Group 4 Members: Isaac Supeene and Braeden Soetaert

Design:

Models: Game The Game itself is the model class, and will satisfy the GameContract. The Game holds all the state of a Connect 4 game.

Controllers: Controller (GUI, CLI, AI views will delegate model changes to this class)

The Controller class will be a class that grants players control of the game, and satisfies the ControllerContract. The AI and human players will each be given a controller. Actions on the UI or on the command line can cause functions on the Controller class to be executed.

Views: GUI, CLI, AI   CLI should be an alternative to the GUI, not a way to provide input to a game already running under the GUI. The AI should also have a “view” in that it is notified of game events. Note that the CLI can also invoke the GameManager to start, end, save and load games.

Views must satisfy the ViewContract. When the game is created, it may be provided with several Views, which will be notified on important game events - i.e., a turn was played, the game ended, a chat message was sent, etc.

The GameManager will be what the player interacts with when she selects menu items, such as starting a new game. The game manager controls everything outside the context of a Game itself, such as starting a game, saving and loading a game, and ending a game. At the start of a game, the GameManager creates the game and the AI player, and gives the controllers to the AI player and the human player.

For our contracts, we will implement the ViewContract, the ControllerContract, the GameContract and the GameManagerContract.

What can we do on a computer than we can’t do on a printed board?

One thing that can be done is to have a computerized opponent so that players can play the game by themselves.. Another thing that could be done is to save a game. This can be done with a printed board but it requires physical space and care. Games can also have score values and leaderboards when they are computerized. With a connect 4 game a ranking system would make more sense than just a base score system because your performance usually depends on your opponent.

What is a computerized opponent? Remember, not everyone is an expert. What are its objectives? What characteristics should it possess? Do we need an opponent or opponents?

A computerized opponent is an AI for players to play against when they are playing the game alone. Its objectives are varied depending on the goals set. These computerized opponents can be there to help players learn the game, as well as being there to challenge the players to reach new heights. The computerized opponent should attempt to win the game and should play quite fast so the user is constantly engaged. One opponent is a good start. This opponent should either be of entry level skill or average skill level. More opponents would be better, where the opponents have different degrees of difficulty so players can progressively get better. Another option that is implemented in some games that we will not be implementing is higher level opponents who “cheat” in some way. One such way could be that the computer has a random chance to place 2 tokens at once on a turn.

What design choices exist for the Interface components? Colour? Font? Dimensions of Windows? Rescale-ability? Scroll Bars? ….

Colourwise, it would probably be best to stay with the classic colours of connect 4 which are red and black. For TOOT and OTTO it would probably be best to use black and white or similar colours for readability. For the dimensions of the windows, it would probably be good to set a minimum size for the game where the game board can still be accurately seen and clicked upon easily. Other than that we could allow scalability so that the windows can be resized. Scroll bars are probably not a good idea as for a connect 4 game, you typically need to be able to see the entire board setup to see the best possible move. On top of those points we should also have a menu that allows the user to end the current game and start a new one, as well as exit the game client altogether. Another important interface component is going to be a spot that indicates whose turn it is currently, to better facilitate gameplay.

What are the advantages and disadvantages of using a visual GUI construction tool? How would you integrate the usage of such a tool into a development process?

Advantages:

- Allows you to visualize the GUI easily as it will be displayed for you.
- Easier to align elements relative to each other.
- Overall faster GUI creation

Disadvantages:

- Limited by the GUI construction tool’s functionality, some customization you would like may not be present in the tool.
- Have to conform to the GUI construction tool’s interface and design strategies for the rest of your code.
- Have to code specifically for the tool. Can not easily swap tools without writing more code.

Such a tool could be integrated into the design process by first visualizing the GUI, and then creating the GUI with the tool. The code that is specific to the tool should be separated from other code in such a way that the GUI could be created manually or with a different tool in such a way that the bulk of the code does not require changes when the way the GUI was made is changed. Just the separated GUI tool code would need to be changed/swapped out using delegation.

What does exception handling mean in a GUI system? Can we achieve consistent (error) messaging to the user now that we are using two components (Ruby and GTK2)? What is the impact of the Ruby/GTK2 interface on exception handling?

Exception handling in a GUI system means handling the normal exceptions that are generated in the backend code as well as potential exceptions thrown from the GUI. These exceptions need to be displayed to the user through the GUI. Also, if an exception is fatal, then the exception needs to be displayed to the user and the user must be able to read the exception before the GUI closes.

It is more difficult to achieve consistent messaging to the user when we have both ruby and GTK2. To achieve consistent messaging to the user, we need to display a message to both the GUI and command line at the same time. We can not just display the messages on one because that particular display may not be seen by the user. For instance, if a user starts the program from ruby, and then covers that ruby terminal with the GUI, then no messages that are written to ruby will be noticed by the user. In summary, it is possible to achieve consistent messaging while we are using 2 components. In the GUI we could have a window that shows messages or a popup for errors. We can achieve this by sending the messages to all views so that all views can display them. However, just because we could achieve consistent error messaging, does not mean we would want to, especially if popups were used in some way. This is because all the messaging and error messages may not need to be displayed in the GUI as they are more of a developer tool.

The impact of the interface on exception handling is that we need to catch exceptions and print them to the GUI, as well as to STDOUT and figuring out what the exception means (if it is fatal or not).

Do we require a command-line interface for debugging purposes????? The answer is yes by the way – please explain why.

Yes we require a command line interface. The reason for this is that it allows us access to the full functionality of our program and the error messages that may be contained within, rather than just the subset that the GUI would allow access to.

What components do Connect 4 and "OTTO and TOOT" have in common? How do we take advantage of this commonality in our OO design? How do we construct our design to "allow it to be efficiently and effectively extended"?

The components of the two games are very similar. The only differences are that the playing pieces are different colours in Connect 4 and they are different letters in OTTO and TOOT. The other difference is the win condition. We can take advantage of this commonality by making the base game and then specifying the win condition and pieces to be used. With this design, we can have new variants added to the game by specifying different pieces and win conditions. As stated below, the different pieces to be used can just be specified by the type of view that is used and it can render the pieces differently.

Specifically, with respect to our MVC design, the Game class can accept the win condition as a parameter (or we can use polymorphism for this), and different views can be implemented for each game - for example the view for Connect 4 can render red and black tokens, whereas the view for OTTO and TOOT can render Ts and Os. These will be set up by the Game Manager when it's asked to set up a particular type of game.

What is Model-View-Controller (MVC this was discussed in CMPE 300 and CMPUT301)? Illustrate how you have utilized this idea in your solution. That is, use it!

MVC is a way to organize your program into 3 parts. Models which contain data, views which allow the models to be viewed in different ways, and controllers which manipulate the data in the models. Both the command line interface and the GUI are views in our system while the playing pieces and board are models in the system. The GUI and command line interface will also serve as controllers to modify the pieces.

Different articles describe MVC differently; are you using pattern Composite?, Observer?, Strategy? How are your views and controllers organized? What is your working definition of MVC?

We are using the strategy pattern of MVC for our views and controller. We also use the observer pattern so that the models are not inherently tied to our views and then views can be added easily so our design will be extendable. Our views (GUI, CLI, and AI) will change the model state by delegating to the Controller class. For organization, we have 3 views (GUI, CLI, and AI), a Controller class that can make changes to the model, and a Game class that is our model of all things in the game. Our working definition of MVC is that we have a model that notifies its registered views when it changes, and views that update when a model changes, and the views delegate to the Controller class to update the models when the views want to change the model.

Namespaces – are they required for this problem? Fully explain your answer!

Namespaces are not really "required" for this problem. Namespaces could be useful though in that we could put the code into a namespace and use that namespace as a way to validate input that is passed into the CLI. Another use for namespaces in this problem is that if, in the future, our game is included into a library of games then namespaces would be good to separate the games from each other when they may have similar names.

Iterators – are they required for this problem? Fully explain your answer!

Yes, as we will need them to check the win condition. For the win condition we will need to check adjacent slots for played pieces to see if the game is over. End condition should not require this as we can just count turns so we don't need to check the entire board each time.

What components of the Ruby exception hierarchy are applicable to this problem, etc? Consider the content of the library at: http://c2.com/cgi/wiki?ExceptionPatterns Which are applicable to this problem? Illustrate your answer.

None of the exception patterns that are detailed in the library really stands out for this problem. This is due to the fact that our program should encounter very little in the way of exceptions. The main place we may encounter exceptions would be in the CLI view where invalid input could cause an exception. One pattern is the Designing with Exceptions pattern and realizing that exceptions don't need to be everywhere and that unneeded exceptions would just clutter our code. Also, as always, we will be using the DesignByContract pattern to design our code using contracts. Another one that comes to mind is the CatchWhatYouCanHandle pattern as the views will have to catch exceptions so that they can properly display the error to the user through their view.