, in object-oriented programming, convenient units would be either objects or classes.
- *Domain bias.* The definition is 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.* While a certain amount of vagueness seems to be unavoidable if the definition is to be broad enough to cover different technologies, the definition should give clear criteria that can be used to distinguish components from other artifacts.

For these reasons, we decided on the definition given by Szyperski [122]. However, for the purposes of this paper, we remove his restriction that components cannot have persistent state, because many existing composition environments violate this condition. We believe that the resulting definition is not biased towards a technology or domain, and it stipulates the comparatively concrete criteria of deployability and reusability.

A component is:

- *A unit of third-party composition.* Components can be used to build an application by an organization that is different from the organization that developed them. In other words, components are reusable.
- *A unit of independent deployment.* Components can be deployed easily and individually. Components do not have to be compiled or otherwise manipulated in a non-trivial manner by the person reusing them. They have been specifically prepared for reuse.

2.2 Component Model

A component model is a standard that components adhere to. Without any kind of component model, it would be hard to make components interoperate. As a simple example, components that want to communicate with each other through remote procedure calls need to agree on a standard for passing parameters. What kind of standards a component model should exactly define is a matter of debate, and depends very much on the specific goals of the component model in question. 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 re-

strictive it is, the easier it is to develop components. Weinreich and Sametinger [138] propose several criteria that a component model should fulfill: it should contain standards for interfaces, naming, metadata, interoperability, adaptation, composition, and packaging. With the exception of interoperability, these component model features are among the ones covered below.

The term "component model" is most often used for the industrial component standards Com [21], Java Beans [49], and the Corba Component Model [23]. However, all of the composition environments surveyed in this paper include a component model of some kind, even though it may not be explicitly defined.

> A component model is a specification of the common features of a set of components that make it possible to compose these components into applications.

2.3 Composition Environment

Without a composition environment of some kind, components are useless. Since components are usually not executable programs of their own, applications built from components require special environments. 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 relations between components, and to execute the resulting application. In the complex case, it may include a wide variety of supporting tools that make application composition and related tasks easier.

> A composition environment is a program that is used: (1) to build an application out of components, and (2) to execute this application. A composition environment uses one or more component models.

2.4 Connection

Connections are what turns a set of components into an application. They define how the components interact to form a larger whole. Ideally, all dependencies or communications between components should be modeled as connections — if this is the case, components are context independent, because their whole context is created by the connections. Connections that can be changed without changing the components that they connect are called connectors (see Section 3.1.4).

> A connection is a logical link between two or more components. It represents an exchange of data or control between the components.

2.5 Configuration

The configuration of an application is the result of the application builder's work. It consists of connections between components and adaptations of components. An adaptation is something about a component that the application builder has modified in order to make it fit better into the application. Configuration is a useful concept to separate the structure of an application from its constituents. The constituents are the components; the configuration is everything else.

The configuration of an application specifies:

- the names of the components that are part of the application,
- how these components are adapted,
- the connections among the components.

2.6 Interface

Interfaces specify the services that components provide or require. Each component can provide or require several interfaces, and each interface can be provided or required by several components. A typical example of an interface is a set of method specifications.

An interface is a specification of (part of) the functionality of a component.

3 Features

In this section, we define and describe the various features that are relevant to composition environments. We will use these features to characterize the environments surveyed in Section 4. Section 3.1 describes component model features, such as components, interfaces, and connections. In Section 3.2, process level features such as scripts and diagrams are discussed. Table 1 gives an overview of all the features contained in these sections. Table 2 shows our categories of features, and how they interrelate. The left column shows the steps of the composition process. The second column represents component self-description, which is related both to the composition process, and to the component model. The last two columns show the remaining categories of component model features.

The features selected are the ones that are most relevant to composition environments, and have been cited most often in the literature. Both component model and process level features are essential to realize end-user composition. A good component model will make composition easy and expressive. But in reality, composition is often difficult because programming or other complex procedures are necessary to configure components, or even source-level changes to components need to be performed. Composition is often unexpressive because only components from a limited domain can be used, or because configuration is restricted in some way.

Without appropriate process level support, the benefits of a good component model will be accessible only to technical experts. Composing an application from prefabricated components is a significantly less complex task than programming the same application from scratch, and thus there should be no need to be familiar with complex tools such as programming languages in order to perform this task. Instead, users of composition environments should receive tool support in all steps of the composition process.

Component model	Process
Components	Searching and selecting
Components and component instances	Remote search
Component identity	Search criteria
Interfaces	Leverage self-description
Interfaces	Syntax
Interface instances	Semantics
Interface identity	Quality of service
Interface location	Non-technical
Versioning	Configuration
Directions of interface instances	Scripts and diagrams
Self-description	Ad-hoc adaptation
Syntax	Constraints
Semantics	Consistency
Quality of service	Distributed applications
Non-technical	Execution
Configuration	Execution in the environment
Connection semantics	Packaging
Connectors	Runtime changes
Connector types	
Connection cardinality	
Anticipated adaptation	
Composite components	

Table 1. Overview of the features discussed in this section.

Process		Component Model	
searching and selecting	self-description		components
configuration		interfaces, configuration	
execution			

Table 2. Relationship between the features of composition environments.

3.1 Component Model Features

We describe in this section the features of a component model. Thus, we are defining a component metamodel [85]. We use this metamodel as a basis for the discussion of the component models used in composition environments. While it would be desirable to be able to compare component models from a completely neutral and unbiased point of view, this is not possible. We need a terminological framework that we can use as a basis for comparisons. The metamodel presented here has been chosen as a compromise between neutrality among the existing approaches, and our intent to focus on those features of component environments that we believe to be the most relevant ones [62].

3.1.1 Components

Components are the building blocks of applications. They are usually licensed from another organization, installed, and then adapted and connected to other components to form an application. While the definition in Section 2.1 already restricts possible component concepts, the exact characteristics of a component are defined by the component model in question. While there is substantial variation, the common property of the various component models is that components are those artifacts that can be composed into applications.

Component Category

We distinguish five categories of components. Each component model surveyed defines its components as members of one of these categories. See Section 3.1.4 for a discussion of the relation between these categories and connection semantics. The categories are:

- Process. A process is an executable program running in an operating system.
- Object. An object is a set of data with associated behavior, is generally created by instantiation from a class, and usually belongs to exactly one process.
- Procedural library. Libraries of procedures are supported by many operating systems, for example in the form of dynamic link libraries.
- Class. A class is an abstract data type specification. Classes can be instantiated into objects. Szyperski argues that components should not be identified with data types [125].

- Class library. A class library is a set of classes.

Component Types / Component Instances

The relationship between components, types, and instances may be confusing. The meanings of these terms differ significantly among component models. This is illustrated by Table 3, which compares the realization of component types and component instances in C2 (see Section 4.1.1) and Java (see Section 4.2.1). C2 components are either processes in the distributed case, or objects in the localized case. Its component types are abstract specifications of components (expressed in the SADEL language), which may include syntactic and semantic constraints, but do not include concrete implementations. There is no concept of component instances in C2. Java components are classes. Classes can be instantiated into objects; thus objects are component instances in Java. There is no concept of component types in Java, although it would be possible to introduce such a concept analogously to C2 component types.

We define component types [70, 72] as a high-level mechanism to classify components according to certain concepts of substitutability, or to express architectural constraints. Composition environments may or may not support different component types. In systems where components are objects, classes are obvious candidates for component types, but component type systems may be more complex than programming language type systems.

Component instances may exist in systems where components are associated with data types. Instances of a component are the instances of the associated data type. Thus, each component may have one or more instances at any point of time in a given application. Modeling of both components and component instances may be useful in systems that infrequently create new instances, or that have a limited number of instances per component. For example, many graphical user interfaces have a fixed, small number of instances per component, which makes component instance modeling feasible.

	C2	Java
Component Type	abstract specification of a component	—
Component	process / object	class
Component Instance	—	object

Table 3. Comparison of component types and component instances in C2 and Java.

Global Component Identity

Since components are reusable, it has to be possible to identify them across organizational boundaries. In a global market, global identification becomes desirable. Components can be identified either through a name space [2, 54], such as the Uniform Resource Locator name space [130], or through unique decentralized identifiers [15].

Name spaces provide a human-readable classification of names, based on conventions about the structure of the name space. Name spaces should be global to accommodate a global component market, so that each name space element is worldwide unique. Name spaces are logically centralized (the name space has a root), but can often be used in a distributed way through selection of appropriate conventions. For example, by using

Internet domain names as the basis of 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.

Versioning

Components may have versions [22]. Versions are 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, which may or may not imply backwards compatibility.

3.1.2 Interfaces

In component models, interfaces are used to realize information hiding. When components specify their functionalities through interfaces, their implementations can remain hidden.

Interfaces

An interface is a specification of an abstract data type. It consists of specifications of the operations that define the data type, but does not specify its implementation. Components may use instances of interfaces to describe the services they provide or require. Interfaces can be associated with all levels of component self-description (see Section 3.1.3), so that they can specify not only the syntax and semantics of an operation, but also properties such as performance.

Component models that do not support interfaces are untyped. Without interfaces, there is no way to restrict connections, and thus 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.

Since interfaces specify types, interfaces may require an underlying type system that determines type compatibility. In the simplest case, two interfaces have to be identical to be compatible, but the various subtyping relations known from programming languages can occur [20].

Interface Instances / Ports

An interface instance, also called a port, is an association between an interface and a component. It represents a set of services that the component provides or requires; thus, each port has a direction: it is either a provision port (also called out-port), or a requirement port (in-port). While an interface is just a specification of an abstract data type, an interface instance is a part of a component that expresses that the component provides or requires an implementation of that interface.

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.

Component models differ in the number of ports they allow per component. Environments may have 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. Untyped systems provide at most one port in each direction.

Global Interface Identity

Similar to components (see Section 3.1.1), interfaces may be identified either through a name space, or through random unique identifiers. Without mechanisms such as these, name conflicts can occur when interfaces from different sources are used in one application.

Interface Location

Composition environments and components need to be able to access interfaces (i. e., abstract data type specifications) in order to perform type checks, and to receive information about components. Therefore, there has to be a way to find an interface by its interface identifier. Environments differ in where the interface itself is stored.

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

Versioning

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

3.1.3 Self-Description

Self-description [10, 57, 62, 93, 97, 140] gives components and interfaces the ability to specify or describe themselves at various levels of detail; self-description is a way of providing component meta-data. Description that is contained in a component has many advantages over externally stored description. 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 features [88], and is

categorized into four levels (syntactic, semantic, quality of service, and non-technical), each of which is described in the following.

Syntactic Self-Description

The first level of self-description, syntactic self-description, describes the operations of an interface at the programming language level. It is needed to check compatibility between interfaces; if there is no syntactic compatibility, one of the components will have to be adapted for both to be connected.

Semantic Self-Description

Semantic self-description [12] (second level of self-description) specifies the behavior of the operations in an interface. Semantic description can be formal, semi-formal, or informal. Informal specifications often take the form of natural language keywords. Since complete formal specifications are hard to use, assertions [34, 110] have been developed as a pragmatic form of incomplete formal specifications.

Quality-of-Service Self-Description

The third level of component self-description [64, 107] is concerned with all technical issues that are not functional. Examples include 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). Quality-of-service description can be applied both to interfaces and to components as a whole.

Non-Technical Self-Description

Non-technical self-description, the last level, includes all other, non-technical properties that may be of interest to component users. Examples include the price of a component, the author of a component or interface, and licensing information. Similar to quality of service, this level is applicable both to interfaces and components.

3.1.4 Configuration

Configuration features determine how components are adapted and connected to form an application. Connections are necessary to make components talk to each other; adaptations are desirable so that components can be modified to fulfill the requirements of an application better. Composite components are a means to structure large applications into smaller parts.

Connection Semantics

Connections can have different meanings in different component models. Without specifying the semantics of connections, one cannot deduce the semantics of composed applications. A connection can either 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 remote procedure calls). The first

case is type-based or code-based semantics, the second case is service-based or instance-based semantics [63].

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 [86] divide data into discrete messages that are exchanged at certain points of time.

The connection semantics employed in a given system correlates with the component category (Section 3.1.1) used. Components that are processes and objects usually employ instance-oriented semantics; components that are procedural or class libraries usually employ type-based connection semantics.

Connectors

Connectors [96, 114] are explicit connections (also called external connections). They are connections that exist independently from the components that they connect. In component models that do not provide connectors, components have to be modified whenever a connection is being changed.

Often, connectors have a certain functionality that does not logically belong to either component, but rather to their connection, for example, a specific network communication protocol. Component models that provide connectors with functionality or user-defined connectors raise the question what the exact difference between components and connectors is. In distributed systems, components can easily be identified with hosts and connectors with communication connections between hosts. In localized systems, however, no obvious analogy exists, and thus the distinction between components and connectors may depend on a specific architecture or a specific architectural style. Table 4 briefly summarizes the differences between components and connectors.

Component models may allow composite connectors [38]. 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 [35].

Components	**Connectors**
provide the bulk of application functionality	only small, specialized amounts of functionality
many different components available	only a few different connectors available
can have any number of ports	often have a fixed number of connection points (usually 2)
related to network hosts; localized	related to network wires; may be distributed

Table 4. Comparison of components and connectors.

Connector Types

Environments may provide several types of connectors with different connection semantics or different quality of service characteristics. Some environments additionally let the user define new types of connectors. Mehta et al. survey possible connector types [74].

Connector Cardinality

The cardinality of a connector determines how many ports can be linked to it at each end. A basic connector has cardinality 1-1; more complex connector cardinalities are 1-n (or n-1) and n-m. Connector cardinality may depend on the type of the connector.

Anticipated Adaptation

Components are more reusable if they can be adapted by an application builder to fit the needs of a certain project. Towards this goal, component developers can provide means to adapt components through anticipated adaptation [14]. 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 configuration steps.

Heineman and Ohlenbusch [41] 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 fragile base class problem [80, 122].

Composite Components

Applications easily become confusing when they consist out of a large number of components. Composite components may alleviate this problem by providing internal structure. Composite components are components that are not implemented in a programming language, but are instead composed out of other components. Components that are not composite are called atomic. Without support for composite components, large applications become hard to understand. Composite components hide part of the complexity and thus make application composition more scalable.

Composite components can be hierarchical or non-hierarchical. In a hierarchical model, every component is an immediate part of at most one composite component.

3.2 Process Level

Process level features make composition environments usable. They guide and inform the application builder so that the process of application composition becomes as easy as possible. While many accepted criteria for the design of usable environments apply, we focus on those user level features that are characteristic of composition environments. We categorize them according to the application composition process [62]; first, the builder searches for components, then self-description is used to get information about the com-

ponents and to select appropriate ones, third, the application is configured, and last, it is executed.

3.2.1 Searching

Searching is an integral part of the application composition process. When requirements are determined, components to fulfill these requirements have to be found. Every component that is found and selected may depend on other components, which also have to be searched for, although tools that help in this task exist [42].

Remote Search

Components may be developed by an organization that is different from the organizations using it. The organizations that want to use a component 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 builders have to manually locate components and integrate them into the local repository. At the other extreme, a composition environment may be completely distributed and hide the actual location of components from the user.

3.2.2 Leveraging Component Self-Description

Self-description of components (see Section 3.1.3) can be leveraged by environments to guide the user, to provide information for searching and selecting components, and to guarantee application consistency analysis. Composition environments differ by the amount and way in which they make use of self-description capabilities defined by the underlying component model.

All levels of self-description can be used by environments for the following purposes:

- User guidance. Components provide information about themselves that is used to inform the user about components, and to guide the process in which the components are used. Diagrammatic notations can make use of component self-description in many ways, for example they can use color schemes to visually categorize components into groups.
- Searching and selecting. Component description is used to search for matching components, and to select among a set of components returned by a search in the component-based development process.
- Consistency and analysis. Environments may guarantee consistency of a composed application by checking connections for agreement between provided and required features when they are specified in the tool, or in a separate consistency analysis step later. In either case, component self-description is needed to ensure that connections are legal.

3.2.3 Connecting and Adapting

In order to create an application out of a set of components that have been found, the components need to be connected and adapted. This activity is also known as the configuration step. Connecting means that the abstract dependencies between components

are instantiated with concrete relations, typically by defining a connection between two ports. Adapting means that individual components are changed to make them fit better into the system that is being built.

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 programming skills.

Scripts are text files that are written in a scripting language. A scripting language [98] 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; they are typically Turing-complete programming languages though. For application composition, even simpler notations may be sufficient. Assuming that all complex tasks are handled by components, a composition language only needs to be able to define connections and adaptations.

Diagrams are specifications that are expressed in a graphical notation [117]. Typically, notations consist out of boxes and arrows between them; in the context of a composition environment, boxes often represent 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 full-featured visual programming languages to simple diagrams that express nothing more than connections between components.

Environments use scripts, diagrams, as their composition notation, or they may require compositions to be expressed programmatically. They may support more than one composition mechanism, in which case they have to ensure consistency between the different models that they employ.

Ad-Hoc Adaptation

Ad-hoc adaptation (as opposed to anticipated adaptation, see Section 3.1.4) is the task of adapting components to use them in a way that was not anticipated by the component developer. Ad-hoc adaptation, also called external adaptation, is often used with legacy components that are no longer supported, so that the application builder has to adapt them to function with current technology. Since components are encapsulated and hide their implementations, adaptations cannot change the components themselves. Instead, wrapping techniques have to be used.

A wrapper [37, 43] is 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 component systems, wrappers can be either components or connectors. A component can wrap another component simply by using it as a server, and providing a different inter-

face. A connector can act as a wrapper by translating communication between the components it connects.

Shaw discusses types of ad hoc adaptation, many of which require source code modification, though [113].

Constraints

Constraints are logical expressions that constrain the structure of an application [1]. Constraints can be either configuration constraints, or component constraints. Configuration constraints regulate the way a set of components interoperate. Topological constraints are configuration constraints (for example, a constraint that forces the architecture of the application to be layered). Component constraints limit the kinds of components that can be used in an application, for example, a constraint specifying that every component has to support a given interface.

Constraints are useful as application-specific extensions to the underlying component model. Using a set of constraints in an environment is comparable to an ad-hoc specialization of the underlying component model. Thus, constraints provide a flexible way to extend the environment. Architectural styles [116] 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. [69] 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 constraints given by components, such as restrictions to which other components a component may be connected, or with application constraints defined by the application builder (see above).

Consistency checks can be done either at design time, or at runtime. Design time consistency checks are performed either on-the-fly, or as a separate analysis step [108]. On-the-fly checks are done while a specific design step is performed, for example, while a connection is made between two components. If the check fails, the user will be prevented from creating that connection or warned about a possible error, for example with the help of design critics [109]. Separate consistency checks are performed once an application is 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. Runtime 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 [101]. Therefore, many composition environments have integrated support for distributed applications (applications that are distributed over several hosts on a network).

3.2.4 Executing

Executing the composed application, is, of course, the goal of the application composition process. But execution is also a part of testing the application, and thus a process step in its own right. A flexible, usable composition environment will allow the user to build and run applications in quick iterations.

Execution of Partial Applications

Composition environments must provide ways to execute composed applications. Environments differ in the way they do this. They may require an application to be completed, or they may allow execution of incomplete applications or even individual components for testing purposes.

Packaging

Once an application has been built out of 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 deployed (installed and executed), packaging may be more or less complex. In the simplest case, the components and their configuration are copied into a file that can directly be executed on the host platform. In more complex cases, the generation of installation scripts may be necessary.

Runtime Changes

Environments may support the changing of configurations (connections and adaptations) at runtime. For example, applications for which the acceptable downtime is very small may have to be updated at runtime, because updating of individual components instead of updating the application as a whole may reduce downtime.

Since configuration changes at runtime are still a research issue [94, 95], environments usually support them in a limited manner only. It is often possible to change configurations dynamically as long as no guarantees are needed that no program state is lost. Environments may provide mechanisms such as transaction support, roll-backs, and state extraction to support dynamic changes.

4 Approaches

Composition environments combine solutions from various areas of research. Work related to composition has so far been spread among many communities both in research and industry. In this section, we will look at each of these areas, and present exemplary approaches from them. The areas that we discuss are: software architecture, programming languages, visual programming, programming environments, computer networks and interoperability, operating systems, and plug-in systems. We list representative composition environments from each area, and briefly discuss similar approaches, where appropriate. While we try to cover each of these areas equally well, we focus on those approaches that show the current state of technology. At the end of this section, Tables 6 and 7 give a detailed comparison of the approaches according to the features introduced in Section 3. Table 5 gives a quick overview of the terminologies of the different approaches.

	Component	Connection
C2	component	connector
Koala	component	binding
Unicon	component	connector
Java	class file	class path
Jiazzi	unit	connection
Visual Age	part	connection
Visual Basic	control (VBX)	—
Bean Box	bean	connection
Vista	component	link
Code Broker	—	—
Agent Sheets	agent	—
EJB	enterprise bean	—
Pipe and Filter	filter	pipe
Win 32	dynamic link library (DLL)	—
Dot-Net	assembly	—
Plug-In	plug-in	—

Table 5. Comparison of the terms for components and connections in the surveyed approaches.

4.1 Software Architecture

Software architecture [27, 73, 99, 116, 139] is concerned with the large-grained structure of applications. 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 design of the runtime structure. Software architecture deals with architectural components and connectors. Components in software architecture are not necessarily components in the sense of this paper, but may be so-called abstract components [18], which are pure design artifacts without realization in the implementation. Both instance-oriented application design and the concept of connectors, which is not limited to instance-oriented design, are contributions of software architecture. We discuss two architecture environments that support deployable components.

4.1.1 C2 / Archstudio

C2 is an architectural style developed in 1995 [126]; Archstudio is an environment to support building applications in the C2 style [8, 68, 95]. C2 was originally developed for adding graphical user interfaces to legacy software, but it has proven suitable for large-

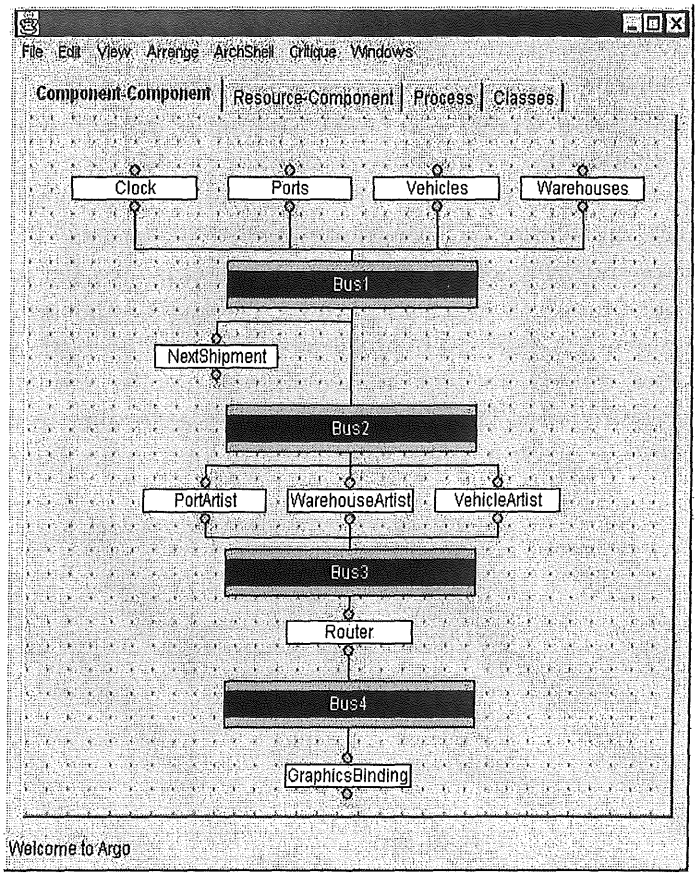

Figure 2. Representation of a C2 style architecture in Archstudio.

scale distributed applications in general. Archstudio contains tools to allow display and modification of architectures (Argo/C2, see Fig. 2, and Archshell), and to map architectural changes between model and implementation.

C2 structures applications in a layered manner; components communicate only by exchanging asynchronous messages through connectors. There are two types of messages, events and requests. Depending on their type, messages can flow only upwards or downwards in an architecture. Connectors are heavy-weight and can have complex implementations of their own [25].

Component Model. Components are Java objects with their own control threads. Each 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. Both predefined and user-defined connector types are supported, and connectors may be distributed. There is no support for anticipated adaptation or composite components.

Process-level support. Applications can be built and modified by using either scripts, a command-line shell, or a graphical tool. They can be tested and executed in the environ-

ment and exported in the form of scripts. There is no support for component development.

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 environment. Various interesting connector technologies have been implemented in C2 [25]. Because of the fact that C2 connectors are heavy-weight and have variable cardinality, it is well suited for experimentation with novel connector types.

Related Approaches. C2 is extended by an architecture description language, C2 SADEL, that focuses on component evolution, and a supporting tool, Dradel, which has been integrated into Archstudio [71]. Several architecture description languages (for example, Darwin [65], Weaves [39], or Wright [7]) address similar issues as C2; most of them, however, do not have tool support for deployable components.

4.1.2 Koala

Koala [90, 91] is a component model and associated tool set for embedded software in TV sets. Koala is based on the Darwin architecture description language [65]. Components are implemented as C modules. The Koala language describes these components, and allows textual modeling of applications out of them. When the application is compiled, components are statically bound to each other. Procedures that are connected to each other are in this way are resolved, if necessary, by adding macro definitions to the modules that result in replacing the names of the two connected procedures by a common name.

Component model. Koala components have provision and requirement ports, but no self-description beyond this level. Koala interfaces are sets of C functions, and are explicit entities. Connectors are explicit, but no connector types exist. Connection semantics are type-oriented, since connections establish matchings between functions. There is no support for component adaptations. Composite components, called compound components, can easily be defined as Koala scripts. Koala has specific rules for component evolution which constrain what may change between versions of a component.

Process-level support. Koala applications are defined using a textual notation. There also exists a graphical notation for visualization purposes. Since embedded system have no user interface, no post-design user support is needed.

Summary. Koala combines the architectural modeling features of Darwin with a component model. While the applicability of Koala's techniques is certainly wider than the very specific domain that it is built for, it is geared towards high-performance applications. As a consequence, deployability of components was not one of its design goals. Possible extensions should easily be able to add this feature though.

4.1.3 Unicon

Unicon [115, 143] is an architecture description language that defines various types of connectors, and supports implementation of applications using these connectors. Most of its connection mechanisms are source-code based, but it also provides an extension of the pipe-and-filter mechanism for the composition of deployable components (see Section 4.6.1, Pipe and Filter).

4.2 Programming Languages

Programming language research, especially in the area of object-oriented languages, has developed many mechanisms to support design of the compile-time structure of programs by making program code more understandable and more reusable. Examples of such mechanisms are inheritance, polymorphism, encapsulation, and explicit interfaces. Lately, many programming systems have started to support dynamic linking [36], which is meant to make libraries independently deployable. Dynamic link libraries (DLLs) extend reusability from the source code to the compiled code. We discuss Java, a popular object-oriented programming system that puts a strong focus on dynamic linking and deployability, and Jiazzi, an extension of the Java platform.

4.2.1 Java Platform

The Java programming system [119] was introduced by Sun Microsystems in 1995. It consists out of a programming language [40], a class library [48], and a virtual machine [142]. The Java virtual machine and the Java standard class library are together known as the Java platform. Design goals of the Java 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;
- Object-orientation and ease of programming: The object-oriented core concepts of encapsulation, inheritance, and polymorphism are built into the platform. For example, programs for the Java platform cannot use pointer arithmetic, and there is internal support for automatic garbage collection. There are no program files; the only kind of executable files are class files which each encapsulate one class.
- Internet-wide deployment: Class files can be loaded from remote hosts on the Internet, and dynamically linked at program runtime.

Components for the Java platform are class files, or Jar files, which are sets of class files that are bundled and compressed into one file. Each class file contains either the implementation of a class or of an interface. Interfaces are definitions of data types that do not include method bodies; they are meant to be inherited from by classes, which provide full implementations. 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 runtime environment) that determines where the runtime environment looks for class files. The class path can include local directories, files, or remote locations specified by URLs. Additionally, in order to execute a program on the Java platform, the main class has to be set. The main class is the class with which program execution is to start; it needs to have a specifically labeled main method. When an application is executing, the runtime environment searches for classes that are needed by using the class path. If several classes with the same name exist, only the one that is listed first in the class path is used.

Component model. Components (class files) are encapsulated and deployable [28, 59]. They provide syntactic self-description of provided features through the reflection mechanism of the Java platform. Interfaces are entities of the same rank as components, since they are also stored as class files. Both components and interfaces are identified through a naming convention that guarantees globally unique names by including the component developer's Internet domain name. Components are limited by the fact that

they are data types (classes) at the same time. This makes it impossible to create large-grained components without compromising principles of object-oriented design, and it prevents the creation of composite components [125]. The class path is an explicit connector; the class path facilitates changing the architecture of an application without having to change its components. However, use of the class path is tedious, because there is no mapping between the structure of the class path and the application architecture. Components do not describe their required features, so creating an architectural description requires program analysis.

Process-level support. The Java platform is provided in the form of a command-line based interpreter. Therefore, there is no process-level support beyond the connecting and executing of applications.

Summary. While severely limited as a composition environment, Java was the first object-oriented system with a precise specification of compatibility of deployable components. Class paths are a simple, though limited, means to compose applications. Together with binary compatibility, they make it easy to substitute deployable components for each other.

Related approaches. Binary Component Adaptation [53] and Load-Time Adaptation [29] 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).

4.2.2 Jiazzi

Jiazzi [67] is a script-based tool that adds a component concept to the Java platform. Unlike Java Beans and similar technologies, it is aimed at realizing deployable, type-based components. Jiazzi is based on research geared at introducing connectors to programming languages [35].

Components, called units, are sets of Java classes, specified through scripts that either map a unit to a single Java file ("atoms"), or define a compound, which is a composite component that refers to several atoms. Another kind of script, called signature, is used to specify packages in the Java code. The Jiazzi Linker generates an application out of scripts and class files; it does so by modifying the constant names (such as names of classes) in the class files: it 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. Figure 3 shows two example Jiazzi scripts: "ui", which is a user interface component specified through the signature "ui_s", and "linkui", which is a compound that connects "ui" with another component called "applet".

Component model. Components are largely deployable, but suffer from a lack of encapsulation: each component consists both of a set of class files and the add-on scripts before linking, and it is not possible to prevent component implementations from communicating with each other in ways not specified in the scripts. Components are type-based, and provide syntactic self-description of both provided and required features. Signatures serve as interfaces; they specify Java packages (sets of classes), and are thus more abstract than Java interfaces, which specify only individual classes. The scripting language

```
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 3. Example of two Jiazzi scripts defining an atom and a compound.

provides explicit connectors, and composite components. There is no support for adaptation.

Process-level support. The user interface is limited to the simple scripting language and the execution of the Linker. The separate linking process (the modification of the class files) makes it possible to avoid almost any performance loss when using this environment, but reduces interactivity. Since generated applications exist in the standard Java distribution format, no additional packaging process is needed; 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 an environment with a simple component-based notation (in-ports, out-ports, connectors) that works on top of the Java Virtual Machine. It shows how a tool that does not support these concepts can easily be extended to include them.

Related approaches. Arch Java [6] is an extension of the Java language that allows for connectors and ports, and can enforce their use. However, it allows composition only at compile time, and does not support deployable components.

4.3 Visual Programming Environments

Visual programming [117] is a subarea of programming language research focusing on graphical, instead of textual, notations for programs. While visual programming languages are not commonly used, visual design notations (such as the various notations included in UML [89] and software architecture diagrams) are widespread. Composition environments often include a kind of visual notation with an expressiveness that is somewhere in between UML-like design diagrams and Turing-complete visual programming languages. The area of visual programming environments is concerned with the design of tools to support software development tasks. A composition environment is such an environment, though the underlying process and many of the process steps are different than in traditional programming environments. We survey Visual Basic, Visual Age,

and the Bean Box, three popular commercial environments, and Vista, an academic visual programming environment.

4.3.1 Visual Basic

Microsoft Visual Basic [30, 78, 129] (see Figure 4) 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. Visual Basic programs consist out of 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 program code must be inserted into event handler procedures associated with the individual events that can happen in a form, for example the pressing of a button. Controls are usually predefined, but it is possible to import custom controls into the environment. Since the introduction of Visual Basic, many other IDEs 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 file format for custom controls [66]. Controls are well encapsulated and can easily be exchanged, thus promoting component reuse. They provide syntactic self-description that is used by the environment to display and adapt them visually. Custom

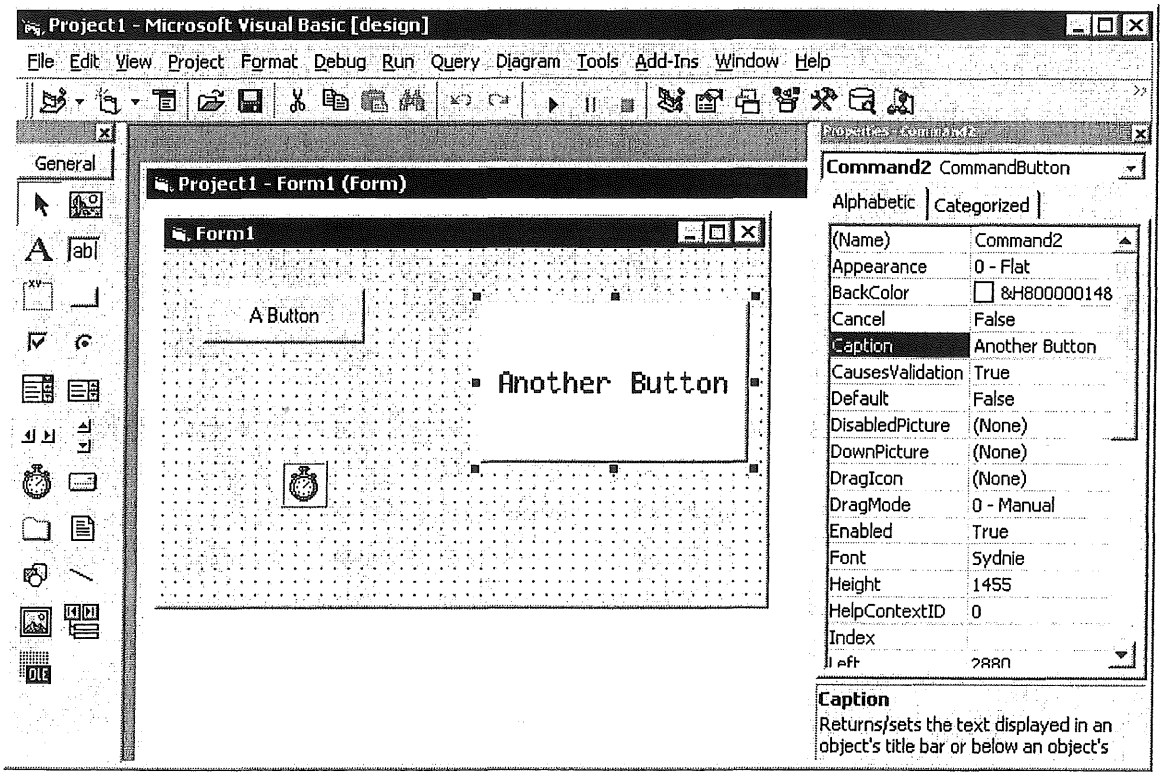

Figure 4. Composition of a GUI in Visual Basic, version 6. The form shows two button controls and a timer control; the property sheet for one of the buttons is visible.

controls that are displayed in the graphical environment are instances of the types defined by their files. Event handlers can be considered as connectors. However, they often contain functionality of their own and thus should better be classified as components. In this case, there are no explicit connectors. The architecture of an application is not visible, because the visual environment shows only top-level component instances, that is, those that are immediate constituents of the application. Other components that these might depend on are not shown. Composite components do not exist.

Process-level support. Predefined components can be visually arranged and adapted in the environment. To connect components, it is usually necessary to write textual event handler code. 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.

Summary. Visual Basic opened components to the mass market. The combination of an easy-to-learn programming environment with a robust file format for deployable components proved to be commercially very successful.

4.3.2 Visual Age

Visual Age [131, 132] (see Figure 5) is an integrated development environment for the Smalltalk language published by IBM in 1994. Since then, versions for C++ and Java have been added. Apart from the traditional features of an IDE, such as program editing and debugging, Visual Age has support for component-oriented visual programming. The goal of Visual Age is to increase the separation of labor among programmers by reducing the amount of technical skill needed to build an application. It builds on and extends the metaphor of graphical user interface builders (tools to visually create dialog windows out of predefined visual elements such as buttons, check boxes, and text fields) by including support for non-visual components and different kinds of explicit connectors.

To compose an application, 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. The interface of each component consists of attributes, actions, and events. Connections are created by linking these interface 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 its parameter), and
- attribute-to-action (the action is triggered whenever the attribute changes).

New components can either be defined visually, or through scripts (i.e., Smalltalk methods). After a new 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. 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 components and explicit connectors. Components have syntactic self-description; the direction of ports depends of the types of con-

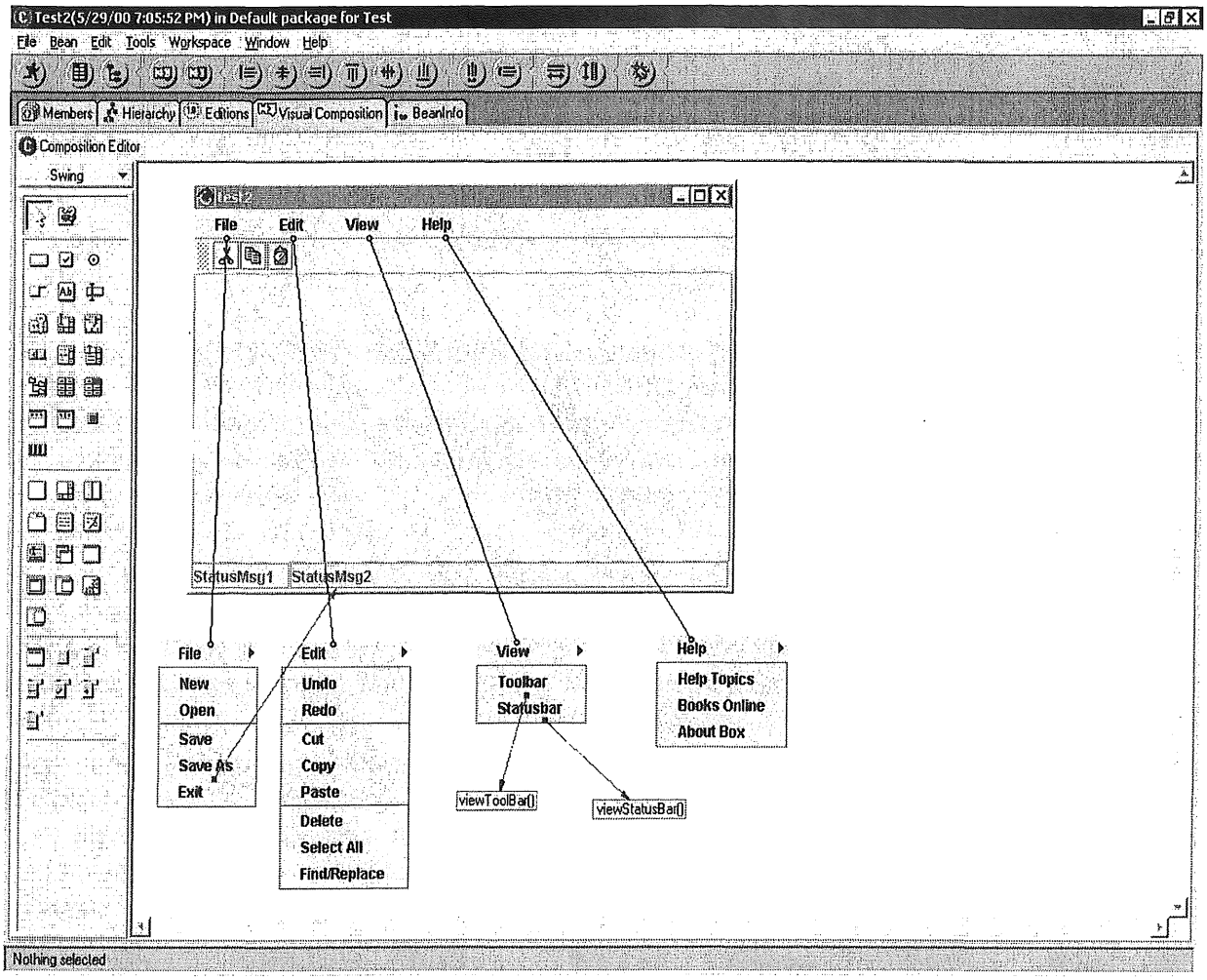

Figure 5. Screen shot of the Visual Age composition editor. Several visual parts (GUI components) are connected. Blue connections are attribute-to-attribute connections (linking the selection status, of a menu bar element to the visibility status of a pull-down menu) green ones are event-to-action connections (linking the pressing of a menu item to a method).

nectors attached to it. For example, an action port has always direction In, whereas an attribute port can have both directions. Connectors are explicit and serve as communication channels between components.

Process-level support. Visual Age has extensive support for connecting and adapting components visually. Since it is a full IDE, textual representations in the underlying programming language can also be used. There is limited support for reverse engineering textual programs into visual representations. Newer versions of Visual Age include remote repositories that can be used to store and retrieve components in local area networks. There is no deployment support beyond the support of the underlying platform.

Summary. Visual Age was one of the first commercial tools to realize visual, component-based programming. Unlike other development environments, Visual Age is not limited

to GUI components. It is, however, limited by its identification of components with objects.

Related Approaches. Many integrated development environments support similar mechanisms for visual composition. Examples are Microsoft Visual Studio [79], Borland Delphi [13], and JBuilder [51]. 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 configuration tasks.

4.3.3 Bean Box

The Bean Box is a protypical environment for Java Beans [11, 49], a component architecture for the Java platform introduced by Sun Microsystems in 1996. Its original purpose was to test Java Bean components for compatibility, and 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 Bean Box, for example JBuilder [51].

The Bean Box is based on the technologies of Visual Age (see above), but simplifies them. It consists of a workspace with a toolbar; components can be dragged onto the workspace, and can be adapted and connected there. There is no support for component development. Only three of the connection types of Visual Age are supported, event-to-action, attribute-to-attribute, and attribute-to-action.

Component Model. 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; any other classes in the Jar file could only be support classes used by the Bean class. Jar files contain meta-information to identify Bean classes. In the environment, Bean classes are instantiated when a Bean is dragged into the graphical builder. Connectors are explicit; they are realized as individual classes that are generated when a connection is established. The Java Beans component model defines syntactic self-description that goes beyond the self-description provided by the Java platform, especially through naming conventions. For example, there is a convention saying that methods that return the value of an attribute have to be named as "get" concatenated with the attribute name. Limited semantic description is also available. Simple method calls between components are allowed, but cannot be modeled by explicit connectors since there are no input ports for method calls. There are no composite components. Ohlenbusch and Heineman give a formal specification of the types of ports in the Java Beans component model [87].

Process-level support. Components can be connected and adapted through property sheets. The environment leverages the self-description of the Java Beans standard through specific dialogs that display information about components. Applications can be executed in the environment; the Bean Box can also generate an "applet", that is a Java program executable in a Web browser, that encapsulates the connections that were made between Beans.

Summary. The Bean Box is an attempt to combine the technologies of visual development environments such as Visual Age with the Java language. Through the associated Java Beans component model, the Bean Box shows the power of self-description, and how it can easily be realized using naming conventions.

Related approaches. Arabica [111] is an extension of the Bean Box with focus on architectural concerns. It supports composition in the C2 architectural style (see Section 4.1.1) by generating wrappers that turn C2 components into Java Beans, and enforcing the C2 style rules. The Bean Markup Language [137] 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 composition notation for the Java Beans component model can be realized.

4.3.4 Vista

Vista is an environment for visual composition of applications [75, 76]. While originally developed for the composition and manipulation of multimedia applications, it supports a generic concept of components and connectors. A Vista component consists of an interface, a behavior, and a presentation, which is responsible for displaying it in the graphical environment. The environment is relatively independent from a 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.

To create an application, components are dragged into the visual editor and connected in the usual, graphical manner. However, the application composer does not only need to get appropriate components, but must also get access to an appropriate component model. Just like the development of components, the development of new component models is non-trivial and requires technical knowledge.

Component Model. Vista allows for user-defined component models, however these have to adhere to and be expressed in an underlying model, a meta-model. The meta-model has explicit representations of components, ports, and connectors (called "links"). Components need to 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). Components are instantiated by the environment when they are being dragged into the editor; but the concrete meaning of "instantiation" is left to the component, so that both instance-oriented and type-oriented component models are possible. Composite components can easily be created graphically. There is no support for adaptation.

Process-level support. The usual support for graphical configuration exists. Because of the flexibility of the system, however, the concrete user-interaction model is determined by the designer of the component model. Composed applications can be executed in the environment.

Summary. Vista is a fully-configurable visual programming environment with support for code-based components and connector support. It represents the first attempt to combine a visual environment with a complex component model.

4.4 Software Reuse

Software reuse research [56, 120] has been concerned with the problems of identifying reusable code, storing it in a reuse repository, and enabling software developer to retrieve reusable code from the repository when they need it. However, the task of composing applications is generally not supported; the code retrieved from a reuse repository has to be adapted and integrated with other code manually by the developer. Often, only source code components are supported. However, reuse environments provide relevant support

for searching and selecting of components. We discuss Code Broker, which focuses on the task of finding components, and Agent Sheets, which focuses on usability.

4.4.1 Code Broker

Code Broker [141] is a software 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 code editor, the system uses a machine-learning technique to check the natural-language comments and the signature of the method to look for similar methods in the repository. Possible matches are presented to the developer; the developer can receive more information about them, if desired. The system adapts itself to the developer and the task at hand, and can also be adapted manually; for example, methods about which the developer already knows are excluded from the search results. The current prototype of Code Broker is implemented as an extension to the Emacs text editor and uses the Java standard class library as its component repository. It uses the Java documentation to search for methods.

Component model. Components are deployable by their virtue of being Java components. However, there is no composition support.

Process-level support. Code Broker 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 done on an automated basis; the user does not have to decide on search criteria and initiate searches, since this is done opportunistically by the environment. The only kind of component self-description used is the syntactic information from method signatures, but extensive use is made of semantic description from other sources than the components themselves. There is no additional support for configuration or execution.

Summary. While not a composition environment in itself, Code Broker shows how much automated support can be provided for searching and selecting, tasks that are typically still done manually. Composition environments should integrate these techniques.

4.4.2 Agent Sheets

Agent Sheets [5, 105] 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 consist out of simple event-action sequences that allow the agent to change its depiction and to cause events in the adjacent agents. Simulations based on Agent Sheets have been successfully used for elementary school education.

Figure 6 shows a simple Agent Sheets 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 nothing under them, they crash. 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.

Component Model. Components can internally be complex, their interfaces, however, are restricted to simple interactions with their four adjacent components. There is no support

for self-description or any configuration support beyond mere connecting of components. The grid on which agents are places acts as an explicit connector; it decides which components are adjacent to which and thus how events are routed.

Process-level support. There is extensive support for the graphical creation of applications from agents. Applications are configured by placing components on the graphical grid. There is no support for the technical steps of the development process; applications are assumed to be built from a small number of predefined components, so that no searching or selecting procedures are needed. Applications need not be complete in order to be executable.

Summary. Agent Sheets is an extremely usable composition environment. Building applications has a similar touch-and-feel as moving pieces on a game board. However, it is limited to application domains that can be represented as a two-dimensional grid, such as map-based simulation.

4.5 Computer Networks and Interoperability

Computer networks have in common with component systems the fact that applications are composed out of heterogeneous, decoupled parts. In a networked, or distributed, system, each part of the application may run on a different platform and be under different administrative control. In a component-based application, each part may have been developed by a different component developer. Systems that are both network-based and component-based combine the properties of both. Since its runtime structure is an essen-

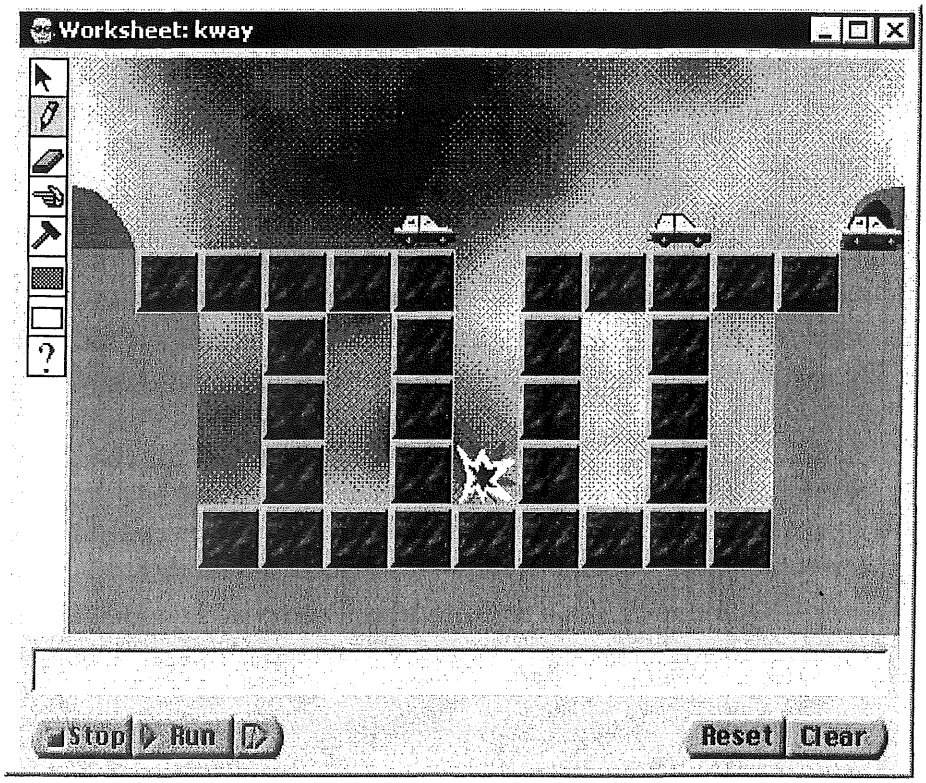

Figure 6. An Agent Sheets worksheet simulating cars on a bridge.

tial attribute of a network-based system, software architecture is often used to analyze such systems. Waldo et al. [135] argue that there are fundamental differences in the design of distributed versus local applications because of network latency, network reliability, and similar issues. These differences are analogous to the differences between distributed and local composition environments. We discuss Enterprise Java Beans, an industrial client-server component standard, and the Worldwide Web, an example of an Internet standard that allows composition of independently developed components into networked applications.

4.5.1 Enterprise Java Beans Containers

Enterprise Java Beans (EJB) [26, 82, 112, 127] is an extension of the Java Beans component model for client-server applications. It provides components with common services such as transaction processing or lifecycle management. An Enterprise Java Beans container is a runtime environment that can execute Enterprise Java Beans. Examples of EJB containers are Bea Web Logic [9], IBM Web Sphere [46], and Oracle Application Server [92]. 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. EJBs use Java Remote Method Invocation for communication between client and server [134].

Component model. Components are deployable, because they can be used by the container without human intervention. The units of deployment are so-called 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 syntactic self-description of Java Beans is extended in many ways and is stored in XML-based Deployment Descriptors; for example, to differentiate between data types that have state and those that do not, or to include information about required data types that are not implemented in the same component. There is no concept of component identity. Composite components are supported in a rudimentary way. However, to create a composite, the self-description of contained components has to be manipulated, which violates the principle of encapsulation. Connectors and anticipated adaptation are not supported.

Process-level support. None. Unlike the Bean Box, 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.

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 Virtual Machine) can be extended to include the needs of such applications. As a trade-off, it ignores aspects of usability.

Related Approaches. Bean Bag [93] is a repository for Enterprise Java Beans that extends deployment descriptors by adding semantic and non-technical information. The Corba Component Model [23, 136] 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.

4.5.2 The Worldwide Web

The Worldwide Web [32, 33] is a distributed, Internet-based hypermedia system developed in the early 1990s. Unlike previous Internet systems such as FTP, its goal was to reference remote data instead of requiring users to copy them to their local systems. Unlike previous hypermedia systems, the Web made integration of documents published by many different organizations possible, and provided a uniform user interface to them. The Web provides information through resources located on origin servers. When a user agent (that is, a Web browser) requests a resource from a server, the server creates a representation of the resource, and sends it along with metadata to the user agent. The user agent then uses the metadata to determine the way in which the representation is displayed to the user, for example, by using an appropriate plug-in.

Component Model. Components fall into two categories: server-side components and client-side components. Server-side components are resources located on servers; these may be simple, static data (such as hypertext pages or pictures), or complex programs that generate dynamic replies (such as the results of database queries). Client-side components are viewers used to display or interpret representations of resources, such as image viewers, Java virtual machines, or Shockwave plug-ins. An application is composed out of one server-side component, and a variable number of client-side components. An example of a complex Web application is an online auctioneer; the server component contains a large database with many bids and auctions, client components are the various tools customers use to display auction information and embedded information such as pictures or movies.

Connections are always established between one server component and one client component. A user enters a URL into the Web browser, or clicks on a link pointing to that URL, and this causes the browser to request the corresponding resource. URLs thus serve as connectors between the resource named and the client components active on the user's desktop. Proxies and user agents can filter incoming representations, and can thus adapt them. It is not possible to adapt components as a whole, though.

Process-level support. The composition notation is simple — a short character string describing the location and name of the desired resource. Support for component searching is not integrated, but is provided through additional services, such as Web search engines, for example Google. These search engines can deploy optional self-description provided by resources. Only a small number of connections can be deployed at a given time, and it is not possible to persistently store connections, so that composite components and application packaging are not applicable.

Summary. The Web is specified as an open system in the tradition of Internet standards. This made it possible to implement a wide variety of Web servers, browsers, proxies, and other supporting applications. As a consequence, the Web has become an immensely successful system. Although it is a composition environment with a very specific domain (distributed hypertext applications), it exemplifies the importance of well-defined standards for a composition environment.

Related approaches. Many other Internet technologies are based on similar open standards, such as Internet Mail [47, 118] or Usenet [45]. Java Server Pages [50] and Active Server Pages [3] provide ways to integrate Web server applications with composition environments (Java and MS Windows, respectively). Java Applets [48] integrate mobile

Java components into Web pages; they are defined by the server, but executed on the client host.

4.6 Operating Systems

Operating systems support installing, configuring, and executing applications, and, to a small degree, connecting them. Operating systems also include many interoperability techniques. Connection mechanisms include messaging services, data exchange through a central registry, and remote procedure call capabilities. Thus, operating systems can be considered as precursors of composition environments [122, 123]. The main difference between operating systems and composition environments is their implementation of the performance versus configurability trade-off: operating systems focus on performance with little concern about configurability; composition environments focus on configurability with less focus on performance. We survey Pipe and Filter systems, which provide a simple, yet popular technology found in many operating systems, and two versions of the MS Windows platform, Win 32 and Dot-Net, which is the current industry standard for personal computer operating systems.

4.6.1 Pipe and Filter

Pipe and filter systems [116] are used in command shells for the quick and easy connection of text-based tools. The C Shell [52], which is frequently provided with Unix-like operating systems, is an example. A Unix program has one standard input port ("stdin"), and one standard output port ("stdout"). Programs are called filters, and they can be connected with pipes. A pipe links an output port to an input port. It acts as a stream buffer to which the input port writes and from which the output port reads. Input and output are usually interpreted as text strings that are structured only by convention; for example, space and tab symbols are usually considered separators. Tools that are connected with pipes are typically text-based operating system commands (such as "print directory"), or simple string manipulation tools that can sort or otherwise modify text streams.

Component Model. Pipe and filter systems are designed for quick ad-hoc connection, and this is usually all that they can do. Components have a predefined number of ports; each of those ports is untyped, as a result of which the validity of a connection cannot be checked. Components are processes, that is, running instances of Unix programs. Pipes are explicit connectors.

Process-level support. The C Shell allows users to quickly connect and execute larger applications by using the filter symbol ("|"). Using this symbol, several tools can be connected at once in a command line. If so desired, connections can be stored in a shell script so that they can be used more than once. Anticipated adaptation of components is possible through command line parameters by adding the parameter symbol ("-") and the parameters after the name of the tool.

Summary. Pipe-and-filter impresses through its simplicity. It is by far the simplest composition environment that is commonly used. On the other hand, it is limited by its command-line interface.

Related Approaches. Unicon [115, 143] is an architecture description language that supports modeling and executing of Unix pipe and filter architectures. It extends the capa-

bilities of shells by handling arbitrary topologies, so that filters with more than one input or output port can be used. As a consequence, branches and loop backs can be modeled.

4.6.2 MS Windows / Win 32

MS Windows is an operating system for Intel PCs developed by Microsoft Corporation. Originally developed as a graphical front-end to the MS Dos operating system, MS Windows grew into a stand-alone system that includes a host of technologies. Win 32 is the programming platform that the 95, 98, Me, and NT 4 versions of MS Windows are based on. Components in MS Windows are dynamic link libraries (DLL files); some of the limitations of DLL technology were removed by the Component Object Model (COM) standard introduced in 1995 [11, 15, 21, 121]. COM makes it possible to extend a DLL by adding procedures without making recompilation of all the clients of the DLL necessary.

To install applications and components, automated installation scripts are usually used. When a new component is installed, it is registered in the Windows Registry for a set of services. Clients can locate these services without having to know the component that implements them.

Component model. Component describe the services they require and provide through interfaces. Interfaces are identified by randomly generated unique numbers. Registry entries advertising services can be considered as explicit connectors, since the registry decides which server component is used for a specific service. However, there is no way to display the architecture of applications. For anticipated adaptation, the registry can also be used by providing keys that can be filled in with appropriate values. There are no composite components.

Process-level support. The user has little control over configurations, because they are usually changed automatically (without user interaction). Users familiar with the technical details of MS Windows can use the registry editor to manually edit the list of provided services in a script-like manner. Further, MS Windows includes tool support for the syntactic description of components, both of provided and required features, and to retrieve versioning information.

Summary. MS Windows exemplifies the attempt to realize component reuse in the setting of a large, performance-oriented operating system. While the performance focus minimizes usability, the Windows / COM approach works well for many commercial applications.

4.6.3 MS Windows / Dot-Net Framework

The Dot-Net Framework [31, 58, 77, 102, 104, 106] is the platform planned for future version of MS Windows. It consists of a virtual machine (the "Common Language Runtime") and a standard library, and is an extension of the Win 32 platform and the Microsoft Foundation Classes (MFC) library. It was announced in 2000, and is currently available as a beta version. The main goals of the Dot-Net Framework are support for language interoperability and support for Web services.

Applications consist out of "assemblies" of files. While assemblies are not files themselves, operating system extensions make it possible to treat them in the same way as individual files. Assemblies are components that have self-description. Unlike in Win 32,

entries in the central registry are not necessary for component interoperation. Assemblies can be private or shared. Private assemblies are used by only one application, and only need to be copied to the directory of this application in order to be used. Shared assemblies are shared among applications, and they are installed by using a simple command line tool ("gacutil.exe", the Global Assembly Cache Utility). All further configuration is done internally by accessing the self-description of the assembly.

Component model. Components are deployable and type-oriented. Syntactical component 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. A component has a globally unique identifier, in which versioning information is cryptographically encoded, so that updates can be provided only by the originally manufacturer of the component. There is a global name space of interfaces. No concept of connectors is present; since dependencies are specified through component identities (as opposed to interfaces or ports), the environment can resolve each requirement in a unique way. There are no composite components. It is unclear at this time whether there will be support for anticipated adaptation.

Process-level support. There is no Process-level support so far. Especially, there is no way to visualize the overall architecture of an application; all architectural information is encapsulated in the components and cannot be separated from them.

Summary. Dot-Net is intended to be the future platform of MS Windows. It attempts to unify the strengths of Win 32, such as performance and backwards compatibility, with those of the Java Virtual Machine, such as platform independence and component deployability. In how far this will be successful remains to be seen.

4.7 Plug-In Systems

Systems with a plug-in architecture, such as Netscape Navigator [103] and Adobe Photoshop [4], provide an easy, low-overhead way of extending their functionality. A complex, large application, often called the plug-in framework, defines an API that other manufacturers can use to extend the functionality of the application through plug-ins. The extent to which a plug-in can cooperate with the framework, or how closely coupled the two products are, depends on the API in question. All plug-ins are optional; a plug-in can extend the functionality of the framework, but it is not required for its use. Also, the number of plug-ins is unlimited; 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 plug-in API, and may differ for each framework. Since plug-ins provide a functionality, composition is type-based. Since there is only one framework in each system, the architecture of composed applications is limited to two levels (platform level and component level), and there is no need for composite components or component adaptation. Because of their specialized nature, plug-in systems typically have only a small number of available components per framework.

Process-level support. Since plug-ins are developed for a 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.

Summary. Plug-ins are an ad-hoc solution for platforms that want to interoperate with a small number of very specific components. As such, they are very useful, but their scope cannot easily be extended.

Related approaches. Compound document standards, such as Opendoc [100] or OLE [17], 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 components, in compound document systems each program can act both as framework and as component. This makes it possible, for example, to embed spreadsheets into a text document, and conversely to embed text documents into a spreadsheet. Compound document technologies have largely been replaced by component models such as COM and Java Beans.

4.8 Comparison

Figure 7 gives an overview of the historical relations between the approaches and the research areas. On the horizontal scale, approaches are laid out by area; on the vertical scale, approaches are laid out by approximate chronological sequence. An arrow from A to B means that A influenced B, or that B is in some way based on A.

Tables 6 and 7 summarize the surveyed composition environments and their properties. Each row corresponds to one feature, each column to one approach. As can easily be

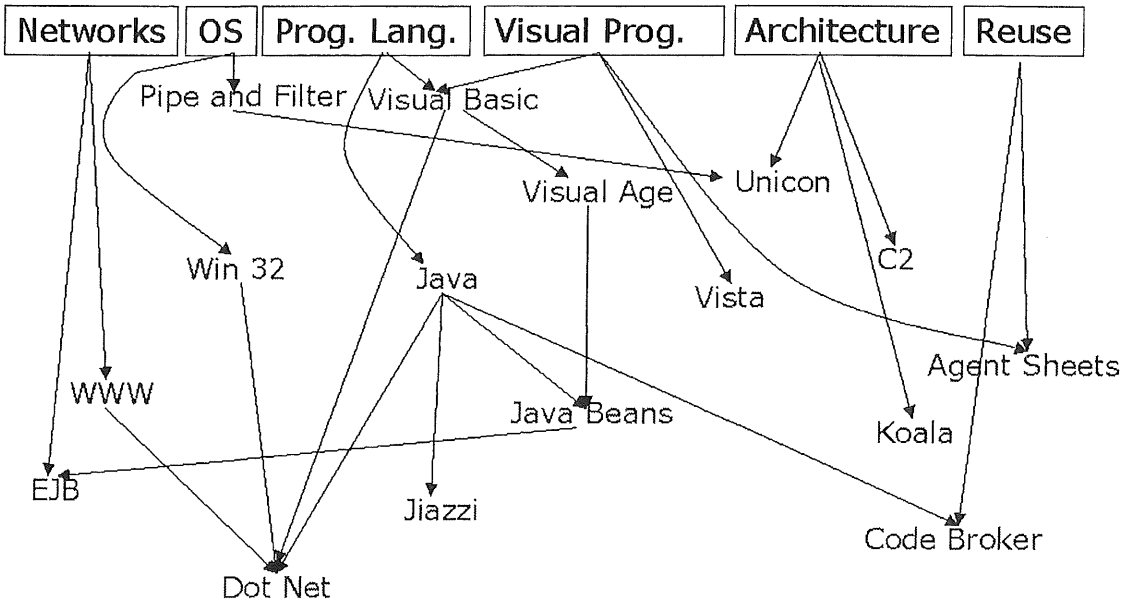

Figure 7. Historical relations among composition environments.

seen, the approaches vary in their focus; none implement functionalities from the whole spectrum of features described in Section 3. Almost all approaches focus either on component model concerns or on process level concerns, but not on both. Features related to self-description are not represented very often, even though there is theoretical agreement on the importance of component self-description. It is interesting to see that approaches stemming from very different research areas often focus on similar issues in the feature spectrum.

		Archstudio	Koala	Java	Jiazzi	Visual Basic	Visual Age	Bean Box	Vista
1.1 Components	Categories	process	proc.lib.	class	class lib.	proc. lib.	object	object	user-def.
	Types	+							
	Global identity		+						
	Versioning	+	+	...		...			
1.2 Interfaces	Interfaces	untyped	+	+	+			+	
	Instances (ports)	P1 R1	P* R*	P* R0	P* R*			P* R0	P* R*
	Global identity		+	namesp.	namesp.			namesp.	
	Location		ref.	ref.	ref.			ref.	
	Versioning	+				+			
1.3 Self-Description	Syntax		+	+		+	+	+	
	Semantics						informal	informal	
	Quality of service								
	Non-technical								
1.4 Configuration	Connection Semantics	event	type	type	type	event	event	event	user-def.
	Connectors	+	+	+	+		+	+	+
	Connector types (no.)	user-def.		1	1		3	2	user-def.
	Connection cardinality	n-n	1-n	1-n	1-1	n-n	1-n, 1-1	1-n, 1-1	user-def.
	Anticipated adaptation					+	+	+	+
	Composite components		non-hier.		non-hier.				non-hier.
2.1 Searching	Remote Search								
2.2 Lever. Self-Description	Syntax					+	+	+	+
	Semantics						+	+	
	Quality of service								
	Non-technical								
2.3 Configuration	Composition notation	scr, dia	scr, dia	prog	scr	prog, dia	prog, dia	prog, dia	scr, dia
	Ad-hoc adaptation	...							
	Constraints	+							+
	Guaranteeing consistency	fly, analysis		runtime	analysis	on-the-fly	on-the-fly	on-the-fly	on-the-fly
	Distributed applications	+		+			?		
2.4 Execution	Partial applications	+		+		+	+	+	+
	Packaging	+	+	+	+	+	+	+	
	Runtime changes	+		...				+	

Table 6. Comparison of approaches, part 1. Legend: **+** yes, (empty) no, **...** some, * variable number, **P** provision ports, **R** requirement ports.

		Code Broker	Agent Sheets	EJB	Web	Pipe/F.	Win 32	Dot-Net	Plug-In
1.1 Components	Categories	NA	object	class	process	process	proc. lib.	proc. lib.	proc. lib.
	Types								
	Global identity				+		...	random id	
	Versioning			...	+		...	...	
1.2 Interfaces	Interfaces			+		untyped	+	+	untyped
	Instances (ports)		P4 R4	P* R0		P1 R1	P* R0	P* R0	P1 R1
	Global identity			namesp.			random id	random id	
	Location			ref.			cop.	cop.	
	Versioning			+					
1.3 Self-Description	Syntax		+	+	+			+	
	Semantics			informal				informal	
	Quality of service								
	Non-technical			...				...	
1.4 Configuration	Connection Semantics		event	type	event	stream	type	type	type
	Connectors		+		+	+			
	Connector types (no.)					1			
	Connection cardinality			1-n	1-n	1-1	1-n	1-n	1-n
	Anticipated adaptation		+	+				+	
	Composite components			non-hier.				non-hier.	
2.1 Searching	Remote Search								
2.2 Lever. Self-Description	Syntax	+	+						
	Semantics	+							
	Quality of service								
	Non-technical								
2.3 Configuration	Composition notation	prog	dia	prog	script	script	prog	prog	scr
	Ad-hoc adaptation								
	Constraints								
	Guaranteeing consistency			runtime				runtime	
	Distributed applications			+			+	+	
2.4 Execution	Partial applications		+	+	+	+	+	+	N/A
	Packaging			+		+	...	+	
	Runtime changes		+						

Table 7. Comparison of the approaches, part 2. Legend: **+** yes, (empty) no, **...** some, * variable number, **P** provision ports, **R** requirement ports.

5 Conclusions

This survey has classified issues surrounding composition environments, and described and compared several such environments with the help of this classification. As a result, we can now characterize what is missing from existing systems for enabling end-user composition of applications.

Most importantly, an integrated approach to composition is needed. Composition environments have been built in both industry and research, but none of them integrates all capabilities that are desirable. Operating system or networking based approaches focus on performance, but require extensive technical experience from the user. Visual programming environments, on the other hand, provide user guidance, but do not sufficiently address the underlying component model. We need to combine solutions from these areas in order to build composition environments that are both usable and performant.

Many existing approaches are domain dependent in some way. They support only a specific kind of program (such as programs that focus on graphical user interfaces), or support only a specific architectural style (such as client-server architectures). While domain specific solutions are useful, there seems to be a lack of universal, domain independent solutions in the area of composition environments. The degree of universality that a composition environment could achieve is an open question.

Several important concepts of composition environments seem not to be employed as frequently as one might wish. Component self-description, connectors, composite components, and component adaptation are generally accepted as good things by researchers, but are not used in some of the most popular environments. The reason for this seems to be that many questions about their realizations are still open:

- Should interfaces be referenced or duplicated, or do intermediate solutions exist?
- Should self-description be static or dynamic?
- In what format should data be described?
- Should description be required or optional?
- How much of it can be automatically generated and checked?
- Should connectors be light-weight or heavy-weight (i.e. with complex implementations of their own)?
- What are the trade-offs between the different connection semantics, and can they be combined?
- Should composite components be hierarchical or non-hierarchical?
- How can adaptation be expressive without breaking encapsulation?
- Are diagrams or scripts preferable as composition notations?
- How much support for end-users is desirable, and how different is it from support for developers?
- Which process steps have to be integrated into a composition environment, and which can be left to external tools?

Scalability seems to be a key issue in the design of composition notations. While many technologies work fine on a small scale, when a large number of components are in-

volved, they become unusable. Especially diagrammatic approaches seem to suffer from this problem: diagrams that show more than a few components typically require careful manual layout to stay readable. But some text-based approaches, such as pipe-and-filter systems and Java class paths, are also severely limited in size. Scripting languages are an alternative, but they tend to deteriorate over time by becoming more and more complex. Many of these scalability issues have been solved on the programming language level (for example, through the use of name spaces), but the solutions have not been ported to composition environments. Both architectural solutions (for example, new mechanisms for composite components) and solutions on the user interface level (for example, new layout algorithms) might be feasible. It is not clear yet how powerful exactly a composition notation should be. Novel language features needed by composition notations, such as type-system support for non-functional properties, are still a matter of research [124].

Compared to the large number of systems supporting source code components [56, 81], few environments for deployable components exist. We believe, however, that deployable components are fundamentally different from pieces of source code, because their implementation is hidden and cannot easily be changed. It is known that requirements elicitation [19], testing [60, 97], and maintenance [133] present new problems in the context of deployable components. Many software engineering techniques still need to be applied (and adapted, if necessary) to deployable components. For example, the Unified Modeling Language [89], which provides vast support for source-based design, does not provide sufficient support for design with deployable components [61].

Software architecture addresses many issues that are also addressed by component technology, so a clear definition of the commonalties and differences between the two research areas would be desirable [84]. In general, it seems that architecture research focuses more on the overall structure of applications and on the connections between components, while component research focuses more on the structure of the components themselves. But both problem areas are intertwined. Each component model introduces some architectural constraints; what constraints are acceptable is an important question in the design of a component model. On the other hand, architecture description languages would be much more effective if they could enforce architectures instead of just specifying them [6]; for this to happen, they need to be integrated with component technologies. Composition environments should unify research from both communities.

As we have shown, a large number of open research questions exist in the field of composition environments. We believe that composition environments constitute an important research area with the potential of having a large impact on the future of software engineering.

6 Research Plan

We will briefly outline possible topics of future research in this section. We believe that end-user composition of applications is a promising technology, and we identify some preconditions that need to be met for this technology to succeed.

It is our opinion that there is a large space in the usability/expressiveness spectrum that has not been sufficiently explored. By reusing premanufactured components and developing adequate mechanisms to build applications from them, it might be possible to fill this void. One half of this problem is to provide useful, usable components. Component self-description, our first subsection, will be the biggest part of the solution to this problem. The other half of the puzzle is to provide tools and notations that make it easy to build applications without restricting expressiveness; we will investigate this in the context of process support. Third, remote services are a research area of strong current interest; to integrate them with composition environments seems promising. Finally, we will investigate a possible case study for the use of a composition environment.

Self-Description

Component metadata, realized in the form of component self-description, are essential for reusability of components. Reuse can succeed only when the effort needed to understand a component is significantly less than the effort needed to reimplement it. Component self-description helps to make components understandable. It can provide information both to users and to tools. Information provided to tools can be modified before it is presented to users.

We are planning to define mechanisms to realize self-description in flexible, extensible ways. It should be possible to implement only a minimum of self-description, for example for components intended to be reused by in-house projects, so that extensive documentation is not needed. But it should be equally easy to give full self-description of all the potentially relevant properties of a component without interfering with the component's functionality. Further, self-description should be dynamic, so that self-adapting components or components that wrap external functionality, which might change over time, are possible.

Quality-of-service (not just performance) self-description seems especially promising. It poses the question of how to describe non-functional properties. As far as possible, they should be described in machine-readable notations, so that automatic evaluations can be performed. Questions to be answered are: Which non-functional properties of software are suitable for such notations? What do the notations look like? To what degree can a composed application be described automatically based on the descriptions of its constituent components?

The role of interfaces in component self-description has not been adequately researched yet. Many component properties that should be described are, in fact, properties of interfaces, part of the contracts that exist between components. But interfaces generally exist in the context of a type system, and that raises the question how non-functional specifications affect this type system. In other words, we need to find ways to represent self-description in interfaces without negatively influencing their traditional tasks.

Process Support

The different steps of the application composition process require a variety of tool support. Of course, the desirable amount of tool support may be unlimited, because it is always possible to include more sophisticated tools into an integrated environment. Therefore, the question becomes relevant which tools should be closely tied to a composition environment, and which tools may be added externally.

We are planning to identify those tools that need to be closely integrated, so that implementation of a user-friendly composition environment becomes possible without having to bother with less essential functionality that can be added later. Starting with the core functionalities of configuring an application out of components and then executing it, we will prioritize desirable features of composition environments and investigate if they can benefit from inclusion into the "kernel" of the composition environment.

For example, integration of component download from a component repository on the Internet should be integrated into the core composition environment, as we have shown in previous research [62]. The problem of maintenance of component-based applications becomes easily untractable because of the potentially large number of components from different sources in one application. Maintaining hundreds of components from dozens of sources manually is very difficult. Therefore, automated support for notification about updates, retrieval and integration of updates is desirable.

Remote Services

Remote services are services that are offered through a network and that can be used by software that is not part of the organization offering them. When they are provided through the Worldwide Web, remote services are often called Web services. A reusable service makes it possible to acquire another organization's development effort, just like a reusable component. Analogous to the connection semantics defined in Section 3, downloading a component establishes a type-based connection between the component user and the component provider, while accessing a remote service establishes an instance based connection between client and server.

Thus, remote services try to solve the same problem as components. We believe that it makes only sense to take a closer look at the commonalties and differences between the two. It seems promising to explore possibilities to integrate support for both components and services in one environment. It is known that components and services complement each other: components are well-suited for tasks that are frequently executed and do not require a large amount of data, while remote services are suited for infrequent tasks that depend on a large amount of data. For example, a container library would be a good example of a component, while a natural language translation system would be a good example of a service.

Self-description seems to be a very promising candidate for unification. Both services and components require self-description in order to be usable by third parties. Thus, when developing self-description mechanisms for components, we will explore how they can be extended to remote services. Similarly, we will investigate which other parts of a composition environment might profit from integration between components and remote services.

Case Study

To evaluate a prototypical composition environment, an appropriate case study will have to be defined. Performing a case study for a composition environment means identifying a domain with a sufficient number of available components, and using those components to build a number of different applications using the composition environment. Since deployable components have to be compatible to the composition environments, a domain with open source components will be needed. These open source components will have to be converted manually into deployable components of the appropriate format.

The selection of the application domain for this case study will mainly depend on the availability of components. However, the area of applications for space-constrained platforms with graphical user interfaces, such as handheld computers, seems especially promising. A user interface is necessary so that end users have access to the system, and can configure it according to their needs. Space-constrained systems have the property that only a limited amount of software can be installed at a given time, which increases the need for component technologies. In systems with a large storage space, such as most workstations, software can just be duplicated inside of preconfigured, monolithic applications. Limited space systems make it desirable for both developers and users to create applications in a flexible, space-saving, component-based form.

Bibliography

1. Gregory D. Abowd, 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.
2. Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In *Joint Modular Languages Conference 2000.* Springer, Berlin, 2000, 77-89.
3. Active Server Pages Guide. http://msdn.microsoft.com/library/psdk/iisref/aspguide.htm. 2002.
4. Adobe Photoshop: Application Programming Interface Guide. Version 6.0 Release 1. http://partners.adobe.com/asn/developer/gapsdk/download/win/Photoshop60sdk.zip. 2000.
5. AgentSheets. http://agentsheets.com. 2002.
6. Jonathan Aldrich, Craig Chambers and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *2002 International Conference on Software Engineering.* 2002, to appear.
7. Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology* 6, 3 (1997), 213-249.
8. ArchStudio 3. http://www.isr.uci.edu/projects/archstudio/.
9. BEA WebLogic Server. Version 6.1. http://www.bea.com/products/weblogic/server/index.shtml.
10. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau and Damien Watkins. Making Components Contract Aware. *Computer* 32, 7 (1999), 38-45.
11. Dietrich Birngruber, Werner Kurschl and Johannes Sametinger. Comparison of JavaBeans and COM/ActiveX - A Case Study. In *5. Fachkonferenz Smalltalk und Java.* Erfurt, 1999.
12. Alex Borgida 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.
13. Borland Delphi. Version 6. http://www.borland.com/delphi/.
14. Jan Bosch. Adapting Object-Oriented Components. In *Object-Oriented Technology.* Springer, Berlin, 1998, 379-383.
15. Don Box. *Essential COM.* Addison-Wesley, Reading, 1997.
16. Paul Brereton and David Budgen. Component-Based Systems: A Classification of Issues. *Computer* 33, 11 (2000), 54-62.
17. Kraig Brockschmidt. Developing Applications with OLE 2.0. http://msdn.microsoft.com/library/en-us/dnolegen/html/msdn_devwole2.asp. 1994.
18. Alan W. Brown and Kurt C. Wallnau. The Current State of CBSE. *IEEE Software* 15, 5 (1998), 37-46.

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

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

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

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

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

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

25. E. M. Dashofy, 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.

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

27. Elisabetta Di Nitto 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.

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

29. Andrew Duncan 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.

30. Laura Euler, 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.

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

32. Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henryk Frystyk Nielsen, Larry Masinter, Paul J. Leach and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1.* Request for Comments 2616. Internet Engineering Task Force, 1999.

33. Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology* (to appear).

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

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

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

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

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

39. Michael Gorlick 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.

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

41. George T. Heineman 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/.

42. André van der Hoek, 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.

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

44. Jon Hopkins. Component Primer. *Communications of the ACM* 43, 10 (2000), 27-30.

45. M. Horton and R. Adams. *Standard for Interchange of USENET messages.* RFC 1036. Internet Engineering Task Force, 1987.

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

47. *Internet Message Format.* RFC 2822. Internet Engineering Task Force, 2001.

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

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

50. JavaServer Pages Specification. Version 1.2. http://java.sun.com/products/jsp/. 2001.

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

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

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

54. Rohit Khare. 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.

55. Wojtek Kozaczynski. Composite Nature of Component. In *Proceedings of the 2nd Workshop on Component-Based Software Engineering.* Los Angeles, 1999. http://www.sei.cmu.edu/cbs/icse99/papers/.

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

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

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

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

60. Chang Liu and Debra Richardson. Software Components with Retrospectors. In *International Workshop on the Role of Software Architecture in Testing and Analysis.* Marsala, 1998.

61. Chris Lüer and David S. Rosenblum. UML Component Diagrams and Software Architecture. In *1st ICSE Workshop on Describing Software Architecture with UML.* Toronto, 2001, 79-82.

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

63. Chris Lüer, 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.

64. Mark Lycett 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/.

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

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

67. Sean McDirmid, 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.

68. Nenad Medvidovic, 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.

69. Nenad Medvidovic, 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.

70. Nenad Medvidovic, David S. Rosenblum and Richard N. Taylor. An Architecture-Based Approach to Software Evolution. In *Proceedings of the ICSE '98 Workshop on the Principles of Software Evolution (IWPSE '98).* Kyoto, 1998.

71. Nenad Medvidovic, 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.

72. Nenad Medvidovic, 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.

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

74. Nikunj R. Mehta, 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.

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

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

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

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

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

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

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

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

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

84. Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys* 27, 2 (1995), 262-264.

85. Jim Q. Ning. A Component Model Proposal. In *Proceedings of the 2nd Workshop on Component-Based Software Engineering.* Los Angeles, 1999. http://www.sei.cmu.edu/cbs/icse99/papers/.

86. David Notkin, 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.

87. Helgo Ohlenbusch 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.

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

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

90. Rob van Ommering. Koala, a Component Model for Consumer Electronics Product Software. In *Development and Evolution of Software Architectures for Product Families.* Springer, Berlin, 1998, 76-86.

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

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

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

94. Peyman Oreizy, 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.

95. Peyman Oreizy, 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.

96. Peyman Oreizy, David S. Rosenblum and Richard N. Taylor. *On the Role of Connectors in Modeling and Implementing Software Architectures.* Technical Report UCI-ICS-98-04. Department of Information and Computer Science, University of California, Irvine, Irvine, 1998.

97. Alessandro Orso, 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.

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

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

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

101. František Plásil 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.

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

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

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

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

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

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

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

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

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

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

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

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

114. Mary Shaw. *Procedure Calls Are the Assembly Language of Software Interconnection.* Technical Report CMU-CS-94-107. School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994.

115. Mary Shaw, 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.

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

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

118. *Simple Mail Transfer Protocol.* RFC 2821. Internet Engineering Task Force, 2001.

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

120. *Software Reusability.* Ellis Horwood, New York, 1994.

121. Kevin J. Sullivan, 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.

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

123. Clemens Szyperski. Independently Extensible Systems —Software Engineering Potential and Challenges—. *Australian Computer Science Communications* 18, 1 (1996), 203-212.

124. Clemens Szyperski. Modules and Components — Rivals or Partners? In *The School of Niklaus Wirth: The Art of Simplicity.* D-Punkt, Heidelberg, 2000, 121-132.

125. Clemens A. Szyperski. Import is Not Inheritance—Why We Need Both: Modules and Classes. In *ECOOP '92 European Conference on Object-Oriented Programming.* Springer, Berlin, 1992, 19-32.

126. Richard N. Taylor, 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.

127. Anne Thomas. Enterprise Java Beans Technology. 1998.

128. Vincent Traas 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.

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

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

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

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

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

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

135. Jim Waldo, Geoff Wyant, Ann Wollrath and Sam Kendall. A Note on Distributed Computing. In *The Jini Specification.* Addison-Wesley, Reading, 1999, 307-326.

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

137. Sanjiva Weerawarana, 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.

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

139. E. James Whitehead, 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.

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

141. Yunwen Ye and Gerhard Fischer. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *Proceedings of the 2002 International Conference on Software Engineering.* 2002, to appear.

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

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