CONTRACTUAL SPECIFICATION OF COMPONENT USING VIRTUAL INTERFACE

Eustache MUTEBA Ayumba
Researcher/Lecturer
Correspondent of IMIA, NEM and EASST in Democratic Republic of Congo
Email: emuteba@hotmail.fr

Abstract
This paper describes a method for providing a reliable reusable component using a virtual interface. Specification of interfaces between functionality entities, supporting dynamicity and variability of software, so that new functionalities can be learned and automatically be integrated in future applications. Therefore the virtual interface needs to be designed using method and techniques of self-composition of components on demand on the basis of essential criteria such as components specification, version, cost and availability.

Keywords: Contractual Specification, Software Reuse, Component, Virtual Interface

INTRODUCTION
To date computer science technology is based on standardized communication infrastructures that are used by different organizations. Idea and challenge beyond software reuse is developing component that can support interoperability and reusability through a framework called a contract. This contract can allow others developers to build her system by assembling reusable components. The contractual specification of component is a contract templates that is specifies what an existing component will do if it is incorporated into a client application. This paper describes a method for providing a reliable reusable component using a virtual interface. Specification of interfaces between functionality entities, supporting dynamicity and variability of software, so that new functionalities can be learned and automatically be integrated in future applications. Therefore the virtual interface needs to be designed using method and techniques of self-composition of components on demand on the basis of essential criteria such as components specification, version, cost and availability.

FORMALISATION OF VIRTUAL INTERFACE

Contract vs Interface

Idea of contract is borrowed to those related to human activities where there are commonly two parties: a provider, which performs some task for the other a client. Contract's content is a protocol that specifies how much should be done by the provider and whether the client can accept or reject the contract. Thus applying to software, a contract leads the interaction between two modules or methods. In the other words, a module views as a “component is a black box entity with a set of required and provided services”. Remark in that the notion of “black box and white box is introduced to specify the degree of reusability. The degree, to which, a software module or other work product can be used in more than one computing program or software system”. The composition based on the contract enables communication between components to provide a service and to request a service. To be use and integrating in custom application, component must adhere to a binary form. The point of contact of custom application and the component is the runtime binary representation. It comes that the contract ‘is a specification stating what the client application needs to use the interface. Vice-versa, it states what the provider application has to implement to meet the service
promised by means of the interface. In the other words the interface specification is a contract that specifies what an existing component (provider) will do if it is incorporated into a client application (request). Ideally, a contract should cover all essential functional and non-functional aspect”.

Furthermore, the interface specifies the externally visible operations or the limited part of the behaviour by which components interact, and has no implementation of its owns. Ideally, the interface specification can contain the following pieces of information:

**a. Operation semantics**
It is a description of each operation using:
- informal text,
- pre/postcondition
- invariant

**b. Interface protocol**
It provides the constraints on the order in which operations may be called. Non-functional aspects and others functional aspects can be considered like the constraints.

**c. Service level**
It covers guarantees regarding the qualities or non-functional requirements such as:
- timing constraints, CPU budgets restrictions, memory restrictions, availability, mean time between failures, mean time to repair, throughput, latency, data safety for persistent state, capacity, and son on.

**Traditional Interface vs Virtual Interface**

The Virtual Interface (VI) is an interface that allows the system so called contractor to know the needs of custom application before to provide the services and at composition time provides the plug-compatible component. With the virtual Interface, services are automatically adapted to the current component available particular:
- the component can provide two kinds of interface: Server interface and Virtual Interface,
- the specification of the server interface is a set of functionalities in traditional way,
- the specification of virtual interfaces is a machine-interpretable that dynamically learned the request of custom application and automatically integrate plug-compatible component. This is the self-assemble on demand on the basis of the contract generated by the Virtual Interface when exploring custom application. This is an intelligent service that helps service contractor in creating services for different contexts.

The VI is designed to collects information from different context sources (networks, servers, devices, platforms, ...). In order to have a better indication of the user needs. Assuming the VI can provide a solution for the software reuse problem by finding an appropriate component to be used by a specific client application. The client, in one hand, is initiated the processing as following:
- announce the task that needs to be performed,
- receive a contract to a suitable contractor,
- award a contract to a suitable contractor,
- receive and synthesize results.

In the other hand, the provider is reacting as following:
- receive task announcements,
- evaluate his capability to respond,
- respond (decline or bid),
- perform the task if his proposal is accepted,
- report the results.

It comes that the Virtual Interface as a contract contains the following information:

**Contract:**
A request-response protocol

Data-Field

Execution-Information

Message

Message-Client

Message-Provider

Context

Context-Data-Client

Context-Data-Provider

Results

The Request-Response Protocol is transactions divided into two parts, contracted and the contractor, one for requests and the other for responses. A Request-Response Protocol can be defined to facilitate easier testing and integration of components that is validating a VI. Transaction data fields are used for performing a transaction. Two kinds of data fields are considered the execution information that controls the execution of the transaction, and the fields that are sent messages form the contracted or the contractor.

The most types of message are query, reply, explanation, command, permission, refusal, offer, bid, proposal, confirmation, acceptance, agreement, retraction, denial.

The context can take in account the context data client and by the context data provider.

The context permits to specify the components and their relationships in area of interest. The semantics methods enable the representation and manipulation of context information through machine-interpretable. The machine-interpretable can be based on two steps:
- Extraction of information,
- Verification and validation.

The core of the machine-interpretable as an algorithm is described as:

```
Rule (number)
  <domain> "Context"
  <Pattern> "Described the context"
  => <Condition> "semantic constraints on the matched pattern"
  <Action> "tell what to do if the condition is true"
```

The result reports the conclusion of transaction between the client and the provider.

**CASE STUDY: A PRACTICAL EXAMPLE**

The case bellow illustrates in pseudocode such as language Java and is an example of the business component “Movable Machine” that emphasizes the software reuse and evolution by:
- a simple component,
- a server interface,
- a versioning problem,
- a virtual interface,
- a simple client application.

**Primary Approach**
Provider - Component "Movable Machine"

The provider has to declare a component for a car that implement the interface movable machine as follow:

```
// Ver 1.00
Class CCar implements IMovable{
    boolean start() {
        // Body of code to do here
        ...
    }
    void stop{
        // Body of code to do here
        ...
    }
    boolean turn(int degrees) {
        // Body of code to do here
        ...
    }
    double fuelRemaining(double amount) {
        // Body of code to do here
        ...
    }
    boolean changeSpeed(double kmperhour) {
        // Body of code to do here
        ...
    }
}
```

With Interface

Consider typical operations used on machines than move. These operations are start machine, stop machine, turn machine, fuel remaining and change speed. IMovable as interface defines what all movables machines must be able to do. They may do more, but this is a minimum functions.

```
// Ver 1.00
Interface IMovable{
    boolean start();
        // Do whatever is necessary to start
        // a machine and return true if it worked
    void stop;
        // Do whatever is necessary to stop a machine
    boolean turn(int degrees);
        // Do whatever is necessary to turn a machine
    double fuelRemaining(double amount);
        // Return the amount of plane fuel remaining
    boolean changeSpeed(double kmperhour);
        // Accelerate or decelerate
}
```

Request - Client Application

The client application can reuse this component CCar through an interface IMovebale as follow:
module MovableClient {
    Interface IMovable {
        boolean start();
        void stop();
        boolean turn(int degrees);
        double fuelRemaining(double amount);
        boolean changeSpeed(double kmperhour);
    }
}

If a client wants to use the component "Machine Moveable" specifically invokes an operation start() on the interface IMovable.

import class RemoteControl {
    private IMovable machine;
    RemoteControl(IMovable m) {
        machine = m;
    }
}

public static void main(String args[]) {
    //Invoke the operation start()
    boolean okay = machine.start();
    if (!okay) displays("No response on start");
}

**Evolution - Versioning**

The new component CCar is redefined as:

// Ver 1.1

class CCar implements IMovable
    boolean start() {
        // Body of code to do here
    }
    // Other old operations to put here
    boolean changeSpeed(double kmperhour) {
        // Body of code to do here
    }
    // New operation
    int fail(int status) {
        // Body of code to do here
    }
}

The new version of the interface IMovable() defines new operations of component CCar. This is a traditional interface IMovable2() that inherits directly from IMovable.
The **new interface IMovable2** is defined as:

```java
// Ver 1.1
interface IMovable2 extends IMovable {
    int fail(int statut);
}
```

---

**New Approach**

**Provider - Component "Movable Machine"**

The provider has to declare a component for a car that implement the Virtual Interface movable machine to handle the evolution as follow:

```java
// Ver 1.00
Class CCar implements VIMovable{
    boolean start() {
        // Body of code to do here
        ...
    }
    void stop{
        // Body of code to do here
        ...
    }
    boolean turn(int degrees) {
        // Body of code to do here
        ...
    }
    double fuelRemaining(double amount) {
        // Body of code to do here
        ...
    }
    boolean changeSpeed(double kmperhour) {
        // Body of code to do here
        ...
    }
}
```

---

**New Realise by Virtual Interface**

When using a Virtual Interface, the interface is defined at the high level and can give possibility of different extension.

The Virtual Interface VIMovable is defined as:

```java
// Ver 1.00 of the basic component
interface VIMovable {
    boolean result;
    struct requestResponseProtocol;
    {
        boolean executionInformation;
        string messageClient;
        string messageProvider;
    };
    struct Context;
```
{  
    boolean client  
    boolean provider  
    string contextClient  
    string contextProvider  
  };

String MachineInterpretable(string interfaceSignature) {  
    ...
    // Code for exploring the available interface based on context
    ...
}

DISCUSSION

Component specification, in general, is the process of stating, as precisely and completely as possible, how a projected component product will behave. Component specifications form a kind of contract between the developer and the purchaser. So long as the delivered component meets its specifications, it is considered acceptable. In addition, the specifications form the groundwork on which the component is built. For these reasons (and for many others) the specification stage is vital to the overall success of a development effort. The contract provides a basis for reasoning about the properties of component assemblies and provides a basis for certification of component. This means that a component can be replaced at runtime with a component with the same functionality eventually from other vendor (upgrading).

In many case, as long as the implementation respects its contracts, revisions pass unnoticed by clients. A new realise adhering to the original contract but changing performance can break client.

Thus the Virtual Interface is used in order to notify dynamically and automatically the client application about the evolution of the component without to cause any break.

REFERENCES

Component Object Model (COM), DCOM and related capabilities - Software Technology Review.