Really Awesome Team Name
Project: Mastermind

Richard Laughlin, Daniel Wallach, Aaron Damrau
Jason Greaves, Jonathon Shippling

April 5, 2012

Part I

Design Narrative

0.1 High Level Architecture

Our team's design is based primarily around the Model-View-Controller Architecturual pattern. It is important to note that we use a variant of this pattern which groups the View and Controller together into a unified GUI. This was done because it is how Java's Swing is designed, and we are using java as our programming language. We chose Model-View-Controller as the overall architecture because it separates the concerns of the system into distinct groups

We can group our code into three main categories: The GUI, The Event System, and The Model.

0.2 The GUI

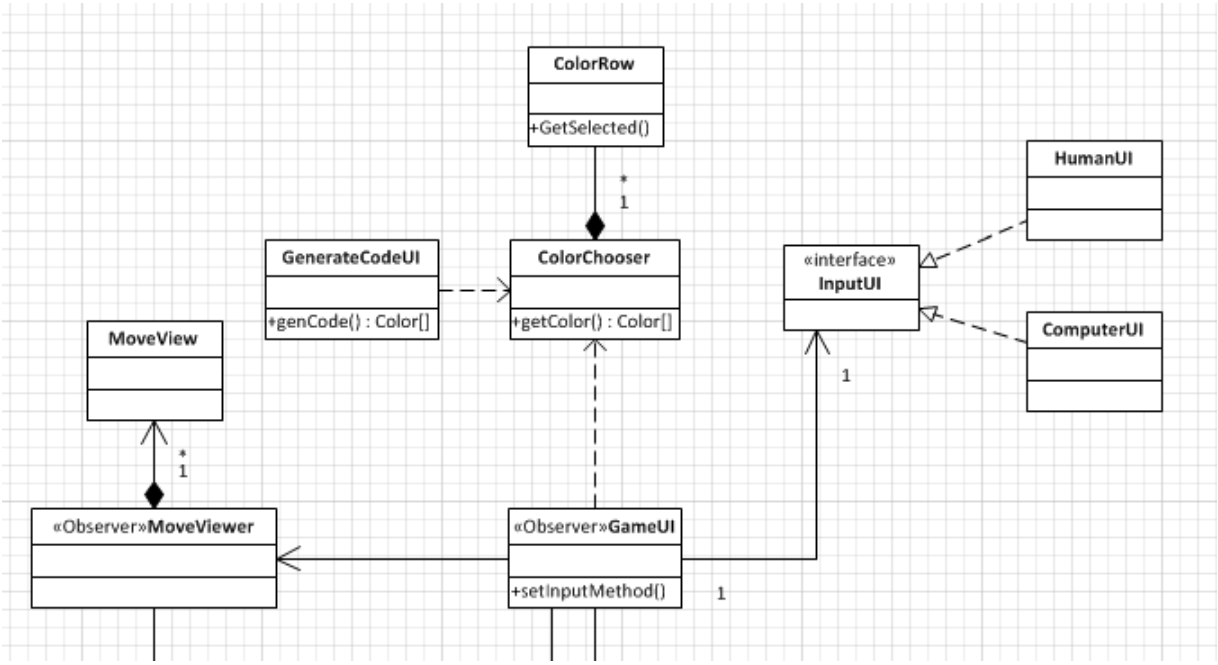


Figure 1: A closeup view of the GUI design.

This system makes up all of the classes that are used to display the interface to the user, and to process their input. When the system first starts a GameUI is created. The GameUI instance then opens a GenerateCodeUI window to fetch the secret code from the user. This secret code is then sent via the event system to the model. More information on exactly how this is accomplished is discussed in the following section. Once this process is finished the GameUI then prompts the user for log file related information and again sends this information to the model via the event system. Finally the GUI is ready to begin accepting input for the actual game.

The GameUI window is composed of three panels: A MoveViewer panel which is used to display the current moves of the game, and two InputUI panels which are used to display the interface for the users that are currently playing the game. The MoveViewer panel is also made up of several sub-panels of the MoveView class, one for each possible slot on the game board. We chose to implement the GUI in this way because it allowed us to create classes that focus on specific tasks in place of a massive, singular GameUI class.

The InputUI interface is implemented by two classes: HumanUI and ComputerUI. The HumanUI displays ColorChooser objects which allow the user to make and submit color code sequences for processing. By contrast, the ComputerUI simply displays a message to the user to indicate the computer is currently thinking. We implemented this section in this way because it allows us to show the UI for only the user currently taking his turn. It also allows us to dedicate specific UI elements to the codemaker and codebreaker.

The GUI is connected to the rest of the system by means of the Observer pattern, and the event system. When the message is leaving the GUI for the model, the event system is used to route the event to the intended target. By contrast, when the message is incoming (e.g. an update) the message is processed by the GameUI. We chose to do this because it created single path in and out of the GUI. This prevents unnecessary coupling between seemingly unrelated classes.

0.3 The Event System

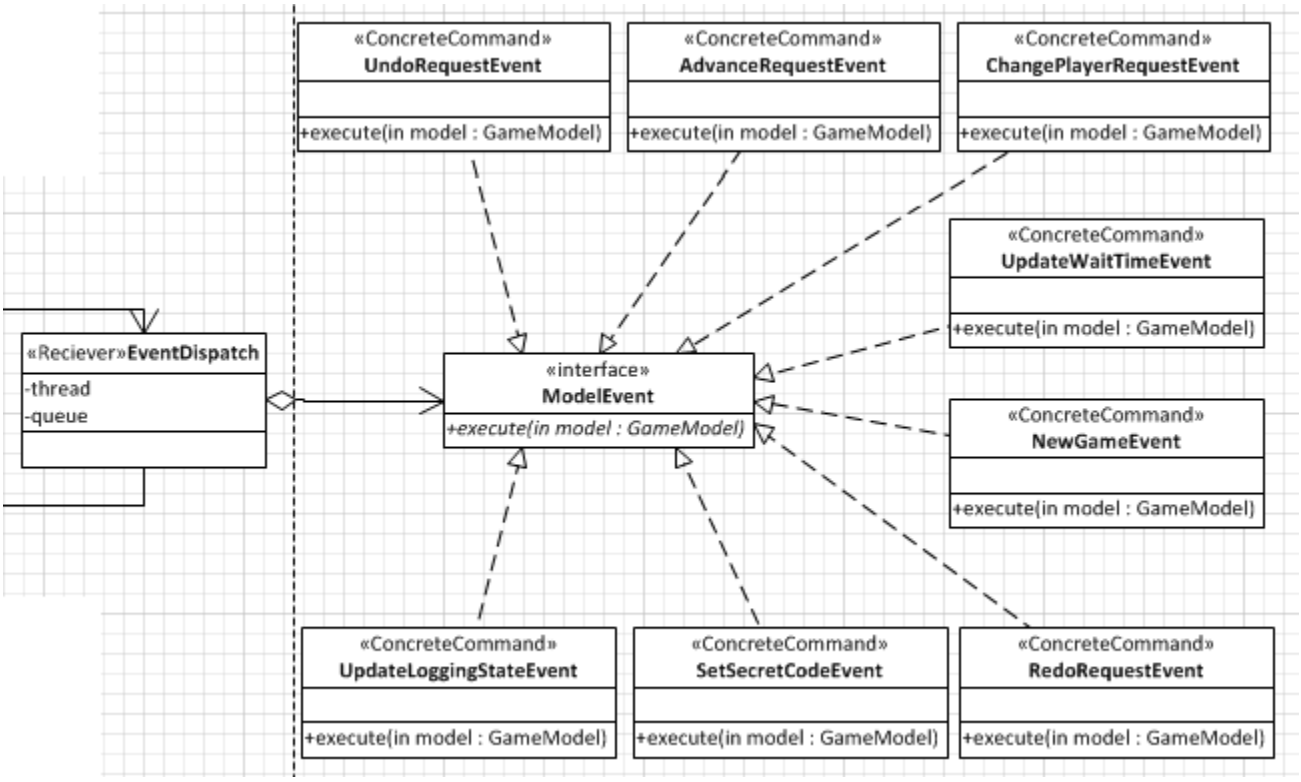


Figure 2: A closeup view of the event system.

This system is important because it allows the view to queue and dispatch events to The model on a different thread than the GUI thread. We implemented this to prevent screen lag or stalling while the system processes the actual game logic. It also has the benefit of disconnecting the sender of an event from the receiver. This allows the source to ignorant of what actually sent the event in question.

The ModelEvent interface is a great example of the command pattern. Each of the ConcreteCommands implement a single execute method which takes a GameModel (discussed in the next section) as a parameter. This allows each command to execute a particular set of function calls on the GameModel. The EventDispatch class queues incoming ModelEvent objects for later processing. These events originate in the GUI, but could also originate in other more interesting places such as a network. The event dispatching thread continuously polls the queue and invokes the events on the GameModel. We chose to implement this system in this way because it allows external sources to perform common functions such as advancing the game, undoing a move, redoing a move, setting the secret code or logging filename without having to know anything about the internals of the GameModel. This is done because it meets the requirements of the Law of Demeter, giving us the freedom to change the internals of the model in any way we choose without affecting a large amount of code.

0.4 The Model

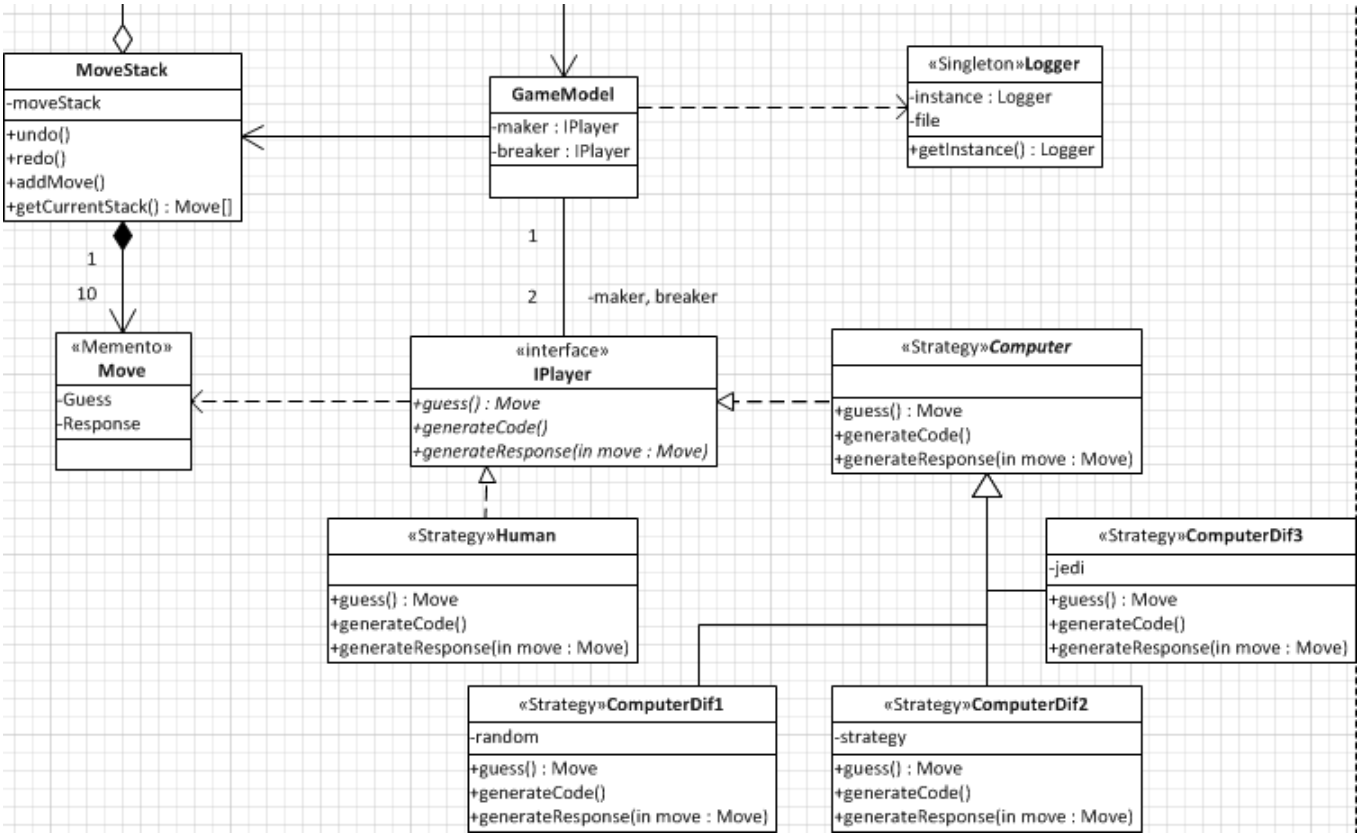


Figure 3: A closeup view of the model.

The model subsystem is the fundamental piece to the system. It is in this collection of classes where game logic is actually defined and used. The main player in this group is the GameModel class which contains a MoveStack, and two IPlayer instances (the codemaker, and codebreaker).

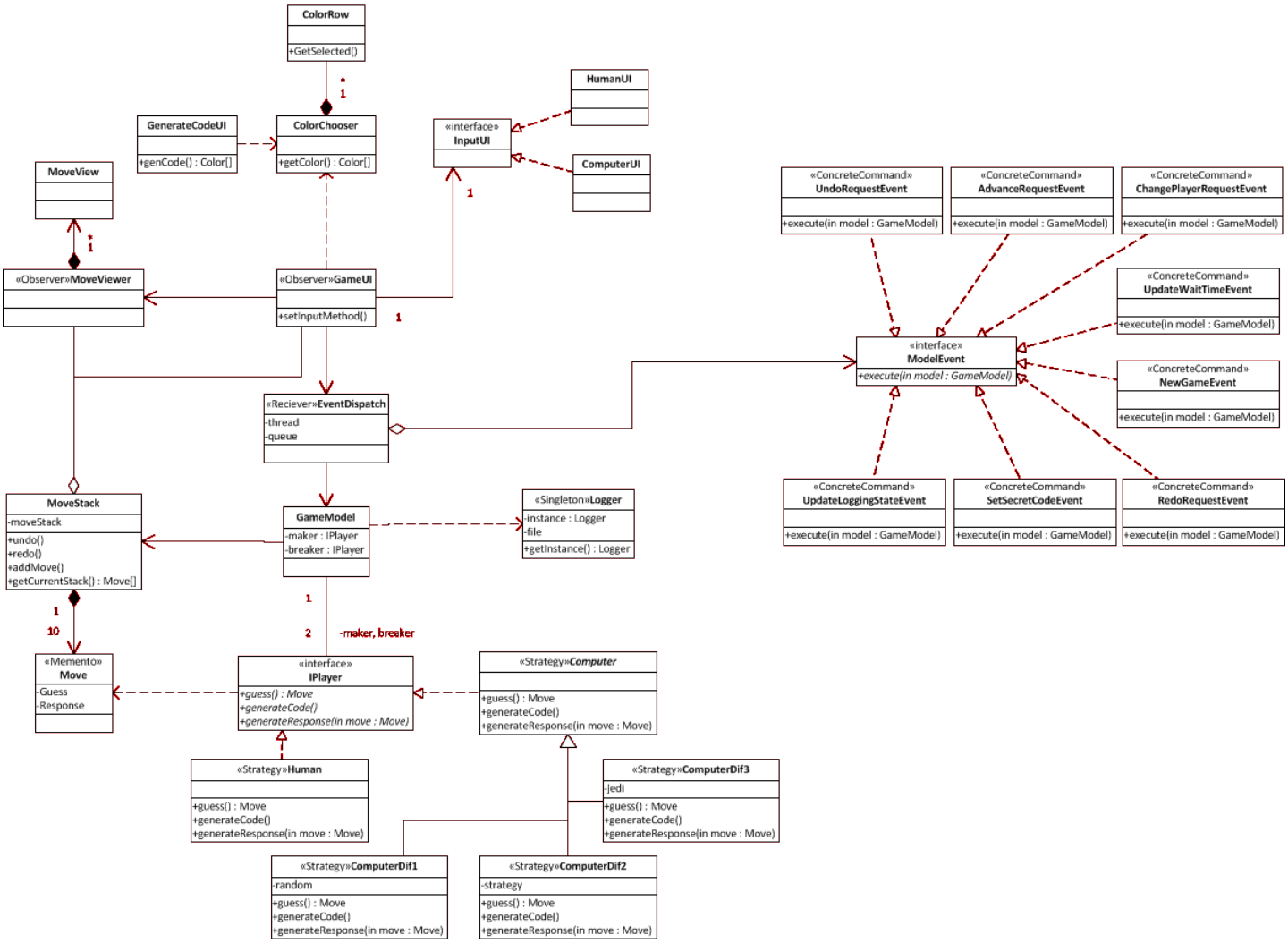
The MoveStack implements Observable and is what defines the current moves on the board. It does this with a stack of Move objects. It also contains a cursor to the most recent Move on the stack. This allows the system to undo moves without having to remove them from the stack. If the player decides to redo a move, it can simply increment the cursor back to the next Move on the stack. By contrast, if the player decides to make a new move, the undone, but saved moves are thrown out and the new move is added. Whenever a change is made, the MoveStack class updates all observers. This setup allows our system to meet the move undo requirements. Currently the only observer is the MoveViewer in the GUI. This allows us to update the GUI automatically. We decided to implement this because it allowed for a straightforward and simple way to handle any combination of moves, undos, and redos. This also provides a simple way for the GUI to determine whose turn it currently is.

The IPlayer interface allows for different strategies to be implemented to make moves. The Human strategy makes the moves that are selected on the UI. It obtains this information from the event that is sent when the UI submit button is pressed. We currently have two complete Computer implementing classes: ComputerDif1 and ComputerDif3. ComputerDif1 makes random moves as is required in the design documents. ComputerDif3 "cheats" and always guesses the correct secret

code. This subsystem is an example of the Strategy pattern. The IPlayer interface allows for easy swapping of implementing classes. This was necessary to allow users to change the AI of the Codebreaker during the middle of the game.

The final class in the Model is the Logger class. This class is a singleton and contains methods that are relevant to logging the progress of the game. As events and processed by the MoveStack the Logger is notified. We implemented this class as a singleton because having more than one logger does not make sense, and could result in problems if the same file is opened twice.

0.5 Combined UML Diagram



0.6 State of Implementation

Our implementation is currently considered complete. We are not aware of any bugs or issues that affect the ability of the system to function as the requirements specify. If you happen to find any, we would love to hear about them!

0.7 Description of Program Flow

The following sequence diagrams should provide a reasonable understanding on the general flow of our implementation:

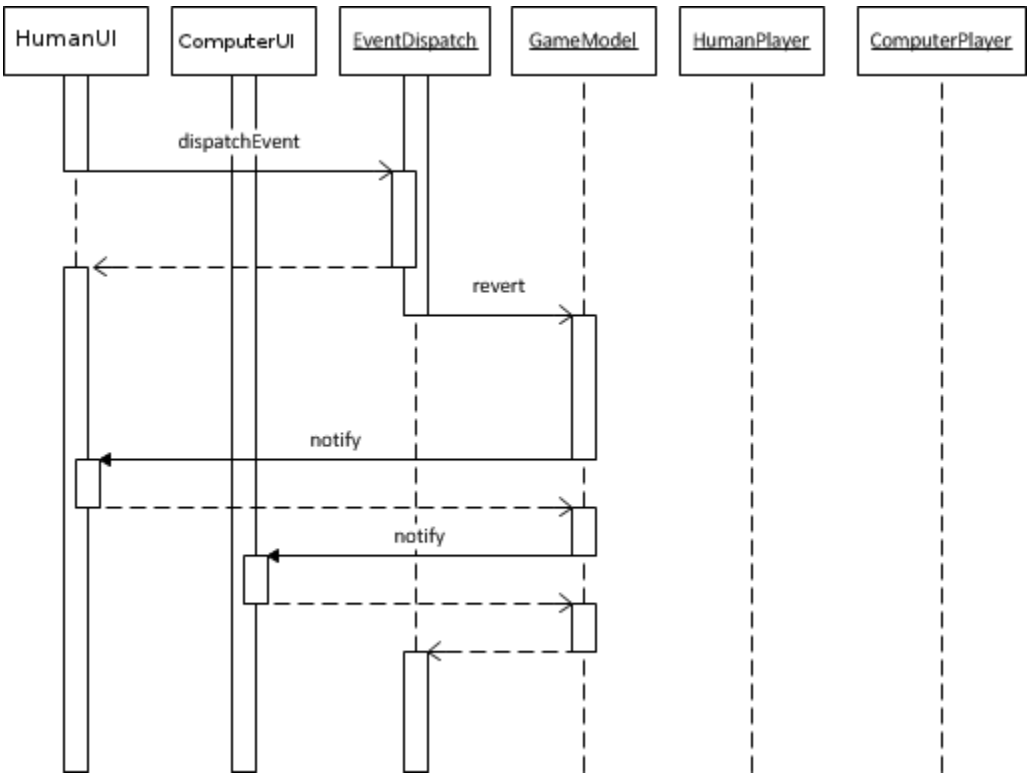


Figure 4: Human player undoes a move

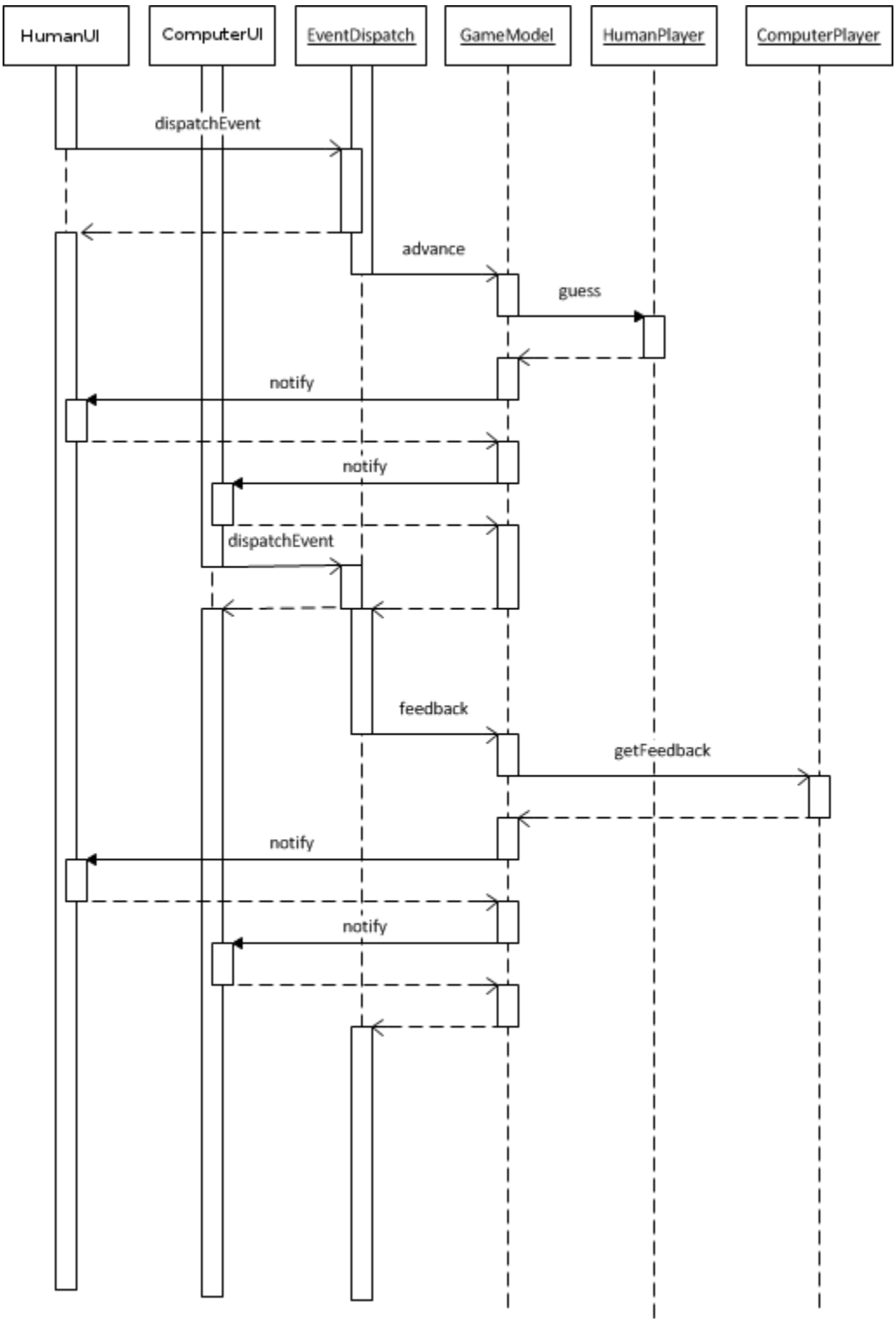


Figure 5: Human player makes a move

Part II

CRC Cards

Class: IPlayer	
Responsibilities: This interface process generateCode, guess, generateResponse, and getName classes.	
Collaborators	
Uses:	**Used by:** Human, Computer, ComputerDif1, ComputerDif3
Author: Jonathon Shippling	

Class: Human	
Responsibilities: This class implements the IPlayer class. It is for when the current player is a human player and thus is getting it's inputs from an actual player.	
Collaborators	
Uses: IPlayer	**Used by:** GameModel
Author: Jonathon Shippling	

Class: Computer	
Responsibilities: Computer is an abstract class that sets up the how a certain type of computer plays the game. It has the universal methods that are used by all types of computer players, like the waitTime, a method should have every computer wait a certain number of seconds to simulate thinking.	
Collaborators	
Uses: IPlayer	**Used by:** ComputerDif1, ComputerDif3
Author: Richard Laughlin	

Class: ComputerDif1	
Responsibilities: This is the Computer difficulty that chooses random colors as the code. Every single time. It is known as the Simple difficulty	
Collaborators	
Uses: Computer	**Used by:** GameModel
Author:	

Class: ComputerDif3	
Responsibilities: This is impossible mode computer that will guess you're the correct code on the first try every single time. It is known as the Jedi difficulty	
Collaborators	
Uses: Computer	**Used by:** GameModel
Author:	

Class: GameModel	
Responsibilities: Class that keeps logic and whatnot. Does things like start and set logging, start a new game, setting the initial code, changing maker and breaker types, setting the wait time, and getting the move stack.	
Collaborators	
Uses: IPlayer, MoveStack, EventDispatch, Logger	**Used by:** ComputerDif1, ComputerDif3, Human, EventDispatcher MoveViewer
Author: Richard Laughlin, Aaron Damrau	

Class: Logger	
Responsibilities: Logs the activity of the game when created. Logs moves, undos, redos, and player changes. If logging is turned on mid match, previous moves are logged, but undos and redos before logging was turned on are not.	
Collaborators	
Uses:	**Used by:** GameModel
Author: Daniel Wallach	

Class: Move	
Responsibilities: A Move is a class that has two arrays of Colors. One is the guess generated by the code breaker, and the other is the feedback generated by the code maker.	
Collaborators	

Uses:	**Used by:** MoveStack, GameModel, Logger, MoveView, MoveViewer
Author: Daniel Wallach	

Class: MoveStack	
Responsibilities: A class that contains a linked list of moves. It extends observable and is observed by the MoveViewer class. MoveStack handles adding moves, doing undos and redos, and checking for wins.	
Collaborators	
Uses: Move	**Used by:** MoveViewer, GameModel
Author: Aaron Damrau, Richard Laughlin	

Class: EventDispatch	
Responsibilities: This class routes events from Observers to models. It sets if the game is running and creates a GameModel	
Collaborators	
Uses: GameModel	**Used by:** GameUI, PlayerSelectionListener
Author: Richard Laughlin	

Class: ColorChooser	
Responsibilities: Used for choosing a color for a guess, response, and setting the initial code.	
Collaborators	
Uses: ColorRow	**Used by:** HumanUI, GenerateCodeUI
Author: Daniel Wallach	

Class: ColorRow	
Responsibilities: Sets up the row of colors for the ColorChooser	
Collaborators	
Uses: GameModel	**Used by:** ColorChooser
Author: Daniel Wallch	

Class: ComputerUI	

Responsibilities: A JPanel that shows a computer thinking.	
Collaborators	
Uses: InputUI, GameUI	**Used by:** GameUI
Author: Jason Greaves	

Class: GameUI	
Responsibilities: This is the complete view that the user will see. It contains the panel to choose colors from, the moves done, and a menu that has the buttons for setting log, wait time, or player type.	
Collaborators	
Uses: GameModel, ComputerUI, HumanUI, GenerateCodeUI, EventDispatch, MoveView	**Used by:** ComputerUI, HumanUI, MoveViewer, GenerateCodeUI
Author: Jason Greaves	

Class: GenerateCodeUI	
Responsibilities: This is for the code maker to make the initial code for the code breaker to guess.	
Collaborators	
Uses: GameUI	**Used by:** GameUI
Author: Jason Greaves	

Class: HumanUI	
Responsibilities: This is for the UI for when a player is of the type Human. It will display the colors the player can choose and use them to make the code to guess or as the feedback to send.	
Collaborators	
Uses: InputUI, GameUI	**Used by:** GameUI
Author: Jason Greaves	

Class: InputUI	
Responsibilities: Interface that processes setting the initial turn, turning callback, and setting game overs.	
Collaborators	

Uses: GameUI	**Used by:** GameUI
Author: Jason Greaves	

Class: MoveView	
Responsibilities: View of the moves that have been made. Shows both guesses and feedback and buttons for redo and undo.	
Collaborators	
Uses: Move	**Used by:** MoveViewer, GameUI
Author: Daniel Wallach	

Class: MoveViewer	
Responsibilities: Observes a MoveStack and gets updated by it after every change it goes through.	
Collaborators	
Uses: MoveView, MoveStack	**Used by:** GameUI,
Author: Jason Greaves	

Class: PlayerSelectionListener	
Responsibilities: Sets the UI depending on what type of player is selected	
Collaborators	
Uses: EventDispatch, GameUI	**Used by:** GameUI
Author: DanielWallach	

Class: WaitTimeSelectionDialog	
Responsibilities: Dialog Box that sets the time	
Collaborators	
Uses:	**Used by:** GameUI
Author: Richard Laughlin	

Class: AdvanceRequestEvent	
Responsibilities: Represents a request to advance the game	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: ChangePlayerRequestEvent	
Responsibilities: A ModelEvent that changes which AI is managing a particular player	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: ModelEvent	
Responsibilities: Represents an event that will be run on the given GameModel	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: NewGameEvent	
Responsibilities: Starts a new game	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: RedoRequestEvent	
Responsibilities: Redoes the last undo	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: SetSecretCodeEvent	
Responsibilities: Dialog Box that sets the time	
Collaborators	
Uses:	**Used by:** GameUI
Author: Richard Laughlin	

Class: UndoRequestEvent	
Responsibilities: Represents a request to undo the last move	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Richard Laughlin	

Class: UpdateLoggingStateEvent	
Responsibilities: Sends information to the game to pass onto the logger	
Collaborators	
Uses: Logger	**Used by:** GameUI
Author: Daniel Wallach	

Class: UpdateWaitTimeEvent	
Responsibilities: Sets the computer's wait time	
Collaborators	
Uses: GameModel	**Used by:** GameUI
Author: Daniel Wallach	

Part III

Design Pattern Usage

Our system implemented several design patterns.

1. Observer (Push updates to Views)
   The **MoveStack** class is the *ConcreteSubject*.
   The **MoveViewer** class is the *ConcreteObserver*.
   This pattern allowed us to update the GUI only when necessarily, and prevents the GUI from knowing too much about the data it is displaying.

2. Memento (Saving and Restoring Moves)
   The **MoveStack** class is the *Caretaker*
   The **Move** class is the *Memento*.
   The **IPlayer implementing classes** are the *Originators*.
   This pattern allowed us to keep track of what moves have been made, and what moves have been undone.

3. Command (Queueing and Dispatching)
   The **EventDispatch** is the *Invoker*.
   The **ModelEvent** is the *Command*.
   The **ModelEvent implementing classes** are the *ConcreteCommands*.
   Currently, the *Client*'s role is filled only by the **GUI**.
   The **GameModel** is the *Receiver*.
   This pattern allowed us to decouple requests in the GUI from the actual implementation of those requests in the Model.

4. Strategy (Codebreaker and Codemaker Implementations)
   The **GameModel** class is the *Context*.
   The **IPlayer** class is the *Stategy*.
   The **IPlayer implementing classes** are the *ConcreteStrategies*.
   This pattern allowed us to meet the requirements by facilitating switching of different player types in the middle of games.