ECE 421 Design Document

Kirby Banman

Ryan Thornhill

Mar 17, 2015

1) What can we do on a computer than we can't do on a printed board?

   Quickly prototype and change the prototype. Easily distribute prototypes to any number of people anywhere. Neither of these properties are true of printed boards.

2) What is a computerized opponent? Remember, not everyone is an expert.What are its objectives?

   A procedurally simulated game player. The main objective of such an AI is to play the game with a human if that human does not have another human to play with. The human player may want to just have fun, or to improve their game. The AI should have different difficulty levels to match those different goals for people of different skill levels.

3) What characteristics should it possess? Do we need an opponent or opponents?

   The AI should be defeatable and configurable (difficulty level). For a single game instance, there are 2 players, so only 1 opponent is required. Across all possible game instances, several possible opponents should exist (different configurations)

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

   Color and font are important interface decisions, but they are not the most important decisions to make. More important decisions include:
   - How can we keep the interface portable across screen sizes, color blind users (or users with other accessibility needs? Window dimensions, rescalability, and scroll bars matter for the former, color and font matter for the latter.
   - How simple can we keep the interface? i.e. How few GUI components can we possible use, and how cleanly can we relate them? (visually and in view/controller code)
   - How can we minimize the use of words to explain? i.e. Make the GUI visually obvious to use.
   - Placement of important messages
   - Animation of gameplay

5) What are the advantages and disadvantages of using a visual GUI construction tool?

   Advantages: fast prototyping, easy changes to existing UIs, immediate visual display of UI as its being designed. No possibility of erroneous layout definitions.

   Disadvantages: builder (as opposed to programmatic construction) may not be as flexible (UI builder may not expose all possibilities). Some UI's are inefficient to build

by hand and could be done more efficiently in code using a loop. Sometimes UI builders are difficult to use (Glade's UI is not always intuitive), displayed appearance may only reflect GUI appearance on a single platform.

6) How would you integrate the usage of such a tool into a development process?

   Early (in the design and requirements gathering steps) a fast UI builder tool can be used to gather feedback from clients by creating quick mock ups. The tool can be used again in the implementation phase to build the production ready UI. Finally, in the maintenance phase, UI changes can be made quickly and easily with such a tool.

7) What does exception handling mean in a GUI system?

   As specified in the below link, the Ruby GTK system uses signals to represent events. Signals are hierarchical. For instance, the Gtk::Window defines several signals, and all things inheriting from Window will share the signals. These signals introduce a new failure mode, signal handler crashes. This means our code will have to resistant to all errors that can occur in a terminal based solution as well as added GUI errors.

   When we encounter exceptions in a GUI based system we can now prompt the user with a dialog explaining the error instead of a stack trace printout to the console. This also means that we may need to spend more time creating user friendly messages for the exceptions we catch.

8) Can we achieve consistent (error) messaging to the user now that we are using two components (Ruby and GTK2)?

   Yes, both Ruby and GTK exceptions can be caught as they always have been. The only difference is that we can now prompt the user with these errors graphically instead of through the terminal.

9) What is the impact of the Ruby/GTK2 interface on exception handling?

   In order to avoid falling into exception handling anti-patterns we will have to be aware of the GTK exception types. Furthermore, when using a GUI we can't adopt a "let it fail" methodology as we have in previous projects because we cannot let the UI crash because of an exception. We will have to be more diligent in trapping exceptions than we have in the past.

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

    The GUI is a complex, error prone system, and reporting errors can be difficult. The

CLI can be made very simple, and reporting errors is usually easy. If the application is exhibiting bugs, circumventing the GUI and interacting over the CLI to replicate the bug will tell whether the bug is in the GUI or the backend.

11) What components do Connect 4 and “OTTO and TOOT” have in common?

Both games use boards that are grids, and both games use tokens that are inserted into grid positions. Also, the interaction paradigms are identical. In both, tokens are added to the board by choosing a column. Finally, win condition evaluation is very similar. Both look for rows of four tokens in a specific pattern (though the pattern varies between either game). The number of players and the turn-based play is also identical.

12) How do we take advantage of this commonality in our OO design?

Both games will share a vast majority of the code. We will be using a GameType object and factory that will build many of the objects that differ between the two games. By leveraging the abstract factory pattern we should be able to support the two game types attaining very high code reuse.

13) How do we construct our design to “allow it to be efficiently and effectively extended”?

From the perspective of MVC, the model should be constructed such that it can be extended to implement both game types, and allow human and computer players. Also, the interface between controllers and the model should be extendable such that the Commands sent by the controller to the model should be redirectable through a network to a server and broadcast to both participating game instances.

14) 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 an architectural style for human interfacing applications. It represents a decomposition of functionality into three partitions. The essence is to have a Model component of software which represents and maintains data, a View component that presents the Model to the human user, and a Controller component that the user uses to mutate the Model. Depending upon the flavor of MVC in use, the Model may automatically update the Views, and the Controller and View Objects may significantly overlap.

15) 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?

Our flavor of MVC will use the Observer and Strategy patterns:
The strategy pattern will lend itself well to the differing levels of AI. The user will essentially select the AI algorithm at runtime. This pattern will surface again for the win condition. Depending on game type we have to define an algorithm to determine if a player has won. This algorithm will be selected when the user selects game type.

Our View objects will use the Observer pattern to observe Model objects so that they update their views every time Model objects change.

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

Namespacing is a fundamental part of software engineering. For this particular application, we'll be interacting with the GTK toolset. GTK is a very heavy, complex stack, and it is in its own namespace in the Ruby environment. Since this project will have the most classes of any project we have completed so far we will namespace our classes with modules to group alike functionality. This will increase the cohesion and readability of our code.

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

Detecting a game win condition will require iterating over the entire game board on each turn. Clever iteration techniques could aid in winning pattern recognition. The same technique will be used in the AI algorithm to compute the next move. The hard AI will potentially compute the best possible move for the current board state. This will again require iteration over the whole board. Also, iterating over model observers will be necessary to call their update methods.

18) 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.

The exception handling code will largely surround GTK exceptions, but some StandardError exceptions from the Ruby hierarchy will apply. Potentially the IndexError if we choose to store the gameboard data in an indexable model object.

Design by contract pattern is heavily used in this project and will hopefully absorb many of the errors or bugs we encounter. Within our contracts we use the Bouncer pattern as a way of validating input to class methods. Let Exceptions Propagate will probably also be useful to allow graceful handling of exceptions at the GUI level.