KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2007
Cluj-Napoca (Romania), June 6–8, 2007, pp. 332–340

ARCHITECTING AND SPECIFYING A SOFTWARE COMPONENT USING UML

DRAGOŞ PETRAŞCU(1) AND VLADIELA PETRAŞCU(2)

Abstract. The current paper experiments a component-based approach for the *LCD Wallet Travelling Clock* case study. It proposes a component architecture and tries to formally specify its building blocks using UML and OCL. Following this architecture and specifications, a JavaBeans implementation has also been developed.

1. Introduction

Software components can be thought of as *units of composition with contractually specified interfaces and explicit context dependencies only* [6]. Ideally, they should be black boxes, enabling third parties to reuse them without knowing the details of their inner structure [4]. The interfaces they provide should be the only access points. Therefore, specifying precisely these interfaces becomes of utmost importance for both clients (which rely solely on that specification when accessing a component's services) and implementors (which are provided with an abstract definition of a component's internal structure).

At least three levels can be identified when specifying software components: syntactic, semantic, and nonfunctional. Without disregarding its significance, the last of these is not covered by our current paper. As for the syntactic specification, it conforms to the following general model: A component implements a set of named interfaces (also known as *provided* or *incoming interfaces*), while making use of the services offered by another set of named interfaces (the *required* or *outgoing interfaces*). An interface consists of a set of named operations, each of which has a number of named parameters (in, out, or inout). Types are associated to all parameters. But even though it is the basis for client code type checking and component interoperability verifications, this kind of specification is missing semantic information. Nothing can be said about the effects of invoking one of the services exposed by an interface, except what might be guessed from the name given to

2000 *Mathematics Subject Classification.* 68N30, 68Q60, 68N19.
Key words and phrases. component specification, UML, JavaBeans.

that operation and the names and types of its parameters [4]. Nevertheless, this is the only type of specification used with dedicated component-based approaches as COM, CORBA or JavaBeans (COM and CORBA use different dialects of the IDL for syntactic specification, while JavaBeans uses the Java programming language).

So as to overcome the above mentioned defficiencies, several techniques for components semantic specification were provided in the literature. Most of them propose a *design by contract* approach in order to formally describe the semantics of the services offered by a software component. We have followed the general guidelines given in [2], which introduces a process inspired, on its turn, by Catalysis [5].

2. Component specification using UML

The approach proposed by [2] enhances the general syntactic model introduced earlier with new concepts. The core one is that of a *contract.* Two types of contracts can be distinguished when specifying software components: usage contracts and realization contracts.

A *usage contract* is a run-time contract between an interface offered by a component object and its clients. Each such contract is represented by an *interface specification* that consists of the following:

- all the services that compose the interface with their signatures and associated behavior;
- the interface's information (state) model and any constraints (invariants) on that model.

The information model associated to an interface is an abstraction of that part of a component's state that affects or may be affected by the execution of operations in the interface. It does not impose any implementation restrictions. It is merely an abstraction that helps in specifying operations behavior. Each such operation is considered as a fine-grained contract in its own right. Behavior is described in terms of pre/post-condition pairs. A precondition is an assertion that must be true before the operation is invoked. It is the client's responsibility to ensure that it holds prior to making the call. It is a predicate expressed in terms of the input parameters and the state model. The postcondition is guaranteed by the component's implementor, after the execution finishes, provided that the precondition was met. It is also a predicate, involving both input and output parameters, as well as the state just before the invocation and just after.

A usage contract is represented by means of an Interface Specification Diagram, with associated constraints. An Interface Specification Diagram is a usual UML Class Diagram, just enriched with some specific stereotypes. It contains the interface to specify and its information model. The interface (including the signatures of its provided services) is figured as a classs having the `<<interface`

`type>>` stereotype. The information model is represented by a collection of associated types (classes), at least one of them having a composition relationship with the interface. The types that are part of the information model, excepting the built-in ones, are stereotyped as `<<info type>>`s. All the constraints (operations pre/post-conditions and information model invariants) are formalized using OCL.

For illustration purpose, we consider an `ISpellCheck` interface [4], implemented by a simple `SpellChecker` component. `ISpellCheck` offers a single method, `isCorrect`, that checks whether a certain word is correctly spelled or not. The interface specification makes use of a basic information model, consisting of a set of strings. There are no invariants associated to this model. The corresponding usage contract is presented in figure 1.

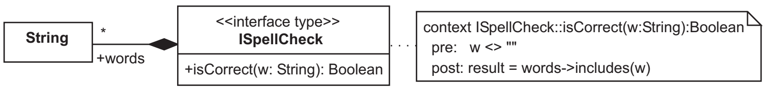
String
*
+words
<<interface type>>
ISpellCheck
+isCorrect(w: String): Boolean
context ISpellCheck::isCorrect(w:String):Boolean
pre: w <> ""
post: result = words->includes(w)

Figure 1. `ISpellCheck` - Interface Specification Diagram

While a usage contract is a run-time contract with the client, a *realization contract* is a design-time contract with the component implementor. The realization contract is represented by the entire *component specification*, that, apart from the usage contracts, includes information regarding the following:

- all the interfaces provided and required by the future component;
- inter-interface constraints (relationships between the information models of the different interfaces);
- interaction scenarios with other components, required in order to implement the provided services.

The required and provided interfaces are illustrated by means of a Component Specification Diagram. This is another variation of the UML Class Diagram, an example being given in figure 2a. Componenent Specification Diagrams form the building blocks of Arhitecture Specification Diagrams for component-based systems.

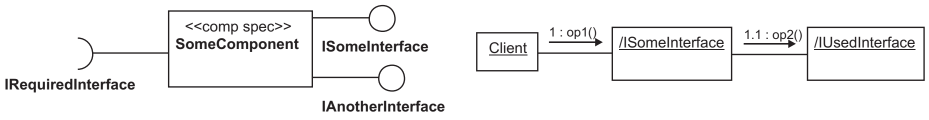

Figure 2. a)Component Specification Diagram b)Collaboration Diagram

When a component implements several interfaces and/or requires the services offered by other interfaces, certain relationships may exist between their associated information models. These inter-interface relationships can be expressed using OCL constraints. Suppose that the above mentioned `SpellChecker` component implements another interface, `ICustomSpellCheck`, having a state model that also consists of a collection `words` of strings, and offering services that allow a potential client to add (remove) words in (from) the collection. Then, the fact that the two interfaces work on the same state model can be expressed in OCL as follows: `ISpellCheck::words = ICustomSpellCheck::words`.

Required interactions with other components, in order to implement the provided services, are best described using UML Collaboration Diagrams, as the one in figure 2b.

3. Case study: LCD Wallet Travelling Clock

3.1. General requirements.

We have followed the above mentioned component specification guidelines in a case study. The object of our case study was the LCD Wallet Travelling Clock (already introduced in [3]), which we have tried to approach (architect, specify and implement) from a component perspective.

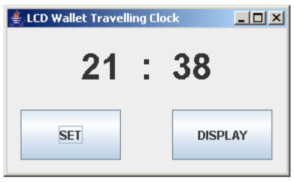

Figure 3. LCD Wallet Travelling Clock

Next, we give a brief description of the clock's requirements. There are two kinds of events that influence its behavior. Firstly, there is an internal *tick* event (generated from inside the clock by an inner ticker) whose occurence causes the time to advance by one second. The same event controls the showing or hiding of the vertical dots separating the two main display sections of the screen. Secondly, there may be external events, generated by the user depressing one of the two buttons offered by the clock (the *display* button and the *set* one). By starting in the default state in which the clock is showing the current time (under the format `hour:minute`) and repeteadly depressing the display button, the display states are went through: date display (under the format `month_day`), seconds display (under the format `_:seconds`, the underscore character indicating an empty display zone), then again time display and so on. Analogously, by starting in the default display state and repeteadly depressing the set button, the setting states are visited: month setting, day setting, hour setting, minute setting, then again time display and so on. While in a setting state, the depressing of the display button causes the value of the component to be set (month, day, hour or minute) to increment by one unit.

3.2. **Component architecture and specification.** Our aim was to give a JavaBeans component implementation for the LCD Wallet Travelling Clock. By analysing the general behavioral requirements described earlier, we have decided to factor the functionality offered by the clock in two provided interfaces: one that allows client code to send `pressDisplayButton` and `pressSetButton` requests, named `IClockKeyboard`, and the other, `IReadOnlyClockDisplay`, used to obtain the values to be shown on a screen-like part of a user interface. In order to accomplish this, `IReadOnlyClockDisplay` exposes the following three operations: `getLeftSide`, `getMiddle`, and `getRightSide`. Besides, the clock component requires the services provided by a `PropertyChangeListener`, in order to notify the user interface about changes occured in the displayed values. The graphical component has the ability to register/unregister itself as a clock listener, by means of `[add|remove]PropertyChangeListener` services, also offered by the `IReadOnlyClockDisplay` interface. All the mentioned interfaces, as well as the dependencies they cause among the clock component and the `ClockFrame` (playing the role of a visual interface) are represented on the architecture specification diagram in figure 4.

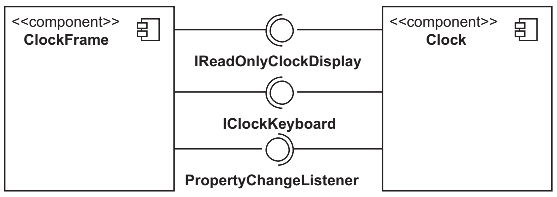

FIGURE 4. External view of *Clock* component

By now, we have offered an external (user) view of our clock component: the interfaces it provides and requires and the way it interracts with its environment. Following, figure 5 shows its internal structure through a component-connector-port architecture. Both diagrams employ the new UML 2.0 component concepts and associated graphical notations.

Basically, the clock behavior is ensured by a `ClockController` component object. As shown on the architectural diagram in figure 5, the `ClockController` offers the same two interfaces provided by the clock (`IReadOnlyClockDisplay` and `IClockKeyboard`); all the messages that the latter receives, requiring services from one of its interfaces, are further delegated to the controller (this is indicated by the delegation connectors that link the two ports on the left with the corresponding provided interfaces). In order to grant this functionality, the `ClockController` component requires the services of three other interfaces, namely `IClockTicker`, `IClockMemory`, and `IClockDisplay`. These are implemented by the `ClockTicker`, `ClockMemory`, and `ClockDisplay` components, respectively. All three are observed components; they have the ability to notify a potential listener (the controller in this case) when certain events (ticks) or state changes occur. Therefore, the interfaces they provide should offer `[add|remove]XListener` type services, requiring, at the same type, services provided by `XListener` type interfaces. As can be seen, the `PropertyChangeListener` interface requied by the clock component is actually required by its `ClockDisplay` (another delegation connector that links, this time, a required interface to a port).

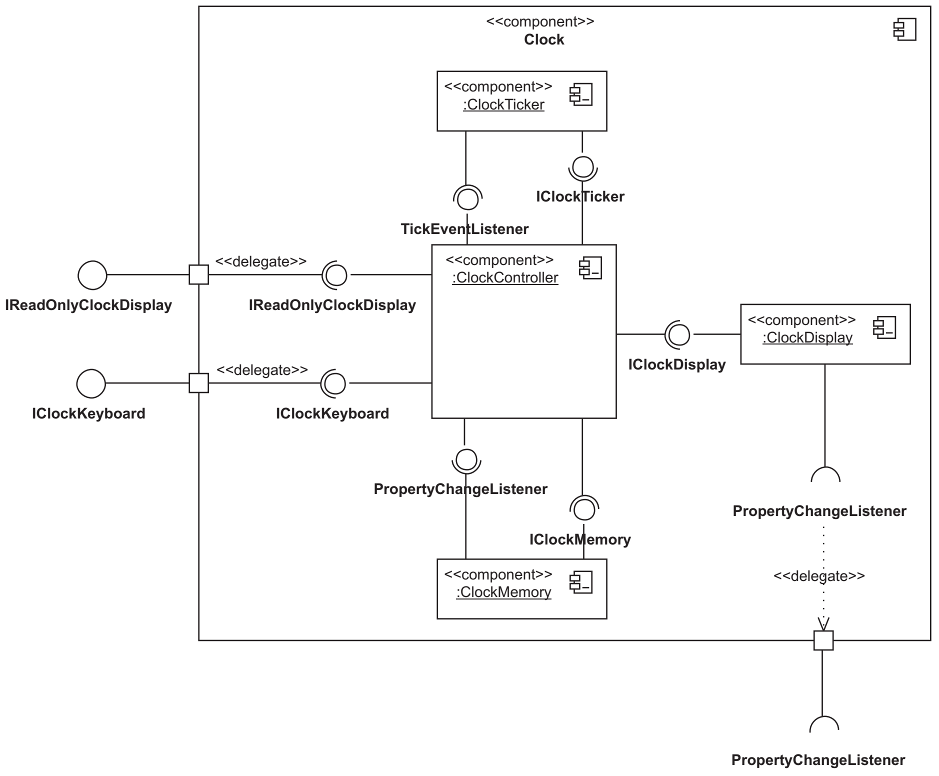

FIGURE 5. Internal architectural view of *Clock* component

In order to accomodate the space constraints of the paper, we will not enter the specification details of all interfaces. We will only insist on `IClockMemory`, by illustrating its information model, as well as some of the operations' specifications.

Figure 6 shows the Interface Specification Diagram for `IClockMemory`. It depicts the interface itself, together with the types that make up its information model. As shown by the diagram, the `IClockMemory` interface is represented as a class having the `<<interface type>>` stereotype, that lists all its services inside

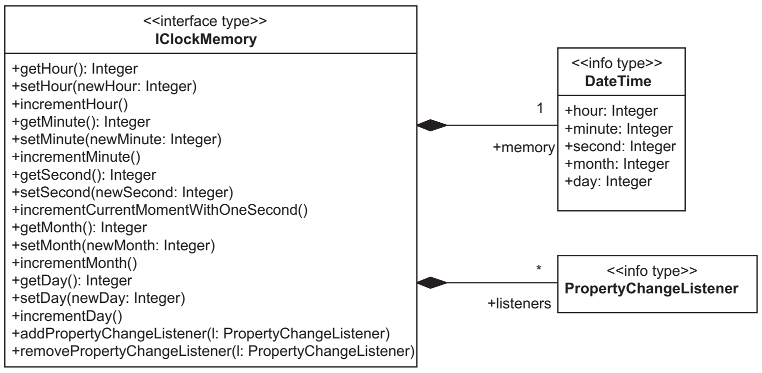

FIGURE 6. `IClockMemory` - Interface Specification Diagram

the operations compartment. Some of these services enable the handling (setting, getting or incrementing) of values stored in the clock memory (i.e. hour, minute, second, day, and month), while the others allow the management (adding or removing) of listeners registered with it. In order to ease the precise specification of these two types of services, we have associated to the `IClockMemory` interface an information model consisting of two classes, namely `DateTime` and `PropertyChangeListener`. Both have the `<<info type>>` stereotype (since they represent informational types) and are linked to the interface via composition relationships.

To conclude with the usage contract corresponding to `IClockMemory`, listing 1 shows some of the OCL pre/post-condition pairs that specify the operations' semantics. The entire OCL specification has been validated with OCLE 2.0 [1].

3.3. **Implementation issues.** Up to now we have concentrated on creating a component architecture and specifications. We have tried to make them as independent as possible from technological issues. The *provisioning* and *assembly* [2] phase is concerned with providing realizations for previously defined component specifications (either by finding existing implementations or developing them from scratch), as well as binding them as architected.

We have implemented our *Clock* component all from scratch using the JavaBeans standard. Each component specification has its counterpart in a JavaBean class (its realization). The bean implements all the provided interfaces appearing on the component specification diagram. As for the required interfaces, they were managed by storing their references inside the component object. This dependency allows calling their services whenever needed. The real component objects supporting these services were plugged inside the client component by means of specialized "plugging interfaces" that we have created. We regard these so called

Listing 1. IClockMemory - OCL constraints. ocl

```
context IClockMemory::getMonth():Integer
  post: result = self.memory.month

context IClockMemory::setMonth(newMonth:Integer)
  pre:  newMonth >= MIN_MONTH and newMonth <= MAX_MONTH
  post: self.memory.month = newMonth

context IClockMemory::incrementMonth()
  post: self.memory.month = self.memory.month@pre mod NO_OF_MONTH  + 1 and
        if self.memory.day@pre > noOfDays(self.memory.month)
        then self.memory.day = MIN_DAY
        else true
        endif

context IClockMemory::incrementCurrentMomentWithOneSecond()
  post: if self.memory.second@pre < MAX_SECOND
        then self.memory.second = self.memory.second@pre + 1
        else self.memory.second = MIN_SECOND and
          if self.memory.minute@pre < MAX_MINUTE
          then self.memory.minute = self.memory.minute@pre + 1
          else self.memory.minute = MIN_MINUTE and
            if self.memory.hour@pre < MAX_HOUR
            then self.memory.hour = self.memory.hour@pre + 1
            else self.memory.hour = MIN_HOUR and
              if self.memory.day@pre < noOfDays(self.memory.month@pre)
              then self.memory.day = self.memory.day@pre + 1
              else self.memory.day = MIN_DAY and
                if self.memory.month@pre < MAX_MONTH
                then self.memory.month = self.memory.month@pre + 1
                else self.memory.month = MIN_MONTH
                endif
              endif
            endif
          endif
        endif

context IClockMemory::addPropertyChangeListener(l:PropertyChangeListener)
  post: listeners = listeners@pre->including(l)

context IClockMemory::removePropertyChangeListener(l:PropertyChangeListener)
  post: listeners = listeners@pre->excluding(l)
```

"plugging interfaces" as playing the role of assembly connectors, linking a component's required interface to a corresponding provided one.

Several patterns have been applied during the design process. Among them, we may mention the *Observer* pattern, used for managing the raising of events by the `ClockTicker` as well as the state changes occured in the `ClockMemory` or `ClockDisplay`, the *State* pattern, employed for handling the `ClockController`'s dynamic behavior, or the *Factory Method* pattern, used in building the `Clock` itself from components. Readers interested in all these details are strongly encouraged to contact the authors.

4. Conclusions and future work

The current paper belongs to a series of works trying to apply different formal specification techniques for the *LCD Wallet Travelling Clock* case study. This time, we have experimented a component-based approach. We have proposed a component architecture and we have tried to formally specify its building blocks using UML and OCL. Following this architecture and specifications, a JavaBeans implementation has also been developed. As a future research direction, we intend to study the opportunities given by formal specifications in the field of test automation. Precisely, we will try to derive JUnit test cases based on previously described OCL constraints.

References

[1] OCLE Homepage. `http://lci.cs.ubbcluj.ro/ocle/index.htm`.

[2] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[3] Vladiela Ciobotariu-Boer and Dragoş Petraşcu. X-Machines Modeling. A Case Study. In Militon Frenţiu, editor, *Proceedings of the Symposium "Colocviul Academic Clujean de Informatică"*, pages 75–80. Faculty of Mathematics and Computer Science, "Babeş-Bolyai" University of Cluj-Napoca, România, June 2005.

[4] Ivica Crnkovic and Magnus Larsson (editors). *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[5] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

(1) Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Str. Mihail Kogălniceanu nr. 1, RO-400084 Cluj-Napoca

E-mail address: `petrascu@cs.ubbcluj.ro`

(2) Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Str. Mihail Kogălniceanu nr. 1, RO-400084 Cluj-Napoca

E-mail address: `vladi@cs.ubbcluj.ro`