Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

A Development Method Boosted by Synchronous Active Objects

Claude PETITPIERRE
EPFL, Laboratoire de Téléinformatique, 1015 Lausanne, Switzerland

Abstract. This paper presents a novel development method for interactive and distributed applications. The benefit that this method provides is that the design gives clear guidance towards the implementation. The method is based on three main elements: a concept of synchronous active objects that is closely related to that found in CSP; the Java environment; and a selection of the diagrams defined by UML. This approach alleviates many of the most serious problems that are encountered when using GUI builders, which hide the application structures and so make it difficult to devise sound architectures. The final part of the paper briefly describes an application generator that will help a developer to implement a design which is developed according to the approach advocated here.

1 Introduction

The weak points of most current development methods lie in the coding phases [1]. Although much of the literature describes the implementation process in detail [2] [3], there is a lack of a comprehensive set of approaches which guide the development precisely from the design down through the implementation phase. This paper shows that the concept of synchronous active objects, based on CSP's key concept [4], leads to a simple methodology that works well for the development of interactive and distributed software applications. The approach presented here is tailored to the Java environments and libraries, and based on UML [5]. It is supported by a set of program generators (available on the Web [11]) which make it possible to develop prototypes of interactive and distributed applications very quickly.

After it became clear that the "waterfall" class of development methods was generally unworkable or produced unexpected results, software specialists began to define iterative methods, which cycle over a number of phases such as analysis, design, implementation and test, with more success, but still with many failures too. Taking a cues from the "gurus" who could create wonderful programs by just rolling up their sleeves and working hard until the system works, a further step was taken into what is called eXtreme Programming (XP) [6]. This method stresses the communication between the team members and the customers, the simplicity of the code, the feedback from testing, and the developer's courage and pride. The extreme developer takes pride in an affordable working result obtained in timely fashion, rather than in a dazzling, sophisticated, expensive, and possibly late, solution.

This paper presents concepts that complement rapid-cycle approaches, such as XP, with a set of documents and rules that lead quickly to realistic prototypes. It exploits the benefits of UML by selecting a set of diagrams and concepts from this language and by

proposing a precise way of using them. The introduction of synchronous active objects into the method makes it possible to create working prototypes directly from the UML diagrams, which allows a team to discuss the architecture of the application and to involve the stakeholders in the project and get their feedback early. The prototypes can be created, rebuilt, refined, and completed several times, approaching ever-closer to the goal, and they may even persist into the maintenance phase of the product. The proposed approach supports multi-tiered architectures in which the GUIs, the business logic, and the local or remote accesses to the databases are embedded in separate components (or tiers).

Section 2 briefly defines the concept of synchronous active objects. The proposed method, summarized in section 3, iterates over the following steps: design of collaboration diagrams (section 4), design of use cases (section 5), refinement of the collaboration diagrams (section 6), design of state charts for the active components (section 7) and creation of prototypes. The prototypes can be created with the help of a program generator (section 8) and a compiler capable of handling synchronous objects. The generator takes as input a diagram very close to a collaboration diagram. Finally, an example of development that leads to a prototype is presented in section 9.

2 Synchronous Active Objects

The concept of synchronous active objects is described in [7], [8], [9] and [10], but is briefly summarised here for the sake of completeness. A synchronous active object is implemented as a Java active object, namely an object that has a method *run* executed on a thread dedicated to the object. A call to a synchronous active object (either from a passive or an active object) is blocked until the synchronous object explicitly accepts that the called method be executed, by means of the statement written below.

```
accept someLocalMethod;
```

The *accept* statement is also blocked until a call has actually been performed, which realises a rendezvous like the one specified by CSP. The synchronous active objects also define a *select* statement that allows them to specify the set of (blocking) calls and acceptations they are ready to execute. This *select* statement, presented below, exactly implements the CSP alternation shown on the right of the *select* statement.

```
public void myM(int i){
    local = i;
}
public void run () {
     . . .
P:
  select {
  case
      activeObject.m(x);
      // statement possible
  case
      accept myM;
      // statement possible
  case
      activeObj.met(y);
      // statement possible
  }
Q: . . .
}
```

CSP equivalent

```
P =
(
   activeObject_m ! x → Q()
|
   thisObject_myM ? x → Q(x)
|
   activeObj_met ! y → Q()
)
```

A call to a method is equivalent to sending some parameters into a channel. An *accept* statement, accepting a method that stores the parameters locally, corresponds to the input of the parameters. In the example given above, there is a single local variable, named *local*. *Q(x)* corresponds thus to the object on the left, when its location counter is set after the selection (at *Q:*) and *local* contains *x*. If no value has been passed during the transition from *P* to *Q*, *Q( )* keeps the value previously available in *local*.

3 The Documents Used to Drive a Project

UML is only a language and may be used within various development processes. In the following, we define our own process by selectinga subset of UML documents and their relationship. The approach is supported by the synchronous objects defined above, which make it possible to map the design to a prototype implementation that can be executed. This prototype can be generated with the help of a graphical generator that represents a diagram very close to a collaboration diagram.

- The ***collaboration diagrams*** provide a "geographical" view of the application, a kind of playground showing the components, namely the actors, the sources and the sinks of data as well as the communications established between these elements. The lines that represent the communications can be decorated with the message transmissions or the operations that correspond to the use cases.
- The ***use cases*** describe the scenarios that may be played in the application playground described by the collaboration diagrams. They are written in plain text. There may be one or more use cases per collaboration diagram. Thus, they may be represented by the same set of components on several versions of a collaboration diagram. A typical use case traces the retrieval of some data from a GUI, the forwarding of those data to some central server, and their introduction into the database.
- The ***class diagrams*** define the datasets and their structures. They specify passive classes, with their attributes, methods, and the aggregations they form. We will not be concerned with class diagrams further in this article, because they are well treated in the relevant literature.
- The ***state chart diagrams*** are used to specify the behaviours of the components of the collaboration diagrams, namely the sequences of events the components receive or produce. A state chart diagram is thus defined at the intersection of the use cases and a component. State chart diagrams are particularly well adapted to describing the behaviour of active components.
- If the aforementioned diagrams have been correctly designed, they should be easily amenable to a ***prototype***. A prototype helps to verify that the assumptions made by the developer are correct and that she or he masters the technology on which the project will be based. A prototype is the key "document" to which the members of the project team can refer in order to understand and extend the application. Every (new) member of the development team should be able to view the early prototypes, so as to understand the structure of the application. While the project is progressing, the prototypes may, step by step, become the definitive application.

All the diagrams listed above, as well as their relationships, are detailed in the following sections.

4 Collaboration Diagram

Most development methods suggest starting a project with the design of the use cases. However, use cases involve actors, or components. We therefore propose to start the design by drawing a collaboration diagram, which corresponds, as previously mentioned, to the playground on which the actors and the components will exist. The first collaboration diagram should not represent an abstract situation, but a concrete one, with the central system, some terminals, the databases, the GUIs, special devices, and so on. A concrete example makes it easier for the developer to imagine the operations that must be carried out on the elements. For example, instead of representing all clients by the client class, it is better, at least at the beginning of the project, to represent them by a few hypothetical client components.

A collaboration diagram contains the following components:

- The ***sources*** and the ***sinks*** of data (the keyboard, the display, the printer).
- The main ***data pools***, such as the database servers and the aggregations of data kept in the local memory. A data pool is both a source and a sink of data.
- The ***controllers***, which include the data conveyors, the data transformers and the schedulers.
- The ***actors*** (human, devices) who create events and start sequences of operations.

Figure 1 is an example of a collaboration diagram. The rectangles correspond to the components listed above. The rectangles with thick borders represent units that, unlike passive objects, can react on their own, not only when they are being called (humans, devices, active objects). The rectangles with the thin borders correspond to passive objects. Passive objects should only be used as terminals. Their internal structures must be described in the class diagrams. The behaviour of the standalone components may be described by state chart diagrams, as explained below.

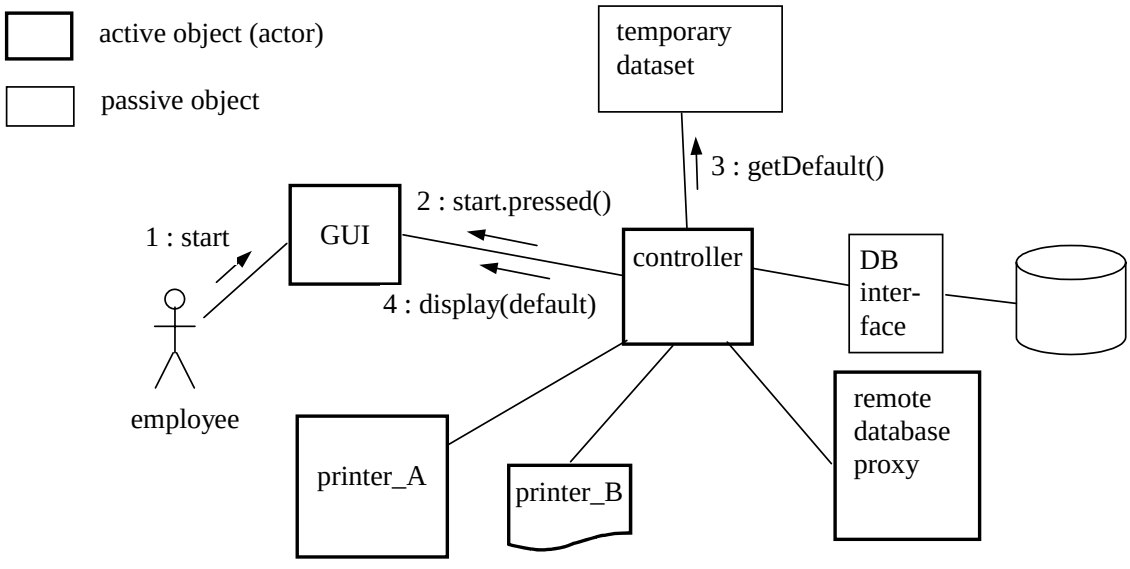

Figure 1. An example of a collaboration diagram.

5 Definition of the Use Cases

A use case is a scenario played on a collaboration diagram. In a system based on synchronous objects, all actions performed by the components can be expressed by calls or acceptances. A use case is defined by a name and by the flow of events and operations that

are exchanged between the components of a collaboration diagram. A use case specifies part of the functionality of a system. The flow of events is written in plain language. The actors involved in the events use the names of the components defined in the collaboration diagram. The description of the events should follow the format given below as much as possible, to assure the coherence of the use cases. They are attributed sequential numbers.

Use case X

1. actor verb element acted upon.
2. actor verb element acted upon
3. . . .

For example, with respect to Figure 1:

Entering a new person

1. the employee clicks button start in the GUI
2. the controller executes *start.pressed* (namely, has its call accepted)
3. the controller gets the default person frame
4. the controller displays the default frame on the GUI
5. and so on

The reader may wonder why the use case diagrams defined by UML are not presented. The reason is that these diagrams carry very little information, and besides being shown as illustrations on the cover of the report to the customer, they have very little utility. The collaboration diagrams include the same information, and, unlike the use case diagrams, they can express scenarios that involve a succession of actors and components.

6 Refinement of the Connections Between the Components

The collaboration diagrams contain several kinds of communications that must be specified within a distributed application: the communications between the GUI and the local body of the application, between the body of the application and the remote sites (possibly in both directions), between the server body and the databases, and so on.

After a rough version of the collaboration diagrams has been devised, they must be refined with respect to these communications. This refinement actually amounts to devising the software architecture, namely attributing the application functions to a set of processes and then defining their interconnections to each other and to the devices they use (databases, GUI, sensors). This architecture should be adapted to the performances and security required by the application, taking into account the location of each pair of communicating components: these may be located in the same program (loaded in the same memory partition), in the same computer (but as a program started separately), or on different computers. In this paper, we assume that the system is realised using synchronous active objects.

6.1 Analysis of the Remote Accesses

In the following, we will first discuss the weaknesses of a naive implementation and then show how the introduction of a controller clarifies the program structures and avoids meddling with the *synchronized* keyword, a dangerous concept when it is used at the application level. Note that the sequence diagram used for the explanations is an auxiliary feature, and is not intended to be a part of the core method.

This example, extracted from the application discussed in the second part of this paper,

is the following: we assume that an alarm triggered by some sensor (in the event that a temperature reaches some limit) must be sent to a central server, which in turn broadcasts the alarm to several services (for example the police and the fire department).

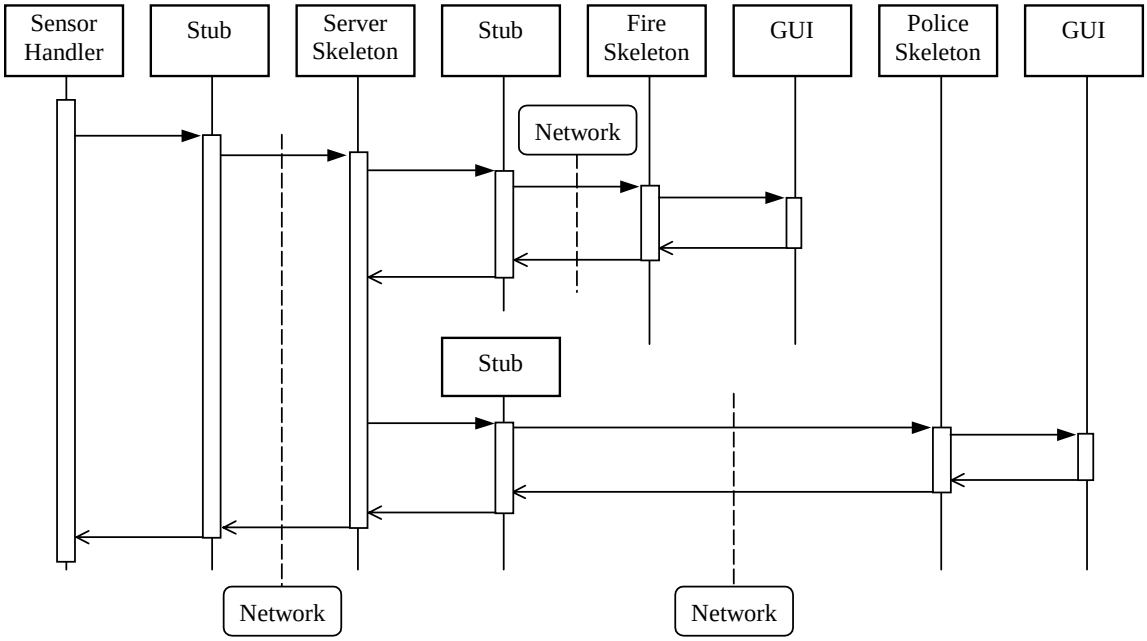

Figure 2. Element involved in the broadcast of an alarm.

The communications illustrated in Figure 2 represent both local and remote calls. The calls that cross a dotted line with the indication *Network* are remote communications, which, we assume, are implemented by remote method invocations (RMI). When the sensor handler triggers an event, its program performs an RMI to reach the server skeleton. The latter calls the stub that leads to the fire department skeleton and then the stub that leads to the police skeleton. As the sequence diagram clearly shows, the sensor handler is blocked until the data have been broadcast to all services concerned with the broadcast alarm. This use case requires several network traversals (three in Figure 2). Thus, if the network is congested, the sensor may be prevented from performing its surveillance task for unacceptably long periods of time.

Moreover, several sensors located at different places may interleave their calls. An RMI skeleton starts a new thread each time a client calls a remote method, and so these methods may access several data structures in the server concurrently, introduce race conditions and corrupt the data structures. These interactions therefore need to be coordinated, which may slow down the operation even further.

The coordination may be done using either of two different philosophies: a defensive philosophy and a constructive one. Under the defensive philosophy, the developer wraps her or his data within monitors defined by the *synchronized* keyword. This approach is simple for simple applications, but it does not scale up well, as it may serialize the requests of all sensor handlers and even lead them to deadlock. The developer must also assure that the stubs are re-entrant, that the accesses to the database do not exceed the allowed number of simultaneous accesses, and so on. This approach is defensive in the sense that it only determines and prevents what is forbidden, but allows the system to run at its own pace otherwise.

Under a constructive philosophy, the developer designs an active distributed structure. In the case of the server defined in the example, one or several active objects may be installed

in the server and all alarms may be sent to this or these objects. The objects execute the requested operations, letting the methods of the skeletons return to their callers as soon as they have delivered their messages. Several approaches may again be envisaged to connect a skeleton method to the active objects: creating a new object each time one is required, retrieving one object out of a pool of active objects, or dedicating active objects to the different roles required by the application.

Creating a thread for each new call makes it easy to pass the data to the object that must process them: they can just be stored in the parameters of the constructor. However, the creation of threads has its drawbacks. It is both time and memory consuming, and the number of threads that a program may execute simultaneously is limited by the underlying operating system. Some means must thus be devised to count the number of active threads, to suspend a caller if the upper number of threads is reached, and to resume the activity of that caller once a thread is again available.

Managing a pool of threads is not that much of an improvement, even if it is often described in the relevant literature. First of all, once a thread has been retrieved from the pool, the data to process must be transmitted to the threads, which requires a specific construct. If the threads perform different kinds of activities, it may happen that all threads are used for one kind of activity at the expense of other activities. This problem could be solved by using several pools, but the following solution has the same effect and solves the problem in a simpler manner.

The most effective way to entrust active objects with activities is to use channels or synchronous calls. With this approach, a set of threads is dedicated to each kind of task, which makes it easy to determine the number of threads attached to each one. All the threads belonging to the same group await messages from the same (limited or unlimited) channel, or accept a method dedicated to the reception of messages. An object that wants an active object to perform a given action sends a message to the corresponding channel, or calls the synchronous object that performs that action. If there is a single controller, the latter call may be a simple synchronous call. If the application uses several active objects, the following statements can transmit the operation to the next ready object in the set:

```
select {
    cases (i<N)      // instantiation of N cases, with the values of i : 0 ≤ i < N
        task[i].perform(dataPiece);
}
```

A refined collaboration diagram, corresponding to the situation described in Figure 2, is shown in Figure 3, without the indication of the use case events. The controller disconnects the incoming calls from the outgoing calls.

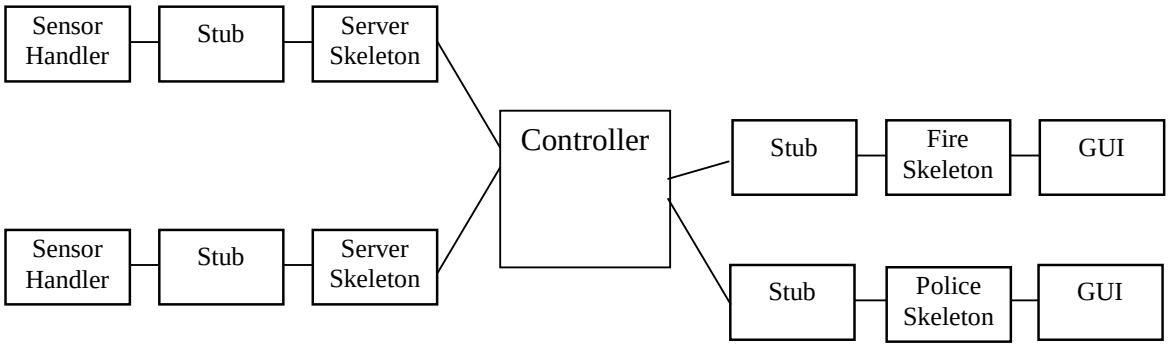

Figure 3. Architecture based on a controller.

6.2 Connecting the GUI to the Program

The use of synchronous objects to manage a GUI has been described in [7]. The source code provided below illustrates the basic principle.

```
select {
case
    buttonA.pressed();
    // perform operation triggered by button A
case
    textField.read();
    // treat the text entered in this text field
case
    buttonB.pressed();
    // perform operation triggered by button B
}
```

This code prepares three parallel calls to the GUI. Then, as soon as one of the GUI elements defined in the *select* statement has been activated by the user, the GUI element (namely its listener) accepts the corresponding method, which in turn releases the case of the *select* that contains that method. This allows the program to "read" the GUI, which is exactly the inverse of what an application based on listeners does.

The use of a controller to retrieve the GUI events has many advantages. It allows the program to perform the requested operations independently from the GUI system thread, and the GUI to handle the subsequent events (window update, data type-ahead and so on) even if the requested operation lasts a long time (such as may occur with remote operations). It also provides a simple means for implementing an automaton that defines the sequences of operations that may be requested by the end-user. The task of specifying this automaton may appear to be an annoyance within the method, since it could be avoided by the use of listeners (because listeners allow a developer to consider each operation independently of the other). However, an automaton allows an exact specification of what the end user can do on her or his terminal. It is a direct implementation and documentation of the use case that describes the behaviour of the end user actor. Finally, it can be represented directly by a state chart diagram (see below).

6.3 Active Stub

In the standard Java environment, the stubs use the threads of the callers to perform the remote call. Thus, if a call to a stub is performed from a listener, the whole GUI is blocked for the duration of the remote call. If a controller is used, as described in the previous paragraph, the GUI is released during the treatment of the events, which is sufficient in simple cases. However, the controller is still blocked, which does not allow it to handle several stubs in parallel or to get a cancellation requested by a user. Therefore, we propose to attribute a thread to each stub. Actually, each time a remote object, namely its implementation on the server side, is called by a client, a new thread is instantiated. There are then as many active threads as calling clients on the server's side, so it is legitimate do the same on the client side.

We have now devised a synchronous active stub, which possesses its own thread. Our stub defines the three following calls:

```
activeStub.remoteMethod(x);

activeStub.post_remoteMethod(x);
activeStub.ready_remoteMethod();
```

The method defined on the first line is blocking, like the method available in the basic RMI library, but it can be decomposed into the two calls defined in the last two lines. The *post* call transmits the parameters to the active stub, which carries out the message transfers in parallel with the caller. Method *ready* is accepted when the data have returned and the call is completed. These two calls can be used in relation with a *select* statement in the following way.

```
activeStub.post_remoteMethod(x);
select {
case
    activeStub.ready_remoteMethod();
case
    cancelled.pressed();
    activeStub.cancel();
    break currentLoop;
}
```

In this code, the remote call is started and its termination is awaited in parallel with the awaiting of a cancel request. If the *ready* event occurs first, the program continues normally. If the cancellation occurs first, the program executes the second case, discontinues the normal execution and takes whatever action is required in this situation. The two kinds of calls only differ by the way they are used, which may be specified in the kind of diagram defined in the following.

7 State Chart Diagrams

The same component may appear in different use cases and thus in several collaboration diagrams. The sequences of events related to this component define a state machine that can be specified by a state chart diagram. Figure 4 represents a set of collaboration diagrams and a representation of the state chart diagram attached to one of its components.

The specification by state charts is mainly useful for the controllers. If these are based on synchronous active objects, all transitions defined by the state charts are either synchronous calls or *accept* statements. The refined collaboration diagrams may contain auxiliary elements, such as the channels, the stubs, the data pools, sinks and sources, but the latter do not need to be specified by state chart diagrams, because they would be trivial.

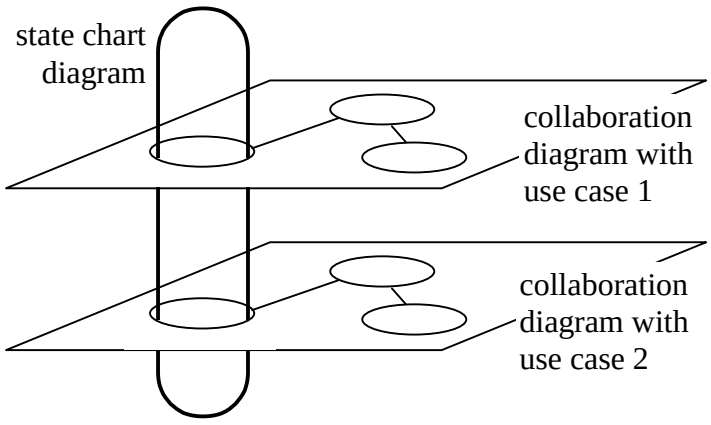

Figure 4. Intersection of the state chart and the collaboration diagrams.

7.1 Implementation of the State Chart Diagrams with Synchronous Objects

The components of a collaboration diagram are implemented either as passive or as active objects, insofar as they do not model unimplemented actors, such as humans and devices. The passive objects contain the data. They are specified by the data types and by the methods used to access and transform the data (class diagrams). The active objects take care of the application behaviour. They may be defined by state chart diagrams.

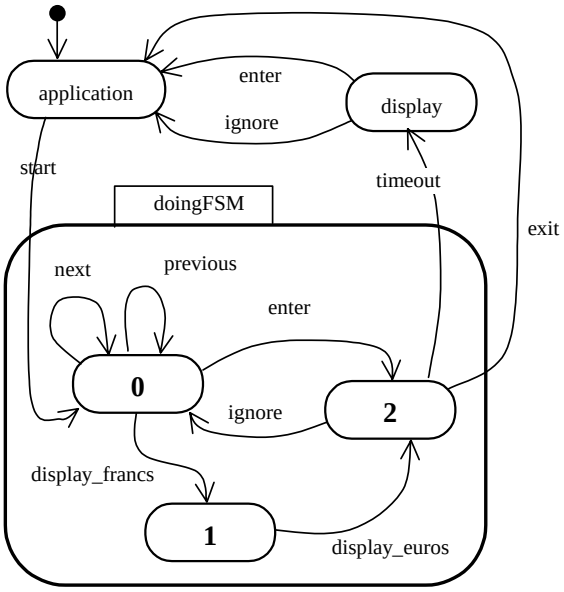

Figure 5. State chart diagrams.

Figure 5 presents an example of state chart implemented in the source code written below. This code is assumed to run in the body of a synchronous object. Method *run* implements the external state machine, while method *doingFSM* implements the composed state. Each finite state machine is placed within a *switch* statement, itself placed within an infinite *for* loop. The states are represented by the values of variable *extState* for the external machine, and by variable *state* for the internal state machine. Of course, in some cases, other statements may also be used, but jumping to another state clearly amounts to performing a *goto* statement, which is naturally not available in Java. Note that a state machine is allowed to perform a goto, since it may jump from any state to any other state. The *break label* statement in Java is the thing closest to the *goto*, but still admitted as a statement compatible with structured programming.

```
void run () {
    for (;;) {
        switch (extState) {
        case application:
            start.pressed ( );
        case doing:
            switch ( doingFSM() ) {
            case exitTrans:
               extState = display;
               break;
            case timeoutTrans:
               extState = application;
               break;
            }
```

```
            case display:
                select {
                case
                    ignore.pressed( );
                    extState = application;
                    break;
                case
                    enter.pressed ( );
                    extState = application;
                    break;
    }   }   }   }

    int doingFSM ( int state ) {
        for (;;) {
            switch (state);
            case 0:
                select {
                case
                    next.pressed ( );
                case
                    previous.pressed ( );
                case
                    francs = text.readInt ( );
                    state = 1;
                case
                    exit.pressed ( );
                    return exitTrans;
                case
                    waituntil (currentTimeMillis(10000);
                    return timeoutTrans;
                }
                break;
            case 1:
                text.setText ("€ "+francs / 1.5);
            case 2:
                select {
                case
                    enter.pressed ( );
                    state = 0;
                case
                    ignore.pressed ( );
                    state = 0;
                case
                    exit.pressed ( );
                    return exitTrans;
                case
                    waituntil (currentTimeMillis(10000);
                    return timeoutTrans;
    }   }   }   }
```

One could also resort to the state design pattern [2], but experience shows that it makes the writing of code more difficult than necessary.

The implementation of the state chart diagrams is particularly straightforward if the GUI, the stubs, and all device interfaces are realized with synchronous active objects.

7.2 Validation of the State Charts

The implementation of the state chart diagrams in the various components must naturally be capable of executing every flow of events defined in the use cases. Actually, it may happen that some new event flows, which did not appear in the specification, may be

executed. These new event flows should not impede the functioning of the program if the user happens to execute them. When the application is not too large, namely early in the development, automatic validation techniques may help to study and isolate those functionalities.

8 An Application Generator

A program generator, which we have made available on the Web, allows a developer to draw a graph that is very close to the refined collaboration diagram presented in this paper, and to generate a program directly from this graph (Figure 8). The following components may be introduced into the application:

Initialisor	An object that collects the instantiation statements of all the parts of the application.
Controller	A synchronous active object that contains the main body of the application, most often described by a state chart diagram.
Display	A graphical user interface with synchronous accesses. The elements that must appear on this display may be specified by a limited GUI generator.
Implementation/Stub	The ends of an RMI communication. The developer defines the interface of the remote object. A sub-generator then creates an active stub and a standardized remote object. The program generator also calls the *rmic* (Java RMI compiler) to generate the basic stubs and skeletons.
Database	An SQL interface to a database. The developer defines SQL tables and a set of SQL operations and inputs them to a generator. The latter creates classes that can contain the data of the records stored in the tables of the database, and a class that allows a local program respectively to introduce the data of these objects into the database, and retrieve them from the database. The interface executes the SQL commands provided by the developer with the actual data. In the case of retrieval commands, the interface extracts the data from the text returned by the SQL commands. The generator also creates a remote object that can provide a remote access to the database interface described above.
User Object	The developer can enter objects that she or he has defined elsewhere into the graphical editor of the application.

This generator delivers programs based on synchronous active objects. It compiles them and creates the scripts and auxiliary files needed to deploy and run the application.

9 An Example – The Information System of the Town of Verdun

The following presents a development example. We assume that the town of Verdun has asked us to propose an information system for the management of its infrastructures. We want initially to define and validate a basic architecture for this system, then to create a first prototype that shows what kind of functionalities such a system may offer, and finally in this way to have an evolved prototype to present to the officials responsible for the project.

9.1 Requirement elicitation

The definitive system will be used by the full set of town services (public works, police, fire department, environmental and agriculture services, and the planning service responsible for granting construction permits). They will use it to manage the town infrastructures and to coordinate the interventions on these infrastructures, sites and roadwork, to alert the response teams, and to log the alarms provided by the various kinds of infrastructures, for latter analysis.

The infrastructures include the public buildings, the roads, the water networks (drinking water, sewage water, rain water, as well as rivers and channels), gas and electricity distribution, telecommunications, and so on. The proposed system must allow a continuous remote supervision of the installations (for example, level of the water in the channels, traffic overload, fire alarms) and transmit the alarms to a central server and to the responsible services. It must also handle the documents related to these infrastructures, the planning, and the coordination of the works they must undergo.

9.2 First iteration

The first prototype will handle an alarm database (descriptions of the possible alarms, actions to undertake when they occur, alarms that have occurred with time of occurrence), and the broadcast of the alarms to the services that have logged in.

9.3 Services considered

The following services will be considered for the prototype, but at this level of detail there will not be much difference between them.

- the public works
- the police department
- the fire department

9.4 Alarms

- The alarms are generated by electronic surveillance systems and by the persons who have access to the system. The surveillance systems will be simulated by programs that generate alarms, either at random or when a button is clicked.
- The alarms must be transmitted to a central server, which must forward them to the services and to the response teams, and store them in a database.
- The system will have facilities to show which infrastructures have produced which alarms.

9.5 Basic collaboration diagram

According to the method presented at the beginning of this paper, the first step requires the design of a rough collaboration diagram defining the actors and components that will play a role in the application. This first diagram (Figure 6) will be completed in the subsequent phases. Thus, it does not need to be very elaborate. This diagram contains two identical sensors and three services. The definitive system will contain more of them, but we assume that if two or three stations may be handled, the extension to *n* stations will not be a problem.

9.6 Use cases

The actions and events that compose the use cases refer to the actors and the components by the names available in the collaboration diagrams. They have not been represented in the diagram, but to do so is trivial. Here is a list of possible scenarios for the application analysed in this section:

1. Login of a service

1.1 A service opens a connection to the server
1.2 The service sends its location and an identifier to the server
1.3 The server verifies the identifier
1.4 The server opens a connection to the service for the broadcast of alarms

2. Alarm logging and broadcast

2.1 A sensor sends an alarm to the server
2.2 The server logs the alarm in the database
2.3 The server stores the alarm in memory
2.4 The server broadcasts the alarm to the services that have requested it
2.5 Each service displays the alarms

3. Alarm acknowledgement

3.1 A user selects an alarm in a service
3.2 The service sends the alarm identifier to the server
3.3 The server cancels the alarm in memory

The developer may now choose where to continue: refine the collaboration diagrams (subsequent section) or specify the class diagrams (the latter are out of the scope of this paper).

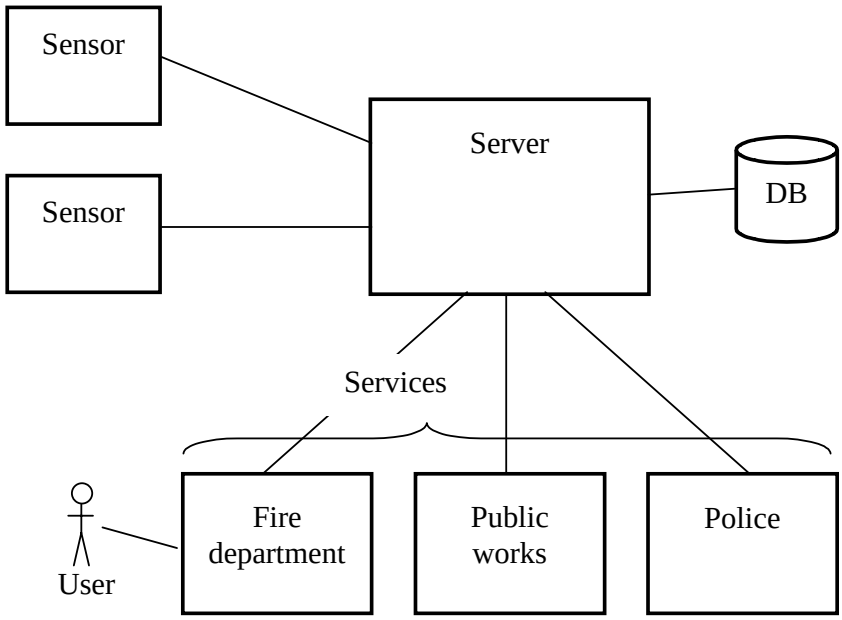

Figure 6. Collaboration diagram of the application.

9.7 Refinement of the remote connections

One of the previous paragraphs explained how to refine the connections generated by the

use case scenario 2. The extension of this analysis to the whole application introduces only one additional controller. Figure 7 shows the complete diagram, with controllers in the server and in the clients.

9.8 Controller of the service

We propose that the handling of the events produced by the GUI in the service stations should be performed by the same controller that already handles the alarms. It is obviously safer to have the same controller display the alarms and update the GUI, because it automatically keeps track of the state of the GUI, and thus does not risk trying to display the alarms when the GUI is in an invalid state.

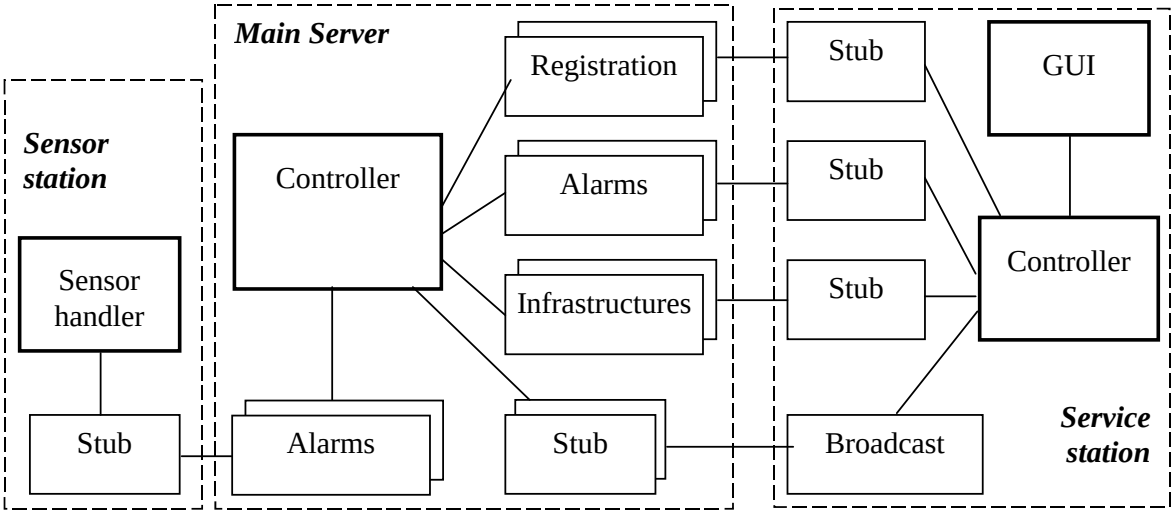

Figure 7. Refinement of the collaboration diagram.

10 Fast Prototyping of the Example

The picture in Figure 8 shows the kind of graphs that appear on the editor of the program generator. The generator creates all the .jar files, scripts and auxiliary files needed for execution.

The approach described in this paper has seen practical use by 7 groups of 5 students within the framework of a one-semester project on software engineering. Using the techniques described here, all these groups were able to present an application having one or several central servers, several services and alarm producers, distributed access to the database, and an on line HTTP access used by the standard browsers. All this was achieved without overstepping the available time. These applications were multi-threaded and handled client-server as well as symmetrical communications.

11 Conclusion

This paper has described a development method that leads directly from design to executable prototypes. It has shown that the synchronous active objects we have defined can greatly help the design of distributed interactive applications. To be fair, other object-oriented approaches derived from CSP could provide similar advantages.

The approach is systematic enough to allow a program generator to produce fast prototypes and the basic architectures on which industrial software applications may be built. In fact, the program generator is itself based on these objects. The program is very

flexible and readable. For example, the approach makes it easy to give the ability (to the end-user of the generator) to interleave the handling of the various components. For example, one could begin the creation of a component, create another component needed by the first one, and then terminate the specification of the first one.

This approach has successfully been used by a substantial set of students.

12 Acknowledgements

The program generator was developed in large part by Riadh Kortebi. My thanks also go to Mark Madsen who carefully revised the paper.

References

[1] J.A. Whittacker and S. Atkin, Software Engineering is not Enough, IEEE Software, August 2002, pp 108-115.
[2] E. Gamma et al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[3] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, 2000.
[4] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall.
[5] J. Rumbaugh, I. Jscobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley.
[6] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley.
[7] C. Petitpierre, Synchronous Active Objects Introduce CSP's Primitive in Java, CPA'02, Reading, September 2002.
[8] C. Petitpierre, A. Eliens, Active Objects Provide Robust Event-Driven Applications, SERP'02, Las Vegas, June 2002.
[9] C. Petitpierre, Synchronous C++, a Language for Interactive Applications, IEEE Computer, September 1998, pp 65-72.
[10] 1999 - C. Petitpierre, Implementing Protocols with Synchronous Objects, in ed. D. Avresky, Dependable Network Computing, Kluwer, November 1999, pp. 109-140.
[11] http://ltiwww.epfl.ch/sJava

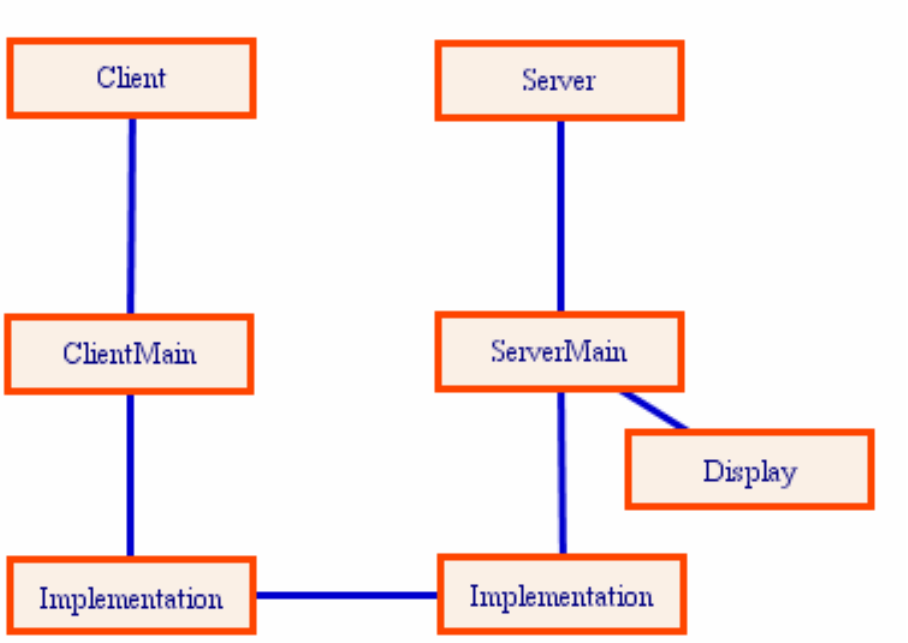

Figure 8. A picture of the kind of graphs handled by the graphical editor.