

Designing Concurrent and Distrib Control System

After a decade of real-world experiences,
the applicability of the G++ pattern language
is well documented.

THE object-oriented paradigm claims to promote reuse, reduce development time, and improve software quality. Originally these properties were considered at the level of components, and were achieved through the use of libraries of reusable classes organized in hierarchies of inheritance. However, attention is now being shifted from individual components to the whole architecture by giving more importance to the fundamental role that patterns of relationships between the elements have in any design.

**Amund Aarsten,
Davide Brugali, and
Giuseppe Menga**

uted ms

The approach described here is based on the concepts of patterns and pattern language:

- **Pattern**—A cluster of cooperating objects linked by certain relationships and the rules that express a link between a context, the design problem, and its solution—become the module of a unitary architectural model.
- **Pattern Language**—The sequential and organic structuring of patterns for a specific application domain—becomes the method for the development process.

This article has two objectives: to interpret the pattern language concept in the domain of software engineering, and to show its use in software architecture design. We do this by summarizing the G++ pattern language [1], the result of 10 years of experience in developing concurrent and distributed control systems, originally in the field of computer-integrated manufacturing (CIM), and by using it to describe a recent project involving cooperative autonomous mobile robots [5].

The G++ Pattern Language

The G++ pattern language addresses the problem of designing large software control systems, made up of layers of concurrent control modules installed on a distributed computer architecture,

by following an evolutionary development process that derives the final implementation from a prototype of the logical design. Patterns in this language are structured, following Alexander [2], in a tree as shown in Figure 1; each circle denotes a pattern and hence a design decision point, and the arcs, which link patterns, represent the temporal sequence of decisions. The development process results by following the graph from the root to the leaves, as outlined here.

The language is derived from the “divide and conquer” paradigm [6] and deals with issues such as the organization of control modules in a hierarchy of inclusion and use relationships, and the visibility requirements between the control modules, concurrency, and distribution.

In addition, in large distributed control architectures, a prototyping phase is mandatory before attempting to integrate modules of control with the physical process. The adoption of an evolutionary design approach, which moves smoothly from a prototype to the physical implementation, is then the correct solution; the pattern language takes this into account.

The Application Domain

As an example of the theory outlined in this article, we consider the problem of developing a control architecture for cooperative autonomous mobile robots.

The robots are said to belong to an agency,¹ which is able to fulfill a variety of tasks specified by the “Mission Supervisor.” An “Agency Manager” manages the robots and provides the Mission Supervisor with a centralized point of access to the agency. The “Agency Manager” decomposes tasks into sub-tasks and activates those robots that may be needed to execute each sub-task. It also has the responsibility of monitoring the results in order to verify their correctness.

Mobile robots usually operate in unstructured environments with little a priori information, where robustness in front of significant variability of the operational condition is necessary. Therefore, mobile robots must have a high degree of autonomy, being able to employ different strategies and exhibit dif-

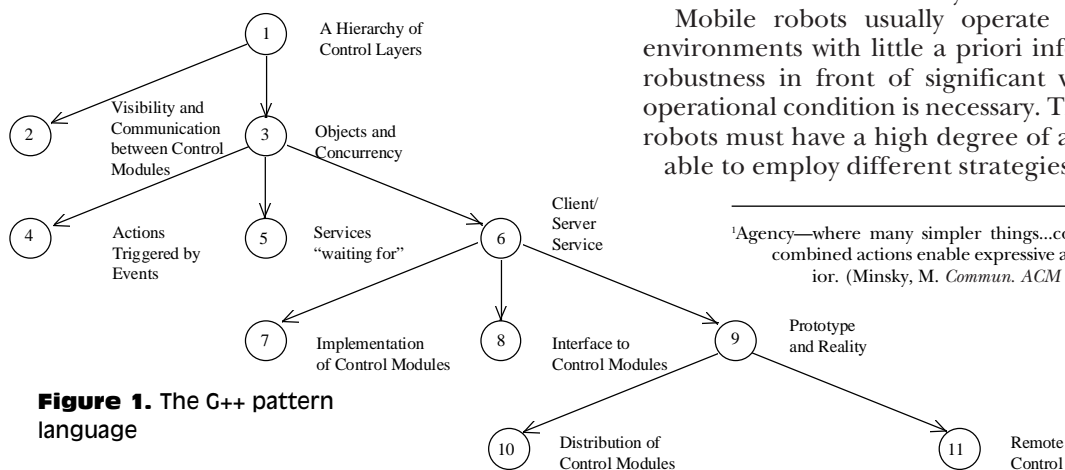


Figure 1. The G++ pattern language

¹Agency—where many simpler things...come together and their combined actions enable expressive and knowledgeable behavior. (Minsky, M. *Commun. ACM* 37, 7 (July 1994), 54.)

ferent behaviors, and adapting themselves to different environmental conditions.

In an agency, autonomy also implies that the robots may collaborate and, more importantly, initiate collaboration on their own initiative. Collaboration requires the robots of the agency to have a common architecture and a common medium, in order to have awareness of each other and to communicate.

As an example, we consider an agency of robots that have to explore an indoor environment, as described in [5]. Each robot has its own sensors and is able to build a description of its surroundings using sensor information. Every sensor has specific capabilities and is able to detect specific kinds of objects in the environment. Therefore, each robot's description of the environment may be incomplete. In order to adequately fulfill its task, each robot may take advantage of the information represented in the maps of the other robots, for instance in planning a collision-free path to reach its destination in the environment.

Figure 2 shows the analysis model of the robot agency: a *MissionSupervisor* assigns a task to the *AgencyManager*, which manages a set of robots. Each robot maintains a *Map* of the environment and is able to collaborate with the other robots of the agency. Furthermore, each robot has *Sensors* and a *MobilePlatform*.

The remainder of this article presents selected patterns from the G++ pattern language, showing how each pattern was applied in the robot agency project.

Communication Between Objects

Any complex system, by its very nature, follows a development process where an important problem is the integration of the system's components as they are finished. Two situations are possible: building modules that will be plugged into a pre-existing architecture, or building an architecture that must integrate pre-existing control modules. In the former case, modules are built having *visibility* [4] of the environment in which they will operate, while in the latter situation this does not happen. The following pattern offers a guideline to solve communication problems between modules taking into account the two different situations of visibility.

Pattern: Visibility and Communication between Control Modules

Each control module performs services, requests services from

other modules or signals events so as to inform other modules when its state changes.

Context

A complex architecture is decomposed into modules that communicate with each other in order to cooperate, and the way the communication is established depends on their visibility of each other. Control modules in communicating can be imperative, as in the case of a command issued by one module to the other, or they can be reactive, as in the case of event monitoring.

Problem

Visibility and communication mechanisms between control modules must be established for both imperative and reactive communication.

Solution

The two situations have a direct relationship with the two basic mechanisms of communication between objects, here called *Caller/Provider* and *Broadcaster/Listeners*. The *Caller/Provider* mechanism is involved when an object invokes another object method and is deeply rooted in object-oriented programming, so it will not be discussed further. The *Broadcaster/Listener* mechanism is achieved by giving all the objects of the framework the capability of broadcasting and listening to events. The term *event* is used here to indicate

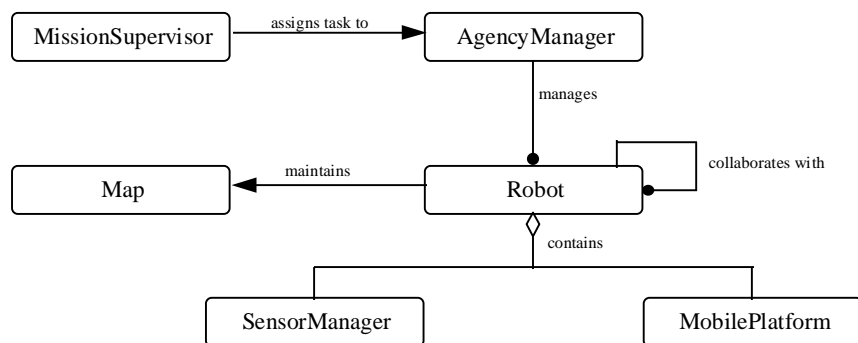


Figure 2. Mobile robot agency analysis

a broadcast message issued at a certain instant of time by an object, identified by a symbolic name and with some data associated with it. There is no redundancy in these two mechanisms as they imply different visibility relationships between the players involved in the communication, and the choice between the two has important consequences on the reusability of the design. The Broadcaster/Listener communication mechanism is widely used in object-oriented pro-

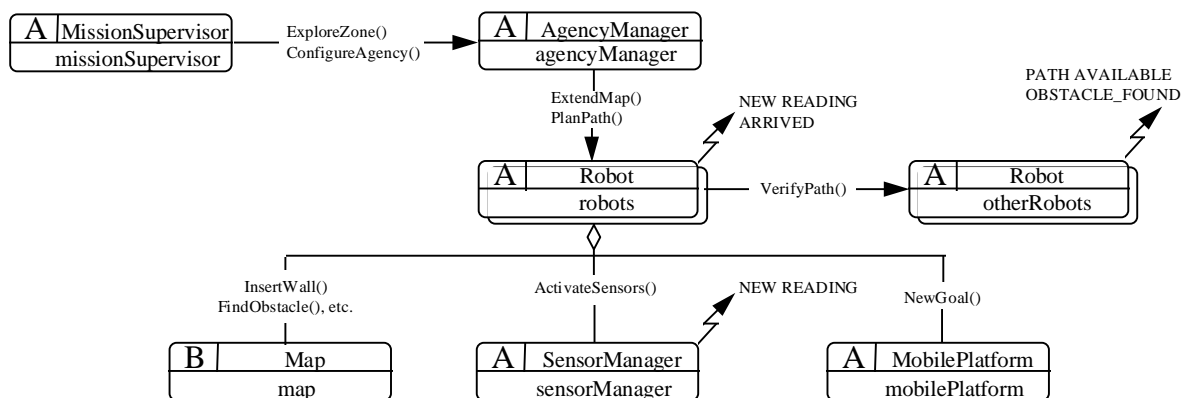


Figure 3. Examples of method calls and events in the robot agency

gramming: for example it is called *Observer* in [8] and it corresponds to the *changed: update* mechanism in Smalltalk.

Example

In Figure 3, high-level control modules have visibility of lower-level control modules and communicate with them using the caller/provider mechanism: the *MissionSupervisor* configures the agency or assigns a task to it, calling the corresponding methods of the *AgencyManager*; similarly, the *AgencyManager* specifies sub-tasks for the robots, which initialize the *SensorManager* and assign new goals to the *MobilePlatform*.

At the opposite, low-level control modules broadcast events, which are monitored by higher level control modules. Using this mechanism the *SensorManager* informs other modules when new readings are available or the *MobilePlatform* that the robots have reached the goal.

The visibility relationships so far explained follow the classical hierarchy that may be recognized in application domains such as CIM, but in the robot architecture presented here, the hierarchy is broken: two different robots (represented by control modules at the same level) are entailed to communicate directly using both mechanisms. In fact, a robot asks for collaboration from other robots using the Caller/Provider mechanism and waits for the answer while monitoring events raised by the partners.

Levels of Granularity in Concurrent Activities

In real systems concurrency occurs at different levels of granularity. These are usually classified in fine, medium, and large grain as in [7]; control modules manage different pools of shared resources and offer

different kinds of concurrent operations to their clients. The following patterns offer guidelines to recognize different granularity of concurrency in a complex system and provide features to conveniently model the concurrent activities according to their level of granularity.

Pattern: Objects and Concurrency

Control modules in the system perform services concurrently and concurrency assumes different scales of granularity.

Context

In a complex system, some activities are made up of simple actions without a predefined order (also called a weak cohesion); others are operations comprised of an ordered sequence of actions, having a stronger cohesion, while yet another group of activities is represented by control modules operating in parallel, such as the individual robots. These three examples of the different levels of granularity were defined as *fine*, *medium*, and *large* in the previous paragraph.

Problem

It is desirable to adopt the correct model for representing concurrency at the different granularities.

Solution

For each grain of concurrency, a different type of object is needed to support the activities. The relationship between objects and processes supporting their activities induces the following, generally accepted, classifications:

- **Passive Objects** depend on client threads of control for the execution of their functions and can be subdivided into *sequential* or *blocking*. The



behavioral semantics of sequential objects are guaranteed only in the presence of a single thread of control. Blocking objects maintain their semantics when more than one thread accesses them, and offer the *wait* primitive that is the indispensable element to block client threads allowing them to interact.

- Threads of control—*ThreadOfControl* is the class whose instances are independent processes. They simply lend the objects that possess them their data structure (context) and their capability to suspend the flow of execution (“*wait*”) of the services they support.
- Active Objects differ from passive ones because they possess, create, and internally manage one or more independent threads of control that govern the execution of their services. They are the natural candidates to represent modules of controls.

Represent fine-grain activities as event-driven, atomic actions, which access sequential objects. Make

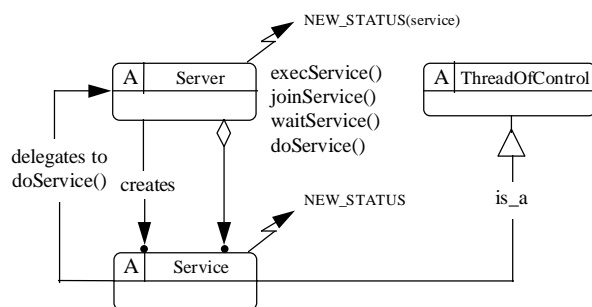


Figure 4. Client/Server/Service

medium-grain activities run in the context given by a thread of control, which can share blocking objects. Large-grain activities are implemented as separate active objects. These concepts are explained later in this article.

Example

The highest level of concurrency may be recognized among the robots belonging to the agency; since they are autonomous, they are conveniently modeled as active objects. Other active objects are indicated with an “A” in Figure 3.

A medium grain of concurrency occurs between the different activities of the *SensorManager* when several sensor devices are available. In fact, it may be modeled as an active object providing several functionalities, depending on the capabilities of the sensors.

The robot’s different strategies and behaviors are modeled as fine-grain concurrency: each sensor event causes parallel transitions along several paths in the

robot’s decision graph.

The *MotionManager* has the responsibility to drive the robot to the specified goal, to react to an unexpected change of goal while traveling and to provide an estimate of the robot real displacement in case of motion errors. Once again these functions are modeled as medium-grain concurrent activities.

Pattern: Client/Server/Service

Modules of control are objects, which must be able to offer multiple concurrent services and have the capability to encapsulate the resources and the services that manipulate them.

Context

Modules of control deal with different topics: concurrency, abstraction, and encapsulation.

Problem

The first requirement of a control module is to offer concurrency—it should be able to offer more than a single service in every moment to its clients.

Another requirement is a common representation of the control modules at the different levels of the hierarchy for standardization purposes, so as to enforce reusability and facilitate the task of distributing the application.

Finally, modules of control should have the capability to encapsulate the resources and the services that manipulate them.

Solution

These requirements are satisfied by the Client/Server/Service model proposed in this pattern and derived by the programming paradigm of the language “Synchronizing Resources SR” [3]. Two classes are needed to support it: the *Service* and the *Server* (see Figure 4).

The Service

The *Service*, derived from *ThreadOfControl*, encapsulates a thread of execution. It maintains a symbolic value corresponding to its execution state, and broadcasts events announcing the new state name every time a state transition occurs. In addition, it can have internal, thread-local data (sequential objects).

The Server

The *Server* is the abstract class that defines the common characteristics of all active objects, and has to be redefined in order to build concrete servers. It offers three methods for creating and executing services: *execService()*, *waitService()*, and *joinService()*, which represent asynchronous, synchronous, and deferred-synchronous service requests, respectively.

Whenever one of these methods is called, a new *Service* is created and set up to execute a private

method of the server. Methods that are executed by the services in this way are called *active methods*.

Server also subscribes to state transition events generated by its services and relays them externally. In this way, clients can monitor the execution state of asynchronous and deferred-synchronous requests without having visibility of, or even knowledge of, the *Services*.

The representation of all service behaviors in the framework as methods of the classes derived by *Server* is an alternative design choice to the command pattern of [8]. Command “objectifies” services, by modeling them as different “behavior objects”. Instead, *Client/Server/Service* uses a pattern which we would call Delegated Environment: service behaviors are methods of the object which offers them, but they can be executed concurrently thanks to the context delegated from instances of the *Service* class. This solution has the double advantage of offering a protected execution environment for each service instance, while at the same time allowing all services inside a server to share the server resources without visibility problems.

Example

Each *Robot* is modeled as an active object, performing simultaneously different activities:

- Monitoring the execution of its own task.
- Cooperating with other robots.
- Managing the mobile platform.

The *Robot* is therefore structured as a *Server* offering different concurrent *Services*. For instance, when the robot asks other robots to verify its planned path according to their local information, the collaborator *Service* waits for reply events (or a time-out). The other robot activities, meanwhile, proceed.

Evolution from Simulation to Reality

Concurrency and distribution add considerably to the complexity of the task of developing large software systems. Simulating some parts of the system can make testing and debugging (some of the most difficult aspects) much easier. In some application domains, simulation becomes a necessity.

Simulation is often employed when the physical architecture of the final system is distributed. For the purposes of this article, there are two basic kinds of distributed systems. One is the distribution of server objects; the other is objects with remote control. These two situations are covered by the patterns *Distribution of Control Modules* and *Remote Control*, respectively.

Pattern: From Prototype to Reality

Any complex application requires prototyping and simulation of the different elements that have to be integrated before

an implementation is derived.

Context

Simulation is typically used in two different situations:

- When part of the application’s functionality is in time-consuming computations, or when the full functionality is not ready yet.
- When the final system is a distributed system. In these cases, the simulation is typically done locally on a single computer; the simulation must then be evolved to a distributed program.

When parts of the system are simulated, a seamless evolution to the real system is fundamental; also, the changes involved must be of a local nature. If we make changes to the non-simulated components, we can no longer trust the results obtained by the simulation, and its whole value would be lost.

Problem

How can we ensure a seamless evolution from the simulation to the final implementation?

Solution

For any object that needs to be simulated, maintain two coexisting versions:

- The *simulated* or *emulated time prototype*, which simulates or emulates the object behavior.
- The *reality object*, which embeds the physical reality.

The reality object could interface to a hardware device driver, encapsulate an external functionality like a relational database, or act as a surrogate or proxy for a remote object as described in the pattern *Distribution of Control Modules*.

When going from simulation to reality, replace the prototype object with the object (e.g., device driver) that interfaces to the reality. Consistency in this transition can be obtained by inheriting the two implementations from a common base class which defines their interface. Alternatively, by using separate interface and implementation objects, the new implementation object can be substituted in the interface object’s use relationship.

Example

The robot control project benefits particularly from simulation, since the major part of the implementation of the control architecture resides with the robots that not only are distributed but are also moving around.

Pattern: Distribution of Control Modules

Control modules can reside on remote computers or peripheral devices, interconnected through a common communica-



tion network. These modules define the physical architecture that must be realized by the final distributed system.

Context

Some of the objects in the simulation will be moved to remote nodes (i.e., the final system involves distributing first-class objects).

Problem

When going from a nondistributed simulation to the distributed reality, the objects that are moved to remote nodes are no longer part of the original program; they must become independent programs in their own right. The rest of the system should not be affected by moving some objects to a remote node.

encapsulate the communication with the physical robots, as shown in Figure 5.

Pattern: Remote Control

Reality can be a remote device, and possibly part of a reactive, event-driven system.

Context

Often, not all the components of the final distributed architecture are first-class objects. For instance, intelligent peripheral devices such as the *MobilePlatform* are not accessed by method calls. These components can usually be viewed as part of a reactive, event-driven system.

Event-driven systems, for the purposes of this article, maintain a state and interact with the outside world through events. The system state can usually be read, either directly or indirectly, through events carrying information.

A common char-

acteristic of event-driven systems is the presence of active data [10], variables that generate events when their value changes. This is useful for, among other things, implementing constraints between related values.

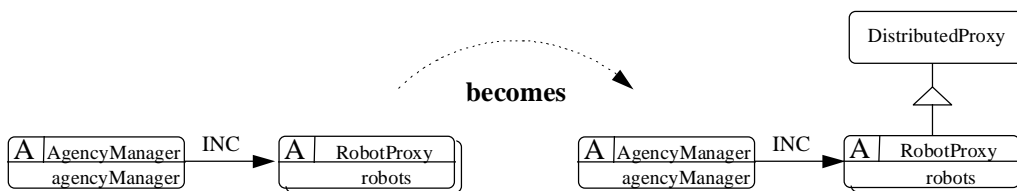


Figure 5. Robot distribution

Moving objects to remote nodes should be relatively easy in order to not prevent the system from changing with the physical architecture; that is, we want to exploit an evolutionary approach.

Solution

Divide the distributed objects into two parts: an interface proxy and the implementation. In the original system, replace the objects that have been moved to other nodes with the proxy objects. The proxies have the same interface as the object they replace, and forward each request over the network or communication link to the remote object they represent. The functionality of the proxy object corresponds to that of the CORBA stub [11]. In the program on the remote node where the implementation object resides, support must be added for listening for requests coming over the communication network and forwarding them to the correct object. This functionality corresponds to that of the server skeleton and the object adapter of the OMG CORBA reference model [11].

Example

In the simulation, the *AgencyManager* has a collection of *Robot* objects. In the final system, this collection is replaced by a collection of *RobotProxy* objects, which

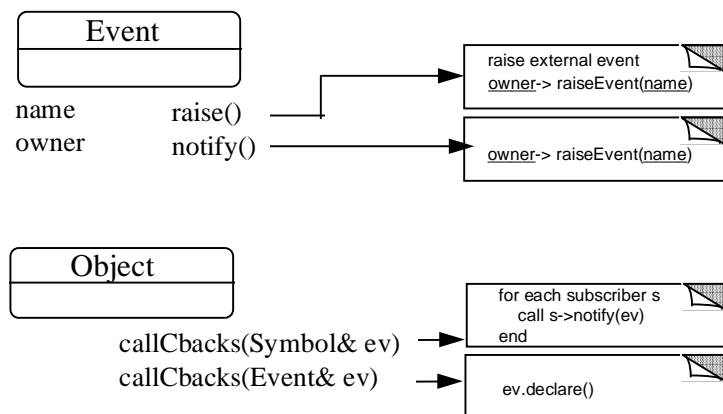


Figure 6. External/logical event bridge

Problem

In the simulation, the control of a peripheral device uses variables and events. In the real system, the device does not have variables in the program's address space.

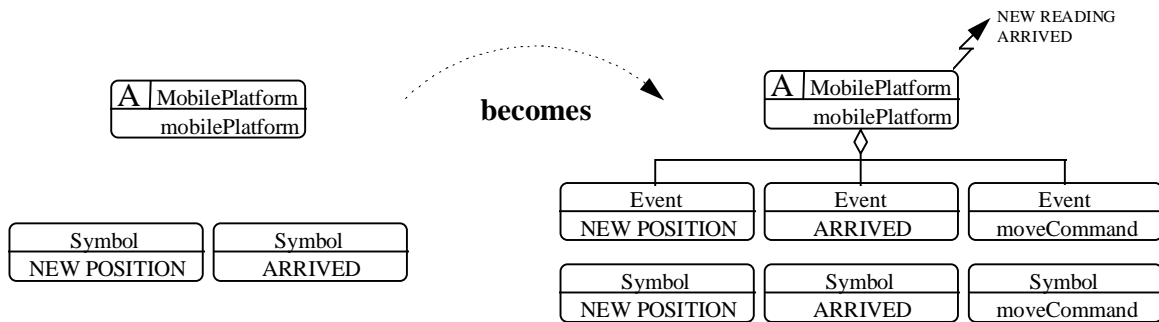


Figure 7. MobilePlatform prototype and reality

Furthermore, the G++ logical events are just a communication mechanism; they are not first-class objects as the events in the real system.

Also, program objects that subscribe to the simulation object's logical events should continue to receive those logical events in the final system. This means a bridge is needed between the logical and the external events.

Solution

Virtualize the data used by the component simulation to access the peripheral state via a proxy class. Make a class Event to encapsulate external events as shown in Figure 6. Event knows the logical event it corresponds to (name) and the program object that should raise the logical event when the external event occurs.

Overload the method used to broadcast logical events (in G++ called callbacks) with a new method taking an Event as a parameter, as shown in the lower part of Figure 6. For each external event that must be translated to and from logical events, declare an Event instance with the same identifier as the desired logical event in the scope of the class in question, and associate the Event with the logical event by setting the Event's name. Calls to callbacks will bind to the new method (since the global event identifier is hidden by the Event object declared in class scope,) ensuring that the logical event is relayed as an external event.

Example

The robot's interaction with the sensors and the mobile platform is an example of remote control. We model the sensor measurements as active data. The mobile platform is controlled by events, and the move progress is also reported back through events.

In the real robot, the object that represented the mobile platform in the simulation is replaced by one that wraps the needed events for communication with the external mobile platform (Figure 7).

Comparison with the "Design Pattern Catalog"


In order to compare the patterns presented here with

those described in the literature [8], they are subdivided and classified as elemental, basic design, and domain dependent. Elemental patterns determine the language (elements of the architecture) in which the whole application has to be written. Of the patterns presented here, Visibility and Communication between Control Modules and Objects and Concurrency fall into this category. Basic design patterns, similar to those discussed in [8] and with the same level of abstraction, propose intermediary design building blocks such as Client/Server/Service and Prototype and Reality, which by their nature are fairly application independent. Distribution of Control Modules and Remote Control are domain dependent and are similar in concepts to [9]. They provide guidelines on using the framework's classes to solve specific problems. They obviously exploit in turn basic design patterns for their implementation.

Conclusion

We have presented a summary of the G++ pattern language for concurrent and distributed systems, the result of experience acquired from real-world projects over the last 10 years.

While there is currently a great interest in individual patterns, we believe that pattern languages have not yet received the attention they deserve, and that they will have an important role in many domains.

The existence of design patterns is often very valuable for solving isolated design problems. In our experience, a domain-specific pattern language does the same to a whole development project. Typically, engineers without experience with object-oriented programming or concurrent/distributed systems are able to develop systems of significant complexity after a period of three months. 

References

1. Aarsten, A. Elia, G. and Menga, G. G++: A pattern language for computer integrated manufacturing. In *Pattern Languages of Program Design*. Addison-Wesley, NY, 1995.
2. Alexander C. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.
3. Andrews, G.R. An overview of the SR language and implementation. *ACM Trans. Programming Languages and Systems* 10, 1 (1988), 51–86.



4. Booch, G. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.
5. Borghi, G. and Brugali, D. Autonomous map learning for a multi-sensor mobile robot using dikiometric representation and negotiation mechanism. In *Proceedings of ICAR'95* (Sant Feliu de Guixols, Spain, Sept. 20–22, 1995), pp. 521–528.
6. Brooks, R.A. A Robust layered control system for a mobile robot. *IEEE J. Robotics and Automation*, RA-2, 1 (Mar. 1986).
7. Chin, R.S. and Chanson, S.T. Distributed object-based programming system. *ACM Comput. Surveys* 23 (Mar. 1991), 91–124.
8. E., Helm, R., Johnson, R., Villisides, J. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, NY, 1994.
9. Johnson, R. Documenting frameworks using patterns. In *Proceedings of OOPSLA '92* (Vancouver, B.C., Oct. 1992).
10. Minoura, T., Pargaonkar, S., Rehfuess, K. Structural active object systems for simulation. In *Proceedings of OOPSLA '93* (Washington D.C., Oct. 1993).
11. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. September 1992.

AMUND AARSTEN is a Ph.D. candidate at the Politecnico di Torino in Torino, Italy; email: amund@polito.it

DAVIDE BRUGALI is a Ph.D. candidate at the Politecnico di Torino in Torino, Italy; email: brugali@polito.it

GIUSEPPE MENGA is a professor and chair of Automatic Controls at the Politecnico di Torino in Torino, Italy; email: menga@polito.it

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/96/1000 \$3.50

Aspects of Software Adaptability

Mohamed Fayad and Marshall P. Cline

IN today's rapidly changing business environment, adaptability is a critical weapon for survival. Businesses must be adaptable in order to meet increasingly narrow market windows. This need for adaptability at the business level has changed the focus in many businesses from efficiency to opportunity, from reducing costs to generating revenue. For example, an efficient but inflexible system might reduce costs, but might also make it impossible for the business to engage in a new revenue-generating opportunity. Those businesses that desire this adaptability are redoubling their efforts to make their underlying people- and software-systems adaptable, particularly for those systems that could inhibit business-level adaptability.

Because of these forces, software developers need to deal with change like never before. This need for adaptable software systems is driving the move toward object-oriented (OO) technology in many circles. Certainly one of the promises of OO has been its ability to make software more adaptable.

Using OO, however, does not guarantee that the resulting software will be adaptable. Adaptability must be explicitly engineered into the software, even with OO. Furthermore, adaptability is not a generic quality of the software system as a whole: Software systems are adaptable in specific, designated ways, if at all. Therefore the adaptability must not only be explicitly engineered into the software, it must be engineered into the software in places where it will do the most good to the business.

It is no longer acceptable if a software system is correct and solves the problem for which it was designed. Ideally the system will be able to grow and change to solve slightly different problems over time. This corresponds to the three stages of the evolution of software development: Build the right thing, build the thing right, and support the next thing.

Build the right thing corresponds to validation. The requirements must be accurately understood. There continues to be a great deal of effort spent trying to figure out what the

right thing really is. In today's world, however, the definition of "right" changes between the time when the perceived sponsor says what they want and the time the software is delivered.

Because the notion of "the right thing" is a moving target, the only way to satisfy the sponsor's desires is to make the software adaptable. Only if the software is adaptable will it be reasonable to change the thing that gets built (which will be "wrong" by then) into the thing that the sponsor wants at that time.

Build the thing right corresponds to verification, correctness, and defect rate. Although everyone has always been concerned with reliability and faithful translation of the requirements into a solution, the market was not always willing to pay a lot of extra money for reliability. Although there still is not a general willingness to pay extra for low defect rates, there seems to be an increasing intolerance for defects. In cases where standards have created a level playing field, customers today appear to have less brand loyalty.