Applying Patterns To Resolve Software Design Forces In Distributed Programming Environments

Joe Hoffert
joeh@cs.wustl.edu
http://www.cs.wustl.edu/~joeh
Department of Computer Science,
Washington University,
St. Louis, MO. 63130, (314) 935-6193

Abstract

Extensible software libraries for distributed programming environments (DPEs) must be flexible and modular. In general, patterns capture widely adaptable collaborations of classes and objects to solve commonly occurring design problems. In particular, patterns are used to resolve the forces inherent in distributed systems and have proven to be a benefit in the development of Playground. Playground is a DPE that provides a methodology and set of software tools for writing interactive distributed applications. This paper uses the Playground environment as an exemplar to discuss the relevance of patterns to extensible software libraries for DPEs.

A version of this paper appeared in the July/August 1998 issue of “C++ Report” under the title “Resolving Design Problems in Distributed Programming Environments.”

1.0 Introduction

Extensible software libraries for distributed applications require extensibility, flexibility, and localization of functionality. Examples of such libraries are the Object Request Broker (ORB) core that CORBA [1] vendors provide, the ADAPTIVE Communication Environment (ACE) communications framework [2], and the C++ class library component of the Playground DPE [3]. This type of software needs to facilitate configuring different options, while also easing the addition of new functionality. In addition, software for DPEs must handle different types of networking mechanisms and heterogeneous operating systems. Such is the case for the Playground environment, which is examined in this paper to illustrate relevant design forces and their resolutions using patterns.

Playground is an object-oriented DPE written in C++ that separates communication from computation for distributed applications. It incorporates the concepts of extensibility and flexibility through the use of patterns. Design patterns [4] capture widely adaptable collaborations of classes and objects to solve commonly occurring design problems. Patterns can aid the development of software that supports distributed applications. They facilitate the flexibility and maintainability that allows changes to be made without rippling through the entire software.

Patterns have proven to be a benefit in the development of Playground. The goal of this paper, therefore, is to show how the use of patterns increases flexibility, extensibility, and maintainability in the development of software supporting distributed applications. The patterns described in this paper are not applicable solely to distributed applications or Playground, however, and have been applied in many other areas.

2.0 Overview of Playground

2.1 Playground Introduction

Distributed applications support interaction among many users and system components. In addition, they facilitate effective communication and synthesis of information. This effectiveness is enhanced when end-users are empowered to integrate and customize applications and their components dynamically to take advantage of new components and new sources of information.

The Programmers' Playground [3] provides a methodology and set of software tools for writing interactive distributed applications. Examples of interactive applications include *teleconferencing applications* that let users join and exit dynamically and *distance learning applications* where teachers and students can interact with the information being presented. Playground provides an abstraction that serves as a *middleware* layer between the programming language and low-level communication protocols. It also provides a uniform approach to communication that accommodates diverse collections of persistent and transient applications.

Playground is based on a model of distributed computing called *I/O abstraction* [3]. I/O abstraction is a model where each program or *module* in a system has a set of data structures that are externally observed and/or manipulated. This set of data structures forms the module's external interface, called the *presentation*. Data structures can be arbitrarily added or removed from the presentation. When a data structure is *published* it is added to the presentation. When *unpublished* it is deleted from the presentation.

A Playground module runs as a process; there can be several processes of a Playground module executing simultaneously. Each module is written independently and modules can be configured by establishing logical connections between the data structures in their presentations. The connections must respect access restrictions established by each module for its own data. As published data structures are modified within a module, communication occurs implicitly "behind the scenes" to other modules according to the connections created in the configuration. As shown in Figure 1, when a module writes to a published variable, the value is automatically communicated to all connected modules.

Playground provides tools to create modules and configure those modules to create distributed applications. There are currently 4 tools supported in the Playground environment (shown in Figure 2): (a) the Playground class library provides all the needed C++ classes to build a Playground module (partial class hierarchy shown), (b) the Mediator tool provides visualization of Playground modules and graphical connection capability between variables, (c), the Euphoria tool provides GUI construction capabilities, and (d) the application management system provides launching capabilities of applications once they have been developed.

Source code for the C++ class library and some sample applications are available for download from the Playground Web page `http://cs.wustl.edu/cs/playground`. Executables for the other components in the Playground environment are also available at that same Web page for several OS platforms including Solaris, OSF/1, IRIX, and Windows NT.

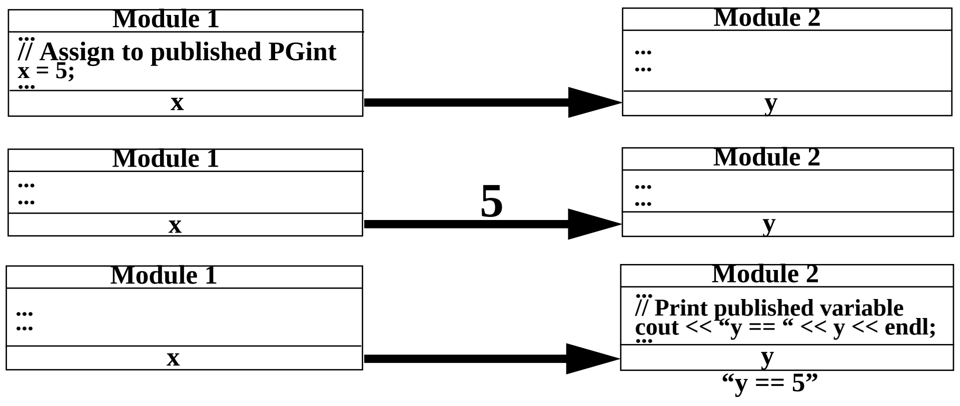
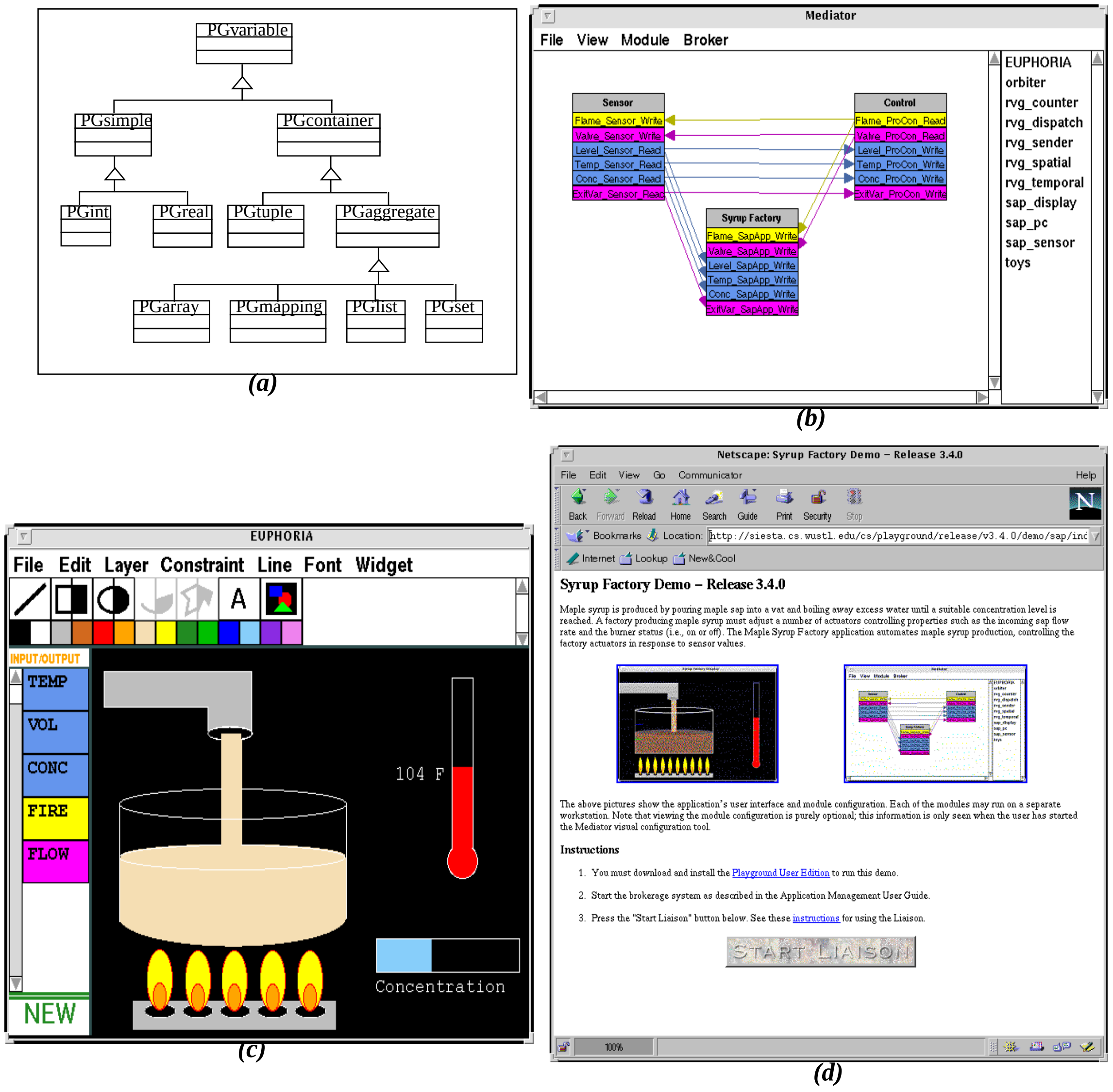


FIGURE 1. I/O Abstraction



Syrup Factory Demo – Release 3.4.0

Maple syrup is produced by pouring maple sap into a vat and boiling away excess water until a suitable concentration level is reached. A factory producing maple syrup must adjust a number of actuators controlling properties such as the incoming sap flow rate and the burner status (i.e., on or off). The Maple Syrup Factory application automates maple syrup production, controlling the factory actuators in response to sensor values.

The above pictures show the application's user interface and module configuration. Each of the modules may run on a separate workstation. Note that viewing the module configuration is purely optional; this information is only seen when the user has started the Mediator visual configuration tool.

Instructions

1. You must download and install the Playground User Edition to run this demo.
2. Start the brokerage system as described in the Application Management User Guide.
3. Press the "Start Liaison" button below. See these instructions for using the Liaison.

FIGURE 2. (a) Playground C++ class library. (b) Mediator visualization tool. (c) Euphoria drawing tool. (d) Application management system via a Web page interface.

2.2 Playground Communication

Playground helps to simplify distributed application programming by treating communication as a high-level relationship among module states. Interaction among modules occurs implicitly as a result of this relationship. In this way, low-level input and output activities are hidden from programmers. For instance, programmers need not be concerned with explicitly initiating communication activities, such as sending and receiving messages. Thus, programmers need not be concerned with the particular communication primitives provided by the OS or the network interface.

Playground provides a library of data types that can be published. These data types are divided into three categories: (1) simple types for storing integer, real, boolean, string, and memory block values, (2) tuples for storing records with various fields, and (3) aggregates for storing homogeneous collections of elements. The fields of tuples and the elements of aggregates may be nested arbitrarily using Playground data types.

Playground modules are written entirely in terms of the module's local state information. A subset of the state information relevant to the module's environment is published. Since all Playground communication takes place through the presentation, a module may modify its local state. Likewise, sometimes data in a module's local state may change asynchronously as the result of some activity in the module's environment.

Relationships between the published variables of different modules are established by creating *logical connections* between the variables. The set of connections defines the pattern of communication among the corresponding modules. Connections can be formed as follows:

1. Graphically by the end-user via the Mediator visualization tool (Figure 3),
2. Programmatically within modules using the Playground C++ class library (Figure 4), or
3. From the Application Management System, which provides an automated application launcher that uses configuration files to relaunch configured applications (Figure 5).

The semantics for logical connection communication are FIFO across the connection. This means that if a published variable has value **a** and later is assigned value **b**, every module connected from that data item will see **a** before **b**. The default semantics are "send-on-update." That is, a value is sent on a logical connection only if the published variable is modified within its module.

2.3 Playground Example Application

An example helps to clarify key Playground concepts. One distributed application may be a process control application that manages the production of maple syrup from the sap of maple trees (Figure 6). The process is to heat up the sap until it becomes the right concentration for syrup. During the process the temperature of the sap may need to be lowered to keep from burning the sap and more sap may need to be added occasionally as the concentration increases and the volume decreases.

One Playground module running on one computer interacts with sensors that are needed. These sensors monitor the volume, concentration, and temperature of the sap. The sensor values are published and made available to the environment. Another module on a different computer controls activators for the heating unit and a valve to add sap. This control module connects to

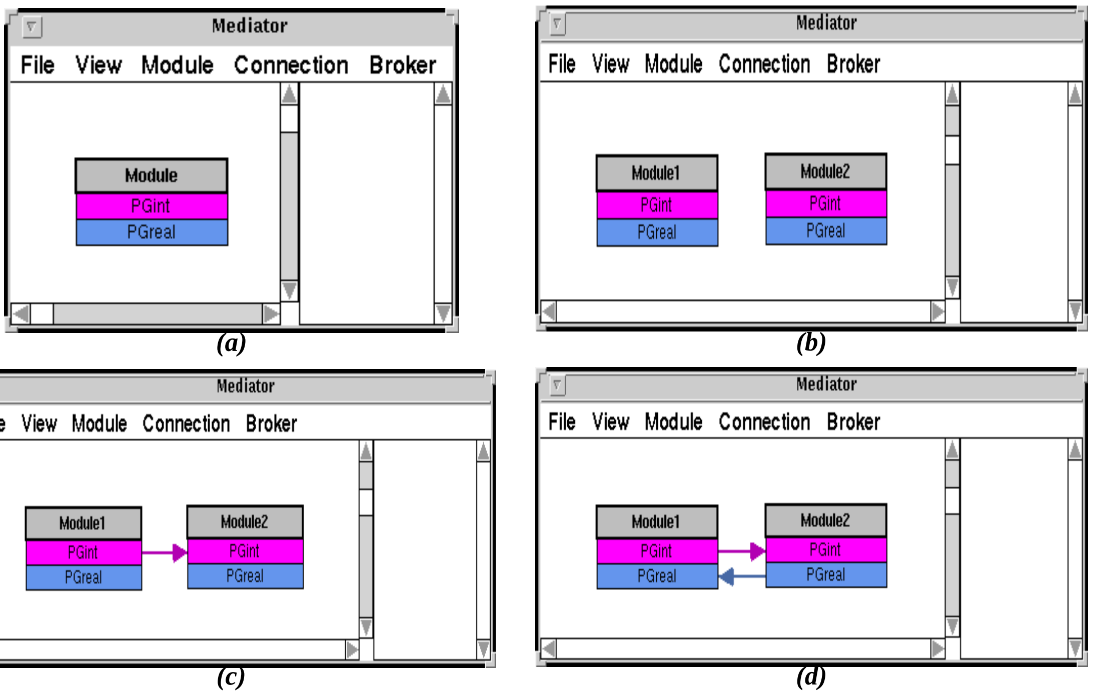


FIGURE 3. (a) Playground module with one published PGint variable and one published PGreal variable as displayed in the Playground visual configuration GUI. (b) Two Playground modules. (c) Connection drawn from Module 1's PGint variable to Module 2's PGint variable. (d) Connection drawn from Module 1's PGint variable to Module 2's PGint variable and from Module 2's PGreal variable to Module 1's PGreal variable. Connections are made by clicking on the source variable and dragging to the destination variable.

```
PG* theVeneer = PG::getVeneer();
theVeneer->connectRequest(module1->getCommunicationID(),
                          pgint1->getNameMember(),
                          module2->getCommunicationID(),
                          pgint2->getNameMember(),
                          PG::UNIDIRECTIONAL);
```

(a)

```
PG* theVeneer = PG::getVeneer();
theVeneer->connectRequest(module1->getCommunicationID(),
                          pgint1->getIndexMember(),
                          module2->getCommunicationID(),
                          pgint2->getIndexMember(),
                          PG::UNIDIRECTIONAL);
```

(b)

FIGURE 4. Programmatically requesting connections between Playground modules. (a) requesting connections via published names. (b) requesting connections via presentation indices

the sensor module's published variables to determine when the sap needs to be heated and when more sap needs to be added.

Another module on a third computer is created to display the process of making maple syrup. It connects to the sensor module's published variables to display the volume, concentration, and temperature of the sap graphically. In addition, it connects to the control module's published variables to display when sap is being added and when the sap is being heated.

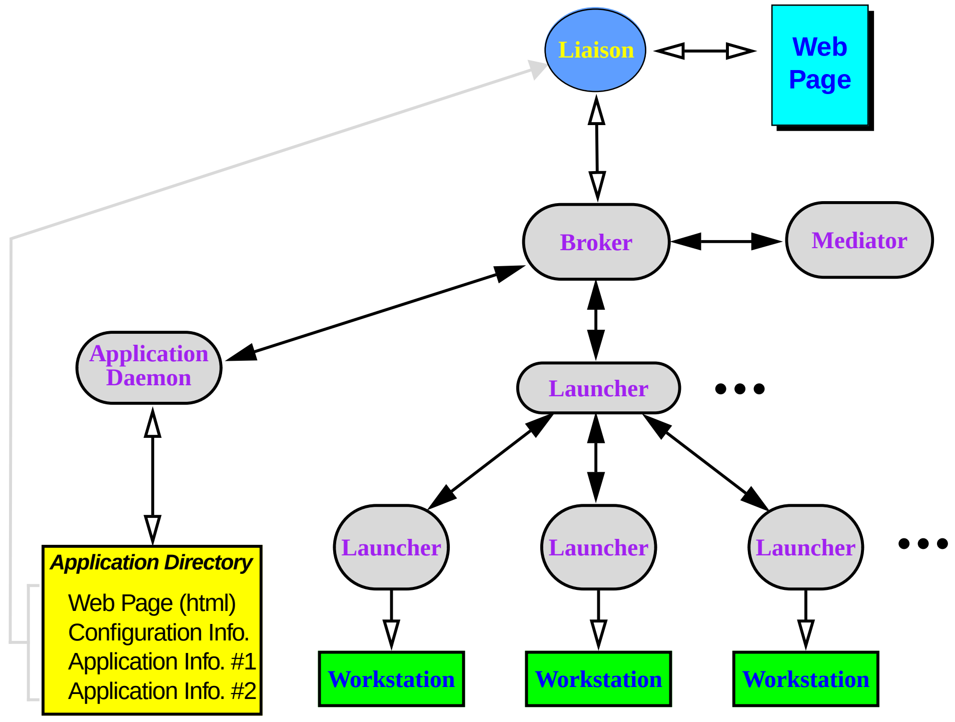


FIGURE 5. Using the application information in the application directory a Playground user can store connections between modules. Later these connections can be made automatically through the Playground GUI or through a Web page (created by the user).

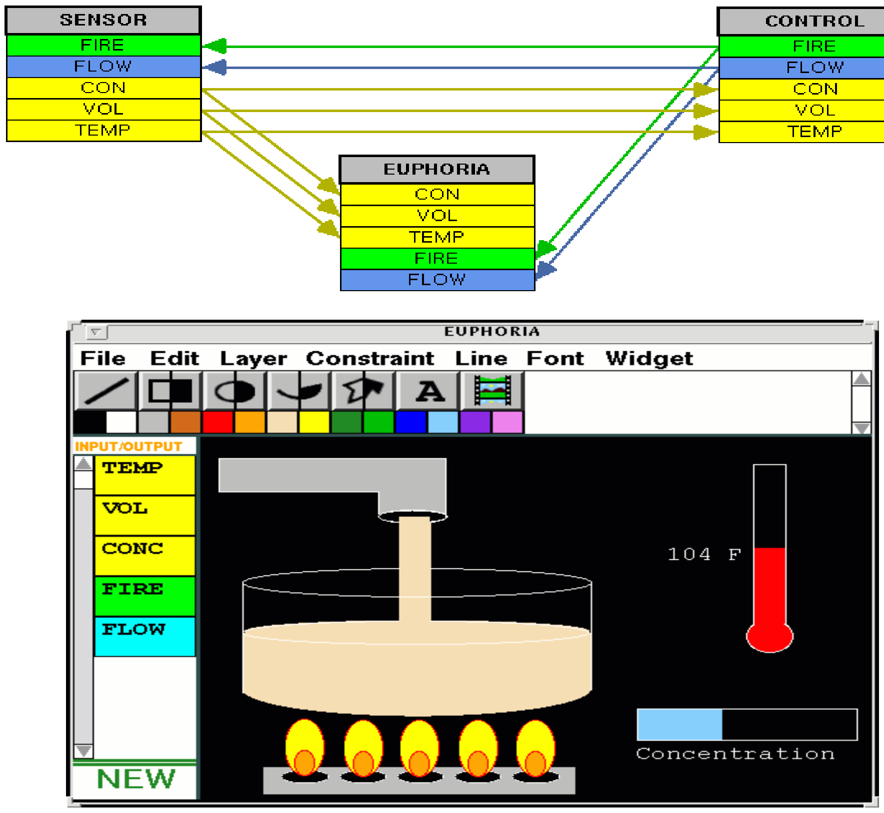


FIGURE 6. Sap Factory Application

2.4 Comparing Playground with other Distributed Programming Environments

A comparison of other DPEs, such as CORBA, DCOM, Java RMI, helps to clarify Playground. DCOM, CORBA, and Java RMI are distributed object models where clients acquire references to distributed objects and then invoke methods on those objects via references. All three environments communicate via remote method calls, which fits well with distributed objects.

DCOM, CORBA, and Java RMI support a *pull model* of retrieval. That is, the client initiates communication between itself and the server. DCOM and CORBA are both programming language neutral, which enhances flexibility and complexity. They include the concept of an interface definition language (IDL) that supports transparent interworking between different programming languages.

Playground has several differences from the environments presented above. (1) Playground is a distributed data model rather than a distributed object model. That is, the data values of the connected objects are distributed rather than the functionality of the objects themselves. Clients receive updates to published variables whenever a server's connected published variable is modified. Thus, the Playground environment implements a *push model*, i.e., the server initiates interaction by sending out updates to clients when a connected server Playground variable is modified. (2) The Playground class library does not support an IDL and is currently implemented only in C++. This does allow the Playground library to take advantage of C++ features such as operator overloading. However, it does not allow the ubiquity of a language-neutral format specification.

3.0 Using Design Patterns In The Playground Library

3.1 The Relevance of Patterns to Playground

Developing distributed applications is hard. There are often multiple types of interprocess communication (IPC) mechanisms that must be supported, as well as multiple operating systems. Moreover, there are often several different communication protocols to transmit desired functionality across components. These complexities must be managed in a flexible and extensible manner.

One goal of Playground is to abstract away low-level complexity. This allows application developers to focus on developing their applications rather than focusing on the low-level details of configuring IPC mechanisms or managing OS calls. Patterns play a crucial role in Playground for abstracting away details and managing the relationships among different components to simplify the development of distributed applications.

3.2 Patterns for Improving Flexibility and Maintainability in Playground

This section discusses several design patterns used to resolve general design forces prevalent in distributed software. The Playground class library illustrates the application of patterns to resolve these forces. The patterns listed in this section are not applicable solely to Playground, of course, and most have been documented in many other application domains.

The intent and usage of the patterns used in Playground is outlined below. Section 3.3 describes these patterns in more depth.

- **The Singleton pattern [4]:** which manages the creation of a single object and provides a global access point to that single object. This pattern resolves the design force of *maintainability* by managing access to a single global object. The Playground software uses this pattern to manage access to the object responsible for functionality at the module level (as opposed to the Playground type/variable level).
- **The Factory Method pattern [4]:** which provides a single method for creating appropriate objects based on the concrete subclass object that defines the factory method without explicitly specifying the subclass name. This pattern resolves the design force of *flexibility* by decoupling objects from the clients that use them. The Playground library uses this pattern to create appropriate incoming message objects and outgoing message objects based on the concrete protocol subclass object that defines the factory method.
- **The Wrapper Facade pattern [5]:** which simplifies the interprocess communication (IPC) programming interface by combining multiple, related IPC calls, like the socket API, into cohesive OO abstractions. This pattern resolves the design force of *maintainability* by abstracting away error-prone details. Playground uses this pattern to create objects that will abstract out tedious and non-portable IPC programming and present a simpler and cleaner interface to the user for managing IPC.
- **The Singleton Factory pattern:** which provides an interface for accessing the single appropriate subclass object (based upon contextual information) from a choice of several subclass objects without explicitly specifying the subclass name. This pattern resolves the design forces of *maintainability* and *flexibility* by managing access of a single global object and decoupling the object from the clients that use it. The Playground software library uses this pattern to select and use the single appropriate OS facade based on the underlying OS on which the Playground module is running. This pattern differs from the Singleton pattern in that there are multiple choices for the singleton object with this pattern. There is only ever one object created but the type of that object will be different for different contexts. The type selection is transparent to the user.
- **The *External* Chain of Responsibility pattern [4]:** which allows the receiver of a particular message to be unspecified and gives the opportunity to handle a particular message to more than one receiver. This pattern resolves the design force of *flexibility* by decoupling the sender of a message with its receiver. The Playground library uses this pattern for processing requests and replies to requests between modules and in handling the events of incoming and outgoing data that take place within a module. This pattern differs from the documented Chain of Responsibility pattern and is qualified as “external” because the order of the objects handling the request is not determined by a containment hierarchy.
- **The Command pattern [4]:** which encapsulates a request as an object allowing for state information to be included with the request. This pattern resolves the design force of *maintainability* by encapsulating all information needed for a message with the message itself. The Playground software uses this pattern to encapsulate messages between modules and to allow these messages to be executed and sent.
- **The Composite pattern [4]:** which composes objects into tree hierarchies that allow uniform treatment of individual objects and compositions of objects. This pattern resolves the design force of *localized functionality* by allowing composed objects to delegate to their composing

elements generically. Playground uses this pattern to allow complex types (such as tuples and arrays) that are composed of simple types (such as integers and floating point numbers) or other complex types to be treated uniformly.

- **The Proxy pattern [4]:** which provides a surrogate object for another object to control access to that other object. This pattern resolves the design force of *flexibility* by allowing arbitrary functionality to be added to the operations of base types. Playground uses this pattern so that local updates of the Playground variables can be sent out implicitly to other connected variables in the environment. It is also used to allow updates from other connected variables in the environment to be retrieved by the variables within a module.
- **The Strategy pattern [4]:** which provides an abstraction for selecting one of several candidate algorithms and packaging it into an object. This pattern resolves the design force of *flexibility* by encapsulating algorithms and facilitating the exchange of them dynamically. This pattern makes it possible in Playground to configure custom strategies that schedule processing of incoming data messages.

Some patterns, such as the Singleton, Wrapper Facade, and Strategy, used in the development of the Playground library were consciously included in the design. However, other patterns were used without realizing they were documented or that those particular design problems had been solved before. The fact that these patterns appeared in Playground shows the relevance and applicability of patterns in managing complexity and flexibility and in solving general design problems for DPEs. It also demonstrates the importance of documenting, cataloging, and disseminating patterns. The remainder of this section deals with each pattern listed in more detail.

3.3 Key Design Challenges Facing Developers

There are many challenges that arise when developing software for distributed applications. The following sections describe specific design challenges common to distributed application development and resolution of these challenges using patterns. The Playground development environment is used to illustrate the challenges and provide a concrete context for the patterns and solutions.

3.3.1 Managing Access of a Single Global Object Using the Singleton Pattern

Context: Often, distributed applications need a single object that is the entry point into the functionality of a component to channel and coordinate functionality at the component level.

Problem: It can be tedious and error-prone to provide global access to a single object and to ensure that one and only one object exists. It can also be non-obvious and non-portable as to how best to handle creation and initialization of this one object, especially in C++.

Solution: Use the Singleton pattern [4]. There is a Playground class (named *PG*) which provides

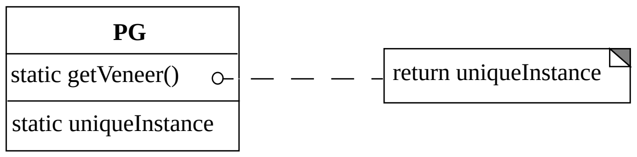


FIGURE 7. The Singleton Pattern

functionality relevant to a Playground module. It provides a facade or “veneer” for the Playground programmer and therefore fulfills the need for a single object that has global access. Although much of the most commonly used functionality in Playground is relevant to Playground variables, some functionality is relevant to a module. Examples of this functionality in Playground are (1) atomically sending out all updated values from any modified Playground variables, (2) checking for any incoming input, and (3) sending out connection requests to other modules. The *PG* singleton object encapsulates this type of module functionality. It uses the Singleton pattern since there is need for only one of these objects. Also, the object needs to be accessed in many different places within the module.

Consequences: The benefits of using the Singleton pattern include easing the management of a class that should only ever have one instance. They also include managing access to that one instance. Retrieving and using the singleton object is made easy and is globally available.

A drawback of using the Singleton pattern is that every time the singleton object is retrieved some processing time is taken to make sure the object has been created and initialized. This checking can be particularly tricky in a multi-threaded environment [6].

3.3.2 Allowing Flexible Protocol Selection Using the Factory Method Pattern

Context: Distributed software should be flexible enough to include or exclude certain pieces of functionality or protocols based upon the needs of the application developer. If a certain protocol is not needed it should not slow down the execution time of the application, nor should it take space in the executable.

Problem: Source code for a particular protocol that is not needed should not be included in compilation and linking of a software library. There should not be any references to any classes or objects relevant to an unnecessary or undesirable protocol. Otherwise, compilation and linking errors will occur when that protocol is not included. Moreover, the protocol should be easily included and used if it is desired. The software should function properly with or without a certain protocol and the programmer should be able to determine this simply by including or excluding object code. However, it can be difficult to support the interfaces for different pieces of functionality generically so that no specific references are included in the base functionality of a library.

Solution: Use the Factory Method [4] to generate specific functionality generically. The Playground library supports different types of protocols for different types of functionality. For instance, the library supports a protocol for requesting connections or disconnections between published Playground variables. The Playground protocols have factory methods to create specific types of messages for specific protocols. The factory method *createOutgoing* creates an initial message object. Likewise, the factory method *createIncoming* is used to reconstruct a

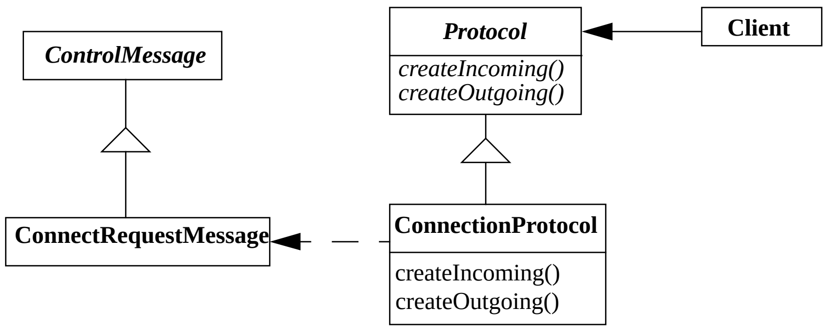


FIGURE 8. The Factory Method Pattern

message object from a streamed message sent by another module. *Protocol* is the abstract class that declares the *createOutgoing* and *createIncoming* factory methods. Each of the concrete subclasses of Protocol define these methods to produce messages specific to that protocol.

Each of the concrete protocol objects can be thought of as a factory that manufactures messages for that specific protocol. In the case of the *ConnectionProtocol* concrete subclass it manufactures messages such as *ConnectRequestMessage*, *ProposedLinkMessage*, *NegotiatedLinkMessage*, and *DisconnectRequestMessage*. In the case of the *AppSignalProtocol* concrete subclass it creates messages such as *KillModuleMessage* and *VerifyModuleMessage* that deal with messages relevant to a module as a whole. Each of the concrete protocol objects are themselves singleton objects since there is only a need for a single factory to create the messages.

A particular protocol registers itself when its object code is included in the executable. When certain functionality is requested by the Playground programmer, the Playground library can check to see if a particular protocol has been included and is therefore available for use.

Consequences: The benefits of using the Factory Method pattern include allowing the Playground library to be decoupled from any protocols that might be included. This facilitates the addition or removal of protocols flexibly when Playground modules are built. It also allows the Playground library to shrink in size when certain protocols are not needed.

A drawback of using the Factory Method pattern is that the Playground library must always check to see if a certain protocol has been included whenever that relevant functionality is requested. It can never assume that the requested functionality is present. If this check is not made errors can occur since invalid protocol objects will be returned when a particular protocol is requested but has not been included.

3.3.3 Encapsulating Low-level IPC Mechanisms Using the Wrapper Facade Pattern

Context: Distributed applications often communicate across heterogeneous hardware and software platforms.

Problem: Different hardware and software platforms support various APIs for interprocess communication (IPC) for similar functionality. Often, these APIs are written in a non object-oriented language such as C. Managing all the low-level details of the IPC interfaces for all the supported hardware and software platforms can be tedious and error-prone. There are certain system calls that need to be made in a certain order to create needed connections. These calls and the order of their invocation may also be OS dependent.

Solution: Use the Wrapper Facade pattern [5]. The *InBandSocketComm* and

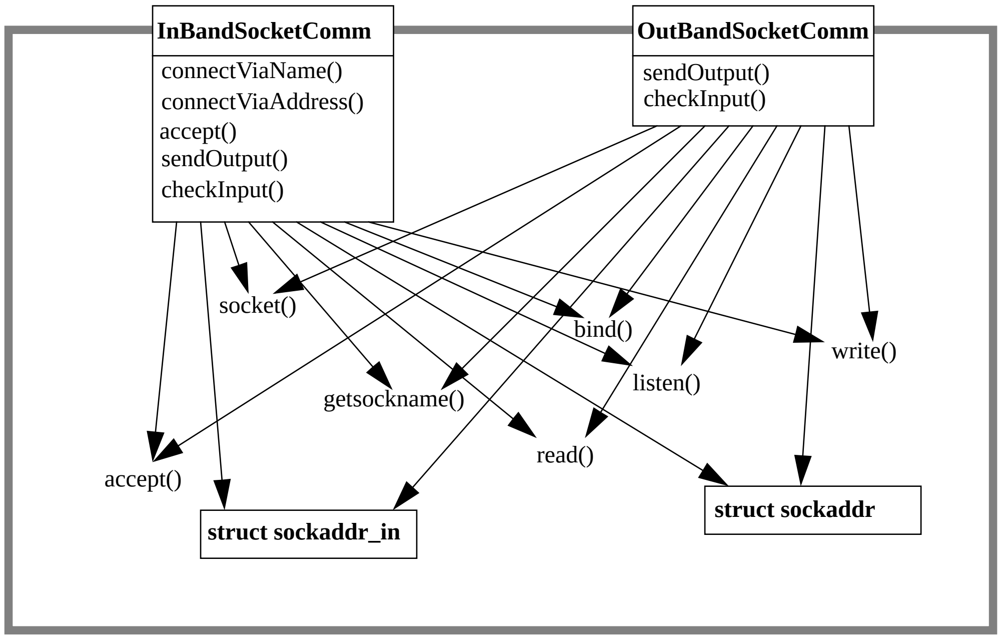


FIGURE 9. The Wrapper Facade Pattern

OutBandSocketComm classes encapsulate socket communication functionality and provide a simpler and cleaner interface for users wanting socket communication. These are the classes whose objects actually make the underlying OS calls to such functions as *socket*, *bind*, *getsockname*, *listen*, *accept*, *read*, and *write* rather than the user having to make these calls.

The *InBandSocketComm* class is used when connections are made between published variables. These connections tend to be long lived (especially as compared to connections used for communication between modules). Therefore the *InBandSocketComm* class has methods to connect and to accept a connection. Once the *InBandSocketComm* objects are connected they simply call *sendOutput* to send out a message or *checkInput* to see if there are any messages waiting to be received. Essentially, these four calls (*connectViaName*/*connectViaAddress*, *accept*, *sendOutput*, and *checkInput*) are all that are needed to send messages between two published variables. Flags can also be passed to the *InBandSocketComm* constructor to determine if the communicator should block waiting for accepts or reads.

OutBandSocketComm provides an even simpler interface. Since *OutBandSocketComm* objects are essentially used to send reliable datagrams the destination of a message is given when it is sent. Therefore the basic communication is encapsulated by the two methods *sendOutput* and *checkInput*.

Consequences: The benefits of using the Wrapper Facade pattern include the fact that the communication interfaces provided by *InBandSocketComm* and *OutBandSocketComm* greatly simplify and clarify communication between modules and variables when using sockets. This pattern also eliminates common errors by encapsulating the intricate details of socket programming [7].

A drawback of using the Wrapper Facade pattern is that encapsulating all related IPC functions into a class can cause a level of indirection that may increase execution time. However, this indirection can be reduced or eliminated without code bloat by judicious use of inlining code.

3.3.4 Managing Multiple OS Interfaces Using the Singleton Factory Pattern

Context: Software supporting distributed applications typically deals with multiple operating systems and their system call APIs. Operating systems such as UNIX, Windows NT, and Windows 95 are prevalent in the market and have a significant installed base. In addition, each OS has strengths and weaknesses over the others, giving it its own niche in application development.

Problem: System calls that provide similar functionality on different operating systems usually have different names and sometimes even different calling sequences. Special casing all these OS calls whenever certain functionality is needed is tedious and error prone. Also, maintaining such a setup is very difficult since the calls may be dispersed throughout the software.

Solution: Use the Singleton Factory pattern. Using this pattern the Playground programmer

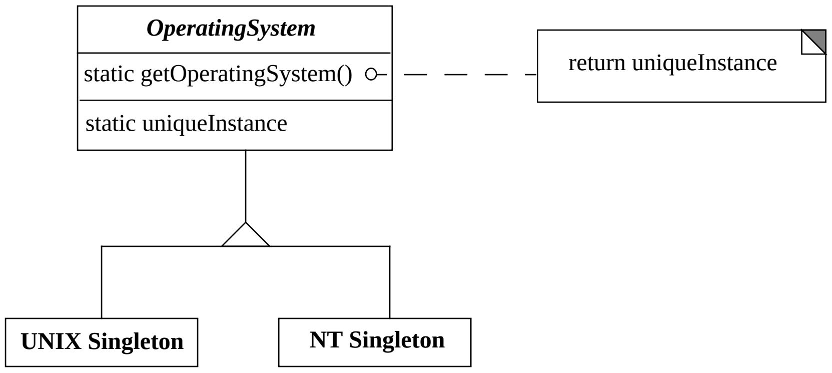


FIGURE 10. The Singleton Factory Pattern

simply requests an OS object from the *OperatingSystem* singleton class and invokes the desired method. The actual object to which the pointer points is OS-specific but the user is shielded from this detail.

The actual functionality of a generic method of the *OperatingSystem* object is mapped to the particular OS call via polymorphism. For example, the interface for the *OperatingSystem* class declares the method *getUserName,* which returns the user's login name. How the username is retrieved differs between the UNIX and Windows NT operating systems so the implementation of this method depends on the current OS context. If the Playground module is being run on the NT OS the *getenv* system call will return the user's name. If the module is being run on UNIX then the *getlogin* system call is made to return the user's name. The context to create the appropriate OS subclass object can be determined using conditional compilation directives for efficiency.

This pattern is really a combination of two documented patterns - the Singleton pattern and the Factory Method pattern [4]. The Singleton pattern is applicable because there's only ever one instance needed for an OS object and it is convenient and helpful to have a single access point for that one object. The Factory Method pattern is applicable because the specific concrete subclass that's created is dependent upon the context (i.e., the OS being run) but the user remains oblivious to the specific concrete subclass created.

Consequences: The benefits of using the Singleton Factory pattern include the simplification of managing different OS platforms since all system calls specific to a particular OS are encapsulated together. Additionally, the user need not know anything at all about the particular OS that might be running and is shielded from having to manage the creation of the appropriate OS object.

A drawback of using the Singleton Factory pattern is the issue of how to handle functionality that one concrete subclass supports and another does not. If the functionality requested can be pieced together from other system calls then the issue can be resolved. However, if there is no way to replicate certain functionality it must either be appropriately designated as such by the concrete subclass (e.g., returning or throwing an error code) or removed from the abstract class and not provided generically to the user. Only the lowest common denominator of functionality can be supported by all the concrete subclasses. Also, making every OS call a virtual function adversely affects execution time.

3.3.5 Handling Requests Using the External Chain of Responsibility Pattern

Context: Processes in certain DPEs must negotiate by sending messages to initialize desired functionality.

Problem: Several negotiations may be occurring simultaneously between distributed components, which means several protocol messages may be passed back and forth between the components at the same time. It can be difficult to manage all the different messages coming in for different negotiations and keep them coordinated with the appropriate functionality being negotiated.

Solution: Use the *External* Chain of Responsibility pattern. Playground modules negotiate and

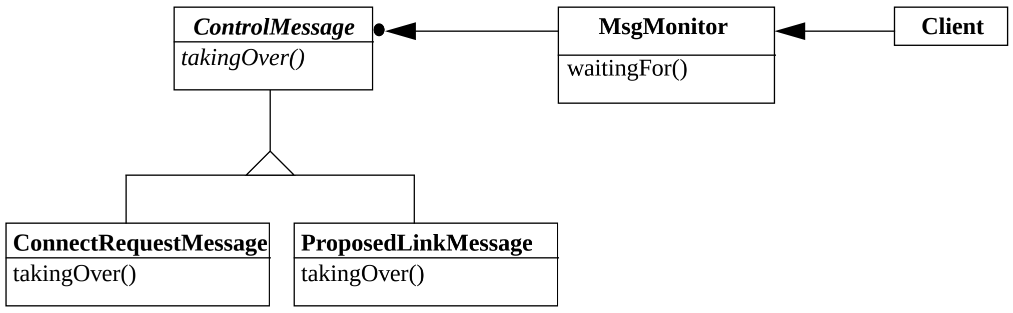


FIGURE 11. The External Chain of Responsibility Pattern

collaborate to achieve certain functionality such as creating connections and removing connections between published Playground variables. In Playground this negotiation and cooperation is achieved using the *External* Chain of Responsibility pattern.

During negotiations, messages are passed between collaborating modules. The initiating module sends a message to a collaborating module and then waits for a reply. Several negotiations can be executing simultaneously which means that several messages in a module may be waiting for replies. A *MsgMonitor* keeps track of all the waiting messages. When a reply message is received, the *MsgMonitor* will query all the waiting messages to see if they want this particular reply message by invoking each message's *takingOver* method. The first message to respond that it wants the reply message receives it for further processing.

The *External* Chain of Responsibility pattern differs somewhat from the documented Chain of Responsibility pattern [4] and is qualified as "external" because the order of the objects handling the request is not determined by a containment hierarchy. It is the *MsgMonitor* object, external to the waiting messages, that determines the order of responsibility.

A good example to illustrate the External Chain of Responsibility pattern is the creation of a connection between two published Playground variables. When a connection is made between variables in two different modules one of the two modules receives a *ConnectRequestMessage*. The module checks to make sure that this connection makes sense from its point of view (e.g., type checking, permission checking, connection property checking). If so, it creates a *ProposedLinkMessage* which will first suspend itself in a *MsgMonitor* to wait for a reply and send itself off to the other module involved in the connection negotiation. When a reply comes back (and more generally when any control message is received) the *MsgMonitor* determines who (if anybody) will handle the newly arrived message. This is where the External Chain of Responsibility pattern is used.

In the connection request example above when a suspended *ProposedLinkMessage* receives its reply it resumes its execution. In the case of receiving a connection status message, it will send a connection status message back to the configuring/requesting module indicating the reason the connection could not be made. Otherwise, it will tell the *NegotiatedLinkMessage* it received to run while it resuspends itself waiting for the result of the *NegotiatedLinkMessage* execution.

Consequences: The benefit of the External Chain of Responsibility pattern is that it allows messages to be decoupled from their receivers (if any) so that they need not manage the overhead of who should process them. The external qualifying functionality of the Chain of Responsibility pattern provides the benefits of the Chain of Responsibility pattern even when no containment hierarchy is available or appropriate.

A drawback of the External Chain of Responsibility pattern relates to processing time. The processing of incoming messages will generally take more time using this pattern because generally some uninterested suspended messages will be queried about an incoming message. On average, for any incoming message received the *MsgMonitor* will query half of all the suspended messages before the incoming message is handled.

3.3.6 Creating Flexible Requests Using the Command Pattern

Context: Functionality between components in a distributed application is often negotiated by sending requests and replies between the components.

Problem: There may be several different IPC mechanisms available to send a request. Managing all the possibilities when sending out a message can be complicated. Packaging up the information needed to send out a request can be tedious. There may also be a substantial amount of information needed to properly process a request that has been received. It can be confusing and error-prone to keep track of all this information.

Solution: Use the Command pattern [4]. In Playground, modules make requests between

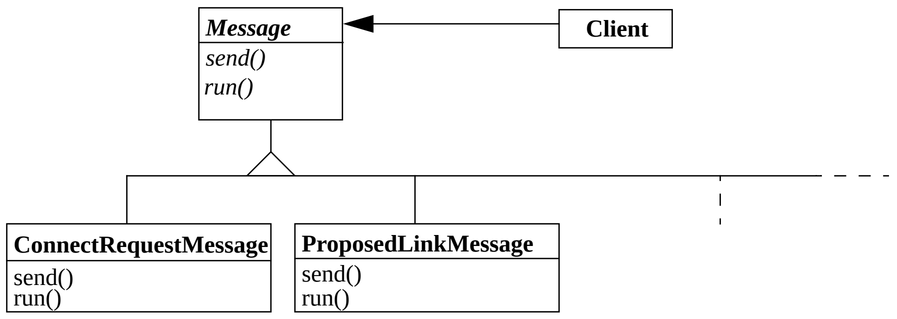


FIGURE 12. The Command Pattern

themselves and other modules. These requests are subclasses of the *Message* class which adds a common interface of *send* and *run* methods. When a module sends a *Message* it does not need to know specifically what the object is or how it should be packaged and shipped. However, the *send* method, defined for all *Message* subclassed objects, knows how to perform this. Likewise, when a module receives a Playground *Message* object it does not need to know specifically what the object is or how it should be processed. The *run* method, defined for all *Message* subclasses objects, will perform the appropriate functionality. Additionally, since a *Message* has state information it can store needed information to be able to undo itself or to perform whatever functionality might be appropriate in a given situation.

Consequences: The benefits of using the Command pattern include making commands (typically some functionality to be performed, a verb that denotes action) more flexible. With commands as objects it is easier to undo commands, send commands to other modules, and run commands once they are received by a module. The Command pattern provides a convenient encapsulation of state that may be needed for later processing.

A drawback of using the Command pattern is that reifying a request into a command object may take more space. This is because all the state information is now always bundled with the request. A command object may be too heavyweight for some applications that send simple requests and do not expect replies or need to keep state.

3.3.7 Managing Various Typed Objects Uniformly Using the Composite Pattern

Context: A software library supplying data types needs to offer several different data types for the application developer to use. There might be the need for simple data types such as integers and floating point numbers as well as the number for complex data types such as tuples and mappings. Also, the complex types should allow arbitrarily complex elements.

Problem: With different data types and their unique interfaces there is still functionality and interface methods that need to be common so that the can be treated uniformly for certain interactions. It can be tedious and confusing to manage both the interface methods unique to a particular data type and the interface methods applicable to all data types.

Solution: Use the Composite pattern [4]. Playground supports many different types of variables

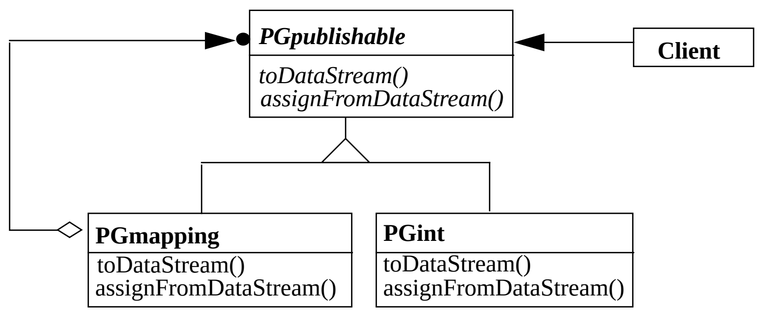


FIGURE 13. The Composite Pattern

including complex types like *PGmapping*s, *PGtuple*s, and simple types like *PGint* and *PGreal*. The aggregate and tuple types can be composed of any Playground type, which can itself be composed of any Playground type. The depth of this nesting is unlimited by Playground and makes possible very flexible and complex types.

Any Playground type, whether it's a simple type, an aggregate type, or a tuple type can be treated uniformly for certain operations such as assignment from the environment and streaming/marshalling. Thus, whether a Playground variable is a simple type (e.g., *PGint*) or a complex type (e.g., *PGarray* of *PGmapping* of *PGtuple*) the same method can be invoked on it to transmit updates to the environment or to assign updates from the environment.

The power of the composite pattern is shown when a complex type receives a call like *update*, which transmits an update to the environment. The *update* method for a complex object will do its own needed processing and then simply iterate through its contained objects and invoke the *update* method on those objects. There is no logic needed to differentiate between the contained objects being simple types like *PGint*s or complex types like *PGmapping*s of *PGtuple*s. When the *update* method is invoked the concrete subclass knows how to handle this call appropriately.

Consequences: The benefits of using the Composite pattern include allowing objects of different types (e.g., simple types and complex types) to be treated uniformly even when only the supertype is known or desired in order to keep objects and classes decoupled. For complex objects this uniform treatment can easily be recursive so that all elements in a complex object are treated uniformly.

A drawback of using the Composite pattern is that it adds an additional abstract superclass level to all the concrete classes that may make the concrete classes more heavyweight. Virtual methods are also implicit in the Composite pattern, which may impact execution time negatively.

3.3.8 Managing Implicit IPC Using the Proxy Pattern with C++ Operator Overloading

Context: In distributed applications communication between distributed components should be as simple and as transparent as possible to the user.

Problem: IPC interfaces, even for a common technology such as sockets, vary substantially from hardware platform to hardware platform and from software platform to software platform. Managing the updating of a variable's value in the logic of a running program along with the sending out of any updates across an IPC mechanism can be complicated and error prone.

Solution: Use the Proxy pattern [4]. Playground variables are essentially C++ base types and

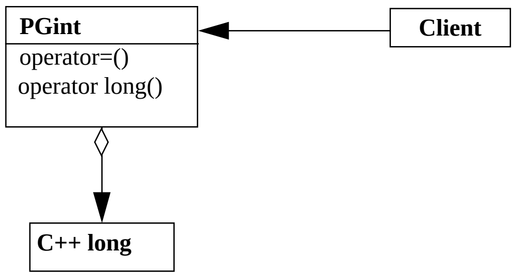


FIGURE 14. The Proxy Pattern

common container types that have distributed functionality added to them. From the Playground programmer’s point of view the Playground types can be treated just like the C++ base types for functionality like assignments and comparisons. The Playground types are proxies or go-betweens for C++ base types that add the functionality of letting the values of the objects instantiated from these Playground types be known outside of the module that created them.

When methods such as assignment or comparison are invoked on Playground variables the Playground variables act as proxies in that they send the assignment or comparison on to the C++ base type but also check for incoming data updates when the value is accessed and send out data updates when the variable is modified programmatically.

Consequences: The benefits of using the Proxy pattern include allowing Playground to enhance the functionality of standard operations of C++ data types so that implicit functionality can be executed.

A drawback of using the Proxy pattern is that it adds a level of indirection that can increase the size of an executable and adversely affect its execution time.

3.3.9 Allowing Flexible Processing Using the Strategy Pattern

Context: For a software library applicable to a wide variety of distributed applications, different approaches to handling certain functionality will be appropriate for different types of applications.

Problem: There is a need to allow the flexibility of selecting and replacing different algorithms to manage certain functionality. This selection and replacement of algorithms may even need to be done at run time.

Solution: Use the Strategy pattern [4]. The Playground library includes a singleton object, the *InBandCommRegistry*, that manages all the individual communication objects sending and receiving data messages. The *InputManager* class is the abstract superclass that defines the

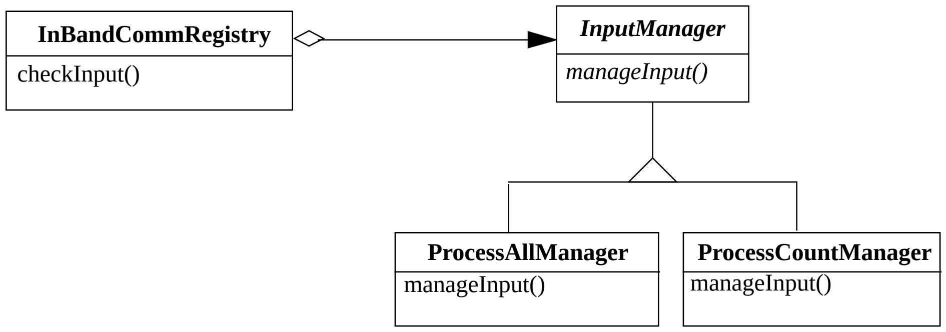


FIGURE 15. The Strategy Pattern

interface for all the concrete strategy objects to be used. The *InBandCommRegistry* class uses whatever *InputManager* subclass object is currently registered to process incoming messages.

Currently, only the *ProcessCountManager* subclass is defined which processes at most only a certain number of incoming messages from each communication object before relinquishing control. However, it is anticipated that other strategies will be needed in the future such as processing as many incoming messages as possible at one time. Different algorithms can easily be swapped in and out at run time depending on the current requirements of the running module.

Consequences: The benefits of using the Strategy pattern include allowing flexibility in regards to algorithms that are used. Different algorithms can be changed at compile time or even run time easily and quickly.

A drawback of using the Strategy pattern is that it adds a level of indirection that can increase the size of an executable and adversely affect its execution time.

4.0 Conclusion

Patterns have been very beneficial in the management of design problems in the Playground environment. They have abstracted out and encapsulated low-level and distracting detail to facilitate clarity on higher-level design issues. They helped the developer organize classes and objects to make designs more flexible and maintainable. They localized areas of design that are complex and subject to modification so that those areas can be modified and maintained more cleanly and easily.

Patterns have also shown the trade-offs that are made when tackling design problems. There are no perfect solutions to conflicting requirements. A balance between requirements sometimes must be made. Often, a designer must choose between several design possibilities. Patterns are helpful in illuminating trade-offs between these different design options.

This article showed how patterns can be used to enhance the flexibility, extensibility, and maintainability of software that supports DPEs. In particular, this article showed that patterns were a significant aid in the development of the Playground class library. The article also underscores the ubiquity of patterns in resolving general design issues. Finally, this article has stressed that the concepts and patterns presented are not solely applicable to Playground but are relevant to a wide range of application domains.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[2] D. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, Ram Sethuraman, “The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications”. *IEEE Transactions on Software Engineering*, 21(9):735-746, September 1995.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[5] D. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible and Maintainable ORB Middleware”, Washington University, Department of Computer Science, 1997.

[6] D. Schmidt and T. Harrison, “Double-Checked Locking - An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently,” in *Pattern Languages of Program Design 3* (R. Martin, D. Riehle, and F. Buschmann, eds.), Reading, MA: Addison-Wesley, 1997

[7] D. Schmidt, “IPC SAP: An Object-Oriented Interface to Operating System Interprocess Communication Services,” *C++ Report*, Vol. 4, November/December 1992