Chapter

Software Architecture and Java Beans

Sylvia Stuurman
Delft University of Technology

Key words: Java Beans, Software Architecture, Component Based Development

Abstract: When thinking about it in theory, Software Architecture and Component-Based Development make an ideal match: the concerns of Software Architecture are high level design, interaction, and configuration of components, while Component-Based Development is centered around the implementation and specification of reusable components.

Together, these concerns seem to be the yin and yang for the development of complex systems out of existing components. Several authors already explained that in reality, there is a gap between the two areas.

In this paper, we investigate the relation between Java Beans and a software architectures description: may Java Beans simply be used as ready-to-use implementations of a software architecture? Which restrictions do they inflict on the software architecture? Where are the mismatches?

1. INTRODUCTION

As has been argued many times, todays complex large-scale software systems ask for a different kind of software engineering than small and simple programs.

On the one hand, there is a need for very high-level design. The level of abstraction should be higher than that of objects, or procedures. Moreover, such a design should be a model of the system-in-use, and not only a model

of the implementation (Allen and Garlan, 1994). Software architecture is an answer for this need (we use ‘software architecture’ in the sense of the definition of Garlan and Shaw (Garlan and Shaw, 1993): ‘Structural issues include gross organisation and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.’).

On the other hand, there is a need for the reuse of components. The ideal with this respect is, for a developer, to be able to shop among different component-providers, and build a system in the same way as building a vehicle out of Lego bricks and pieces. Several standards, both commercial and non-commercial, for component models have arized, such as CORBA, ActiveX, or Java Beans.

Software architecture seems a natural complement for reusable software components: existing component middleware technologies are component-centric, they standardize external component properties. Software architectures are system-centric, with more emphasis on the connections, and the properties of the system as a whole.

One of the problems of building systems out of existing components is the possibility of an ‘architectural mismatch’ between components (Garlan et al., 1995). Components make implicit assumptions about the nature of the components (infrastructure, control model, data model), the nature of the connections (protocols and data model) and about the global architectural structure (for instance about the presence or absence of particular components or connections).

As part of a solution for this problem, it has been suggested (Garlan et al., 1995) that these architectural assumptions could be made explicit using an Architecture Description Language (or ADL). Architectural descriptions could be used to understand the concepts embodied in component libraries (Perry and Wolf, 1992).

However, there are problems to overcome:

ADL’s are created for the specification of software architectures, and software architectural styles. They have not been created with the component standards like CORBA, ActiveX or Java Beans in mind. The two domains use similar, but incompatible models of components and component bindings, revealed when comparing the Interface Description Languages for components, with the possibilities of the ADL used for software architecture.

Moreover, the mapping of components at the software architectural level to components at the implementation level might be feasable, how other architectural elements should be mapped is unclear.

Furthermore, an architecture describes the system as a whole, while reusable components make use of services provided by the middleware infrastructure and the operating system. In fact, these services should be modeled at the architectural level, to get a real mapping between both levels (Oreizy et al., 1998).

Anothor question is **how** software architecture and component-based development can be combined:

One way of combining these domains is to start with the design of the system's architecture. The architecture should be refined until one is able to choose or build existing components, based upon the architectural specification. These components should be connected according to the architecture. Designing the systems's architecture is, in this case, the specification; "filling in" the components, is the implementation. However, when one first completes the software architecture without the components at-hand in mind, the chance that one can really reuse them is very low. The "inevitable intertwining of specification and implementation" (Swartout and Balzer, 1982) is especially valid when reusable components are involved.

Another way of combining software architecture and component-based development is to build a system using existing components, and describe the architecture of such a system in an Architecture Description Language. The description can be used for analysis.

In this paper, we explore the possibilities of both ways of combining component-based development and software architecture, for the component model of Java Beans. On the one hand, we investigate how we can map a software architecture onto an application of connected Java Beans. We use the framework for classifying ADL's by Medvidovic and Taylor (Medvidovic and Taylor, 1997) to cover the different aspects which might or should be included in an architectural description of a system. On the other hand, we summarize the requirements for an ADL, usable to describe the architecture of an application built by connecting beans.

In section 2, we will give a short overview of the features and concepts of Java Beans. In section 3, we discuss the (im)possibilities of mapping architectural elements onto Java Beans. In section 4, we do the same the other way around. Related work is mentioned in section 5, and in section 6, we discuss how to carry on.

2. JAVA BEANS IN SHORT

Java Beans are pieces of software, written in Java, in such a way that it is possible to build applications by connecting beans, in a 'bean-aware' application builder. Such an application builder is able to get information of the bean about its properties, methods, and the events it fires. The user of the application builder may change properties, and connect different beans through events, thus building an application. Everything is done through dragging and dropping, or by filling in property sheets.

In a bean's lifetime, one may discern three different stages: in the first place, a bean should be created. In this paper, we are not concerned with beans programming, and we will just assume the existence of a library of ready-to-use beans. In the second place, a bean is used during the design of an application. The application builder tool discovers its properties, methods and events, the user or developer instantiates the bean, customizes the instances, and connects instances of (the same or different) beans. Of course, a bean may be used multiple times, during the design of different applications. The third stage is the existance (as an instance) in a running application. With "design-time" we will refer to the second stage described above; a bean in a running application is referred to with "run-time".

According to the Java Bean specification (Hamilton, 1997), a Java bean is a reusable software component with at least:

- Support for introspection. Beans are constructed in such a way that an application builder may discover a bean's properties, methods, and events by introspection.
- Support for properties, to be inspected or changed: customization. Properties are a bean's appearance and behavior attributes that can be changed at design time.
- Support for events, for communication between beans. A bean that wants to receive events (a listener bean) registers its interest with the bean that fires the event (a source bean). Builder tools can examine a bean and determine which events that bean can fire (send) and which it can handle (receive).
- Support for persistance. Persistence enables beans to save their state, and restore that state later.

A bean interacts with its environment through its set of properties, its set of methods, and the set of events it fires. Properties are attributes that can be read and written. Methods are normal Java methods that can be called from outside the bean. Events that are fired by beans invoke methods in

beans that have subscribed on the particular class of events. These beans adhere to the EventListener interface. An event-firing bean and an EventListener bean may be decoupled by placing an EventAdapter bean inbetween them.

Some properties of event delivery for Java Beans are:

- Event delivery is multicast: one event that is fired, invokes an associated method in every bean that has subscribed on the event.
- Event delivery is synchronous with respect to the event source: the associated method in the EventListener bean is executed in the thread of the event-firing bean.
- The set of EventListeners for a certain event may be changed dynamically.

2.1 Example

2.1.1 Connections with Events

Imagine a BallThrowing Bean. Throwing a ball is implemented using an event, for which a BallEventObject class is created.

When one uses such a Bean in a bean-aware application builder, one may instantiate instances of the BallThrowing bean, and connect them through the BallEvent. A bean (a BallThrowing bean or any other bean that can handle BallEvents) is connected by stating that the bean listens to the BallEvents sent by the BallThrowing bean, and by specifying which action should be performed when receiving a BallEvent.

A BallThrowing bean class should have a list of BallEventListeners, and methods to add and remove objects to and from that list. These methods are used in the application-buolder, when connections are made and undone.

A BallEventListener bean should have an action method which has a BallEventObject as an argument.

```
public class BallThrower {
   private Vector ballCatchers = new vector();

    public synchronized void addBallEventListener(BallEventListener c) {
      ballCatchers.addElement(c);
    }

   public synchronized void removeBallEventListener(BallEventListener c)
   {
```

```
        ballCatchers.removeElement(c);
    }

}

public interface BallEventListener {
    void catchBall(BallEventObject ball);
}

public class BallEventObject extends EventObject {
}
```

2.1.2 Properties

A bean-aware application builder simply searches for set- and get-methods to find the properties of a bean. The BallThrower bean for instance, could have the number of balls it posseses, as a property:

```
public void setNumber(Integer number) {
    this.number = number;
}

public Integer getNumber() {
    return number;
}
```

Changes to properties may be notificated to other beans. Such a property is called **bounded**. A bean with a bounded property maintains a list of PropertyChangeListeners (beans implementing the PropertyCangeListener interface), and it sends a PropertyChangeEvent to those listeneres when the bounded property has been changed.

A property may be **constrained** as well. In this case, the bean maintains a list of VetoableChangeListeners, which are able to check whether a value of the constrained property is within the constraints. The setProperty method of such a bean raises an exception when one of the listeners uses his veto.

A bean-aware application builder recognizes that a property is bounded or constrained, and offers the user of the application-builder the possibility to indicate which other beans will act as listeners.

2.1.3 Introspection

In the samples of a bean shown above, we have used conventional names and type signatures of methods and interfaces, as a means for introspection. Beans-aware application builder look for set- and get-methods, and addeventlisteners and removeeventlisteners methods, to find the properties of a bean and the events with which it can be connected.

A Java Bean may also explicitly specify its properties, events and methods, using a class implementing the BeanInfo class.

2.2 Status and Environment

Because communication between beans consists of event notification and direct method invocation, it is necessary that beans run in the same address space, in this case in the same Java Virtual Machine.

Another environmental aspect of beans is that they should assume that they are running in a multithreaded environment: several different threads may simultaneously deliver events, or call methods directly.

Several extensions have been proposed:

- InfoBus (Colan, 1998) from Lotus Development is already available. This extension offers a new type of connection between beans: data flows. Beans may subscribe on certain kind of data (based on a name). Other beans produce data. Application builder tools are able to extract from a bean the names of the data it is able to produce. This communication mechanism is known as subscription-based communication (Boasson, 1996). This type of connection is attractive with respect to the introduction of on-line changes (Stuurman and van Katwijk, 1998).
- JavaSpaces (Sun, 1998) is available as a beta version at this moment. JavaSpaces provides a distributed persistence and object exchange mechanism. It is comparable with InfoBus, for communication between beans in different Java Virtual Machines.
- An extensible runtime containment and services protocol has been proposed (Cable, 1998). This protocol supports extensible mechanisms that introduce an abstraction for the environment of a bean, enable the dynamic addition of arbitrary services to a bean's environment, provide a mechanism through which a bean may interrogate its environment, and provide a mechanism to propagate

an environment to a bean. In short, the notion of the context of a bean is introduced in this extension.

- Another extension is the Java Beans Activation Framework (Calder and Shannon, 1998). This framework supplies the services of determining the type of arbitrary data, encapsulating access to data, discovering the available operations on a particular type of data, and instantiating a software component that corresponds to the desired operation.

3. USING BEANS TO IMPLEMENT AN ARCHITECTURE

The idea of using beans to implement a given software architecture looks promising and desirable: beans are components in the architectural sense: loci of computation and data storage; one has the multi-platform benefits of the Java language; there is a possibility to have a visual image of the application, consisting of connected components as a mirror of the software architecture it implements. The idea would be to look for (or build) beans that match the specification of the components of the given architecture, and connect them according to the given configuration.

Which aspects of an architecture are specified depends on the ADL which is used. We will not adhere to one specific ADL, but check the aspects used in the classification framework for ADL's by (Medvidovic and Taylor, 1997). These aspects are: interface, types, semantics, constraints and evolution of components and connections; compositionality, heterogeneity, constraints, refinement, scalability, evolution and dynamism of configurations.

When examining the possibility of a mapping between Java Beans and an ADL, we will mainly look at those aspects that are specific to Benas (as opposed to the general aspects of the Java language). Thos aspects are the most interesting, because they have been specified for the convenience of tool-builders. Next to bean-aware application builders, one may just as easily construct bean-aware software architecture tools.

3.1 Components

The interface of a component in the software architectural sense is the set of interaction points between the component and the external world. An interface specifies the services a component provides, and it might specify the needs of a component. The interface of a Java Bean is its set of properties, its methods, and the events it fires. This information may be extracted from a bean at design-time, so one may use an application builder tool to expose the interface. Mapping the interface of a component, specified in an ADL, to the interface of a Java Bean seems rather straightforward, though, of course, not every aspect of an interface that one can specify in an ADL has a counterpart in Java Beans.

ADL's may model abstract components as types, and instantiate them multiple times. Some ADL's allow abstract component types to be parameterized. A Java Bean may be regarded as a parameterized component type as far as it can be customized. A bean may be instantiated as often as one needs. So, parameterized types are directly supported by Java Beans. On the other hand, not every imaginable component type can be implemented using a Java Bean.

A software architecture specification may contain a model of the component semantics. In a Java Bean however, semantics are not exposed. When using beans in an application builder, the user is obliged to rely on the documentation, supplied with the beans.

An ADL may specify constraints on the abstract state of a component, the implementation, or non-functional properties. With respect to the abstract state of a component: Java Beans have the notion of constrained properties. When such a property is changed, another bean validates the change. A mapping between constraints on the abstract state of a component and constrained properties of a bean seems possible.

ADL's may support design evolution through subtyping and refinement. A mapping between such a support and an implementation using Java Beans might be useful for prototyping. However, subtyping and refinement of Java Beans in an application builder is not supported.

In tabel one, we summarized which aspects of components, described in an ADL, may be mapped to those aspects of Java Beans that are visible for bean-aware tools.

Table 1: Mapping Components in a Software Architecture Description to Java Beans

ASPECT	MAPPING
interface	Possible, beans support a subset

types	Possible, beans support a subset
semantics	Not possible
constraints	Possible, beans support a small subset
evolution	Not possible

3.2 Connections

In an application builder using Java Beans, one glues beans together by connecting them using event notification. An event of an event-firing bean is associated with a method of an event-listening bean. A special case is the notion of constrained properties. A bean with constrained properties is associated with a validator bean. Each time (at run-time) that a property is changed, the change is validated.

InfoBus and JavaSpaces extend this type of connection with the possibility of asynchronous, anonymous data communication. Beans may produce data, and may subscribe to certain kind of data. Producers don't have to wait until every consumer has seen the produced data. Producers and consumers are unaware of each other.

Other kind of connections (create connections for instance) are possible, but cannot be made visible in an application builder, and are 'hidden' in the code of the bean.

The interface of a connection in a software architecture is a set of interaction points between the connection and the components attached to it. Each kind of connection that can be used for Java Beans has its own interface: events are of a certain class and should be connected to an eventsource and a set of eventlisteners; InfoBus connections are associated with a name and should be connected to a set of data producers and a set of data consumers. Of course, not every interface that one can specify in an architecture has a counterpart in a Java Beans application.

Some ADL's distinguish connection types from connection instances. Events in Java Beans are always of a certain class, that can be subclassed. So, for event-based connections, one may map the idea of a connection type to an event connection.

Some ADL's provide means to express the semantics of connections. For the connections possible in a Java Beans application, one should specify the

semantics of these connections once. Of course, in an architecture, one can specify connections with semantics that have no counterpart in a Java Beans application.

Connection constraints may consist of adherence to interaction protocols, intra-connection dependencies, or usage boundaries. In general, Java Beans give no support to translate these kind of constraints.

Some ADL's provide support for connection evolution, through subtyping or refinement. Again, Java Beans give no support.

Table 2: Mapping Connections in a Software Architecture Description to Java Beans

ASPECT	MAPPING
interface	Possible, beans support a subset
types	Possible, beans support a subset
semantics	Not possible (one should first specify beans connections)
constraints	Not possible
evolution	Not possible

3.3 Configurations

Compositionality: some ADL's support situations where an architecture becomes a component in a bigger architecture. Such a composition can be mirrored in Java Beans, where a composition of interconnected beans may be transformed into one new bean.

Many ADL's have the possibility to specify global constraints. In general, it will not be possible to map these constraints to visible properties of a Java Beans application.

Darwin, Rapide and C2 allow specification of dynamism in architectures. Insertion and removal of both components and connections is possible in Java applications, but one cannot extract information about this behaviour by introspection.

Table 3: Mapping Configurations in a Software Architecture Description to Java Beans

ASPECT	MAPPING
composition	Possible

constraints	Not possible
evolution	Not possible

4. USING AN ADL TO DESCRIBE A BEANS CONFIGURATION

The previous section showed that not every software architecture can be mapped onto a configuration of Java Beans. Not every part of an architecture description is translatable into either a Java Bean, or a connection between beans. When using beans to construct a system based on a certain software architecture, one should check the types of components and the types of connections.

Automating such a process is only attractive when one conforms to the subset of architectures that can be implemented using beans.

On the other hand, it seems to be the case that an application built by connecting Java Beans, may be translated relatively easily into an architectural description. One should choose an ADL based on how much of the information, available in a beans application, can be described. In the remainder of the section, we make use of the classification of (Medvidovic and Taylor, 1997), for ADL's, and we take only those ADL's into account that are part of the survey : Aesop, MetaH, LILEANNA, ArTek, C2, Rapide, Wright, UniCon, Darwin, SADL and ACME.

4.1 Beans

The properties, methods and the events a bean can fire, should be translated into an interface specification. All ADL's support specification of component interfaces.

The language should provide the means to specify parameterized types, with the properties that can be changed at design time as parameters. Only ACME, Darwin and Rapide make explicit use of parameterization.

Bounded properties may be translated into constraints on the abstract state of a component. Rapide uses an algebraic language to specify constraints on the abstract state of a component.

Table 4: Mapping Aspects of Beans to an ADL

ADL	INTERFACE SPECIFICATION	PROPERTIES AS PARAMETER	BOUNDED PROPERTIES
Aesop	yes	no	no
MetaH	yes	no	no
LILEANNA	yes	no	no
ArTek	yes	no	no
C2	yes	no	no
Rapide	yes	yes	yes
Wright	yes	no	no
UniCon	yes	no	no
Darwin	yes	yes	no
SADL	yes	no	no
ACME	yes	yes	no

4.2 Connections

Connections between an event source and an event listener should be translated into a specification of a connection with the appropriate interface. The same applies for the dataflow connections of the InfoBus and JavaSpaces extension. This is possible in all of the surveyed ADL's.

The semantics for the Java Beans style event-based and dataflow connections should be expressed in the ADL. It should be possible to express other kind of connections too, when future extensions introduce new types of connections. Rapide, Wright and UniCon support such specifications.

Table 5: Mapping Aspects of Connections of Beans to an ADL

ADL	EVENTS	DATAFLOW	SEMANTICS
Aesop	yes	yes	no
MetaH	yes	yes	no
LILEANNA	yes	yes	no
ArTek	yes	yes	no
C2	yes	yes	no
Rapide	yes	yes	yes
Wright	yes	yes	yes
UniCon	yes	yes	yes
Darwin	yes	yes	no
SADL	yes	yes	no
ACME	yes	yes	no

4.3 Configurations

Since it is possible to compose beans into one bigger bean, an ADL used to describe a bean-based application should support such kind of composition. Most ADL's do support it.

Because Java Beans is still developing, and more extensions are to be expected, an ADL should allow for such extensions.

Table 6: Mapping Aspects of Configurations of Benas to an ADL

ADL	COMPOSITION
Aesop	no
MetaH	yes
LILEANNA	no
ArTek	no
C2	yes
Rapide	yes
Wright	yes
UniCon	yes
Darwin	yes
SADL	yes
ACME	yes

4.4 Implicit Aspects

Above, we inventarized the possibilities of different ADL's to describe those aspects of Java Bean-based applications that are visible for 'bean-aware' tools. However, some implicit aspects of Java Beans should be described too, when distilling the architecture of an application. To name a few:

Threads. Every Java Bean may run in its own thread. At the same time, its methods may be called by other beans, and executed in the thread of the caller. A software architecture description of a beans application should specify this aspect, though it is not available through introspection.

Create-connections. A bean may instantiate other beans at run-time. Such a connection should certainly be described, but again, information about these relationships is not available through introspection.

Run-time change of the configuration. Apart from the possibility to create new instances of beans at run-time, beans are also able to change the

connections at run-time. This will especially be seen very often in applications based on the Activation Framework extension. At this moment, information about these possibilities is not available for application builder tools. However, because the run-time flexibility of the Java system is one of its advantages, an ADL for the description of beans applications should preferably support the specification of dynamism in the configuration. These ADL's are Darwin, Rapide and C2.

Table 7: Mapping Implicit Aspects of Beans to an ADL

ADL	RUN-TIME CHANGE
Aesop	no
MetaH	no
LILEANNA	no
ArTek	no
C2	yes
Rapide	yes
Wright	no
UniCon	no
Darwin	yes
SADL	no
ACME	no

5. RELATED WORK

Reuse of Off-The-Shelf components in combination with the C2 style has been explored in (Medvidovic et al., 1997). They constructed a Class Framework of reusable classes that can be used to implement C2 style architectures, and integrated several OTS components with the C2 style. This integration was done by wrapping OTS objects in C2 components, and mapping events into C2 messages and vice-versa. In this work, the C2 style is the point of departure, and reusable components are adapted in such a way that they can be used to implement C2 style architectures.

A tool to detect architectural mismatches during design has been constructed by Abd-Allah (Abd-Allah, 1996). His method is based on the notion of 'conceptual features', which can be used to detect architectural mismatches. The goal of this work is to enhance the possibilities of reusing

components, by scanning them on assumptions with respect to these features.

6. DISCUSSION

In this paper, we have made a start with combining Java Beans and software architecture.

6.1 Combining Software Architecture and Beans

As we have seen, it is highly improbable that a certain software architecture can be mapped to an application built by connecting existing beans, unless the designer of the architecture has taken such an implementation into account. A more feasable approach to combine beans and software architecture is to build a system using beans, and describe the system's architecture using an ADL. In that case, by choosing beans as components, one restricts oneself to a certain subset of architectural elements.

However, as we have seen, not all the necessary information to describe an architecture can be extracted from beans and their connections. Certain aspects are implicit, and can only been revealed by inspecting the code of the beans in use. Automating such a process is only feasable when beans adhere to standard conventions for the implementation of these aspects. In fact, this would be an extension to the Java Beans specification.

One can imagine an intermediate approach: using beans, especially developed for this purpose, to construct the system's architecture, and implement the system using beans that are specialized versions of the "design" beans. Such an approach would benefit of an extension where one can classify beans as being a specialization of another bean.

Neither of these approaches comes for free: we have to extend the standard for Java Beans to achieve a tight relationship between the software architecture description of a system, and its implementation using beans. On the other hand, the Java Beans specification already offers substantial support for extracting an architectural description: the property of introspection, meant for application builder tools, can be used for a translation into an ADL of the exposed features of a bean.

6.2 Design for Change

An attractive property of both approaches is that changes in the software are automatically handled at the architectural level. On-line change capabilities are needed in several domains (see for instance (Stankovic, 1996)), and the ideal situation would be that such changes can be applied at the architectural level.

Prerequisites for a system with on-line change capacities at the architectural level are:

- The software architecture is reflected in the executable. Parts of the executable from which components can be instantiated are traceable and replaceable.
- Components may be added, deleted or replaced, at execution time.
- Bindings of components through connections occur dynamically. In other words, connections may be added, deleted or replaced at execution time.
- Instantiation of components and connections is possible from outside the system.
- The functionality of components is not directly dependent on other components.
- It is possible to analyze properties of the system at the architectural level. Before a change is applied, the architecture should be analyzed to guarantee that the changed system will meet the changed requirements.

Obviously, using a method based on the combination of a software architecture description and a Java Beans application, it is relatively easy to build systems with on-line change capacities on the architectural level.

7. REFERENCES

Abd-Allah, A. (1996) Composing Heterogeneous Software Architectures, PhD Dissertation, Center for Software Engineering, University of Southern California. *http://sunset.usc.edu/~aabdalla/aaadef.ps.*

Allen, R. and Garlan, D. (1994) Beyond Definition/Use: Architectural Interconnection, in *Proceedings of the Workshop on Interface Definition Languages*, Portland, Oregon, January.

Boasson, M. (1996) Subscription as a Model for the Architecture of Embedded Systems, in *Proceedings of the 2nd IEEE Conference on Engineering of Complex Computer Systems*, Montreal, Canada.

Cable, L. (1998) A Draft Proposal to define an Extensible Runtime Containment and Services Protocol for JavaBeans (Version 0.98). Sun Microsystems.

Calder, B. and Shannon, B. (1998) JavaBeans Activation Framework Specification (Version 1.0). Sun Microsystems.

Colan, M. (1998) InfoBus 1.1 Specification. Sun Microsystems.

Garlan, D. and Allen, R. and Ockerbloom, J. (1995) Architectural Mismatch or Why it's hard to build systems out of existing parts, in *Proceedings of the International Conference on Software Engineering*, Seattle, April.

Garlan, D. and Shaw, M. (1993) An Introduction to Software Architecture, in *Advances in Software Engineering and Knowledge Engineering, volume 1* (ed. V. Ambriola and G. Tortora), World Scientific Publishing Company, New Yersey.

Hamilton, G. (Editor) (1997) JavaBeans 1.01 API Specification. Sun Microsystems.

Medvidovic, M. and Oreizy, P. and Taylor, R.N. (1997) Reuse of Off-The-Shelf Components in C2-Style Architectures, in *Proceedings of the 1997 Symposium on Software Reusability* , Boston, pp 190-198.

Medvidovic, M. and Taylor, R.N. (1997) A Framework for Classifying and Comparing Architecture Secription Languages, in *Proceedings of the 6th European Software Engineering Conference, Lecture Notes in Computer Science*, **1301**, 60-76.

Oreizy, P. and Medvidovic, N. and Taylor, R.N. and Rosenblum, D.S. (1998) Software Architecture and Component Technologies: Bridging the Gap, in *Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures*, Montery, CA, January 6-8.

Perry, D.E. and Wolf, A.L. (1992) Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, **vol 17, nr 4**, 40-52.

Shaw, M. and DeLine, R. and Klein, D.V. and Ross, Th.L. and Young, D.M. and Zelesnik, G. (1995) Abstractions for Software Architecture and Tools to Support them. *IEEE Transactions on Software Engineering*, **April**, 314-335.

Stankovic, J.A. (1996) Real-time and Embedded Systems. Group Report of the Real-Time Working Group of the IEEE Technical Committee on Real-Time Systems. *http://www-ccs.cs.umass.edu/sdcr/rt.ps*

Stuurman, S. and van Katwijk, J. (1998) On-line Change Mechanisms, the Software Architectural Level, to appear in *Proceedings of the the 6th International Symposium on the Foundations of Software Engineering*, Orlando.

Sun Microsystems Inc. (1998) JavaSpaces Specification, Revision 1.0 Beta. *http://www.javasoft.com/products/jini/specs/javaspaces.pdf*

Swartout, W. and Balzer, R. (1982) On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, **vol 25, nr 7**, 438-440.

BIOGRAPHY

Sylvia Stuurman is a researcher at the Software Engineering Group of the Department of Technical Mathematics and Informatics of the Delft University of Technology, the

Netherlands. She finished her study in 1996. Before 1996, she worked as a scientific programmer at the same department.