Component Interface Pattern

Ricardo Pereira e Silva Eng., M.Sc., D.Sc. Roberto Tom Price Eng., M.Sc., D.Phil*

Federal University of Santa Catarina - Computer Science Department
Postbox 476 – Florianópolis – SC – Brazil
http://www.inf.ufsc.br/~ricardo / mailto:ricardo@inf.ufsc.br

* Federal University of Rio Grande do Sul – Informatics Institute
Postbox 15064 – Porto Alegre – RS – Brazil
http://si3.inf.ufrgs.br/Amadeus/Coordenador / mailto:tomprice@inf.ufrgs.br

Abstract

Component-oriented development establishes the building of software artifacts by means of connection of a collection of components, independently produced. The success of applying this development approach depends of the compatibility between connected component interfaces. That compatibility includes interface structural and behavioral features. Component Interface Pattern establishes a structure for building component interfaces that allows to evaluate and to assure the structural and behavioral compatibility between components to be connected.

Motivation

Component-oriented development is being stimulated by the availability of devices allowing component interoperability, regardless of programming language, running platform and executing location. However, the lack of efficient standards for specifying components difficulties the search and selection, functionality understanding and adaptation of components.

In component-oriented development approach an application is produced joining a set of components. “A component is a unit of composition with contractually specified interfaces and explicit context dependencies” [SZY 96]. A component can be implemented using various technologies. So any software artifact with an interface provided can be considered a component. An interface is “a collection of service access points, each of them including a semantic specification” [BOS 97].

A problem related to component-oriented development approach is how to define the external visibility of a component, that is, how to specify its interface. On the other hand, component users need to understand a component by means of this kind of description. So, the absence of standards to describe efficiently what a component does and how to interact with it is an obstacle for that approach.

Usually, interface specification is used to describe a component. Approaches like interface description languages (IDLs) describe the services provided by a component. This is suitable for components made to supply services. Often, function signatures are enough to establish how a client accesses the functionality. A class from a library that implements a *list* is an example of a component that can be adequately described by means of its provided services.

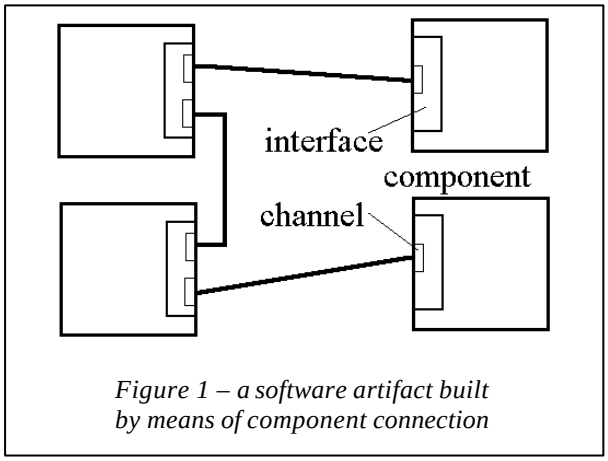

Figure 1 – a software artifact built by means of component connection

Components that just provide services are not the general case in component-oriented development. If an application is built only with components, as shown in figure 1, one component will require the services from other components. So, it is not reasonable a component description including only provided services. Required services need to be specified too. So, in general case, component interaction is bi-directional [OLA 96].

The CORBA Component Model, CCM, part of the CORBA 3.0 specification [OMG 99], introduces an evolution to IDL view. It establishes an interface definition more complex than the traditional view, in that not only provided services are specified, but required services too, as well as required and supplied asynchronous events. Ports define the interface of a component, including:

- Facets: interface views provided by the component for client interaction;
- Receptacles: connection points to required services, supplied by some external provider;
- Event sources: connection points that emit events to one or more interested event consumers, or to an event channel, in an *Observer* [GAM 94] approach;
- Event sinks: connection points where events may be pushed, in an *Observer* approach.

Another component feature to describe is that a component can present more then one view.

CORBA 3.0 facets show that a component can provide more than one view for external devices. This suggests more than one service access point. The service access point of a component interface is called communication channel (or just channel). figure 1 shows components with one and more than one channels in their interfaces. It is possible that just part of required or supplied services need to be accessible at each communication channel of a component interface. So, beside the set of provided and required services, the structural description of a component interface must establish the accessibility of the services at its channels. Table 1 shows an example of this kind of description. The mark denotes the accessibility of a service (provided or required) at a channel. In Table 1 example it is possible to see that the described component will not invoke *requiredMethod1* through the channel *Channel2*. So, the component connected to that channel do not need to supply that service.

		Required services	Supplied services	
		requiredMethod1	**suppliedMethod1**	**suppliedMethod2**
Channels	**Channel1**	✓	✓	
	Channel2		✓	✓

Table 1 – structural description of a component

Besides structural features, component interface specification must describe dynamic features. Contracts [HEL 90] constitute another approach able for describing mutual dependencies between software components. It describes in a formal way the participants of an interaction and its obligations. Contracts allow to describe dynamic

dependencies between components. For example, that when a component executes a service it needs so invoke a service supplied by other component. The main disadvantage of this approach is to produce a description of low level of abstraction, usually too long and hard to understand.

Reuse contract [LUC 97], an adaptation of Helm's contracts, allows to describe component interfaces and dynamic features of a component connection, too. It provides a graphical notation that makes easier to understand dependencies between components. In the example of figure 2 "Component 1" provides the service "x" and "Component 2" provides the services "y", "z" and "t". In "x" running "y" can be invoked ("y" putted between brackets on the "x" arrow).

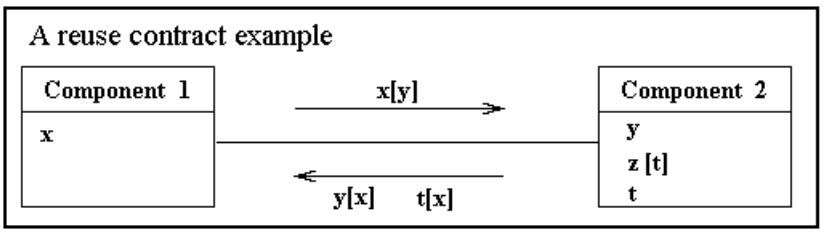

Figure 2 – An example of reuse contract

Reuse contract describes the knowledge relationship between components and the services required in the running of a supplied service. This approach lacks in describing service requirement invoking order in a component architecture. For example, suppose a component that provides two services, *initializeIt* and *executeIt*, and *initializeIt* must be invoked before *executeIt*. This situation can not be identified in a description based on reuse contract.

A more powerful approach to specify dynamic constraints is based on Petri nets. Component interface pattern is based on that dynamic modeling approach, together with a structural description like that shown in table 1 [SIL 00]. Dynamic constraint specification by means of Petri net allowing the description of constraints related to the order of invoking services, like that described in the example above. This kind of dynamic constraint can not be specified with reuse contracts or with CCM approach.

In proposed approach ordinary Petri net is used It is composed by places, transitions, arcs and an initial marking. The only extension to ordinary Petri net is an association of the transitions with a pair channel-service. Each mark on the table that associates channels and services (supplied or required) will correspond to one or more transitions in the Petri net that describes the interface. Figure 3 shows the behavioral description of the interface whose structural description is shown in table 1.

The initial marking establishes that only the transitions T2 and T3 from figure 3 can fire. Both transitions are related to supplied service "*suppliedMethod1*" (one to channel "*Channel1*" and other to channel "*Channel2*"). So, the component will not produce a requirement of service before to receive an invocation of service "*suppliedMethod1*".

Besides to specify constraints related to invocation order of required and supplied services of a component, the approach based on Petri net allows to describe the interaction among a collection of connected components. Based on Petri net property analysis, it is possible to verify behavioral characteristics of the software artifact produced from component connection, like deadlock possibility or the identification of services that never run.

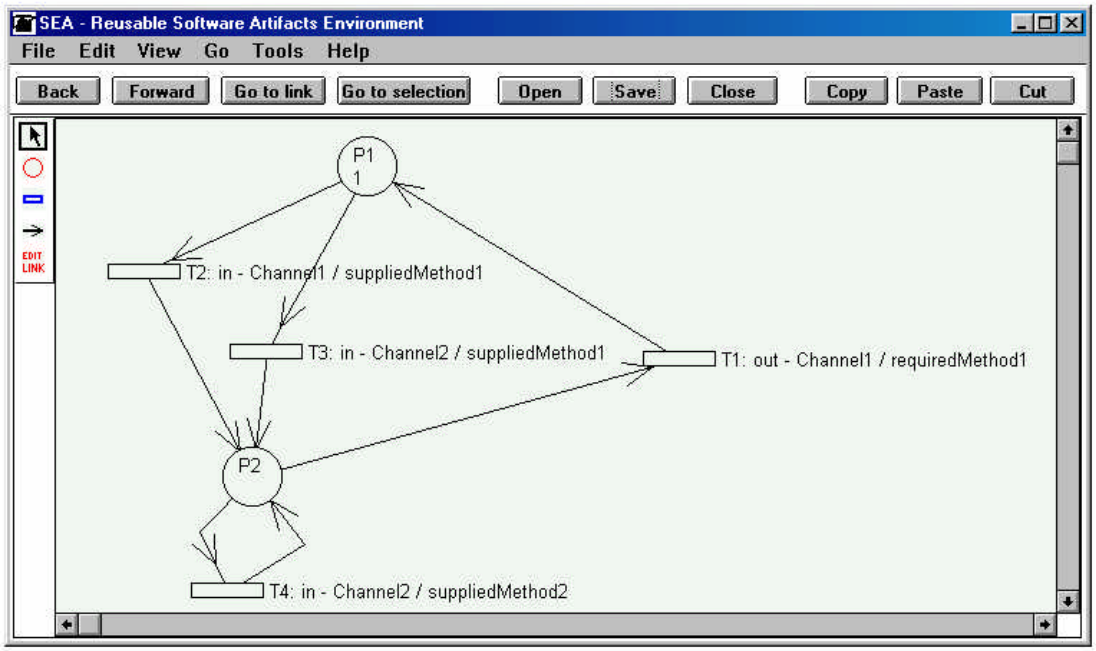

Figure 3 – Behavioral specification of a component interface based on Petri net

Figure 4 shows the behavioral specification of two components and the Petri net produced to specify the connection and collaboration of components. The component at the left side of the figure has one channel, "Ca", one supplied service, "x", accessible at "Ca", and one required service, "y", accessed by means of the channel "Ca", too. The other component has the channel "*Cb*", associated to supplied service "y" and to required service "x". These components can be connected through their channels, "*Ca*" and "*Cb*". They are structurally compatible because the service required for one is supplied by the other. The building of the Petri net resulting from the connection consists in to merge the pairs of transitions associated to a same method and to two connected channels (one of each component). The Petri net at the right side of the figure 4 is the result of the connection of the two components described in that figure. Analyzing that Petri net, it is clear that no transition can fire. In this case, the connection of the two components structurally compatible resulted in a software artifact with a deadlock. So, that components are behaviorally incompatible.

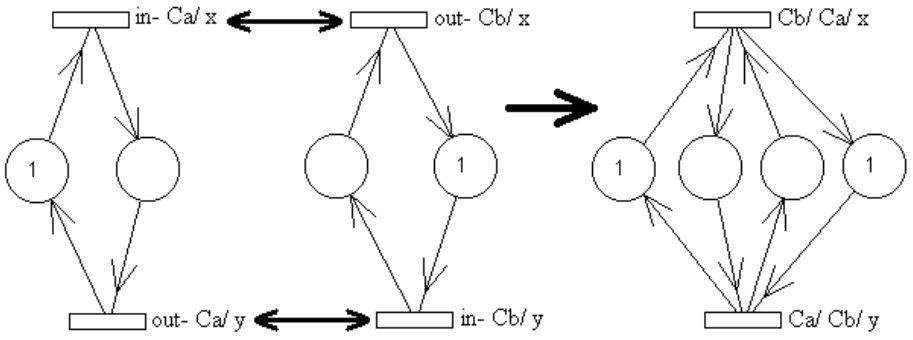

Figure 4 – behavioral specification of the connection of two components

If all components of a software artifact built by means of component connection are implemented following the interface structural and behavioral constraints specified as presented above, it will be possible to assure structural and behavioral compatibility between components connected. That is, it is possible to verify the compatibility before

connecting components. So, the challenge is to include the specified features in the implementation of a component interface. Component interface pattern aims to establish a class structure for building component interfaces that allows to incorporate structural and behavioral constraints, specified as shown above.

Applicability

Component interface pattern allows building a component with an interface precisely specified by means of the structural and behavioral viewpoints described in previous section. This is useful to produce a component that:

- will be connected to other components with interface structures built using component interface pattern;
- will be connected to other components with interface structures built by means of anther approach;
- will be inserted in a software artifact not based on component-oriented approach (similar to reuse a class from a library)

Component interface pattern supplies a complete solution to build a component interface because its structure encloses structural and behavioral constraints. It frees the developer of the need of defining how to implement component interfaces. The disadvantage of the use of this pattern is that the resulting interface is a structure more complex than a *Facade* [GAM 94]. If the component performance is critically affected the pattern application will not be suitable. In *Participants* section is discussed the need of a structure more complex than Gamma's *Facade*.

Structure

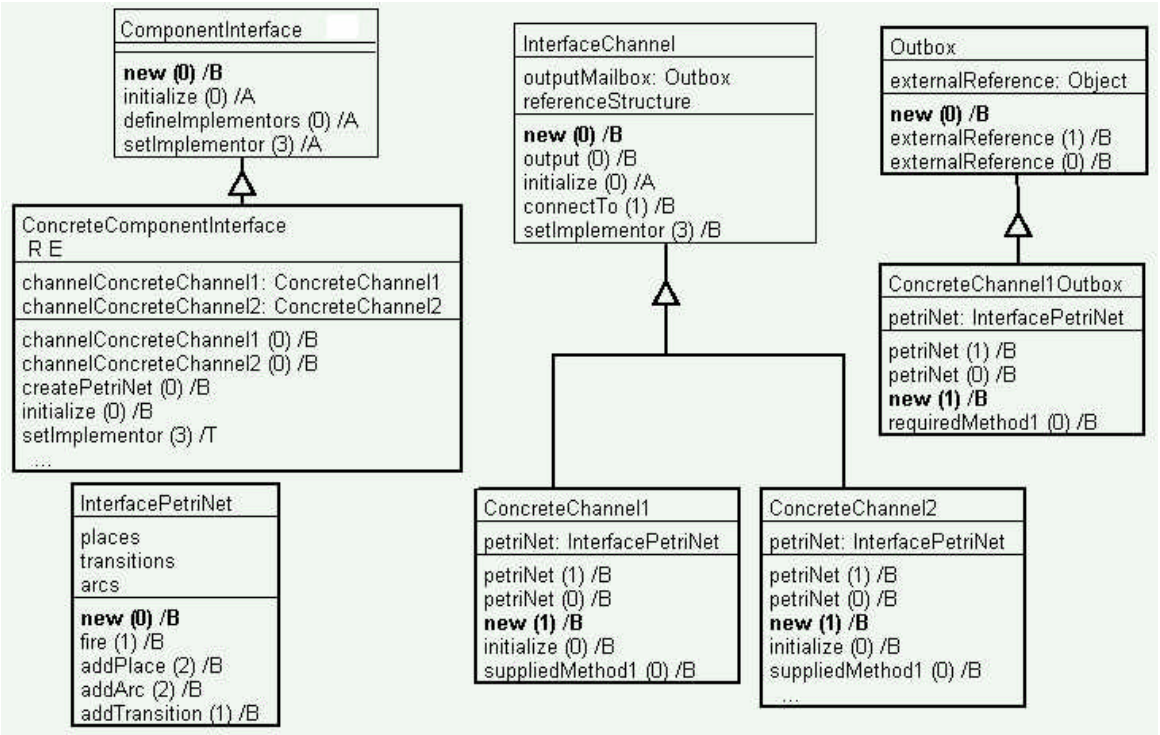

Figure 5 – Component interface pattern structure

Participants

In the following participant description, the subclasses of *ComponentInterface*, *InterfaceChannel* and *Outbox* was defined according to the interface specification from table 1 and figure 3 - so, with figure 5 class structure, too. This means that in general case, an interface implementation will include those three classes as they are here described and their subclasses will be produced according to a specific interface specification. That is, according to its structure (channels and services) and behavioral constraints. The class InterfacePetriNet is included without changes.

ComponentInterface

ComponentInterface is an abstract class that defines part of component interaction protocol. It implements the external view; that is, it corresponds to a *Facade* of a component.

At run time a component is produced generating an instance of a concrete subclass of *ComponentInterface* (see Collaborations section for details about component creation). So, a specific component will have exactly one concrete subclass of this class. A concrete subclass will foresee reference maintenance for the interface channels (attributes that point instances of *InterfaceChannel* subclasses) and will implement component internal structure building.

This class is pointed to the generation of concrete subclasses with specific interface specifications, as well as specific component internal structure.

InterfaceChannel

InterfaceChannel is an abstract class that defines the communication channel, part of the interface. An instance of *ComponentInterface* subclass will aggregate one instance of *InterfaceChannel* subclass for each communication channel. This class is responsible for allowing the access to component supplied services.

It foresees a structure to keep reference to the implementor of each component supplied service (from the component internal structure). Besides that, it has a procedure for establishing the link between a channel and the internal implementor of each component supplied service.

It keeps reference of an outbox (instance of *Outbox* subclass) as an instance attribute – this is useful just for channels that access required services.

It has a procedure for connecting components, that is, for defining the external interface component channel to be connected to it (to send service requirements).

Outbox

Outbox corresponds to a *Proxy* [GAM 94] for component required services. It establishes the link between the component internal element that requires a service and the component that implements it (see Collaborations section for details about component connection). It is the part of a channel responsible for sending service requirements to a connected interface channel (from other component). *Outbox* subclasses will implement component required services in a *Proxy* approach.

So, often in run time, each channel of an interface is composed by an instance of an *InterfaceChannel* subclass that aggregates an instance of *Outbox* subclass. This does not

occur in all cases. As *Outbox* is responsible for sending service requirements, it is not necessary in channels that do not access required services.

The need of two classes related to each channel is due the possibility of a component presents required and supplied services with equal signatures – what it is not possible to insert in a single class.

ConcreteComponentInterface

It is a specialization of *ComponentInterface*.

It must have a specific initialization procedure (overriding the inherited abstract method *initialize*). For details of the initialization, see collaborations.

It must have a specific procedure to build the component internal structure and to define what internal elements are responsible for implementing the services supplied by the component. This is done in the method *defineImplementors* (overriding the inherited abstract method).

It keeps reference of all interface channels (instances of *InterfaceChannel* subclasses) as instance attributes.

It must have a specific procedure to notify the implementor (from internal structure) of each component supplied service for the interface channels. This is done in method *setImplementor* (overriding the inherited abstract method).

ConcreteChannel1, ConcreteChannel2

They are specializations of *InterfaceChannel*. A subclass of *InterfaceChannel* is created to each channel foreseen in an interface specification. In run time each interface (*ComponentInterface* subclass instance) will aggregate exactly one instance of each class that implements each one of its interface channels.

An *InterfaceChannel* subclass implements all the component supplied services accessible at the respective channel. The implementation of a supplied service in a channel consists in a *Proxy* procedure: to pass the invocation to the respective internal implementor.

In building an *InterfaceChannel* subclass, the inherited method *initialize* must be overridden. This procedure (specific for each channel) is responsible for composing the reference structure for pointing internal implementors of the component supplied services and for referring the external reference (an instance of an *Outbox* subclass). Only channels that give access for supplied services need that last functionality in initialize method. For details, see *Code* section.

ConcreteChannel1Outbox

It implements all required services accessible in a specific channel. Similar to supplied services, implementation of required services consists in *Proxy* procedures: to pass the invocation from the internal requirer to the respective external implementor, that is, a channel of a connected component.

InterfacePetriNet

InterfacePetriNet is a concrete class that can hold a Petri net topology and allows proceeding net state evolution by means of transition firing. Its attributes are pointed to store:

- A collection of places, with its respective token number (that is, the net state);
- A collection of transitions;
- A collection of arcs, connecting places and transitions (that is, the net topology).

The *fire* method (with a transition as argument) will update the net state and return *true* if the firing is possible and, otherwise, *false* (without changing the state, in this case).

In run time; all instances of *InterfaceChannel* and *Outbox* subclasses will point an only instance of *InterfacePetriNet*. The fire method will be invoked to each service requirement (required or supplied). The passing of the invocation is conditioned to a *true* return (see C*ollaborations*).

Collaborations

The following collaboration description takes into consideration the class structure from figure 5 and the comments in the beginning of last section. So, the collaborations bellow describe the general case of interaction by means of component interface pattern, but supplied and required services, as well as the interface channels, are defined for each interface specification.

Creating a component

The sequence diagram bellow (figure 6) describes the building of the interface shown in *Structure* section (with two channels).

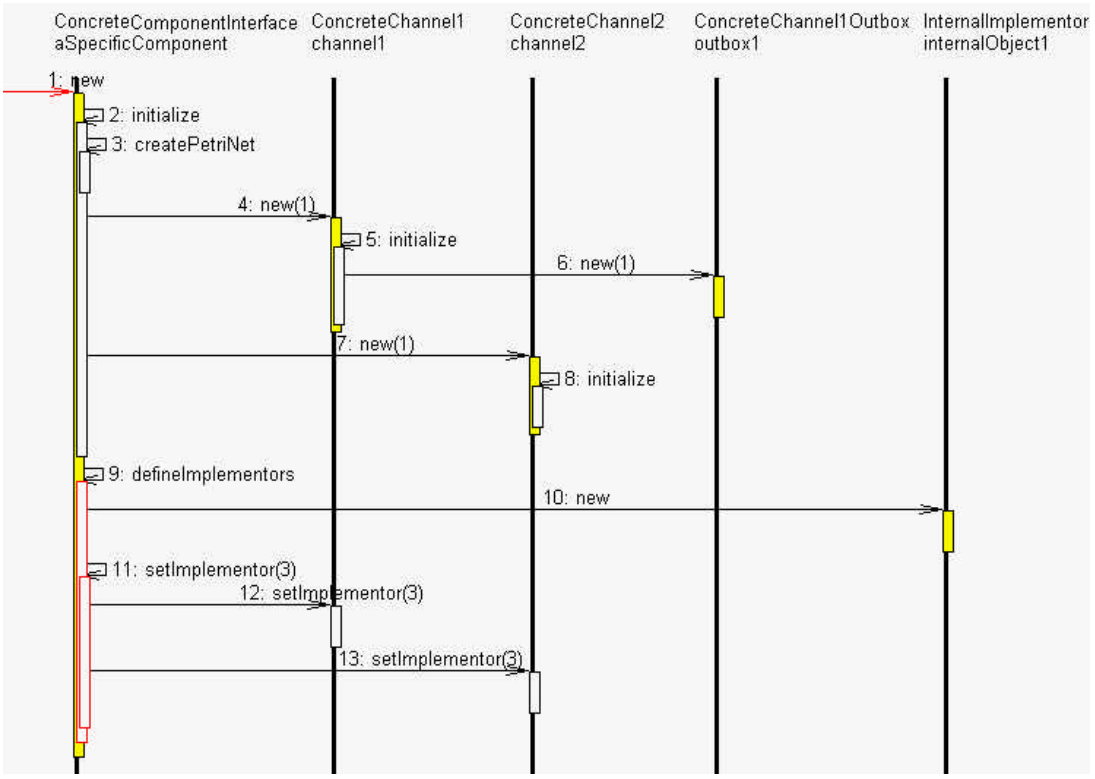

Figure 6 – Component building

The first action of the *initialize* method (*ConcreteInterfaceComponent*) is to create a Petri net (see *Code* section). After, the channels are instanced. The building of a channel includes the instancing of its outbox. This can be seen in channel1 creation (figure 6). The building of channel2 does not include instancing an outbox because this channel does not give access to any required method.

Besides channel building, the *initialize* method starts the building of the component internal structure, what is done by the method *defineImplementors*, to be overridden in a *ConcreteInterfaceComponent* subclass. This method is responsible for generating the remaining component objects, too (the objects that are not part of the interface). Its implementation is specific for each component and is outside the scope of component interface pattern. Component developer is the responsible for defining the component internal elements, that is, the component functionality and how to implement it. The method *setImplementor* communicates to each channel, the implementor of its methods; that is, the component supplied methods accessible in each channel (see its implementation in *Code* section).

After initialize running, a component is built, that is, the objects of the interface and of the internal structure are instanced and the channels are linked to internal implementors of supplied methods. In this state a component is ready to be connected to other components.

Connecting component channels

Figure 7 shows the connection of two components, Component1 and Component2, to the channels of a component, cannel1 and channel2, respectively. The connection of Component1 to the channel channel1 of the component described at the diagram starts with the obtaining of channel reference. Component1 will invoke *connectTo* method from that object. For keeping the external reference (to can invoke required methods) channel1 invokes *externalReference* method of outbox1 (the outbox of the channel channel1), with the connected channel from Component1 as argument.

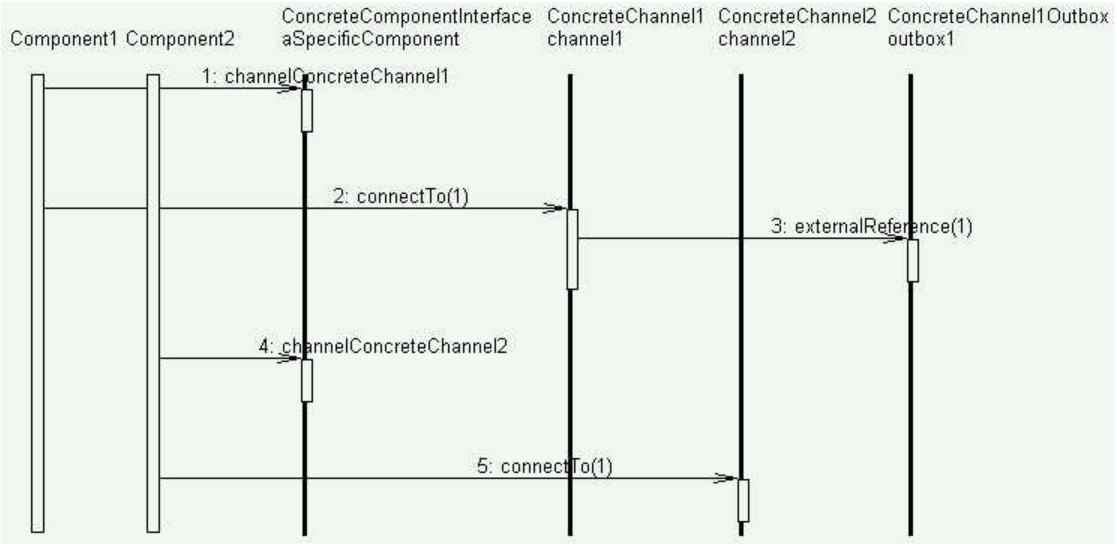

Figure 7 – Component connection

The connection of the component Component2 to the channel channel2 is similar, but it does not occur the interaction with an outbox, because channel2 does not present an outbox.

Sending a service requirement

Figure 8 shows the sending of a requirement from a described component to another component, Component1. The start is a requirement from an internal object sent to an outbox. The outbox invokes the *fire* method from interface Petri net. The request will be sent to Component1 only if *fire* method returns true.

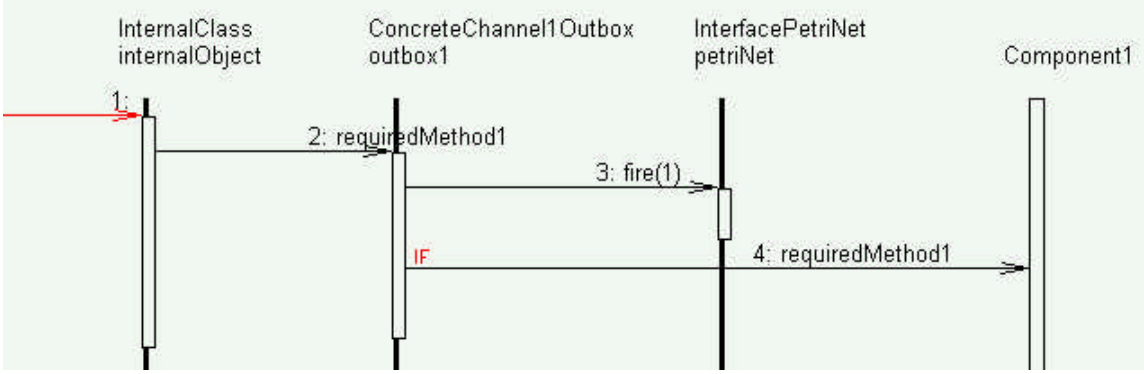

Figure 8 – A component sending a method requirement to another component

Receiving a service requirement

Figure 9 shows the receiving of a supplied method requirement to a described component, from another component, Component2. The start is a requirement from Component2 to a channel. The channel invokes the *fire* method from the Petri net of the interface. The request will sent to the internal implementor, internalObject1, only if *fire* method returns true.

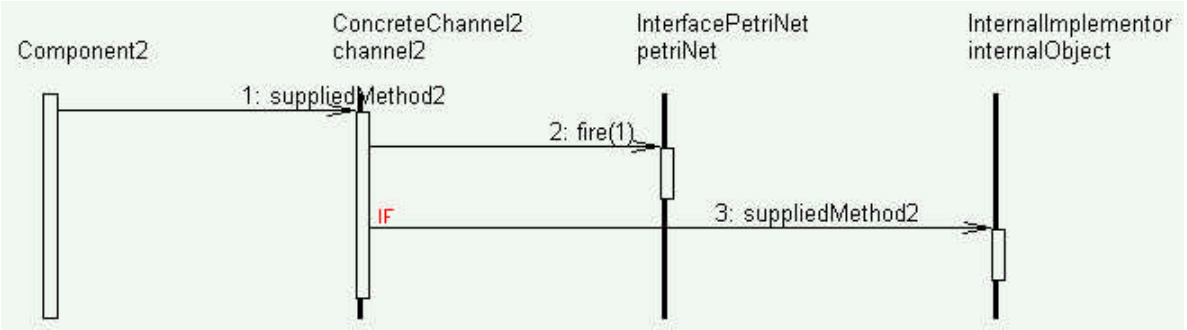

Figure 9 – A component receiving a method requirement from another component

Implementation

Issues to be considered when implementing Component Interface Pattern.

1. **Implementing Component Interface Pattern without a Petri net**. The instance of *InterfacePetriNet* class included in an interface structure avoids improper requirement sending (according to interface behavioral constraints). This occurs because only proper supplied method invocations are passed to internal implementors and only proper required method invocations are made by a component (improper invocations from internal structure, due some error, are not sent). Otherwise, it is possible to implement a component interface without a Petri net (and without its capacity of avoiding improper interaction). In this case tests for requirement passing will not occur, so it increases the performance of the interface (performance increasing is a justification for this option).

2. **Referring subclasses of *Outbox* and *InterfaceChannel***. A feature of Component Interface Pattern is the need of defining new methods, when creating subclasses of *Outbox* and *InterfaceChannel*, that is, methods not foreseen in superclass interface. This will not be a problem if the implementation is done with Smalltalk. Otherwise, that pattern feature is a problem in languages, as C++, in which a variable can refer an object of a subclass of its type, but can not know a method defined only in that subclass – what requires, in this case, a type redefinition in run time. This requires some care when defining the types of elements (attributes, parameters and variables) that refer instances of subclasses of *Outbox* and *InterfaceChannel*.

3. **Heterogeneous systems**. Component Interface Pattern can be used to implement the interface of components that will interact with software artifacts made without this pattern using (components or other artifacts like objects, if a component is part of a program). In this case, the attribute *externalReference* (from *Outbox* class) can refer an object that is not an instance of an *InterfaceChannel* subclass. It is necessary the same care above described for its type definition.

4. **Temporary reference**. An application based on component approach can be different of a collection of components linked by means of permanent references. A component can need to change the component referred (by means of its attribute *externalReference*). A possible situation for this is when a component interacts with one component of a collection. Another attribute can be provided in an Outbox subclass to refer a collection of components (one of then is referred by the attribute *externalReference*).

5. **Broadcasting**. Broadcasting is a kind of communication that can occur in component-based software. In this case a component will not be connected to just other component. So, it needs refer more than one component and it can send messages to all of them. For implementing this situation, the attribute *externalReference* of *Outbox* must keep reference for a collection of objects, instead of just one, and the implementation of required methods in *Outbox* subclasses must pass method invocations to all objects referred. This kind of communication between components is similar to event sending of CCM, as well as to *Observer* design pattern. In this case, the sender acts as the subject and the receivers as the observers.

6. **A component interface implemented as a framework**. When an interface specification is useful for more than one component (a component family, with equal interfaces) it can be implemented as an object-oriented framework[1]. After, the framework can be used for building different components. In a framework that implements an interface component, procedures pointed to build the internal structure are kept abstract. So, the subclass of *ComponentInterface* of an interface will not override the inherited method *defineImplementors*, that is, it will be an abstract class. The overriding will occur in framework using. SEA environment supports the automatic creation of a framework from an interface component specification, as described in *Known use* section [SIL 00].

[1] About object-oriented frameworks, see [FAY 99] [WIA 91] [WIR 90] [JOH 93] [LEW 95].

Code

An implementation of the classes *ComponentInterface*, *InterfaceChannel*, *Outbox* and *InterfacePetriNet* in Smalltalk programming language can be obtained in the address http://www.inf.ufsc.br/~ricardo/download/cip.zip. Next, are discussed some details from their implementation (including their subclasses).

The implementation of a Petri net in an interface is very simple. Places are couples that the first element is the place id (a string) and the second, the place marking (an integer). The transitions are strings (just transition id). The arcs are couples with two strings: origin id (transition or place) and target id (transition or place). Bellow is presented the implementation of the method *createPetriNet* of a *ComponentInterface* subclass (according to figure 3 behavior specification). The methods *addPlace* and *addTransition* check duplicity (*InterfacePetriNet* class). The method *addArc* (*InterfacePetriNet* class) checks if the origin and target are different kinds of object (never from a transition to a transition nor from a place to a place) and if they are contained in the Petri net.

createPetriNet
```
| pnVariable |
pnVariable := InterfacePetriNet new.
pnVariable addPlace: 'P1' with: 1.
pnVariable addPlace: 'P2' with: 0.
pnVariable addTransition: 'T1'.
pnVariable addTransition: 'T2'.
pnVariable addTransition: 'T3'.
pnVariable addTransition: 'T4'.
pnVariable addArc: 'P1' with: 'T2'.
pnVariable addArc: 'P1' with: 'T3'.
pnVariable addArc: 'T3' with: 'P2'.
pnVariable addArc: 'P2' with: 'T4'.
pnVariable addArc: 'T4' with: 'P2'.
pnVariable addArc: 'T2' with: 'P2'.
pnVariable addArc: 'P2' with: 'T1'.
pnVariable addArc: 'T1' with: 'P1'.
^pnVariable.
```

Below is presented the implementation of Petri net firing from class *InterfacePetriNet*

fire: aTransition
```
|inputPlaces outputPlaces fireCondition |
inputPlaces := List new.
outputPlaces := List new.
fireCondition := true.
(self arcs) do: [:anArray |              "determine input places and output places"
        ((anArray at: 1) = aTransition) ifTrue: [outputPlaces add: (anArray at: 2)].
        ((anArray at: 2) = aTransition) ifTrue: [inputPlaces add: (anArray at: 1)]].
inputPlaces do: [:aPlace |     "do all input places have token ? (firing condition)"
        (self places) do: [:aPlaceArray |
                ( ((aPlaceArray at: 1) = aPlace) & ((aPlaceArray at: 2) < 1) )
                        ifTrue: [fireCondition := false] ] ].
fireCondition                                   "yes, fireCondition = true; else, false"
```

```
        ifTrue: [inputPlaces do: [:aPlace |   "remove a token from each input place"
                        (self places) do: [:aPlaceArray |
                            ((aPlaceArray at: 1) = aPlace)
                                ifTrue: [aPlaceArray at: 2 put: ((aPlaceArray at: 2) - 1)]]].
                    outputPlaces do: [:aPlace |   "add a token to each output place"
                        (self places) do: [:aPlaceArray |
                            ((aPlaceArray at: 1) = aPlace)
                                ifTrue: [aPlaceArray at: 2 put: ((aPlaceArray at: 2) + 1)]]].
^fireCondition                                  "return fireCondition"
```

In the creation of an instance of a *ComponentInterface* subclass (constructor method *new* – see *Collaborations*), the method initialize is invoked for the Petri net and the channels. An example of implementation of initialize is presented bellow.

initialize

```
| pnVariable |
pnVariable := self createPetriNet.
channelConcreteChannel2 := ConcreteChannel2 new: pnVariable.
channelConcreteChannel1 := ConcreteChannel1 new: pnVariable.
```

After *initialize* running, the method *defineImplementos* is invoked. This method creates the internal structure objects, it notifies the requirers the external reference and invokes *setImplementor* for linking interface channels to internal implementors of supplied methods. An example of implementation of *defineImplementors* is presented bellow.

defineImplementors

```
| refChannel1 channel1Outbox internalObjectA internalObjectB |
refChannel1:= self channelConcreteChannel1.
channel1Outbox := refChannel1 output.
internalObjectA := InternalClassA newWith: channel1Outbox. “internal structure”
internalObjectB := InternalClassB  newWith: internalObjectA. “internal structure”
self setImplementor: internalObjectA with: 'suppliedMethod1' with: 0.
self setImplementor: internalObjectB with: 'suppliedMethod2' with: 0.
```

The method *setImplementor* in a *ComponentInterface* subclass just pass the requirement to all its channels. An example of *setImplementor* implementation from a *ComponentInterface* subclass is presented bellow.

setImplementor: anObject with: methName with: methParameterListLength

```
channelConcreteChannel2 setImplementor: anObject with: methName with: methParameterListLength.
channelConcreteChannel1 setImplementor: anObject with: methName with: methParameterListLength.
```

At the first moment, when a channel runs *setImplementor*, it verifies if the method identifier (a string with the method name and the parameter list length of the method) is contained in the key collection of its reference structure (a dictionary). In positive case, the object passed is included in the reference structure, using its respective key. The implementation of *setImplementor* from *InterfaceChannel* class is presented bellow.

setImplementor: anObject with: methName with: methParameterListLength

```
((referenceStructure keys) includes: (methName, (methParameterListLength printString)) )
```

```
ifTrue:  [referenceStructure
                at: (methName, (methParameterListLength printString))
                put: anObject].
```

The implementation of a supplied method in an *InterfaceChannel* subclass consists of obtaining the reference of the respective internal implementor object in its reference structure and of passing the invoking to this object. The implementation of an example of a supplied method (*suppliedMethod1*) from an *InterfaceChannel* subclass is presented bellow.

suppliedMethod1

```
| auxVariable runningCondition transitionName |
transitionName := 'T2'.
runningCondition := petriNet fire: transitionName.
(runningCondition) ifTrue: [auxVariable := referenceStructure at: 'suppliedMethod10'.
                            auxVariable suppliedMethod1].
```

The implementation of a required method in an Outbox subclass consists of passing the invoking to external reference. The implementation of an example of a required method (requiredMethod1) from an Outbox subclass is presented bellow.

requiredMethod1

```
| auxVariable runningCondition transitionName |
transitionName := 'T1'.
runningCondition := petriNet fire: transitionName.
(runningCondition) ifTrue:  [auxVariable := self externalReference.
                             auxVariable requiredMethod1].
```

Known use

Component Interface Pattern is used in SEA, an environment that supports the development and use of frameworks and components [SIL 00]. The development of a framework, a component or an application in this environment consists of first building the artifact design specification using UML [RUM 98] [OMG 02] (with extensions), and then checking its consistency and translating it into code, automatically. All UML models of this paper were produced in SEA environment.

In SEA, component-based software development is split into three phases. The first phase is the specification of component interface. The structure is specified defining the channels, the supplied and required methods and the accessibility of each method at the channels, like shown in table 1. The behavioral specification is done by means Petri net, as described in *Motivation* section. Figure 3 shows a Petri net and the respective SEA editor.

The second phase of component-based software development is the specification of the component, using UML. The environment supports automatic conversion of an interface specification in a UML specification that models the interface in an object-oriented approach. The conversion process takes into consideration the classes of Component Interface Pattern. The UML specification produced includes the algorithm of all concrete methods, by means a proper model, what possibly automatic code generation, later. SEA supports the insertion of the interface classes in a component specification, as well as, the automatic building of a framework that implements the interface and, as

described in *Implementation* section, can be used for component building. For both possibilities, Petri net class can be included, or not.

In the third phase, components with compatible interfaces are linked, resulting in component architecture specifications. SEA design specifications includes method body definition and, as the environment has a code generator, the specifications can be converted into code. At this moment SEA has only a Smalltalk generator, but it is not difficult to produce code generator for other object-oriented programming languages. Figure 10 shows some model editors from SEA environment.

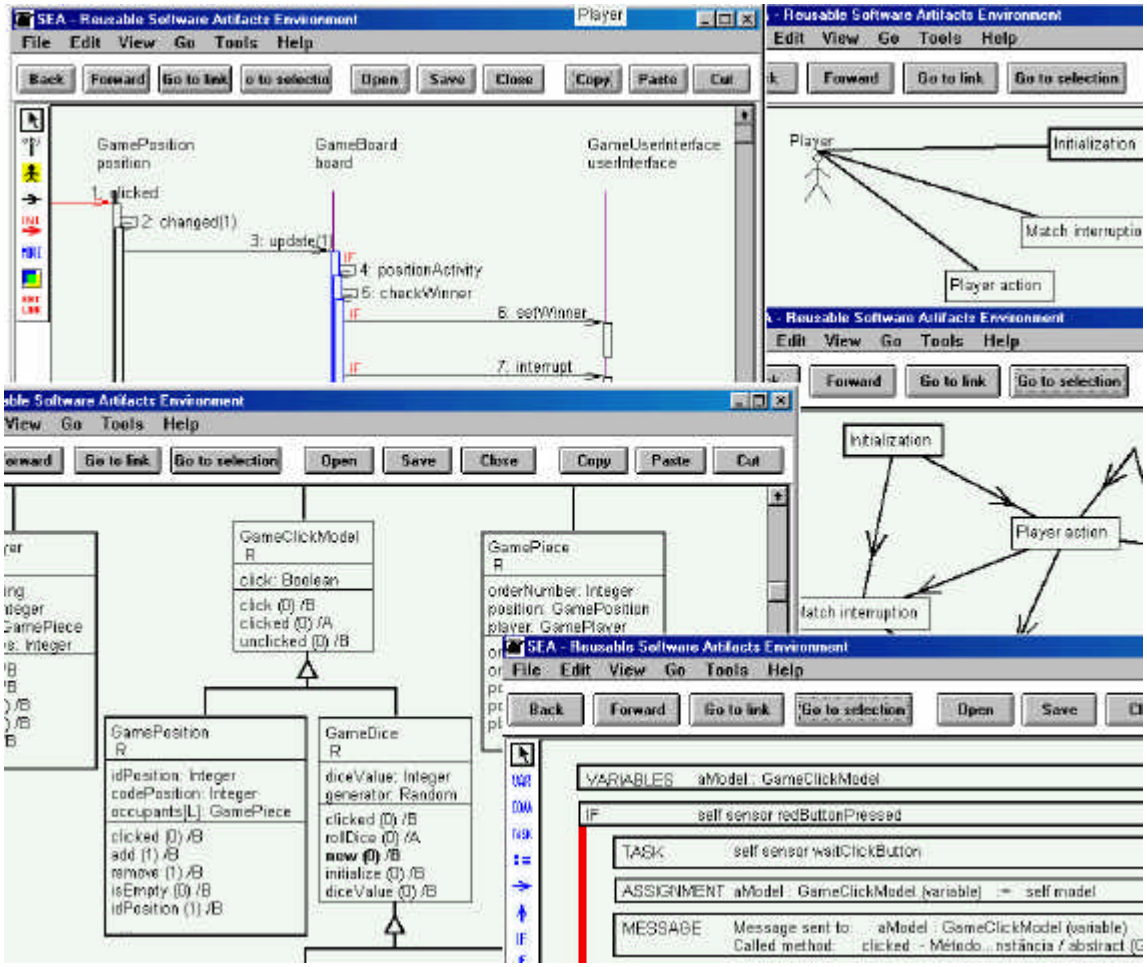

Figure 10 – Some SEA editors

SEA functionality described above shows that is possible to automate the translation of an interface specification into code. Although this, it is not essential this kind of tool support for the pattern application.

Related patterns

Facade. *Facade* is applied in Component Interface Pattern. Concrete subclasses of *ComponentInterface* are facades of specific components.

Proxy. The passing of messages from inside as well as to outside of a component is done in a *Proxy* approach. Concrete subclasses of *InterfaceChannel* define proxies between external elements and internal implementors of component supplied methods.

Equally, concrete subclasses of *Outbox* define proxies between internal requirers and external elements that implement component required methods.

Observer. In broadcasting (see *Implementation* section) the component that sends a message to a collection of components acts as a subject of *Observer* design pattern. The receivers act as observers.

Conclusion

This paper presents Component Interface Pattern, a design pattern pointed to solve problems related to the building software components with interfaces specified structurally and behaviorally. It implements a specific approach of defining component interfaces. In that approach interface structure corresponds to the supplied and required methods, the channels and the accessibility of each method at each channel. The behavioral description adopts Petri net for establishing constraints in method invocation order (component supplied and required methods). The proposed pattern is strongly related to that approach of specifying a component interface and aims to define a direct way of implementing a structure that embodies the specified structural and behavioral constraints.

Component Interface Pattern was adopted in SEA, an environment that supports the development and use of components and object-oriented frameworks, as well as, application development. SEA supports interface specification and the automatic building of the interface of a component by means of translating an interface specification. Otherwise, SEA support is not essential for using component interface pattern.

References

[BOS 97] BOSCH, J. *et al.* **Summary of the Second International Workshop on Component-Oriented Programming**. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 2., 1997, Jyväskylä. Proceedings... Jyväskylä: [s.n.], 1997.

[FAY 99] FAYAD, M. *et al.* **Building application frameworks** - object-oriented foundations of framework design. [S.l.]: John Wiley & Sons, 1999.

[GAM 94] GAMMA, E. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

[HEL 90] HELM, R. et al. **Contracts**: specifying behaviour composition in object-oriented systems. SIGPLAN Notices, New York, v.25, n.10, Oct.1990. OOPSLA, 1990

[JOH 93] JOHNSON, R. E. **How to design frameworks**. 1993. Anonymous FTP in st.cs.uiuc.edu.

[LEW 95] LEWIS, T. *et al.* **Object-oriented application frameworks**. Greenwich: Manning, 1995.

[LUC 97] LUCAS, C. **Documenting reuse and evolution with reuse contracts**. Vrije: Universiteit Brussel, 1997. PhD thesis.

[OLA 96] ÓLAFSSON, A.; DOUG, B. **On the need for "required interfaces" of components**. In Special Issues in Object-Oriented Programming, Workshop of the ECOOP, 1996, Linz. Proceedings... Linz: [s.n.], 1996.

[OMG 99] OMG. **CORBA components**. V.1. 1999. (file orbos99-07-01.pdf, available in www.omg.org)

[OMG 02] OMG. OMG **Unified Modeling Language specification** – action semantics. 2002. (file uml02-01-09.pdf, available in www.omg.org)

[PRE 94] PREE, W. **Design patterns for object oriented software development**. Reading: Addison-Wesley, 1994.

[RUM 98] RUMBAUGH, J. *et al*. **The Unified Modeling Language reference manual**. [S.l.]: Addison-Wesley, 1998.

[SIL 00] SILVA, R. **Suporte ao desenvolvimento e uso de frameworks e componentes**[2]. Porto Alegre: UFRGS/II/PPGC, mar. 2000. PhD thesis.

[SZY 96] SZYPERSKI, C. *et al.* **First International Workshop on Component-Oriented Programming WCOP'96**. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 1., 1996, Linz. Proceedings... Linz: [s.n.], 1996.

[WIR 90] WIRFS-BROCK, R.; JOHNSON, R. E. **Surveying current research in object-oriented design**. Communications of the ACM, New York, v.33, n.9, Sept. 1990.

[WIA 91] WIRFS-BROCK, A. *et al.* **Designing reusable designs: Experiences designing object-oriented frameworks**. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE; EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1991, Ottawa. Addendum to the proceedings... Ottawa: [s.n.], 1991.

[2] Thesis written in Portuguese. Title in English: SUPPORT FOR THE DEVELOPMENT AND USE OF FRAMEWORKS AND COMPONENTS