G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing*

Amund Aarsten, Gabriele Elia, Giuseppe Menga

Dept. of Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi n. 24
10129 Torino - Italy
ph. +39-11-564-(7012), fax (7099)
Email: {amund,gelia,menga}@polito.it

Abstract

This paper presents G++, an object oriented framework of reusable classes and a methodology for developing of distributed information systems.

G++ is presented as a set of "design patterns" structured to form a "pattern language".

Following the intuitions of the architect Christopher Alexander, a design has been considered formed by "*patterns*" of relationships between elements, where each pattern is by itself a rule to enable a design choice from a problem requirement. Indeed, these patterns always come from a language: a "*pattern language*", with which a certain class of problems can be approached.

These ideas have been transferred to the field of object oriented design, where elements are objects and "*patterns*" indicate clusters of cooperating objects linked by certain relationships that can repeatedly be found in the solution of a class of problems; this allows us to call G++ a pattern language, that is, patterns are ordered and structured in such a way to help the designer in his/her decision and to manage at the proper moment the different issues which he/she encounters.

The objective of this approach is to raise reusability in object oriented design from the component to the architectural level, and to offer a conceptual reference model for concurrent and distributed architectures.

Keywords:

Object Oriented Design, Patterns in Programming Languages, Concurrent and Distributed Computing, Prototyping.

*This work has been supported by CNR under grant n. 92.01602.PF69 *Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo*, grant n. 91.01940.PF67 *Progetto Finalizzato Robotica* and by the Chamber of Commerce of Turin - Standards in CIM

1 Introduction

The Object Oriented (OO) paradigm claims to promote reuse. Originally reusability was intended at the level of system components, and component reuse was achieved through libraries of reusable classes organized in hierarchies of inheritance: what is now called a *framework of classes* [26]. However, attention is now being shifted from component to architecture reuse by giving more importance to the fundamental role that patterns of relationships between the elements have in any design. It is general opinion, in fact, that designers have always been driven by personal "*patterns*" when making choices. The architect Christopher Alexander [1, 2] has pursued this idea further.

Alexander suggests that each "design" pattern ...*is a three–part rule, which expresses a relation between a certain context, a design problem and a solution.* Moreover, all these rules – or patterns – are part of a larger system, i.e. a language which embodies the experience gained by generations of designers in solving a certain class of problems. However, if the *pattern language* has to be shared, and become part of the collective knowledge in a certain application domain, it has to be structured, and the structure of the language is created by the network of connections among individual patterns.

In an OO context, elements are objects and a "*pattern*" indicates a cluster of cooperating objects linked by certain relationships repeatedly found in the solution of a problem.

Alexander's definition of a pattern fits perfectly into an OO software design where each pattern describes the translation of one aspect of the analysis into the design, elucidating, at the same time, the use of the framework of classes [15] which are the elements of the architecture. However, the whole set of patterns, together with their structuring principles, becomes a high level language and a design method which accompanies the software life cycle from the analysis to the final implementation.

This paper extends previous works and presents, within the OO context, an attempt to bring the concept of a pattern language closer to Alexander's idea of a whole design method in a specific application domain.

1.1 The application domain

The application domain targeted here is the one of large client-server control architectures. In this sense a "*pattern language*", as defined by Alexander, is the best interpretation that can be offered in such a domain to the idea of a standard architecture.

The Computer Integrated Manufacturing (CIM) domain by its very nature, as a point of aggregation of heterogeneous elements, is a challenging environment in which to define architectural patterns and frameworks of classes and it will be used as a case study. The examples described in the patterns of section 2 are, in fact, taken from the design of a Flexible Manufacturing System (FMS), i.e. a factory shop made up of production cells composed of machine tools, stores and automatic transportation systems, which concurrently manufactures lots[1] of different types of pieces.

However, as the questions raised here related to managing the complexity and the concurrent and distributed nature of the systems are very similar to those of other application domains, these ideas can also be transferred into other fields, such as business administration or telecommunications.

A pattern language for a domain leads naturally to a supporting CASE environment. We have implemented such an environment for G++, merging a framework of C++ classes and tools to automate the development. Examples of application of it can be found in [19].

1.2 Paper organization

The paper is organized as follows: section 2 describes the proposed patterns and their structure through the presentation of examples, section 3 compares the interpretation of patterns presented in this paper with other related results.

2 The Pattern Language

The problem addressed by the pattern language presented here is that of the *design* of large software systems made up of layers of *concurrent* control modules, installed on a *distributed* computer architecture. The chosen

[1] A lot is an administrative entity, representing a group of identical pieces manufactured together.

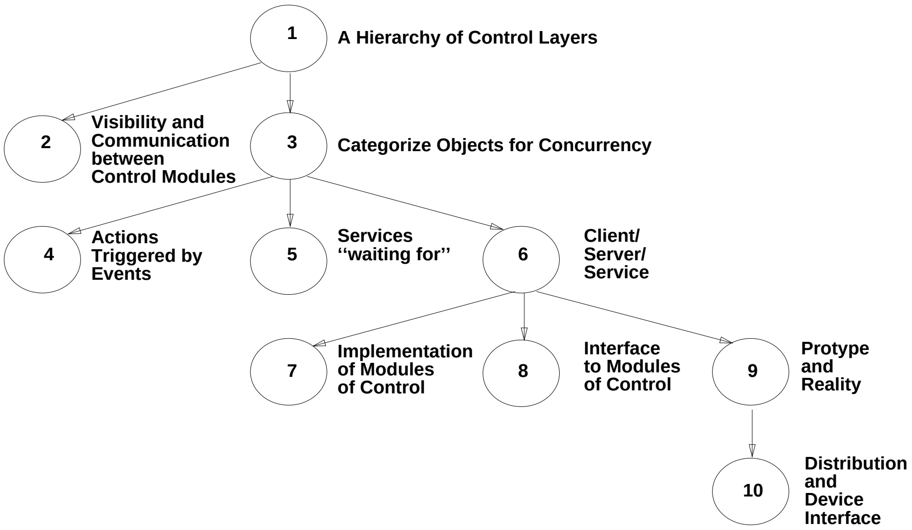

Figure 1: The pattern language for designing distributed application.

development process is *evolutionary*, and it is obtained by exploiting transformations which first map the analysis into a *logical design* to be locally prototyped, and then map the logical design into a *physical design* [7].

Patterns in this language are structured, following Alexander, as in the tree shown in figure 1; each oval denotes a pattern with arcs representing the relationship between larger and smaller patterns. Each pattern then depends both on the smaller patterns it contains, and on the larger pattern within which is contained, and it is the network of these connections between patterns which creates the language.

The template used to describe an OO pattern in inspired by the proposal of in literature [15, 10, 5, 16], by the discussions on the "pattern mailing list" and by the works of Alexander himself and has the following fields:

the pattern name;

the context in which we have to discuss the pattern;

the problems which the pattern addresses, related with the context;

the solution offered by pattern in terms of the framework of classes, as well as a graphical representation; we tried to show the "balance of forces" in the pattern;

an example of the pattern in the case study (e.g. CIM);

the definition and design rules of the pattern, trying to defines rules for an operative use of the pattern;

references to lower level patterns of the language.

As patterns are mostly aggregates of classes, they can be described by means of entity-relationship diagrams, where a rectangle with rounded vertices represents a class (upper identifiers) or an instance (lower identifier); a character on the left side of the label indicates whether the object is active, sequential, blocking and so on. Pure data attributes, methods and names of events issued are represented as a comment on the side of the icon. Relationships are represented using the OMT notation to which have been added the links of use `USE` in the logical design space and of `COM`, indicating interconnection through a communication network involving a stub and a remote context, in the physical design space. The links implying use relationship may also show the method names of the messages flowing through them. The OMT generalization/specialization relationship of the analysis, applied to the design, indicates *implementation inheritance*. If it is necessary to specify *interface inheritance*, i.e. also sub–typing, an `IS_A` label is added to the link.

Pattern 1 : A Hierarchy of Control Layers

Any complex system organizes its functionalities within an architecture of hierarchical layers of control modules.

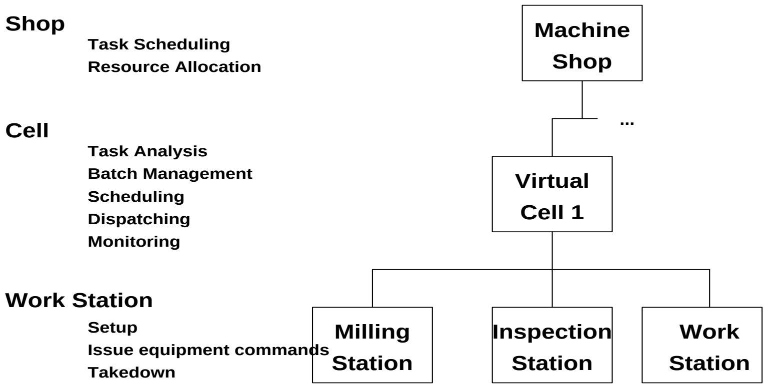

Figure 2: The USA-NBS Reference Model for Manufacturing.

Context

Large, distributed engineering systems, characterized by multiple concurrent activities, are difficult to design and manage; moreover, it is important to be able to reuse some parts in different systems.

Solution

In order to cope with complexity, it is worth to organize functionalities in hierarchical decentralized control architectures. Each layer of the hierarchy encompasses *modules of control* with similar classes of functionalities. A module of control is an entity characterized by the presence of:

- autonomous decision making capabilities;
- series of available services which are offered to the outside (e.g. in manufacturing: planning products, executing piece production, executing a machine operation, storing–moving pieces);
- a pool of controlled resources.

Modules of control are objects that should be organized using two basic structuring principles of inclusion (encapsulation) and use.

Any organization defines its own control modules which are shaped according to its internal principles, which can be functional, logical or simply dictated by tradition. In the manufacturing field, they are traditionally indicated either as business or technical processes, and are arranged as in figure 2 according to [18]. Modules should encompass functionalities in such a way that can be reused for different projects. A pattern language for the definition of modules of control will be highly application specific and it is not discussed here.

Example: The FMS

Figure 3 describes the analysis of a typical FMS for producing mechanical components. This is a production shop composed of two cells – one for machining and the second for assembling. Each cell is made up of a set of numerically controlled machines, an automatic inventory system for raw and finished pieces, and an automatic guided vehicle system (AGV) for moving pieces. Integration is achieved through a network which encompasses shop and cell computers, along with computerized numerical controls (CNC) for the machines and programmable logic controllers (PLC) for the inventory and the AGV systems. The production is characterized, at any period of time, by the presence in the cells of several concurrent small lots of different piece types. Each piece of a lot undergoes a sequence of operations, each one on a different machine, according to the piece type. The cell, with a real time dispatcher, assigns pieces to the machines as soon as they become idle, and issues requests of missions to the AGV system, monitoring the shop–floor (lot production). A database is present and contains the technological data (operation, piece type, machine type) and the management data (order, piece, lot) with their relationships, which allow the shop and the cells to control the production.

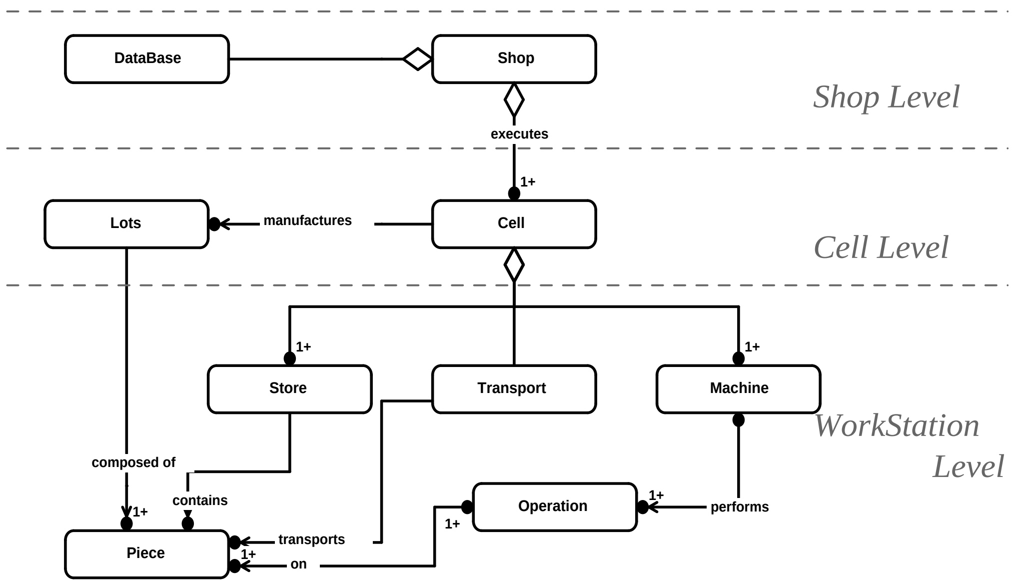

Figure 3: Analysis of the FMS.

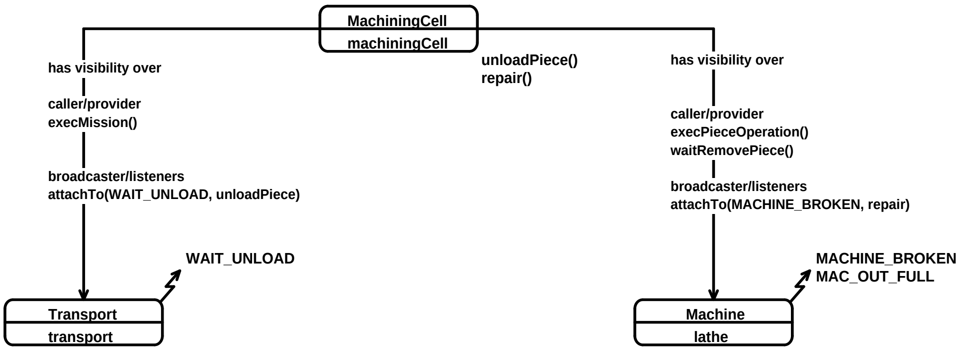

Figure 4: The Communication Mechanisms.

Organize your system in modules of controls, that must be objects with a certain degree of autonomy and capable of offering well defined services managing a set of resources. Give a hierarchical structure to the set of these modules applying the principles of inclusion *or* external use.

Pattern 2 : Visibility and Communication between Control Modules

Each module of the hierarchy performs services, requests services from other modules or signals events so as to inform other modules of its state.

Context

Complex system are developed during a relatively long period of time. Two possibilities are available: the first is building modules that will be plugged in pre-existing designs, the second is building "frameworks" that have to integrate preexisting control modules; in the former case, modules are build having "*visibility*" [7] of the environment in which they will operate, while in the latter this does not happen.

In CIM systems the evolution is bottom-up. A higher layer control module (e.g. a shop controller or a cell controller) acting as a client integrates lower level controllers, which act as servers (e.g. machines, transports and buffers).

However, modules communicate, and the way their communications are established depends on the 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. The two situations have a direct relationship with the two basic mechanisms of communication between objects, here called **caller/provider** (C/P) and **broadcaster/listeners** (B/L) and described in figure 4.

- The C/P mechanism is involved when an object invokes another object method and is deeply rooted in OO programming, so it will not be discussed further.
- The B/L mechanism (Smalltalk dependencies, and X-Window callbacks) [12, 25] 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 a broadcasted message issued at a certain instant of time by an object, identified by a symbolic name and with some data associated with it.

Solution

Reusability in a hierarchy of layers of control will improve, by avoiding visibility loops, not allowing two controls at the same layer to communicate directly, and following instead these rules (see figure 4):

- two modules at the same layer should not be entitled to communicate directly,
- when a client addresses a message (command) to a server, this should be done explicitly in a C/P fashion,
- when a server answers, e.g. by sending monitoring information, this should be done by signalling events through a B/L mechanism.

This pattern is called OBSERVER in [16] and it has the same logic of reusability as the `changed: update:` mechanism of Model–View–Controller mechanism which has already been documented in the Smalltalk browser [17].

Example: Communication in the FMS

The cell for machining, detailed in figure 4, asks for services directly from the machines (`execPieceOperation()`, `waitRemovePiece()` and so on) and from the transport (`execMission()`) (it should be noted here, in fact, that inclusion also implies use); but the evolution of the cell production has to be driven by the events broadcasted by its components, e.g. `MAC_OUT_FULL` when a piece enters the output buffer of a machine.

All modules of a control system are entities that communicate with each other through C/P or B/L mechanisms.
Communication between controls takes place using a C/P mechanism from the layer containing newer (or in any case the more volatile) modules to the layer containing older (or the more stable) ones, and a B/L mechanism viceversa.

Pattern 3 : Categorize Objects for Concurrency

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

Context

The first problem is that there are concurrent activities occurring in a complex system, and that concurrency assumes different scales of granularity. Small grain concurrent activities are made up of simple actions with a weak cohesion, as in a multi-window graphical interface; larger grain concurrency may be represented by operations composed of an ordered sequence of actions, having a stronger cohesion, e.g. the production processes of different (logically concurrent) lots of pieces by a cell controller; even larger concurrency is represented by control modules operating in parallel, such as the distinct machine controllers executing independent operations.

A second problem is that modules of complex concurrent systems interacts in various ways, in particular:

- through entities where services can store data,
- through entities where services enter into conflict over the sharing of limited resources, such buffers of parts or pools of carts for transportation, or
- through entities which perform (sub)services developed over time.

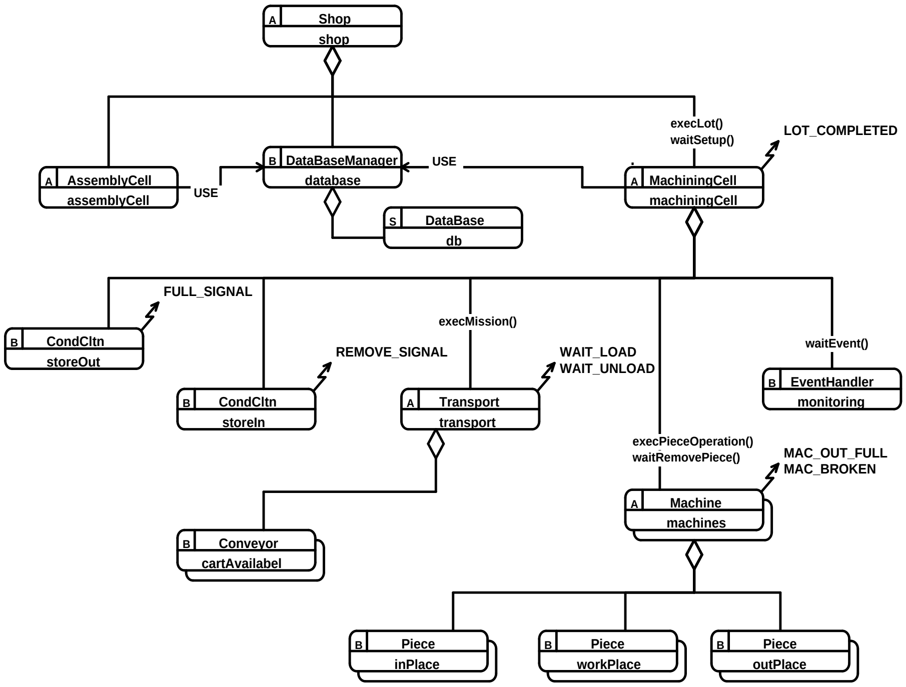

Figure 5: The complete FMS logical design.

Solution

Use concurrent processes to express concurrency in its larger forms. In G++, concurrent processes are encapsulated in proper active objects; in this case, processes represent the *services* offered by an object in response to a request for an operation.

The now widely accepted classification between *passive objects* (sequential or blocking) and *active objects* derives:

Passive Objects do not possess threads of control, and depend on client threads for the execution of their functions. They can be further subdivided into *sequential* or *blocking* objects. The semantics of *sequential* ones is guaranteed only in the presence of a single thread of control. Such classes have an "`S`" in their label. *Blocking Objects*, designed to maintain their semantics when more than one thread accesses them, offer the *wait* primitive which is the indispensable element to block client threads and so allow them to interact. Such classes have a "`B`" in their label.

Active Objects differ from passive ones because they possess, create and internally manage one or more independent threads of control which govern the execution of their services. A class `ThreadOfControl` is used for representing a concurrent service. Active objects are the natural candidates to represent modules of controls; An "`A`" in their label indicates them.

Example: Active Objects in the FMS

In the FMS example, the following control modules, i.e. active objects, are identified in figure 5:

- the shop, `shop`;
- the cells, `machiningCell` for machining and `assemblingCell` for assembly of lots of pieces;
- the machines; a collection of machines is indicated with the name `machines`;
- the transport system, `transport`.

Instead, stores for semi-worked and finished pieces (named `storeIn` and `storeOut`) are modeled as blocking objects, since they act as shared resources between different lot production services.

Model the concurrent activities of a complex control architecture, according to their level of granularity, by actions triggered by events, by sequential processes (services) interacting through shared resources, or by concurrent modules exchanging requests of services. Moreover, in order to support concurrent services, identify sequential, blocking *and* active *objects and represents modules of control as active objects.*

Related Patterns

The role of each of these three basic elements is detailed in a specific pattern (see patterns 4 to 6).

Pattern 4 : Actions Triggered by Events

Communication through events can be either intra–service, generating a small grain concurrency, or inter–service, resulting in a larger grain concurrency.

Context

Developing an application, it is possible to find **intra–service** concurrency requirements (as in the case of activating rules in a rule based decision making system for scheduling production), or **inter–service** ones, when two objects in two distinct concurrent processes, in distinct modules of control, want to communicate by exchanging events with each other (e.g. in the FMS the service of *piece machining* of a workstation raises a failure event listened by the service of *lot production* of the cell, which is waiting for it).

Solution

The B/L mechanism to raise and listen to events, which was introduced with modules of control for communication between older and new one which encapsulated them, is implemented just once in the `Object` class and is inherited by all classes of the framework.

Use the the B/L mechanism to model fine–grained concurrent actions by the interaction between sequential objects bounded to one of the services offered by an active object. Use this same mechanism for communication between lower layers of modules of control to higher ones.

Related Patterns

The interaction between services through events is discussed further in pattern 5.

Pattern 5 : Services "Waiting for"

Services offered by a control module have to wait for a condition to occur or for data to be transferred from/to a shared resource, before they can perform one of their actions in concurrency with others.

Context

In a manufacturing system, the former case is exemplified in the cell by the service of lot production while waiting for the event "`MAC_OUT_FULL`" from a machine, while the latter case is exemplified by the service of transport which is provided by the conveyor system being blocked while waiting for the first cart available in the pool of free carts. Both situations indicate the need of entities which can manage the interaction between services.

Solution

Blocking objects are the entities useful for this problem. In fact, in an OO world the only way a service can enter into a state is to suspend itself by sending a message (asking for a service which implies waiting for a condition) to a blocking object. The flow will be resumed with a transition from this waiting state whenever the appropriate condition on the blocking object occurs.

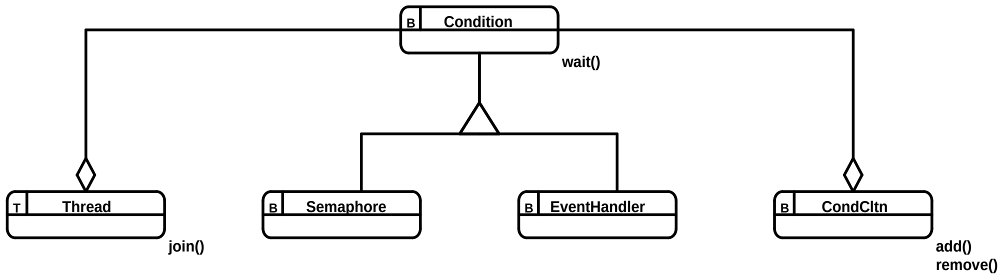

Figure 6: Blocking Objects

`Condition` is the abstract class used by the framework to define any blocking object, from which specific implementations such as timers, semaphores, event handlers and shared queues (bounded buffers, i.e. `CondCollection`) can be derived by inheritance or encapsulation (see figure 6). Conditions, as a way of synchronizing threads, are discussed in [3].

Example: Blocking Objects in the FMS Cell

In the FMS example, there are different blocking objects (see figure 5):

- the `storeIn` and `storeOut` buffers are `CondCltn`, belonging to the `MachiningCell` class. They act as a bounded buffer: the cell service automatically stops if it tries to remove a piece when the `storeIn` is empty or to add a new piece when the `storeOut` is full.
- `inPlace`, `workPlace` and `outPlace` are, similarly, `CondCltn` and act as blocking objects of the `Machine` services.
- `database` is modeled as a blocking object, as it is a shared resource between the cells;
- the `MachiningCell` has an `EventHandler` which is used to block its services while waiting for the events from transport, machines and stores. Here the cell controller uses an EventHandler to monitor its resources (e.g. `MAC_OUT_FULL` or `WAIT_NOT_FULL`) and to take the appropriate actions.

Use blocking objects to model the shared resources of the control module every time interaction between its concurrent services is needed.

Related Patterns

The synchronous request of a service to an active object, as will be seen in pattern 6, is one example that shows the presence of a `Condition` inside a `ThreadOfControl`.

Pattern 6 : The Client/Server/Service: Implementing Modules of Control

Modules of control are active objects which offers multiple concurrent services and are present at different level in the hierarchical architecture of the system.

Context

In the design of control modules, a first requirement if that more than a single service in every moment to there client: e.g., a cell may be managing more than a single lot of pieces.

Another requirement is that we want 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. That is, there is the necessity of representing a system as a structure of hierarchically layered virtual machines [20].

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

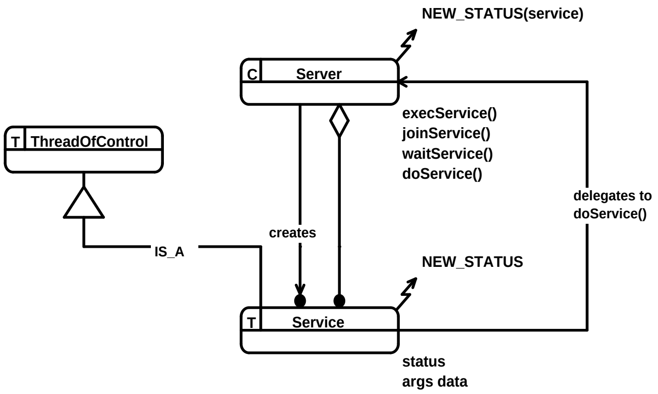

Figure 7: Client/Server/Service.

Solution

These requirements are satisfied by the the Client/Server/Service model proposed in this pattern and derived by the programming paradigm of SR [4]. Two classes are needed to support it: the `Service` and the `Server` (see figure 7).

The `Service` extends the `ThreadOfControl` so that it becomes an extended finite state machine embedding the dynamics of the application and has the following characteristics: it always belongs to a unique `Server` owner, has internal data (sequential objects), maintains a symbolic state value, and broadcasts events with the incoming state name every time a state transition occurs. It rarely needs to be redefined as it delegates the execution of its dynamics to a private behavior method of the server owner.

The `Server` is an abstract class that defines the common implementation of active objects, and has to be redefined in order to build concrete servers.

It contains a `Collection` of `Service` objects: each service executes the server operations in concurrency, while sharing the same set of resources. A server's service can, in turn, use another (sub)server and in this case it acts as a client for that server.

To be accessed, the `Server` offers three public methods: `execService()`, `waitService()`, and `joinService()` which represent asynchronous, synchronous, and deferred-synchronous service requests, respectively.

They accept, as arguments, sequential objects only by value, blocking and passive objects are accepted by reference too.

Whenever each one of these three public methods is called, a new `Service` is forked, it is added to the list of ready services; by default, it is executed the private method `doService()` of its `Server`. `doService()` has to be redefined in the derived concrete classes.

Moreover the `Server` relays its service events externally so that its state can be monitored by other objects.

The Client/Server/Service pattern is fundamental to our pattern language; it contains the essential structuring solution for concurrent programs. Much in the same way that a Smalltalk programmer structures virtually all user-interface code after the Model–View–Controller pattern [17], we use Client/Server/Service for concurrent programming at all levels. Moreover, since the control modules in the CIM reference model of figure 2 are active objects, the pattern applies throughout a CIM control system.

Inherit form `Server` *and redefine* `Server::doService()` *to have a simple active object with multiple concurrent threads of control, which share the Server components such as common resources between them.*

Related Patterns

Modules of control are implemented using this pattern and specializing the `Server` class, as is described in the patterns 7 and 8. The prototype objects mentioned in pattern 9 are often interfaces to `Server` objects.

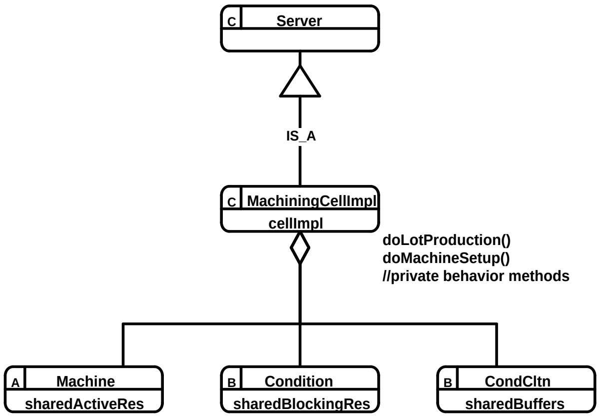

Figure 8: Implementation of Modules of Control.

Pattern 7 : Implementation of a "Multiple Kind of Services" of Modules of Control

Modules of control manage different pools of shared resources and possibly offer different kind of services.

Context

Control modules may need to offer multiple kinds of services, and not simply multiple concurrent services of the same type.

The `Server` class offers an active object with a "standard" interface, represented by the `execService()`, `waitService()`, and `joinService()` methods, and executes multiple instances of the same "service".

Solution

"Multiple Kind of Services" modules of control can be obtained by inheriting from a `Server` (see figure 8) and using the following guidelines:

- The resources needed by the control module must be identified. The resources which are shared by the services of an active object can be only blocking objects (`Condition` or `CondCltn`) or other active objects (`Server`). There are no sequential objects inside the `Server`, as they are suitably encapsulated into blocking objects or bound to the services.
- The different types of services offered must be identified (e.g. for the class `MachiningCell` the services "`LotProduction`" and "`MachineSetup`"), their behaviors must be specified in terms of finite state machines extended by the necessary sequential objects bounded to the service, and implemented by private behavior methods (e.g. `doLotProduction()` and `doMachineSetup()`).

  A graphical notation aimed at specifying service behaviors will add expressiveness to the formalism and because it is possible to animate it, debugging will be easier.

  The specification of a `Service` behavior, obtained with an OO extension of SDL (the Specification and Description Language standardized by CCITT, [21]). Such a specification leads to the automatic generation of the behavior method by the CASE environment.
- The `Server::doService()` method must be redefined in order to become a switch case, which is controlled by a name parameter, so as to access the different types of service. The CASE environment supporting the framework does this automatically when the class is generated.

The practical reason for this behavior is that in the G++ library it is easier for the `exec-join-waitService` triad always fork the `doService()` execution.

The representation of all service behaviors in the framework with the unique `doService()` method of the `Server` is an alternative design choice to the COMMAND pattern of [16]. COMMAND "objectifies" services, by modeling them as different "behavior objects".

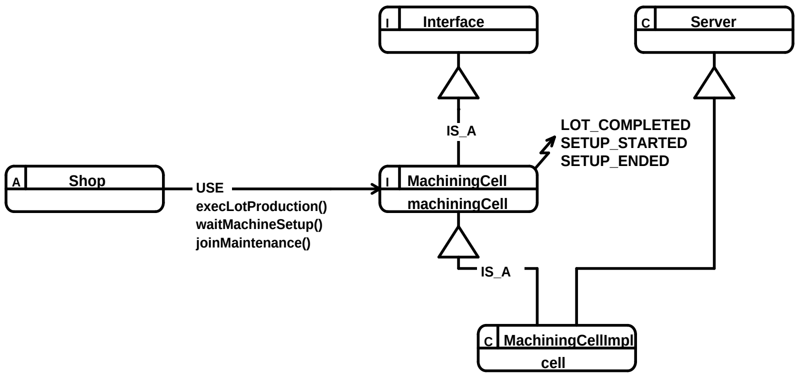

Figure 9: Interface and Implementation linked by Inheritance

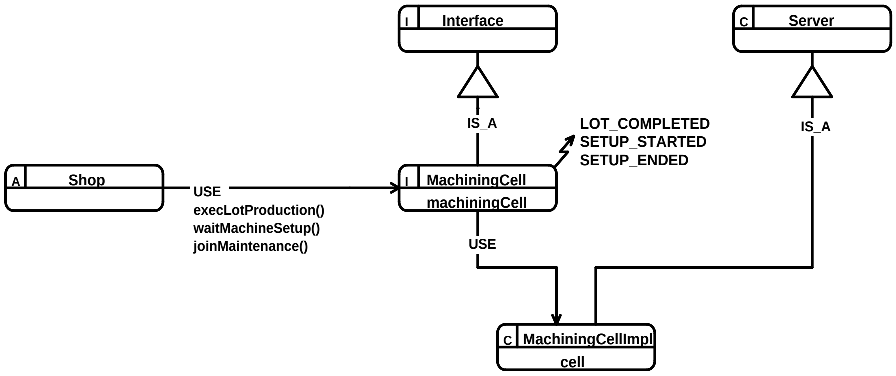

Figure 10: Interface and Implementation linked by Use

Instead Client/Server/Service uses a pattern that we would call *Delegated Environment*: service behaviors are methods of the object that offers them (and not of "objectifing" objects) 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 it allows all services inside a server to share the server resources without visibility problems.

Example: the class `MachiningCellImpl`

At this point it is possible to consider the design of the FMS machining cell, defined by the class `MachiningCellImpl`. It inherits from `Server` and its member data (see figure 5) are the AGV system `transport`, instance of class `Transport`, the collection of machines `machines`, stores of semi–worked and finished pieces (`storeIn` and `storeOut`, of class `CondCltn`).

The service of routing pieces between machines and stores is expressed by the method `doLotProduction()`; more than one service can be active at a certain moment. Other services regards machines setup and maintenance.

Control modules are implemented by inheriting from the `Server`*, by encapsulating or referring to shared resources, which must be blocking objects or other active objects, and by specifying the behavior methods of its services in terms of extended finite state machines.*

Related Patterns

The companion `Interface` class is discussed in pattern 8.

Pattern 8 : The Interface to Modules of Control

Control modules offer different kinds of operations to their clients.

Context

The class derived by inheritance from the `Server` in pattern 8 are "heavyweighted" classes, which carry the property of being active objects but also the resources encapsulated by the module of control they represent. In C++ it is not convenient derive directly by them, if they need to be substituted by proxy objects, e.g. for distributed implementation. Anyway we don't want to change the client code because of this substitution

Solution

Use an `Interface` object to access each `Server` (see figures 9 and 10.) The `Interface` offers the server's operations to clients.

When a new active object is conceived and the services it offers identified, e.g. the `MachiningCell` can perform "`LotProduction`", "`MachineSetup`" and "`Maintenance`", a new class `Interface` is generated, assigning to it the following features for each type of service:

- a link to the implementation object;
- an interface operation, which delegates one-to-one, according to the three synchronization semantics of the specification, its functionalities to the standard three `Server` methods; e.g. `execLotProduction()` is asynchronous and calls `execService()`, `waitMachineSetup()` is synchronous and calls `waitService()`, `joinMaintenance()` is deferred synchronous and calls `joinService()`.

Figures 9 and 10 present two alternative architectural solutions for this pattern, where interface and implementation are linked by "inheritance" and "use" relationships, respectively; in [16] both these solutions are considered in the ADAPTER pattern.

In both cases the `Interface` object can be generated automatically by a CASE environment from the specification of the active object. For active objects, even those that will not be distributed as described in the two following patterns, this separation often favors reusability.

Client control modules have to refer to their server control modules by `Interface` *objects which are linked to their* `Implementation` *through an inheritance or use relationship.*

Related Patterns

This pattern follows the trend among C++ designers to separate an object definition into two distinct interface and implementation classes. The interface class is called a *handle* in [24] and *proxy* in [9]. It also corresponds to the PROXY of [10].

The switch from prototype to reality, as described by pattern 9, is usually enabled by the use of this pattern.

Pattern 9 : Prototype and Reality

Any complex application requires prototyping and simulation of the different elements which have to be integrated before an implementation is derived.

Context

The need for prototyping and simulation is particularly present in CIM control systems, and in other types of applications which interface to hardware controllers or are by their nature distributed. For instance, one cannot test and debug a factory control application using the actual controllers and machines of the factory.

The evolution from simulation to real system should be as seamless as possible. The non-simulated components (that is, the rest of the system) should remain unaffected by this evolution.

Solution

The patterns that follow consider the transition from logical to physical design obtained by substituting in the program the prototypes with their physical counterpart.

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

- the *"simulated or emulated time prototype"*, which simulates or emulates the object behaviour;
- the *"reality object"*, which embeds the physical object.

The *"reality object"* could be a hardware device driver, an encapsulation of an external functionality like a relational database, or a surrogate for an external object (stub) as described in the next pattern.

When going from simulation to reality, replace the simulation object with the object (e.g. device driver) which interfaces to the reality.

Consistency in this transition is guaranteed either by inheriting the two implementations from a common base class which defines their interface, or by application of the pattern 8 substituting the desired implementation object in the interface object's use relationship.

To maintain two distinct implementations of a certain conceptual entity and to switch from one to the other is an application of the BRIDGE pattern from [16]. A reality object which encapsulates an external functionality is an example of the ADAPTER pattern from [16]. Installation can be speeded up by switching from prototype to physical implementation in the same program, which is achieved by using a suitable design pattern such as the ABSTRACT FACTORY of [16], which allows the user to decide at run time the specific class he needs.

Evolution from logical to physical design and from prototype to actual implementation is achieved by maintaining two representations of the same entities (the "prototype" and the "reality") and by transforming the prototype into the physical reality, which is done by exploiting polymorphism or use relationship.

Related Patterns

Going from prototype to reality when reality is a remote (distributed) object is described in pattern 10.

Pattern 10 : Distribution of Modules of Control

Control modules usually reside on remote computers or peripheral devices, interconnected through a common communication network. These modules define the physical architecture which must be realized by the final distributed system.

Context

Complex distributed systems share the need for simulation discussed in the previous pattern. When going from a non-distributed 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.

As for the previous pattern, the rest of the system should not be affected by moving some objects to a remote node. In addition, moving objects to remote nodes should be relatively easy in order to not prevent the system to change with the physical architecture; that is, we want to exploit an evolutionary approach.

Solution

Create large grain objects, in G++ called *remote contexts*, one for each node that will contain remote objects. Remote contexts contain one or more (typically active) objects, along with the services which these objects need in order to function as an install able program.

The remote context supports listening for requests coming over the communication network from other nodes, unpacking these requests 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 [23]. If the objects in a remote context will be making requests to objects residing on other nodes, the installable program must also contain the necessary `Stub` objects, as explained below.

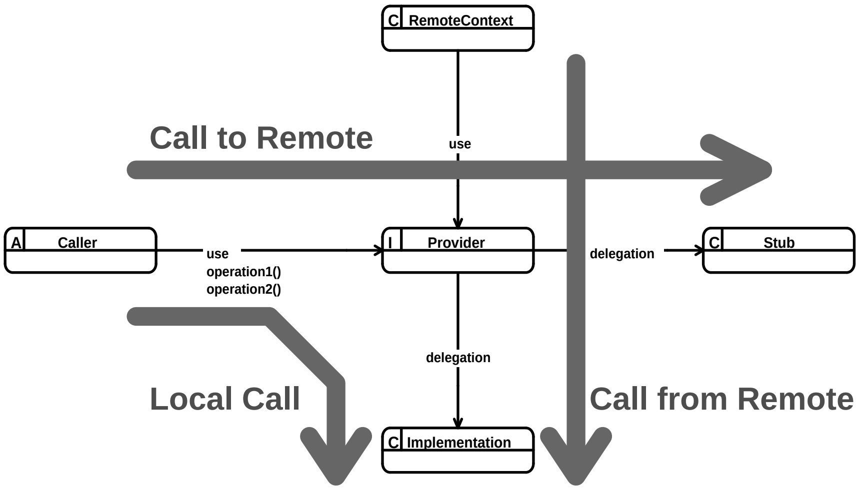

Figure 11: Prototyping and Distribution.

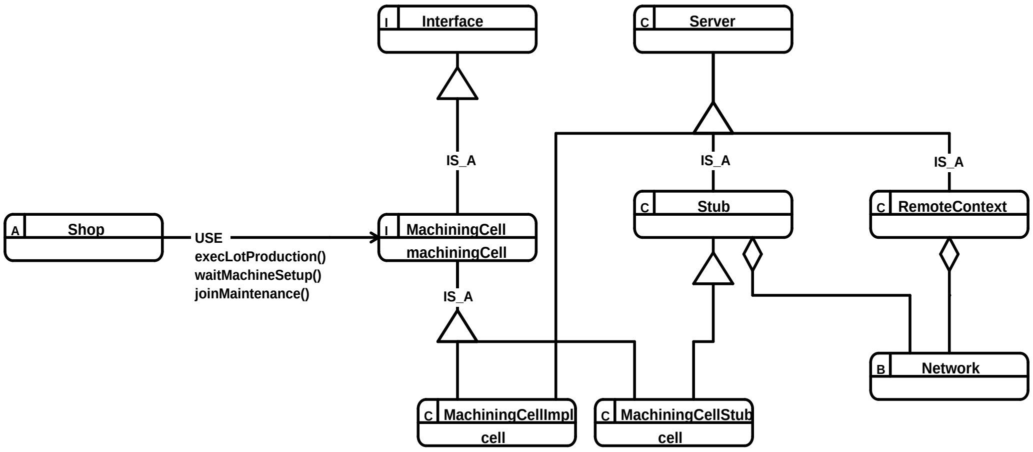

Figure 12: Prototype and Stub linked by Inheritance.

In the original system, replace the objects that have been moved to other nodes with `Stub` objects which encapsulate the distribution and communication interface. These stubs 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. As in the previous pattern, consistency is ensured either by inheritance polymorphism, as in figure 12, or by delegation, as in figure 13. The functionality of the stub object corresponds to that of the CORBA stub.

Each `Stub` and `RemoteContext` encapsulate an instance of the class `Network`. While the former two classes are rarely modified, the `Network` might need adaption to the particular communication platform. Note that a sufficiently intelligent CASE tool could automate the transition to a distributed system by generating the necessary stubs and remote contexts, and replacing the original simulation objects with the stubs. For certain platforms, this transition could be as simple as generating CORBA IDL interface descriptions.

`Stub` and `RemoteContext` are the natural interface between the large grain of concurrency, represented by separate modules, and the smaller one, represented by services inside modules, supported by the G++ model.

`Stub` is an application of the PROXY pattern from [16]. Switching between a simulation object and the corresponding `Stub` is an application of the BRIDGE pattern, again from [16].

Example: Distribution of a Shop and a Cell

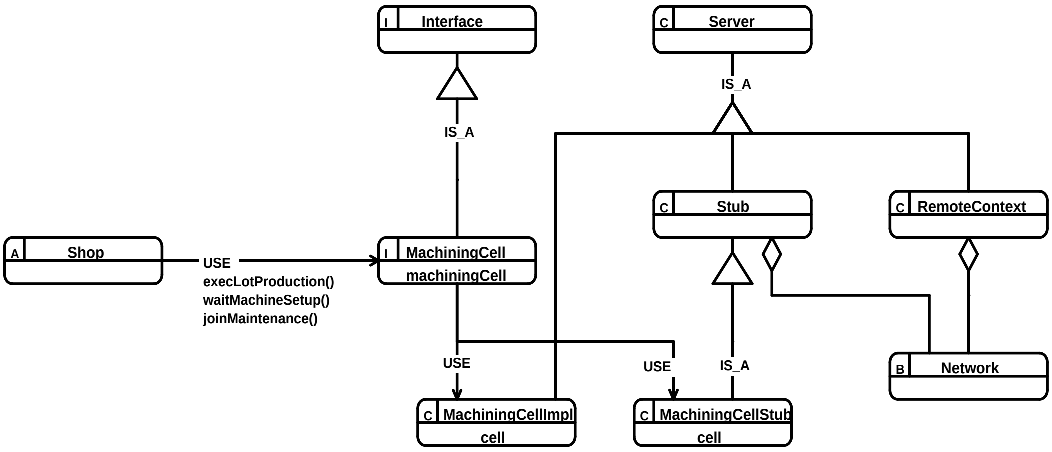

Figure 13: Prototype and Stub linked by Use.

We assume that the `Shop` and `MachiningCell` control modules are installed on respective computers interconnected through a factory-wide backbone network and that cell peripherals, in particular the collection of machines, are connected to the cell computer through shop–floor LANs.

Focusing on the cell distribution, figure 14 shows how in the shop controller, the `MachiningCellStub` substitutes the `MachiningCellImpl`, the prototype of the cell control module. `MachiningCellStub` communicates with its corresponding `RemoteContext`, which encapsulates `MachiningCell`. In turn the `MachiningCellImpl` contains a collection of `Machine` whose implementations, also, have been substituted by their respective stubs.

Evolve the prototype to a distributed application by using `Stub`*s to create alternative implementations of remote modules of control to be substituted in the model, and* `RemoteContext`*s that encapsulate these remote modules of control and provide a context environment for them. These two classes are a matched pair offered by the framework and are rarely modified, as they delegate the implementation of the low level protocols to a* `Network` *object, which wraps the communication drivers (e.g. ORB), and it should be provided for in each different implementation.*

3 Comparison with the "Design Pattern Catalog"

In order to compare the patterns which are presented here with those described in the literature, they are subdivided and classified as *elemental, basic design, domain dependent.* Elemental patterns determine the language (elements of the architecture) in which the whole application has to be written and include *A Hierarchy of Control Layers, Concurrency, Actions Triggered by Events, Services "Waiting for", Prototype and Reality.* Basic design patterns, similar to those discussed in [16], and with the same level of abstraction, propose OO design solutions as intermediary building blocks, such as *Client/Server/Service*, which by their very nature, are fairly application independent. The others, i.e. *Visibility and Communication between Control Modules, Implementation of Modules of Control, The Interface to Modules of Control, Distribution of Modules of Control* are domain dependent and are very similar in concepts to [15]: they provide guidelines on using the framework's classes to solve specific problems. They obviously exploit in turn basic OO design patterns for their implementation.

4 Conclusion

A set of patterns with which to design concurrent distributed systems exemplified by CIM applications, has been defined. They are the documentation which makes it possible to understand and use the many functionalities of event handling, concurrency and distribution offered by a framework of basic reusable classes.

Each pattern proposes a design choice for a particular aspect of the problem, however on the whole the patterns become a language i.e. an organized set of rules which guide the designer from analysis to physical implementation, with the objective of raising reusability from components to architectural level, and of easing

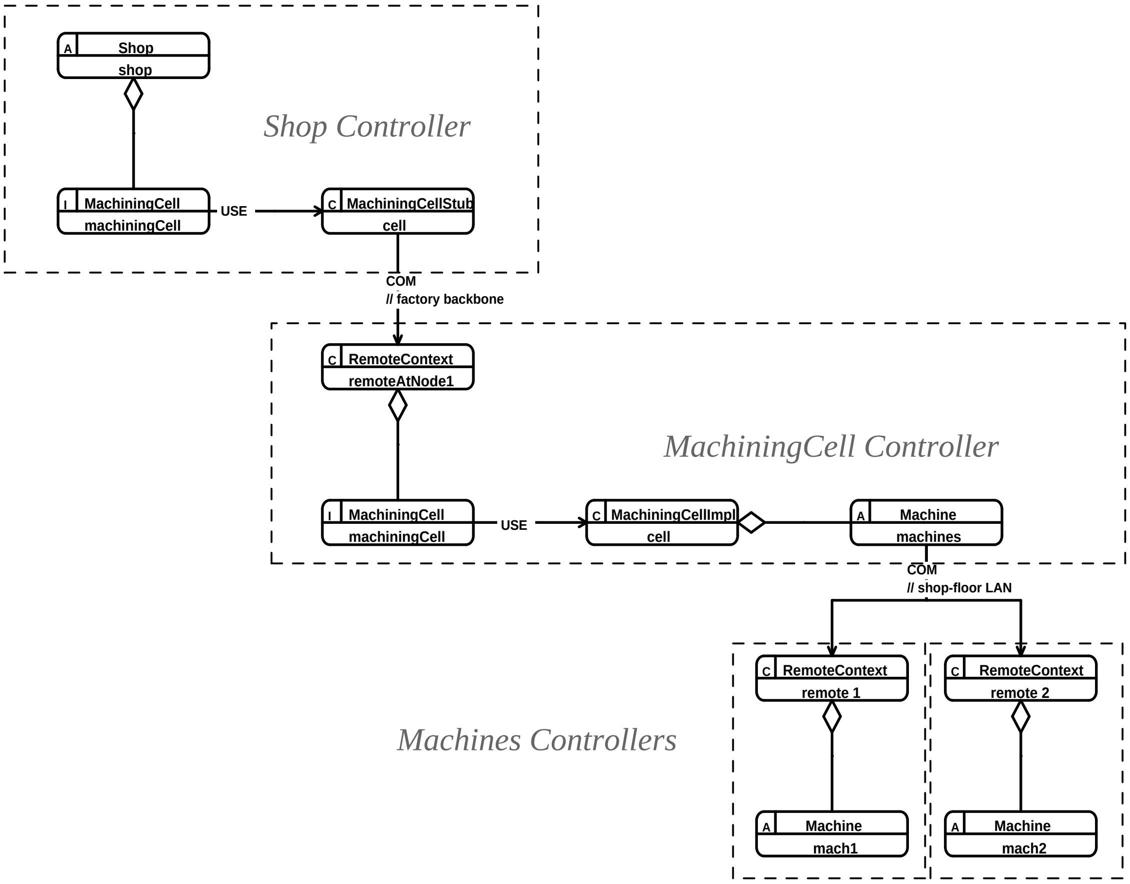

Figure 14: Physical Distribution.

the designer's task, when making choices in the sophisticated environment of OO Programming. This has been made possible thanks to unambiguous rules which reduce the semantics gap between the problem and the software architecture. In fact patterns, as a media in which analysis evolves into design, are an answer to the problems of the seamless transition from analysis to design raised in [14] by Hoydalsvik and Sindre at the OOPSLA'93 Conference.

Finally, one contribution made by this paper is to allow dynamic behavior to be expressed as a property of the whole architecture, instead of it being considered as an attribute of each single object in isolation; this has been achieved by embedding the design into a well defined framework resulting from the application of the pattern language.

Acknowledgment

We wish to thank our "shepherd" at the PLoP conference, Jim Coplien for his help and advice, Prof. Ralph Johnson of University of Illinois as Urbana Champaign together with John Brand, Brian Foote and Don Roberts for their rich suggestions and their carefully review of the paper.

References

[1] C. Alexander. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

[2] C. Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[3] G.R. Andrews, F.B. Schneider. *Concepts and Notations for Concurrent Programming.* ACM Computer Surveys, vol. 15, n. 1, March 1983, pp. 3-43.

[4] G.R. Andrews. *An Overview of the SR Language and Implementation.* ACM Transactions on Programming Languages and Systems, 10(1):51-86, 1988.

[5] K. Beck and R. Johnson. Patterns generate architecture. in *ECOOP'94 Conference Proceedings*, Bologna, Italy July 1994.

[6] B. W. Boehm. *A Spiral Model of Software Development and Enhancement* IEEE Computer, May 1988.

[7] G. Booch. *Object-Oriented Design with Applications* Benjamin/Cummings, 1991.

[8] R. S. Chin and S. T. Chanson. Distributed object-based programming system. *ACM Computing Surveys*, 23:91–124, March 1991.

[9] D. Decouchant. Design of a distributed object manager for the Smalltalk–80 system. OOPSLA'86 Proceedings, pp. 444–451.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissdes. Design Patterns: Abstraction and Reuse of Object Oriented Design. In *ECOOP '93 Conference Proceedings*, Kaiserlautern, Germany, April 1993.

[11] P.B. Gibbson. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, January 1987.

[12] A. Goldberg and D. Robson. *Smalltalk80: The Language and its Implementation.* Addison-Wesley, 1983.

[13] R. Halstead, *Multilisp: A language for concurrent symbolic computation* ACM Transactions on Programming Languages and Systems, vol.4, no 4, October 1985.

[14] G.M. Høydalsvik and G. Sindre. *On the purpose of Object Oriented Analysis.* OOPSLA'93 Proceedings, Washington, D.C. September 1993

[15] R.E. Johnson. *Documenting Frameworks using Patterns.* OOPSLA'92 Proceedings, Vancouver, B.C., Canada, October 1992.

[16] E. Gamma, R. Helm, R. Johnson and J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley. 1994

[17] G.E. Krasner and S. T. Pope. A Cookbook for Using the Model–View–Controller User Interface Paradigm in Smalltalk–80. *Journal of Object-Oriented Programming*, August/September 1988

[18] C. McLean, M. Mitchell, and E. Barkmeyer. *A computer architecture for small-batch manufacturing.* IEEE Spectrum, 20(5):59–64, 1983.

[19] G. Menga, G. Elia, and M. Mancin. G++: An environment for object oriented design and prototyping of manufacturing systems. In W. Gruver and G. Boudreaux, editors, Intelligent Manufacturing: Programming Environments for CIM. Springer–Verlag Ltd., 1993.

[20] D.L. Parnas, P.C. Clements, and D.M.Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):259–266, March 1985.

[21] A. Rockstroem and R. Saracco. *SDL-CCITT specification and description language.* IEEE Transactions on Communication, COM-30, June 82.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design* Prentice Hall, Englewood Cliffs, NJ (USA), 1991.

[23] R. M. Soley. *Object Management Architecture Guide* OMG, Inc. 492 Old Connecticut path Framingham, Ma 01701, November 1990.

[24] B. Stroustrup. *The C++ Programming Language – 2nd Ed* Addison-Wesley, 1991.

[25] D. A. Young. *The X Window System Programming and Application with Xt.* Prentice-Hall, 1989.

[26] R. Wirfs-Brock, R. Johnson Surveying current research issues on object oriented design. Communication of the ACM 33(9) 105-124 Sept. 1990.