Design Patterns for Marine Biology Simulation

Dung "Zung" Nguyen
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3835
dxnguyen@rice.edu

Mathias Ricken
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-1940
mgricken@rice.edu

Stephen Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3814
swong@rice.edu

ABSTRACT

We specify and implement a GUI application that simulates marine biological systems by making extensive use of object-oriented design patterns.

The key design patterns are model-view-control, observer/observable, visitor, command, factory method and decorator. These design patterns help delineate the roles and responsibilities of the objects in the system, establish loose coupling between objects and arrange for the objects to communicate and cooperate with one another at the highest level of abstraction. The result is an application that exhibits minimal control flow, yet is powerful, robust, flexible and easy to maintain.

Our work entails a non-trivial redesign of the current AP Computer Science Marine Biology Simulation case study and may serve as a case study for an introductory "object-first" curriculum.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming.

General Terms

Design.

Keywords

AP Computer Science, object-first, design patterns, closure, inner class, lambda, abstract coupling, loose coupling, pedagogy.

1. INTRODUCTION

Starting in the 2003-2004 academic year, the AP Computer Science curriculum will migrate from C++ to Java with emphasis on object-oriented programming (OOP) [1]. This precipitates the need for first year college curricula to accommodate the upcoming crop of high school graduates with such AP credits. In our effort to upgrade materials for our object-first introductory curriculum, we seek to develop a case study that, in order to be effective, must not only prove to be relevant and challenging, but also build on the student's newly acquired understanding of Java and OOP.

Since the AP Marine Biology Simulation (APMBS) case study [1] is required for all AP students, it is a good candidate for us to explore and expand. This paper presents the result of our work: a similar application that, in comparison, puts much more emphasis on interfaces, abstract classes, design through abstract decomposition and polymorphism. In the discussion that follows, we assume that the reader is familiar with the APMBS and the concepts of OOP, in particular those expressed in the common design patterns presented in [2].

We take the point of view that an OO program should use message passing and exploit polymorphism to keep procedural control flow to a minimum and, as a result, should be declarative in nature. Loose and abstract coupling between collaborating objects is critical in building software that is correct, robust, and easy to maintain due to its flexibility and extensibility. Our case study illustrates how design patterns help achieve these objectives.

Section 2 presents the overall design of the system. As a GUI application, it is based on the well-known model-view-controller (MVC) pattern. The model itself is a system of cooperating objects that interact with each other at the highest level of abstraction by means of messages. Section 3 illustrates how the loose and abstract coupling between objects facilitates the task of maintaining invariants to ensure correctness of the system behavior. Section 4 discusses robustness. To withstand misuse without sacrificing flexibility, robustness cannot simply be patched up by external control code but must be inherently supported by the system structure. Section 5 demonstrates the flexibility and extensibility of the system.

Interested readers can find the complete source code at http://www.exciton.cs.rice.edu/research/SIGCSE04.

2. DESIGN OVERVIEW

The overall specification of the Marine Biology Simulation (MBS) is too involved to completely describe in this short paper. We will present only some key requirements and an overview of the main architecture with a few highlighted implementation details.

The MBS is to be a simulation of the movement of marine life ("fish" – though not technically restricted to such) in a 2-dimensional environment. Fish have a notion of direction and location at all times, though this does not imply that fish know where they are with respect to anything else in the simulation. Fish determine their own movement within the constraints imposed by their immediate surroundings. For instance, a fish cannot move to a location that is blocked.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'04, March *3*—7, 2004, Norfolk, VA, USA.
Copyright 2004 ACM 1-58113-798-2/04/0003…$5.00.

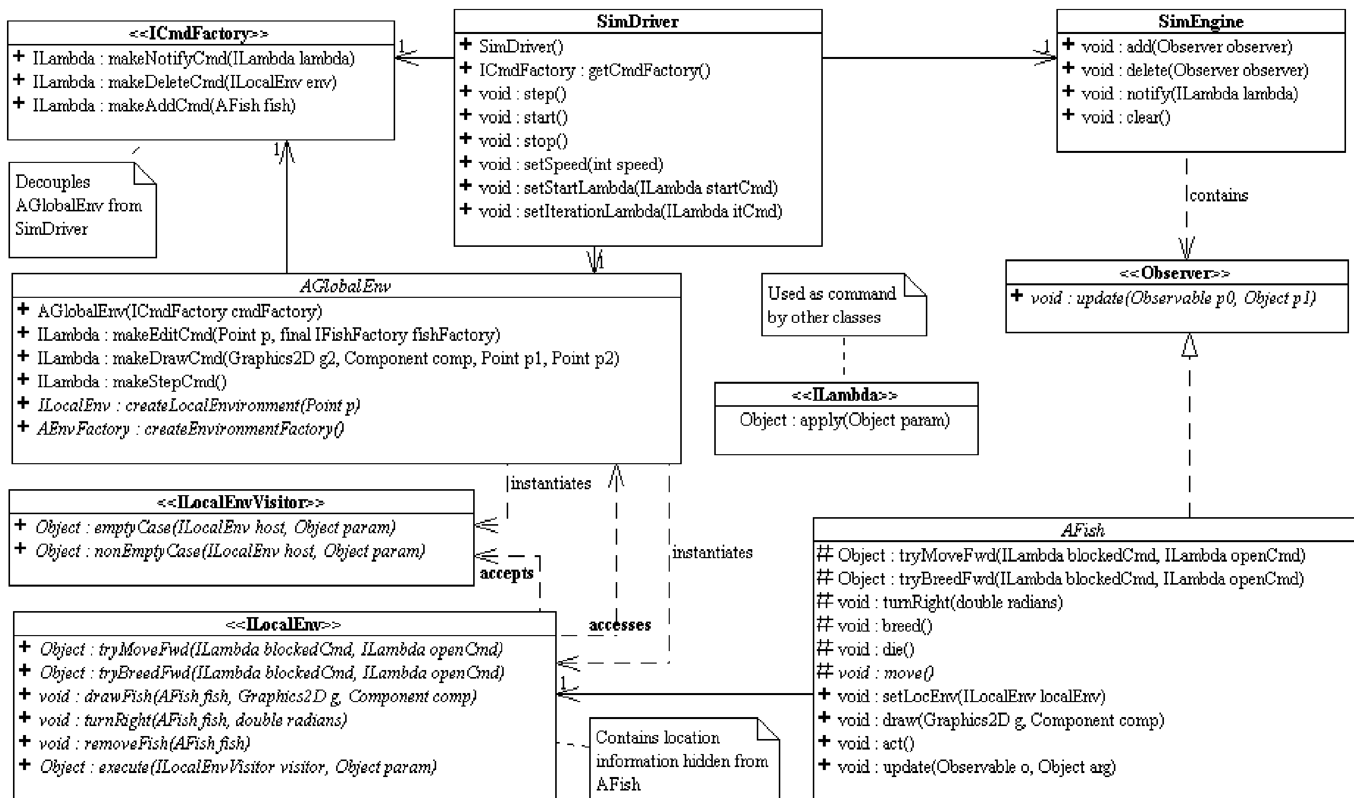

Figure 1. Model Architecture.

Figure 1 shows the UML class diagrams for the key classes in the simulation model. Fish are the main acting agents in the simulation and are encapsulated in an abstract class called *AFish*. The possible actions of a fish depend not on the entire global environment, but only on the subset of that environment immediately around the fish, the "local environment". The local environment of a fish has knowledge of whether or not the space in front of a fish is open and is modeled by an interface called *ILocalEnv*. A fish is only given access to an individualized *ILocalEnv*, which encapsulates coordinate position and direction and keeps them inaccessible to the fish. UML sequence diagrams in Figure 2 and Figure 3 depict how a fish interacts with its local environment: When a fish tries to move forward (tryMoveFwd) or spawn another fish (tryBreedFwd), it provides its private local environment with two actions, blockedCmd for when the forward path is blocked and openCmd for when it is open. These actions are examples of "commands" in the command design pattern and are instantiated as anonymous inner objects of an interface called *ILambda*. *ILambda* has one method, *Object apply(Object param)*, and represents the abstract notion of functions. The associated *ILocalEnv* either

- calls blockedCmd.apply(null) or
- instantiates an appropriate *ILambda*, moveCmd, and calls openCmd.apply(moveCmd). The application of moveCmd, at the discretion of the fish, moves the fish by changing its associated local environment. The moveCmd is a decorated *ILambda* which allows the environment to disable it and prevent its misuse.

Passing commands as anonymous inner objects in the above protocol enables the fish to perform environment-dependent behaviors by cooperating with the local environment and yet remain only loosely coupled with it. Below is a listing of the move method of a fish to provide an example of how command passing is used to control the movement of a fish. This fish moves forward as long as the forward path is open and turns around 180 degrees if the path is blocked. *AFish*.tryMoveFwd simply delegates the call to *ILocalEnv.tryMoveFwd*. *AFish*.tryMoveFwd is a final protected method to prevent the concrete fish subclass from directly accessing the local environment.

```
/**
 * Execute the movement part of a simulation step.
 */
public void move() {
    // attempt to move forward
    tryMoveFwd(new ILambda() {
        public Object apply(Object param) {
            // cannot move forward
            // turn PI radians
            turnRight(Math.PI);
            return null;
        }
    }, new ILambda() {
        public Object apply(Object param) {
            // can move forward, do it
            ((ILambda)param).apply(null);
            return null;
        }
    });
}
```

The above code demonstrates how commands can be used to provide flexible yet decoupled communication in a secure and robust manner.

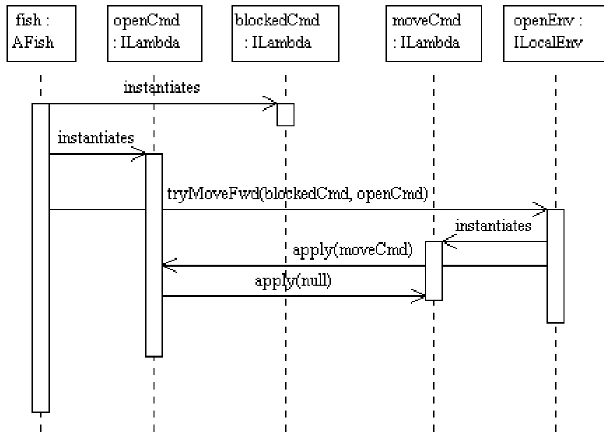

Figure 2. Command passing for moving into an open local environment.

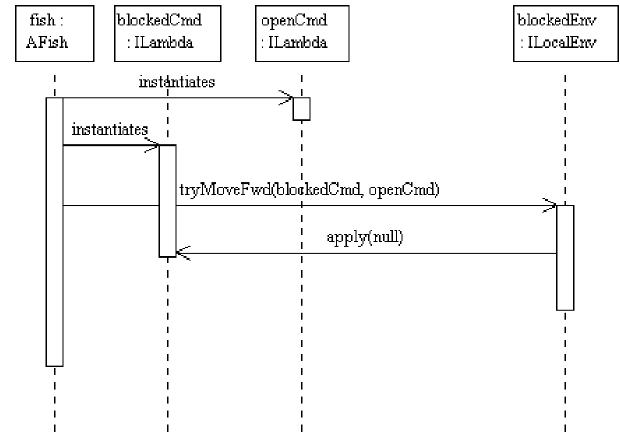

Figure 3. Command passing for moving into a blocked local environment.

There are global constraints, though, that govern the behaviors of the local environments, e.g. discrete or continuous movement and boundary conditions, which can be encapsulated in a global environment class, *AGlobalEnv*. Global environmental control also eases communication between local environments necessary to determine open vs. blocked conditions. The visitor pattern allows the global environment to add, edit and locate fish: *AGlobalEnv* employs an *ILocalEnvVisitor* to perform the appropriate task depending on whether or not the local environment is attached to a fish.

When a fish is initially created, it does not yet have a local environment, but it is already subject to global constraints. At the time the fish is added to the simulation, the global environment is asked to create a local environment for the fish that enforces such constraints; thus the global environment acts as a factory for *ILocalEnv*. By creating the instances of *ILocalEnv* as inner objects, the local environments gain private access to the global environment.

The MBS runs as a sequential synchronous process, where each agent (fish) is allowed to perform its behaviors once per time tick. Therefore *AFish* implements the Observer interface, which enables it to be connected to a central command dispatcher, SimEngine. This class is independent of the simulation and serves only to send each agent an *ILambda* command that, when applied by the agent, will effect the desired behavior, e.g. acting or drawing. An act command and a draw command are sent out by SimEngine to all fish at every time tick of the simulation. These commands are double-dispatching, akin to a degenerate, single-host visitor [3]. When the agent applies the command, it passes itself in as the input parameter. The command's apply method then calls the desired method on that particular fish, such as act() or draw(). This scheme provides a flexible, extensible and robust means of controlling the agents' behaviors.

To decouple the *AGlobalEnv* from the SimEngine, the SimDriver utilizes techniques similar to the command passing interactions between AFish and *ILocalEnv* described above. The SimDriver needs to construct a compound *ILambda* from behaviors specified both in the environment and the driver itself, e.g. the environment's ability to draw and the driver's ability to notify the agents via the SimEngine. This necessitates the use of a factory, which creates *ILambda*s from environmental data or behavior. Space limitation prevents a detailed discussion of this process here.

Figure 4 shows the overall model-view-controller (MVC) architecture of the MBS where

- the view, MBSView, is the graphical use interface,
- the model, is the main simulation driver, SimDriver, that coordinates the global environment and the simulation engine and
- the controller, MBSController, is the coordinator that sets up the system and instantiates and installs the adapters needed to connect the view to the model.

We now proceed to discuss our design in terms of correctness, robustness, flexibility and extensibility. We also contrast it with the current APMBS to illustrate how our use of message passing, polymorphism and loose and abstract coupling leads to a design that meets the aforementioned criteria.

3. CORRECTNESS

Correctness of a program means the program behaves and performs as specified. In order for a program to achieve correctness, it must maintain its invariants. In the APMBS, the fish has a location that must always be synchronized with the location kept in the environment. This replication of data produces a tight coupling between the two objects that makes enforcing the synchronization invariant difficult. The responsibility for maintaining this invariant is partially assigned to the fish, which is a variant component of the system. The designers of the system have no control over the variant code in different fish subclasses; therefore, it is impossible to *a priori* enforce the invariant and ensure program correctness.

In contrast, in our design the coupling between the fish and its environment is loose and abstract. The location is stored and managed by invariant code in the environment. The fish does not have access to it and therefore is unable to do anything the environment does not specifically allow. Thus, the environment strictly enforces its constraints, guaranteeing correct behavior.

The command, factory and visitor patterns are crucial in creating the necessary decoupling to maintain invariance but still allow interaction. Additionally, the null object pattern [4] provides well-defined behavior for uninitialized components.

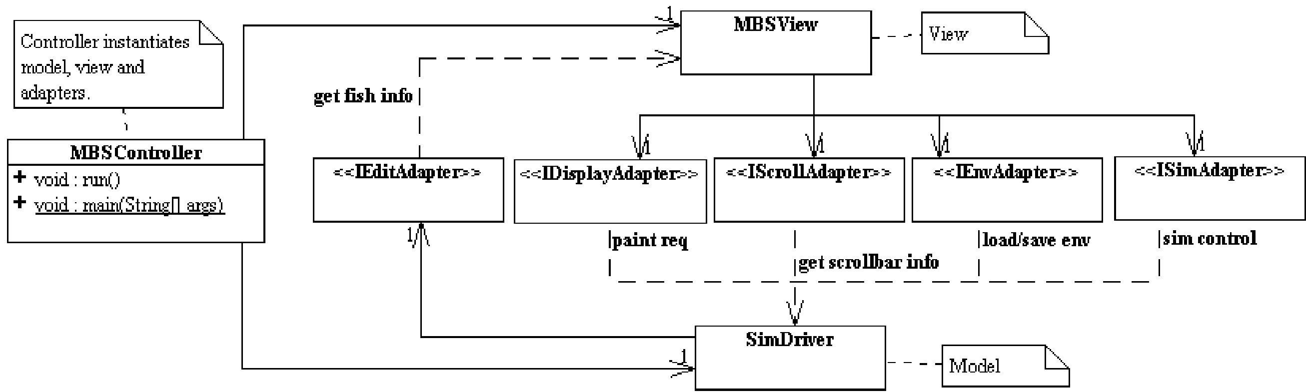

Figure 4. MVC Architecture.

4. ROBUSTNESS

Robustness is the ability to withstand abuses and other unexpected input while continuing to function in a well-defined manner.

The APMBS imposes certain requirements, e.g. that fish can only move to adjacent locations or that the destination must not contain another fish, but does nothing to enforce them. Since the fish are at liberty to violate these requirements, a maliciously or erroneously designed fish can crash the APMBS. For instance, a fish that simply sets its location to valid but random coordinates clearly violates the adjacency requirement, yet is still accepted by the system even though it quickly crashes it. Such a fish will not even compile in our system.

The adjacency requirement in the APMBS, for example, exists only as a coding style stipulation and as after-the-fact value checks. It is not intrinsically built into the system. Inherent robustness is a result of clear delineations of responsibility and knowledge that are enforced by the very structure of the system. This prevents objects from performing operations that are outside their domain and removes the need for programmatic value checks. In our system, the local environment encapsulates the movement information and behavior. It simply provides a command to move a fish the proper amount forward. The fish then can only use or ignore that command, either way, forcing it to be in accordance with the system's requirements. Such robustness is achieved by careful and precise analysis of the components of the original problem.

Robustness without sacrificing a high degree of flexibility and extensibility requires loose and abstract coupling. Loose coupling requires that objects interact with other objects at very high abstraction levels, preventing them from performing potentially dangerous low-level operations. Once again, command, factory, and visitor patterns are central in creating the necessary abstraction.

5. FLEXIBILITY AND EXTENSIBILITY

A flexible system is easy to modify without breaking other components of the software; it has the proverbial capability to "bend without breaking." We can make reasonable changes to the system without breaking all the pieces of the system. That is, a small change in one class only causes a manageable number of changes in other classes.

An extensible system is easy to augment with new capabilities without modifying any of the existing components. We should be able to add more (unforeseen) functionality to the system without incurring any changes to existing code.

Both flexibility and extensibility are based upon modularity. In a modular system the classes organized in a cohesive manner and in such a way that they can be developed and tested in small and independent units. Flexibility and extensibility require modularity, but the reverse does not hold because code cannot simply be modularized in an *ad hoc* manner. A system must be decomposed into the abstract representations of its components. Only if the modules represent the proper abstractions will flexibility and extensibility be achieved.

There are numerous examples of proper abstraction leading to flexibility and extensibility in our MBS:

- The menu options for simulation behavior are now commands and thus can be extended without necessitating changes in the control logic. In the APMBS, on the other hand, `if`-statements have to be written for each option, forcing a change in the control logic whenever the options are changed.
- Environments that are decoupled from the rest of the system mean that new and vastly different environments can easily be added, not only statically but also at runtime. The process of creating a global environment is realized by a series of factory methods. This factory-based process enables environments to create their own options panels, to use an arbitrary number of parameters and to bootstrap themselves. The APMBS lacks this capability and is rigid in that it restricts the environment to only quadrilateral bounded or unbounded regions.
- Existing fish can run in wildly different environments, such as both discrete and continuous coordinate systems, even though they may have been programmed with only one system in mind. The fish in the APMBS do not have this capability since the simulation is not scalable on the environment side. Locations, for example, are hard-coded as pairs of integers for all environments, making it impossible to add environments with continuous movement.
- Complex compound behaviors are easily achieved by composing the modularized behaviors in our system. Since the fish and the environments are decoupled, a new fish or environment can be added without affecting the other class hierarchy. This results in a code complexity that scales linearly. If the APMBS were to support more different kinds of environments, such as a non-discrete environment, the

tight coupling would lead to changes in both hierarchies and quadratic growth.

6. CONCLUSION

The Marine Biology Simulation presented here is intended as a case study for use at or near the conclusion of an objects-first college-level introductory curriculum. It is not intended as a tool for introducing object-oriented concepts. Currently at our institution, at the end of our first year curriculum, we use a final project of similar complexity that requires the same level of OOP/OOD knowledge and skills. Our plan is to replace the current final project with the MBS.

We strongly believe that an objects-first approach that emphasizes abstract decomposition of problems, polymorphic flow control and delegation-model programming will properly prepare students for this case study as well as large-scale OOP. This approach will equip the students with a solid foundation in programming and design fundamentals such as abstract classes and interfaces, separating the variant and invariant pieces of a problem, identification of responsibilities, and core design patterns such as command, state, composite, strategy, factory and visitor. Our case study illustrates how object–oriented design using appropriate design patterns helps express real-world problems in declarative and abstract forms. This reinforces the knowledge and skills learned throughout the course.

It is very important to take students through the problem analysis portion of the development process. In object-oriented design, a crucial ability is to be able to simply and clearly express the abstract interactions that are occurring. It does not suffice to merely show students the code for an OO system such as the MBS. Simple "make another one of these" programming assignments will not illuminate the true nature of the system.

Uncovering the intrinsic way in which objects interact is a vital skill that OOP students need to develop. The MBS is set up to lead students through hotly contested issues such as whether or not a fish "knows" its location. In the end, the students discover that "location" is related to *local* environment-dependent behavior, not to an x-y coordinate value. Debating how decoupled objects can cooperatively interact through abstract message passing is a very useful exercise to explore the nature of programming under the constraints of strictly delineated roles and responsibilities.

The full power of OOP is not realized until the size of the problem or system grows beyond simple "toy" exercises. In the beginning, students need to work with small, easily managed systems. An objects-first approach can utilize scalable techniques that teach "coding in the large" concepts, but on a more comprehensible scale. The MBS project immerses the students in sufficient complexity to accentuate the strengths of OOP. The students easily see the power, utility, flexibility and scalability of the design. The concepts running through the MBS are applicable in a diverse field of problems, which can strongly motivate students to expand their horizons and tackle larger, more complex problems.

7. REFERENCES

[1] Java Marine Biology Simulation Case Study, The College Board, http://apcentral.collegeboard.com.

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

[3] Nguyen, D. and Wong, S. Design Patterns for Decoupling Data Structures and Algorithms, *SIGCSE Bulletin, 31, 1* (Mar 1999), 87-91.

[4] Woolf, B., The Null Object. *Pattern Language of Program Design 3*, Martin, R. C., Riehle, D., and Buschmann, F. eds., Addison-Wesley, 1998.