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

For one, we can provide a computerized opponent so that the game can be played by one person. Additionally, a very useful feature of computer games is that the system can keep track of rules and inform the users the consequences of their choices, behaviour, etc., whereas in a table board game the users themselves must manage the game. We can also let the users know who won the game, whereas in real life, users must figure that out on their own.

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

A computerized opponent is a virtual player, i.e., a player that will interact with another player in the game, without human control - completely controlled by the system. It acts with the intention to win while providing an adequate challenge to human players (neither boringly easy or frustratingly difficult).

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

The opponent should be beatable, but provide some amount of challenge. Few players would be interested in playing against a perfect opponent that virtually never loses and few would want to play against one that seems incapable of ever winning. One characteristic that wouldn't be unusual for an opponent in an application like this would be multiple difficulty levels. Depending on the implementation of this, this could require programming multiple opponents, one for each difficulty level (that's if the AI algorithm used can't be easily adjusted programmatically, necessitating the creation of more than one AI algorithm). While difficulty levels would be a common characteristic in similar games, we will not be implementing this, as the focus of this assignment is inconsistent with the design and implementation of potentially complex AI algorithms. As an alternative, we will implement a relatively simple AI in order to prioritize focus on the more important aspects of this assignment.

There should be at least one human player in the game. Some games provide a mode where all players are computerized and user purely spectate (typically with the goal of studying the techniques exercises by the AI), however we won't provide this feature since our artificial intelligence will likely not be sufficiently sophisticated to justify the implementation of this mode of gameplay. It is believed that there will be marginal benefit resulting from the creation of a gameplay mode wherein our simple computerized opponent engages in a game against a clone of the same opponent, due to the unsophistication of algorithms used to create the computerized opponent.

What design choices exist for the Interface components? Colour? Font?

Dimensions of Windows? Rescale-ability? Scroll Bars? ....

There are endless choices of colour and font for the board, provided the general form of the Connect4 board is kept (it must be a grid, the players' pieces must be clearly distinguished, etc).

Another design choice is whether to make the game always full screen or to permit the game window to be resizable by the user. In the former case, if the window becomes too small to show all the features in the game, scroll bars would be added so that the user can choose the parts of the game that are visible. We will initially make the game full-screen only, making sure that all parts of the game fit in the window, ensuring that no scrollbars are needed. Subject to resource and time constraints, the window resizing feature may be implemented, the feature is of lower priority however.

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

Advantages:

- Iterative feedback: at any given moment the designer can see a prototype of what the GUI will look like
- Asset reuse: pre-made images and windows are available
- Editing with a GUI construction tool can be faster than editing configuration files for less experienced GUI designers, or designers intending to assemble a quick prototype to exemplify their intentions
- Fast and easy

Disadvantages:

- Pre-made assets may impose constraint on a designer, bad for customize and unique design
- Editing configuration files manually can be faster and more precise for experienced designers
- Some visual GUI construction tools can be difficult to utilize (the tools may lack precision, have unexpected effects under certain conditions, etc.)

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

Such a tool will be utilized within our project both as a prototyping tool (to experiment with ideas and implement mockups) and to assist in the creation of the GUI. Such a tool will assist us during design (both graphical and object-oriented), however the functionality produced within the tool should be decoupled from the core logic of the application.

What does exception handling mean in a GUI system?

Exceptions that happen inside the code must be handled comprehensively by an application or, if it is not possible, or if the user must be informed, an error must be shown to the user and the GUI should be restored to a previous, valid state.

An example of the expected behavior is as follows: a user tries to save a file, and a system error occurs. In this case, an error window must be shown to the user and the user must be returned to the file menu once they dismiss the message.

Can we achieve consistent (error) messaging to the user now that we are using two components (Ruby and GTK3 or other GUI system)?

This can be achieved if we catch Ruby exceptions from the backend and wrap them in frontend exceptions, so that both frontend and backend errors are handled the same way. So, if anything goes wrong in the Controller or the Model, the View should have some way of detecting that an exception has occurred in one of these and then create some form of message that tells the user that the error occurred and that either the game has reset or that the program will exit. However, only a subset of the backend exceptions should propagate to the frontend; the majority of the backend exceptions should be handled in a manner which is seamless to the user.

What is the impact of the Ruby/GTK3 interface on exception handling?

The Ruby/GTK3 interface involves View objects being fed a block that will be executed in the case of an event. To handle exceptions, our input blocks will put things in the block within try/catch blocks, where an error dialogue is displayed if an error is caught, letting the user know that something went wrong. So, if the event handler method (from the Controller) throws an exception, the View prints an error message to the user, letting them know about the error and what will happen in response. This also means that if an error is thrown by a method in the Model, it will get thrown by the Controller event handler method that called it and get caught by the View block that called the Controller event handler method and print an error dialogue to the user. So, basically, if the Model encounters an exception it can't deal with and throws an exception or the Controller throws an exception, the View will catch it and print an error message to the user before whatever action is taken to get to an exception-free state, whether that's resetting the game or closing the program.

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

A command-line interface for the program is a way of interacting with the back-end without the need for complicated graphical functionality.

For instance, suppose we know that the back-end has a bug when it receives input of a certain type. During the debugging process, the programmer will need to send that input to the

back-end repeatedly, which can be time-consuming if generating the same input using the front-end requires numerous steps. Using a command-line interface, the generation of that input can be scripted, or stored (in an XML file, for instance).

These features would be especially useful, since most bugs will most likely be in the back-end logic. Within the front-end, events must simply be processed properly, with comparatively little, non-trivial logic. However, since the only events the user can carry out in the front-end are those defined by the developer, it is unlikely that there will be input which the user can provide that will cause an error within the front-end specifically. Errors are much more likely to occur in the back-end, where data is handled and business logic is evaluated. As a result, possessing a tool which increases the efficiency of the debugging of the back-end services of the application is essential.

What components do Connect 4 and "OTTO and TOOT" have in common? How do we take advantage of this commonality in our OO design?

Both games: have the same board structure (a 2-dimensional grid of squares), involve 2 different categories of pieces (one for each player), and involve the formation of patterns of 4 adjacent pieces in the board going in the same direction (vertical, horizontal, or diagonal), resulting in victory for one player or the other. The only difference between the two games is the type of pattern that each player needs to form. In fact, Connect4 could be implemented using O's and T's and be called "TTTT and OOOO".

In our implementation, we will divide our pieces in two categories. In Connect4 mode, each category will be a colour and the patterns needed to win will be four consecutive instances of colour 1 for player 1 and four consecutive instances of colour 2 for player 2. In "OTTO and TOOT" mode, the categories will be the letters T and O, which will be displayed on the pieces, and the patterns will be OTTO for one player and TOOT for the other. All other functionality will be identical within both games.

How do we construct our design to "allow it to be efficiently and effectively extended"?

Numerous variations of Connect4 can be implemented as new modes, using the same, core functionality described above. All that is required is that each variation have a winning condition (such as another pattern). Either way, the core logic within the game will involve checking for patterns in 4 adjacent board locations in a certain direction in order to determine whether or not the board reflects a winning state.

This core logic could be implemented to simplify the process of inspection for more or fewer adjacent pieces following a certain pattern, within other game modes. Such core logic could even be implemented in such a way that patterns other than a straight line are considered,

permitting a very large number of potential rules for the specification of winning conditions, rendering the application prolifically extensible.

Another way to make our design extensible would be to utilize adequate abstractions, exercising foreknowledge concerning the potential aspects and features of the application which may be user-controllable in the future, such as using appropriate abstractions for numerical values within named variables (to, for instance, permit the inspection for winning conditions within boards of arbitrary dimensions).

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

MVC is a design pattern used in several applications and frameworks for user-facing software. There are variations on what constitutes MVC, but the general idea is that the application is divided into three modules:

- The View is the user-facing part ofthe application. Usually this means the GUI. It is an Observer of the Model, and updates when changes to the Model are undertaken.
- The Model is the back-end, the abstractions representing the application's business rules and behaviour. Data schema are implemented within the model. It is an Observable class which notifies Observers when data changes are undertaken, and does not reference any information pertaining to the Views which are observing the model.
- The Controller is the intermediary between the View and Model. The exact functionality of a controller varies according to each interpretation of MVC. Typically, the goal of a controller is to serve as interface between the two and thus facilitate decoupling between the View and the Model. Often, if an event is triggered within the View, the Controller will handle the event, informing the Model as to what event has occurred, or what changes need to be made in response to the event.

We will use MVC in our project, with the GUI, and an Observer class to update the GUI, serving as the View, and the Ruby back-end serving as the Controller and Model (using adequate separations within the code, e.g., by using different classes). The Model class will be an Observable to facilitate updates to the View when changes to the Model are undertaken. The Controller will hold the event handlers that are called by various GUI elements.

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 an MVC architecture with the Observer pattern. Within this approach, the View will have two roles: to pass events to the Controller, which will interact with the Model by communicating that an event occurred, and to observe the Model, update itself when certain data within the Model changes. The role of the Controller is to handle events from the View, calling appropriate functions from the Model in order to update the View appropriately in response to events. The role of the model is to hold the data that represents the state of the application, and to possess methods which alter the data when invoked during certain events. The model is implemented as an observable in order to ensure that any Views observing the Model can register any changes to the data of the Model, and subsequently update accordingly.

Classes start life on CRC cards or a competing notation. Provide a full set of CRC cards produced by your group for this problem.

The CRC cards are provided in Appendix A.

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

Yes. There are several instances where iterators will be useful. For instance, performing identical computations, pertaining to a common action, to all players in the board, or searching for a winning pattern during the enumeration of the board pieces. Searching for a winning pattern is likely the most important use of iterators within this problem, which will involve iterating through every non-blank space on the board and checking whether a winning condition is met in any direction from the given space.

Another possible use for iterators would potentially be within the computerized opponent. Some types of artificial intelligence for this problem may necessitate the use of an iterator in order to go through every move the AI could make before using some means to decide which move is the best. However, as stated earlier, we do not foresee the AI being the focus of the project, so we likely will not implement an algorithm which is that sophisticated.

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 Bouncer Pattern (http://wiki.c2.com/?BouncerPattern) will be used in validating user inputs (there will methods, the only responsibility of which will be to throw an exception if the input is invalid).

Since we're using an Observer pattern, the Observers Should Never Throw Exceptions (http://wiki.c2.com/?ObserversShouldNeverThrowExceptions) pattern applies, since there is no artifact available to catch exceptions which are thrown from an Observer; an Observable is

designed to not possess any knowledge of its Observers. As a result, Observables should not reflect any interest in the failure of Observers.

Model contract

Class name	***Class invariants***
Game	1. @board is not nil 2. @players is not nil 3. @players size is > 0 and <= 2
Connect4	Same as Game (subclass)
OttoNToot	Same as Game (subclass)
Board	1. n_rows > 0 2. n_cols > 0
Piece	1. @categoy is not nil
Player	1. @category is not nil 2. @winning_pattern is not nil
Victory	1. @winner is not nil 2. @positions is not nil 3. @positions is not empty

Class name	***Method***	***Pre-conditions***	***Post-conditions***
Board	initialize(n_rows, n_cols)	1. n_rows > 0 2. n_cols > 0	None
Board	add_piece(col, piece)	1. 0 <= col && col < @n_cols 2. Column col is not full	1. Piece is in column col
Board	valid_columns	None	None
Board	pattern_found(pattern)	1. pattern size is less or equal to number of rows 2. pattern size is less or equal to number of columns	None

Game	initialize(n_rows, n_cols, player_categories)	1. All player categories are valid according to the game's rules (e.g., T or O)	1. There are as many players listed in player_categories as necessary for the game 2. All players have the right winning pattern according to their order (Player 1 has pattern 1, etc)
Game	make_move(player_number, col)	1. There is a player with number player_number 2. Column with number col is one of the available columns to drop a piece	None
Piece	initialize(category)	None	None
Player	initialize(category, winning_pattern)	None	None
Victory	initialize(winner, positions)	None	None

View Contract

ClassName	**Invariants**	**Pre-condition**	**Post-condtion**
BoardView	None	None	None
MenuView	None	None	None
ButtonView	None	None	None

PromptView	None	None	None

Controller Contract

ClassName	**Invariants**	**Pre-condition**	**Post-condtion**
Controller	None	None	None

Appendix A - CRC Cards

Victory	
• Hold data pertaining to victory condition	• Game • Player

Player	
• Hold data about a player	• Game • Victory

Piece	
• Hold data pertaining to a piece on the board	• Board • Game

Board	Observable
• Keep track of board state, such as piece positions • Determine if a given winning pattern appears on the board • Inform other classes on what columns are available to new pieces	• Game • Piece

Game	
• Manipulate the board according to player requests • Determine what's the winning condition for each player • Determine what piece categories exist • Inform the Controller when a player has won the game	• Board • Piece • Player • Victory

OttoNToot	Game
• Hold the rules for the game Otto and Toot	• Game • Board • Piece • Player • Victory

Connect4	Game
• Hold the rules for the game Connect4	• Game • Board • Piece • Player • Victory

BoardView	
• Draw the board and the pieces placed by players	• ButtonView • MenuView • PromptView • gtk.Frame • Board

MenuView	
• Draw the menu on the UI • Menu allow player(s) to select either in 1 player or 2 players mode • Menu allow player(s) to reset the board by clicking the reset button	• Game • gtk.Button • gtk.RadioAction • gtk.RadioButton • board

ButtonView	gtk.Button
• Draw the buttons on the UI • Able to differentiate players by displaying different characters	• players

PromptView	gtk.Dialog
• Display any prompt message to user such as victory, errors, etc.	• Game • victory

Controller	
• The controller class serves as an intermediary software construct used to decouple the Models and the Views within the application. The controller contains methods which may be invoked from the Views within the application for the purposes of the communication of user input, or other potential external events, from the Views. Additionally, the controller has callback methods which may be invoked by observable Models, which subsequently update the state of the Views to reflect the Model changes.	• BoardView • ButtonView • MenuView • PromptView • Game • Board