# Architectural Jigsaw Patterns

Hongyan Sun* and Anders P. Ravn

Department of Information Technology
Technical University of Denmark
Building 344, 2800 Lyngby, Denmark
Tel: (+45) 45933332
Fax: (+45) 45930074
Email: hs@it.dtu.dk and apr@it.dtu.dk

**Abstract**

*Jigsaw patterns*, a graphical architecture description notation, will be presented in this paper. The *jigsaw patterns* shows components and connectors with their interfaces. Connectors are structures or objects implementing the interfaces. Components are structures interacting via the connectors. Each component and connector can be taken as a new system and decomposed. The semantics of the *jigsaw patterns* in terms of ML is also described. The application of this notation is illustrated by a case study "3D-Med" prototype system. The *jigsaw patterns* differs from other architecture description languages (ADLs) and supporting tools in that it focuses on the overall system structure, how subsystems interface and how they decompose further. Other ADLs and tools supports mainly the description and analysis of style-based components and their connections.

**Keywords:** Architecture descriptions, components, connectors, interfaces, jigsaw patterns, system decomposition.

**Workshop Goals:** Learning; networking; exchanging information, discussing new ideas.

*Supported by the Danish Technical Research Council under the Co-design project.

## 1 Background

Some of our colleagues are working on a project, the aim of which is to design and build a low-cost PC-based solution for real-time viewing of 3D medical images. They raised the issue: How to document a complex system which is implemented in a mixed environment, e.g., C++ and Java? The documentation should give an overview of the system architecture and classify interfaces between interacting subsystems. Our first approach to this issue was to use the module system of the functional programming language ML [Paul91, Ullm94] and CML (Concurrent ML) [Repp93] for the architecture description, because the module concepts of ML are simple, yet general enough to define an architecture without introducing the low level data structures of C-based C++ or Java.

The system architecture uses general architecture concepts, i.e. it is composed of components which connect through interfaces to connectors [GaSh93, AlGa94]. The ML signature is used to define an interface and ML functor concepts are used to define components and connectors. The interfaces define access functions [Parn72] to an environment. The connectors are structures or objects defining and implementing the interfaces. The components are structures implementing functions of the system assisted by access functions in the interfaces. We thus isolate the interface implementation in the connector.

In order to visualize these ML textual descriptions we developed a graphical notation, called *jigsaw patterns*, which gives a view of the system architecture. We applied *jigsaw patterns* to the description of the architecture of the prototype "3D-Med" system which is implemented in C++ and Java. The notation helped in the discussions about the project and has been taken over by the team which is currently exploring its use in associated projects. We feel that *jigsaw patterns* has the following benefits compared with the box-and-line diagrams which are often used to describe a system architecture:

- Simple semantics
- Hierarchically decomposable

In the following two sections, we present and discuss the *jigsaw patterns* technique. We use the"3D-Med" system example to illustrate the application of the technique. In the last section we compare the notation to a few architecture description languages (ADLs) [GAO94, LKAVBM95, SDKRYZ95, AlGa94].

## 2 Position

We are interested in component-based system architecture, architecture description environments and supporting tools.

Architecture level design considerations become significant when software systems grow larger and more complex [GaSh93, ShGa96]. Important issues include descriptions of components which constitute the system architecture and relations among these components, i.e. the way these components are connected to build the system. It has been recognized that designers follow some idiomatic patterns [ShGa96, Shaw95a, Shaw95b], e.g. *pipes-filters* [Bach86], *client-server* [Bers92], etc., during system design. These patterns are usually accompanied by box-and-line diagrams that depict the intended system architecture. However, the interpretation of box-and-line diagrams is highly informal and highly specific to the design team, and to the described system. In addition, it is hard to decompose an element, particularly a line, in a box-and-line diagram. As a result, the architecture is poorly understood by developers when the system grows larger and more complex. It also hampers attempts at making system design knowledge explicit and easy to reuse [GAO95, CoLu95, DeCo95].

The *jigsaw patterns* technique, as illustrated in Figure 1, can overcome these problems of box-and-line diagrams. In Figure 1, the component $C1$ could be a *source* or a *sink* process. It has room for one inward stub corresponding to one interface interacting with one connector. The connector $Con\_C1\_C2$ is a *pipe* (or *channel*). $Con\_C1\_C2$ is able to connect with two components via the two interfaces, i.e. outward stubs, which are implemented in the connector. Finally, the component $C2$ is a *filter*. It has room for two inward stubs corresponding to two interfaces interacting with two connectors. The picture could be completed to a *producer/consumer* system by adding a *pipe* connector to $C2$ and a component similar to $C1$.

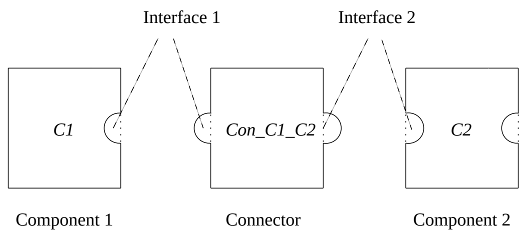


Figure 1. Jigsaw patterns

The notation allows hierarchical decomposition because each component or connector of the system can be treated as a new system to be given an architecture using a finer grained *jigsaw*.

The general discipline is: An inward stub uses services and an outward stub provides services. Thus, **Components** do not implement interfaces, i.e. they do not have outward stubs graphically. **Connectors** do not use interfaces, i.e. they do not use access functions. The rationale for this discipline is to reduce the agenda in the design activity and provide syntactic criteria for classifying a module as a connector or a component.

In some cases, a connector uses the standard linkage convention to glue an object (e.g. in C++ and Java programs), see Figure 2. We call this connector an **Application** module. This, in fact, reflects the ability of composition, decomposition and evolution of *jigsaw patterns*.

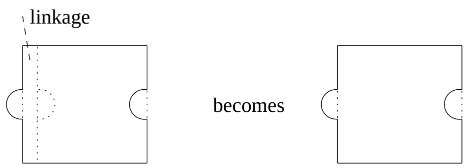


Figure 2. Illustration of an **Application** module

In the following, we shall give the semantics of *jigsaw patterns* in terms of ML. An interface is expressed by a **signature**. A connector or a component is expressed by a **structure**, or a structure generator **functor** if it is generic.

## 2.1 Interfaces

Interfaces of a component define access functions to an environment. The access functions are viewed as inputs from and outputs to an environment which is changed as a side effect. Thus an interface is defined in the following unified form:

> signature *Interface1* =
> sig
> val *output1* : *Domain* → *unit*;
> end;
>
> signature *Interface2* =
> sig
> val *input2* : *unit* → *Range*;
> val *output2* : *Domain* → *unit*;
> end;

These two signatures are two specific cases of the general form of an interface which can contain a variable number of access functions, including mixed input and output functions.

#### Specifications

Above, we have stipulated that the interface is defined by a signature. This is obviously not the whole story. Another part of the definition would be a description of the relation between successive calls of access functions, as seen from both the component and the connector. A formalization involves defining a *state*, recording the side effects of the function calls, for instance a trace of function calls and two predicates over the state: a *component specification* and a *protocol specification*. The component must be developed so that it satisfies its specification by assuming the protocol. Analogously, a connector must obey the protocol under the assumption of the component specification. Obviously a robust connector will make minimal use of component specifications.

## 2.2 Connectors

Connectors implement functions defined in the interfaces of the interacting components. It is a structure which can be expressed by a ML functor:

> functor *Connector* () :
> sig
> val *output1* : *Domain* → *unit*;
> val *input2* : *unit* → *Range*;

```
val output2 : Domain → unit;
end
=
struct
(* implementation of the input and the output functions *)
end;
```

The indexes, e.g. 1 and 2, are used to distinguish the interfaces of the different components, in this case $C1$ and $C2$ corresponding to Figure 1. A connector can have an arbitrary collection of input and output functions corresponding to the interfaces, i.e. outward stubs, which it implements.

A specific connector, $Con\_C1\_C2$ in Figure 1, is then given by:

```
structure Con_C1_C2 = Connector();
```

### 2.3 Components

Components implement functions of the system assisted by access functions in the interfaces. The internal structure of a component is application dependent, and is hidden in the description. A component is expressed by a ML functor as sketched in the follows:

```
functor Component1 (Connector : Interface1) :
sig
val proc : unit → unit;
end
=
struct
(* hidden internal structures *)
fun proc () = ...
end;

functor Component2 (Connector : Interface2) :
sig
val proc : unit → unit;
end
=
struct
...
end;
```

The process *proc* created by the component can be executed both sequentially and in parallel according to the application. *Connector* is the formal name for a connector which connects *Component* to the interacting component through *Interface*. The number of the connectors in the parameter list of this functor is the number of inward stubs in its pattern, as shown in the following examples.

The specific components, $C1$ and $C2$ in Figure 1, are given by:

structure $C1 = Component1(Con\_C1\_C2)$;
structure $C2 = Component2(Con\_C1\_C2)$;

It is not difficult to derive the semantics of an **Application** module from the definitions of the connector and the component in terms of a ML functor:

functor $Application\ (Connector : Interface)$ :
sig
val $input$ : $unit \rightarrow Range$;
val $output$ : $Domain \rightarrow unit$;
end
=
struct
(* *implementation of the input and the output functions* *)
end;

The introduction of the **Application** module makes the notation more expressive, as illustrated in the following section. In an *object-oriented* system [Booc86, GHJV95], it is often the case that an object uses another object in order to provide a new service.

# 3 Case Study: The "3D-Med" System

The "3D-Med" system is implemented in C++ and Java. The architecture of the prototype "3D-Med" system is illustrated in Figure 3. Overall, it contains three components: $UserInterface$, $Viewer$ and $Repository$. $UserInterface$ provides user menus and produces corresponding actions according to user interactions. $Viewer$ processes images and produces 3D image views on the screen. $Repository$ is a data base for storing images. Two connectors, $Con\_User\_View$ and $Con\_UserView\_Repo$, connect these three components. Figure 3 gives the graphical presentation of an overview of the system architecture by means of *jigsaw patterns*.

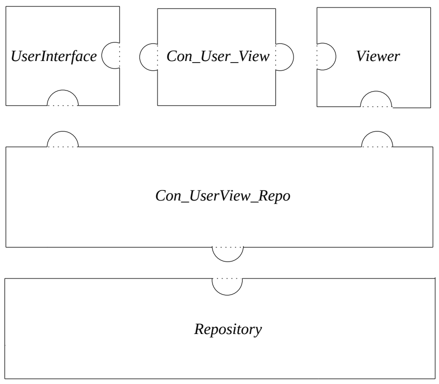


Figure 3. Overview of the "3D-Med" prototype architecture

Each component and connector can be taken as a new system and be decomposed in the same notation. Figure 4 illustrates the decomposition of the component $UserInterface$, and Figure 5 shows the decomposition of the connector $Con\_User\_View$.

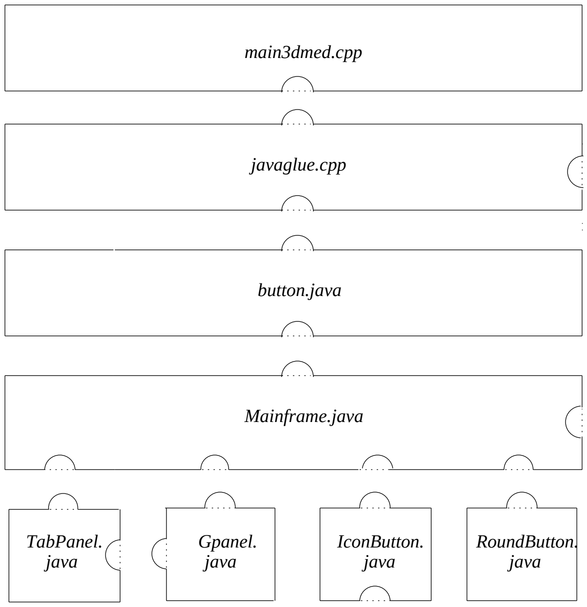


Figure 4. Decomposition of the component $UserInterface$

Figure 4 also gives examples of the **Application** modules. For instance, a standard C++ linkage glues a C++ object in the *javaglue.cpp* module, and a standard Java linkage glues a Java object in the *button.java* module.

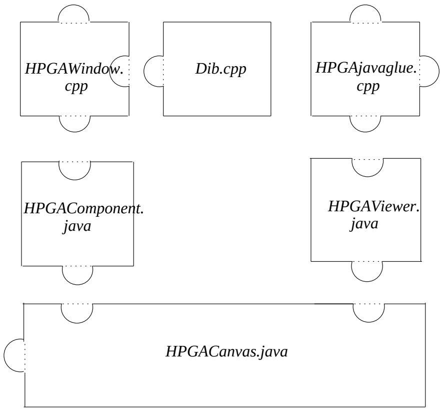


Figure 5. Decomposition of the connector $Con\_User\_View$

Figure 4 and Figure 5 illustrate how a subsystem is decomposed using *jigsaw patterns*. In this particular case, components and connectors are expressed in terms of C++ and Java modules. Here we did not specify the interfaces. This could be done by using a header file or the package concept of C++ and Java.

# 4 Comparison

A number of ADLs and supporting tools have been developed, which include Aesop [GAO94], Rapide [LKAVBM95], UniCon [SDKRYZ95], Wright [AlGa94], STILE [StWe87] and so on. Aesop is a system for developing style-specific architecture development environments, i.e. it supports the use of architectural styles. Rapide is an event-based, concurrent, object-oriented language, which allows architectural design to be simulated and analyzed. UniCon supports architectural abstractions for describing different types of components and connectors. Wright supports specifications and analysis of interactions between architectural components.

STILE is a graphical design and development environment. A system design is expressed by a set of *blueprints*. Each *blueprint* contains a set of graphical notations together with textural specifications to describe implementation details for each part of the system. STILE mainly focuses on description of the behavior of each system component and logical relationships among the components.

These ADLs and supporting tools address important issues of describing and analyzing system architecture with different concerns. However, these ADLs and tools (except STILE) are style-based in common. They try to give formal syntax and semantics to different architecture styles, which are frequently accompanied by box-and-line diagrams as the graphical notation. It is important to describe architecture styles as stated in [GaSh93, ShGa96, Shaw95a, Shaw95b, GAO94]; but it is not always clear that how subsystems interface and how they decompose. The graphical syntax produced by these ADLs and tools are complex, especially when the system is getting complex with multi-styles.UniCon and Wright attack the problem by providing decomposable components. But still, connectors with graphical "lines" are not decomposable. The *jigsaw patterns* technique facilitates this kind of abstraction.

If we carefully examine the examples given in this paper, it has already been illustrated that *jigsaw patterns* can also describe different architectural styles, e.g. *pipes-filters*, *client-server*, *object-oriented*, and *Repository*.

We have mentioned that *jigsaw patterns* alone does not provide information about details of the components, connectors and interfaces. The *jigsaw patterns* is designed as a decomposable substitute for box-and-line diagrams. Textual descriptions like ML or other ADLs are needed. It is possible to develop such an environment using *jigsaw patterns* on top levels, where each level can visualize the lower level decompositions by means of modern graphics tools. At a certain level, we can visualize the accompanying textual description for each component, connector and interface using ADLs or other modeling languages like e.g. UML (Unified Modeling Language) [Uml97].

# References


[GaSh93] David Garlan and Mary Shaw. An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, 1993.

[ShGa96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Shaw95a] Mary Shaw. Some Patterns for Software Architectures, *Second Annual Conference on Pattern Languages of Programming*, September 1995.

[Shaw95b] Mary Shaw. Comparing Architectural Design Styles, *IEEE Software*, IEEE Computer Society, November 1995, pp27-41.

[Bach86] M. J. Bach. *The Design of the UNIX Operating System*, Software Series, Prentice Hall, 1986.

[Bers92] A. Berson. *Client/Server Architecture*. McGraw-Hill, 1992.

[Booc86] Grady Booch. Object-Oriented Development, *IEEE Transactions on Software Engineering*, February 1986, pp211-221.

[GHJV95] E. Garmma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[GAO95] David Garlan, Robert Allen and John Ockerbloom. Architectural Mismatch: Why Reuse is So Hard, *IEEE Software*, IEEE Computer Society, November 1995, pp17-26.

[GAO94] David Garlan, Robert Allen and Jone Ockerbloom. Exploiting Style in Architectural Design Environments, *Proceedings of SIGSOFT'94*, ACM Press, December 1994, pp179-185.

[LKAVBM95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan and Walter Mann. Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995, pp336-355.

[SDKRYZ95] 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*, Vol. 21, No. 4, April 1995, pp314-335.

[AlGa94] Robert Allen and David Garlan. Formalizing Architectural Connection. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento Italy, May 1994, pp71-80.

[CoLu95] Donald D. Cowan and Carlos J. P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design For Reuse, *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, March 1995, pp229-243.

[Uml97] *Unified Modeling Language: UML Summary*, Version 1.0, Rational Software Corporation, January 1997.

[DeCo95] Thomas R. Dean and James R. Cordy. A Syntactic Theory of Software Architecture, *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995, pp302-313.

[Parn72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp1053-1058.

[Paul91] L. C. Paulson. *ML for the Working Programmer*, Cambridge University Press, 1991.

[Ullm94] Jeffrey D. Ullman. *Elements of ML Programming*, Prentice Hall, 1994.

[StWe87] Michael P. Stovsky and Bruce W. Weide. STILE: A Graphical Design and Development Environment, *COMPCON 87*, IEEE Computer Society, Spring, 1987, pp247-250.

[Repp93] John H. Reppy. *Concurrent Programming with Events - The Concurrent ML Manual*, Version 0.9.8, AT&T Bell Lab., February 1, 1993.

## Biography

**Hongyan Sun** is a Research Assistant at the Department of Information Technology, Technical University of Denmark. Her main research interests include methodology for system design and prototyping, system architecture, ADLs and supporting tools, and robotics. She received a Ph.D. in Robotics from the Technical University of Denmark in 1996.

**Anders P. Ravn** is a Reader in the Computer Systems Section, Department of Information Technology, Technical University of Denmark. His main research interests include software engineering, real-time systems, formal methods. He received Dr. Tech. in 1995, M.Sc. in Computer Science and Mathematics from the University of Copenhagen in 1973.