Design patterns for games

Conference Paper *in* ACM SIGCSE Bulletin · March 2002

2 authors:

Dung X Nguyen
Rice University

17 PUBLICATIONS **146** CITATIONS

Stephen B. Wong
Rice University

23 PUBLICATIONS **865** CITATIONS

Some of the authors of this publication are also working on these related projects:

Rice Marine Biology Simulation View project

Proteus View project

Design Patterns for Games

Dung ("Zung") Nguyen and Stephen B. Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
dxnguyen@rice.edu, swong@rice.edu

Abstract

Designing a two-person game involves identifying the game model to compute the best moves, the user interface (the "view") to play the game, and the controller to coordinate the model and the view. A crucial task is to represent the algorithms used in their highest abstract forms. This can prove to be a difficult and subtle endeavor. It is made easier however by the use of good object-oriented (OO) design principles to help maintain a high level abstract viewpoint during the design process. The state pattern is used to represent the game board and is coupled with the visitor pattern to provide state-dependent, variant behaviors. The min-max algorithm and its variants are implemented as the strategy pattern, which allows flexible and extensible computational capability. Careful analysis is performed to properly abstract the interactions between the model and the view. The command design pattern is used to provide callback services between the model and the view. Leading students through this design process enables them to learn algorithms and architectures that easily scale to full-sized applications.

1 Introduction

Standard data structures and algorithms courses cover game trees, and in particular, the min-max principle, to compute the values of the game tree nodes and select the best next move. Even though these algorithms are initially described in high-level abstract terms, this is not reflected in their standard implementations (See for example, [1].) Instead the algorithm code is intertwined with low-level game board implementation details. This not only obscures the essence of the solution, but also makes it impossible to re-use in different types of games. Students end up writing essentially "throw-away" code.

To remedy this situation, we seek to design an OO model for two-person games that enables us to express all facets of the games at the highest level of abstraction. The result is a system of cooperating objects that possesses many of the valued qualities in software engineering: correctness, robustness, extensibility, flexibility, and reusability. We lead students through this design process to teach them about abstract problem decomposition, how design patterns can shape their thinking in OO design, understanding system architectures, and expressing algorithms in their most abstract forms.

We apply the Model-View-Controller (MVC) pattern to the overall architecture of the program. The MVC pattern calls for decomposing the overall system into three subsystems: the model, the view, and the controller. Section 2 details the analysis of the model, which consists of the game tree and the various algorithms that compute the next moves from a given tree node. Section 3 focuses on the interactions between the view and the model and the interfaces through which they communicate. And finally in Section 4, we illustrate our design with the familiar game of Tic-Tac-Toe.

We should stress here that the development of an OO game system must be part of a larger OOP/OOD pedagogy. Before starting such a project, students must be versed in basic OO principles as well as the core design patterns such as the state, strategy, command and visitor patterns [2].

2 The Model

In this paper, for simplicity, we will restrict our discussion to two-dimensional game boards, with only two players. The rules of a game define the legal board configurations plus the winning and the draw (if any) configurations. They are encapsulated by an interface called *IBoardModel*, which abstractly "knows" how to transition from one legal board configuration to another and reject any request for an illegal move. Different concrete implementations of *IBoardModel* represent different types of games. There are a variety of algorithms to compute the next move from a given board configuration. They are abstracted into an interface called *INextMoveStrategy*. The pair, *IBoardModel* and *INextMoveStrategy*, and their interactions constitute the model of the game. A single game management class, GameModel, that holds

instances of the above two interfaces can represent the total game. The relationship between GameModel and *INextMoveStrategy* is called the strategy pattern. Separating the rules of the game and the strategy to make a move in this manner provides the flexibility to apply the same strategy to different types of games, and vice-versa. The discussion that follows will further elucidate this point.

The interaction process between *IBoardModel* and *INextMoveStrategy* is independent of whether the game is Tic-Tac-Toe, Othello, checkers, go, etc. The interaction between the players (human and/or computer) and the board can thus be expressed in abstract terms. The player requests to make a move, which the board either accepts or rejects. Once the move is accepted, the game progresses into one of several states. These states, which are states of the board, and not of the overall game management process, are that player #0 has won, player #1 has won, the game is a draw, or that the outcome has yet to be decided. The game then proceeds in a manner that depends on the current state of the board. This includes terminating the game, or requesting that another move be made.
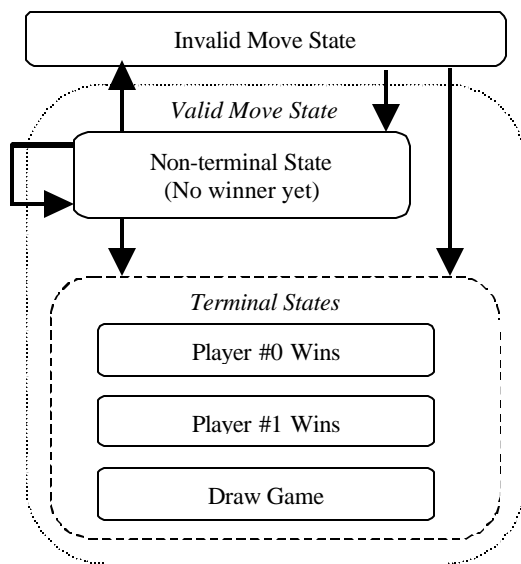

Figure 1. Game board state diagram

The overall state diagram of the board in Figure 1 shows that the board consists of both concrete states and superstates. The state design pattern can be used to model these states. Superstates are depicted as abstract super-classes of the concrete states they contain. At the top level of the hierarchy is an interface called *IBoardState*. The visitor design pattern is used to provide state-dependent abstract behaviors (For a complete discussion of this technique, see [3]). The visitor interface *IBoardStatusVisitor*, contains methods corresponding to the various states of the game board and the board provides a means for executing ("accepting") those visitors. Only

the method associated with the current state of the board is called when the board accepts the visitor.

While it is extremely important to understand the state nature of a problem before attempting to craft a solution, it is not always necessary to represent all the states as classes. When the algorithm used to determine the state of the system completes its task, a visitor's appropriate method can be called immediately, without actually setting the system into a static concrete state. We refer to this as a "virtual state" as the visitor believes that the system has transitioned into a concrete state because its corresponding method was called, when in fact, no actual concrete state exists. There are limitations to this approach however. In particular, it is really only useful if the system expects to immediately transition out of that "virtual" state. Here, the only states that satisfy that restriction are the invalid move state and the valid move superstate.

This design exploits direct dispatching via polymorphism and results in game management code that is devoid of the usual tangle of conditionals needed to determine the game's behavior under different circumstances. Instead, the different cases are encapsulated into separate methods of the visitors and can be thought of as services proffered by the game manager to the board.

To find the best next move, a player would assign values to the children states of the current state and select the one with the highest value as the next move. The function $V(s)$ that computes the value of a state s is commonly based on the min-max principle, which can be expressed as follows:

$V(s) =$
 - ?? 1, if s is a winning state for that player
 - ?? 0, if s is a draw state
 - ?? -1, if s is a losing state for that player
 - ?? Either:
   - o $\max\{V(c) \mid c$ is a child valid move state of s$\}$, if that player moves next.
   - o $\min\{V(c) \mid c$ is a child valid move state of s$\}$, if the other player moves next.

A common approach to computing the max and min is to iterate over the available states. This involves casting the set of available states into some sort of linear ordering. But the statement of the min-max principle is *not* dependent on any linear ordering of the set. It simply prescribes a recursive application of $V(s)$ onto the set. This is akin to the higher order function called "map", familiar to functional programming practitioners. We thus express the min-max algorithm in terms of a map function rather than iteration. Since the actual map function is independent of the algorithm, it belongs as a service provided by the game board, and not as an integral part of the algorithm. The map method provided by the *IBoardModel* takes an abstract "lambda" object, *IBoardLambda*, and a generic

input parameter. The map() method only depends on whether or not the board's state is terminal, hence *IBoardLambda* has two corresponding methods: *apply()* for the non-terminal state and *noApply()* for the terminal states. The map method applies the lambda (with its input parameter) to all available valid move states from the current state. The lambda asks the board to make a move which also checks the status of the board, and recurs if necessary by asking the board to map it again. For efficiency reasons, it is best to accumulate the current extremum value and corresponding best move during the mapping. The responsibility to carry out this task is assigned to an abstract accumulator class, *AAccumulator*. It has an abstract *updateBest()* method that polymorphically performs the min or max determination both trivially and transparently, takes care of saving the maximum value (for this player) or the minimum value (for the other player) and its associated move and stores the result. It also has a factory method, *makeOpposite()*, to instantiate an accumulator for the other player.

Figure 2. UML class diagram of the model (implemented methods have been omitted.)

The min-max algorithm can now be formulated as a class *MinMax* that implements *INextMoveStrategy*. Its method to compute the best next move is rephrased in the following OO terms.

Ask the *IBoardModel, board*, to map the following recursive "helper" *IBoardLambda*, onto all valid moves from the current state with an appropriate initial *AAccumulator*, say *acc*.

1. Ask *board* to make the current move, passing it an *IBoardStatusVisitor* to do one of the following tasks associated with a possible resulting state.
   - ?? Player won: pass the current move and the value +1 to *acc.updateBest()*.
   - ?? Player lost: pass the current move and the value -1 to *acc.updateBest()*.
   - ?? Draw: pass the current move and the value 0 to *acc.updateBest()*.
   - ?? Non-terminal state:
     i. Ask *board* to map this helper algorithm onto the now available valid states with a new accumulator, *nextAcc*, for the other player.
     ii. Pass the current move and the value obtained from *nextAcc* to *acc.updateBest()*.
2. Ask *board* to undo the current move.

When the algorithm is finished, the initial accumulator, *acc*, will hold the best next move with its corresponding value.

The min-max algorithm formulated as shown does not rely on any concrete implementation of *IBoardModel*. Thus it can be used with any concrete type of games, as advertised earlier in the discussion. It involves no external iterative loops, and closely approximates the abstraction of the min-max principle, making the code easier to write, more robust, more likely to be correct, and more flexible than traditional implementations. The Java code for the above OO formulation is almost a word-for-word translation.

3 Model-View Interactions

When setting up a MVC system, one must abstract both the model and view to discover the essential interfaces that they must present to each other. Novices are often tempted to create elaborate interfaces with every bell and whistle available. This approach often violates the encapsulation of which behaviors are strictly in the model or view's domain. In a two-person board game, the model has only three public methods: it must be able to accept a request to make a move by a player to a particular location on the board, reset to an initial condition, and be told which player it represents. On the other hand, the view has four public methods: it must be able to be notified of a valid move, be

told that a specific player has won, be told that the game ended in a draw, and reset to an initial condition. In addition to these interactions, the model needs a mechanism for communicating a rejected move request back to the view.
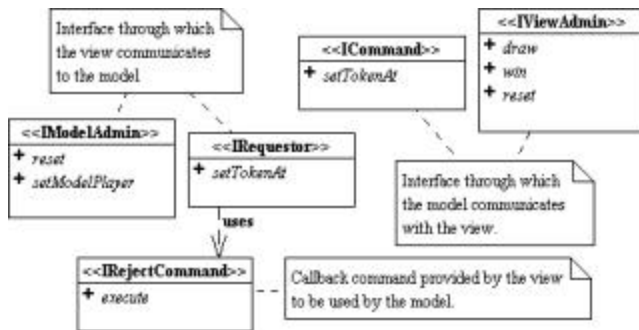

Figure 3. Model-View interfaces

From their respective viewpoints, the model and the view each hold two "commands" in design pattern parlance; a move-making command, *IRequestor* and *ICommand*, and an administrative command, *IModelAdmin* and *IViewAdmin* (see Figure 3). From the larger perspective that includes both the model and the view we see that these commands are actually adapter design patterns which translate the interface the model/view expects to see of the view/model into the actual public interface presented by each. Commands and adapters decouple senders from receivers, here, the view from the model. A discussion of the dynamic behavior of this system is best illustrated with a UML sequence diagram, regrettably omitted here for space considerations.

One of the major advantages of a MVC design, the ability to transparently change the view on a model, is easily demonstrated with our system. A GUI-type interface simply uses event-handlers to execute the request command. For instance, suppose each Tic-Tac-Toe square was displayed as a button. Each button's event handler makes a direct call to the request handler, *IRequestor.setToken()*. Note that no conditionals are needed because each button intrinsically knows its own row and column information. In turn, the GUI view responds to the model calling its *ICommand* object's *setToken()* method by placing the appropriate mark on the specified row and column on the screen. But one can easily replace the GUI view with a console-type view which parses command line input and calls its *IRequestor* object's *setToken()* with the correct information. Likewise, it can respond to the model's call to its *ICommand* object's *setToken()* by typing an appropriate message on the console.

Our MVC implementation exploits the power of inner classes in Java. We use factory methods in the model and the view to dynamically instantiate the *IRequestor* and

ICommand objects respectively as anonymous inner classes. These inner classes have private access to their *closures*, which are their instantiating hosts, even though they are "installed" into another object. Thus the *IRequestor* and *ICommand* objects have direct, internal access to the model and the view respectively. Likewise, the rejection callback command, *IRejectCommand*, is implemented as an anonymous inner class instantiated by the view and passed to the model during its call to the *IRequestor* object's *setToken()* method. If the request is rejected, the *IRejectCommand* has direct access to the view, *in the context in which request was made*. This makes context-sensitive error messages trivial because the *IRejectCommand* knows the specific information pertaining to the request.

The controller's role in the MVC is to "wire" the view to the model. Often this involves instantiating adapters to patch the view and model together. Inner classes for adapters are a favorite again for their unique scoping capabilities. In our implementation however, since both the model and the view are factories for the required interfaces, plus they directly implement the administrative interfaces, the controller's wiring job is trivial.

4 Tic-Tac-Toe Example

In our implementation (Figure 2), GameModel generically models a two-player board game. GameModel does not contain any code specific to Tic-Tac-Toe. It merely moderates the interactions between the board, *IBoardModel* and the strategy to compute the next move, *INextMoveStrategy*. GameModel.getRequestor() is the factory method to instantiate the *IRequestor* for the view. GameModel directly implements the *IModelAdmin* interface.

We implement a Tic-Tac-Toe specific game board, TicTacToeBoard, using the state pattern coupled with the visitor pattern. The board states, as described in Figure 1, are modeled as sub-types of an interface, *IBoardState*. The terminal superstate, *ATerminalState*, is represented as an abstract parent to the concrete Player0WonState, Player1WonState, and DrawState. The invalid move state and valid move superstate are virtual states.

Communication to/from the GameModel and the *IBoardModel* is handled via three main visitors: *IBoardStatusVisitor*, *ICheckMoveVisitor*, and *IBoardLambda*. *IBoardStatusVisitor*, with its four cases corresponding to the four concrete states of the board, is accepted by the *execute()* method of *IBoardModel*. Thus *IBoardStatusVisitor* can be used to implement any generic algorithm on the board that depends on its state. *IBoardModel.makeMove()* takes row/col, and player inputs as well as both an *ICheckMoveVisitor* and an *IBoardStatusVisitor* parameter. The *ICheckMoveVisitor*

is used to provide behavior depending on the virtual invalid move state *vs.* the virtual valid move superstate. Typically it is used to notify the view whether or not a move request was accepted. After the board accepts and performs a move, the board may change state. Thus the *makeMove()* method will automatically execute the supplied *IBoardStatusVisitor* after the move is performed. During normal game play, this is how the view is informed if the game was won/lost or ended in a draw. The min-max strategy uses this to either determine the value function's return or to recur to the next level.

Listing 1 shows the code to implement the GameBoard.getRequestor() method, illustrating the use of anonymous inner classes, and how code normally implemented with conditionals to differentiate between board states is replaced with simple listings of cases. The single line of code in *IRequestor.setToken()* simply delegates the call to the board's (boardModel's) makeMove() method. But in doing so, it creates the visitors the board uses to communicate the resulting state information all the way back out to the view. The *ICheckMoveVisitor* and the *IBoardStatusVisitor* are thus two of makeMove()'s five input parameters.

```
public IRequestor getRequestor() {
  return new IRequestor() { //user makes move using requestor
    public void setTokenAt(final int row, final int col, final
                  int player, final IRejectCommand rejectCommand) {
      boardModel.makeMove(row, col, player,
        new ICheckMoveVisitor() {
          public void invalidMoveCase() {
            rejectCommand.execute(); // Tell view
          }
          public void validMoveCase() { // Tell view
            iCommand.setTokenAt(row, col, player);
          }
        },
        new IBoardStatusVisitor() {
          public Object player0WonCase(IBoardModel host,
                               Object param) {
            viewAdmin.win(0);  return null; // Tell view
          }
          public Object player1WonCase(IBoardModel host,
                               Object param) {
            viewAdmin.win(1);  return null;  // Tell view
          }
          public Object drawCase(IBoardModel host,
                               Object param) {
            viewAdmin.draw();  return null;  // Tell view
          }
          public Object noWinnerCase(IBoardModel host,
                               Object param) {
            makeMove(player);  return null;  //Computer's turn
          }
        });
      }
    };
  }.
```

Listing 1. *IRequestor* factory method

Because they are inner classes, GameModel is the closure of the *IRequestor* object as well as the

ICheckMoveVisitor and *IBoardStatusVisitor* the *IRequestor* contains. The mediation between the various components of the system is taking place through that closure while retaining a proper decoupling of components.

5 Conclusion

This exercise of designing a game system demonstrates the power of properly abstracting a problem and then of matching the code to that abstraction. The benefits of algorithm simplification with increased robustness and flexibility are readily apparent. Tangled stacks of conditionals are replaced with declarative statements listing the relevant cases derived from the state analysis of the problem. The OO design process entails identifying the variant behaviors of the system, encapsulating them into abstract subsystems and decoupling them from the invariant core. Design patterns are used extensively to achieve the proper abstractions and decouplings. The result is a system that can be easily modified, upgraded or adapted. The students are able to scale-up the ideas learned here into full-sized software projects. For instance, a fun exercise for the students is to have them adapt their Tic-Tac-Toe code to handle Othello instead. They discover that the only variant in the system is the board--the rest is unchanged! The lesson is that generic concepts such as game management and min-max principles should lead to generic applicability, not single-use, "throw away" code. The students learn "programming-in-the-large" while studying a system that is still small enough to manage.

A side benefit to the OO design used here is a demonstration of the integration of OO and functional ideas. Concepts such as map, lambda functions, and closures, normally relegated only to courses in functional languages, are shown to be fundamental computer science principles that span programming paradigms.

Games are an effective vehicle for teaching students abstraction and the OO design process. The students easily see the power, utility, flexibility and scalability of the design. This strongly motivates them to expand their horizons and tackle larger, more complex problems.

References

[1] Weiss, A., *Data Structures and Problem Solving Using Java*, Addison-Wesley, 1998

[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, March 1999, 87-91.