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

Save states indefinitely
Save multiple states simultaneously
Computer controlled opponents.
Zero social interaction [because who want to talk to people... really]
Infinite victory conditions. [Doable on a real board, but more easily enforced in a computerized version.]
Ability to choose a clever alias!

2) 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 set of algorithms whose objectives are used to mimic a human opponent. It should provide a realistic and challenging experience to the user, and considering the different skill levels of players in a game, there should be different skill levels available to choose from.

For this specific problem only a single opponent is needed, however the ability to create/handle multiple opponents should be considered for future extendibility.

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

Most of the choices will be based on aesthetics and usability, windows should be able to be resized and the contents will scale while maintaining the correct aspect ratio [subject to capabilities of the library]. Since connect four has a small area, and scroll bars look hideous, they will not be supported (this implies the window will have a minimum size).
The only thing we can say for sure about colour is that we will have colour (and that the colours will be… sensible… and not something ridiculous like yellow text on a white background). And no comic sans will be used.

4) 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?

Ease of implementation, nobody wants to write the code for determining container positions, and you can visually see the gui while designing it. However it is difficult to write test cases and scenarios for GUI's, and GUI tools can have imposed limitations.

The usage of a GUI tool is made easier by using the MVC architecture pattern. This way, the UI can be designed separately from the logic of the game and added fairly easily to the existing model once completed.

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

Exception handling in a GUI system is the same as exception handling in any other system. Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional events requiring special processing – often changing the normal flow of program execution. The only difference with exception handling in a GUI system is the inputs may be in the form of button clicks or keyboard input events, as opposed to a text-based system where exceptions would normally be caused by incorrect text input.

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?

GTK2 defines custom exceptions that may be different from Ruby exceptions, thus we will need to add extra exception handling for cases where an exception is thrown from GTK2. We can achieve consistent messaging by printing a custom message when exceptions are thrown from either GTK2 or Ruby.

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

Debugging GUI's is a ridiculous process, and is much easier to do via command line. It is difficult to write scripts dealing with button clicks or keyboard input events in a GUI; therefore we can simulate these events for debugging purposes using a CLI, thus allowing us to automate testing. This is possible because the model portion of our MVC framework can be tested separately using the command line without the need for the GUI.

7) 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"?

Connect 4 and "OTTO and TOOT" are both games involving forming set patterns of tokens to win the game. In connect 4's case this involves forming a pattern of either 4 consecutive red tokens or 4 consecutive black tokens, whereas in OTTO and TOOT the patterns are exactly that - "OTTO" or "TOOT".

Essentially the only difference between the two games is the "winning" pattern (and corresponding tokens that can be used). Therefore, we can take advantage of this commonality by simply being able to define different "winning" patterns and tokens, or more specifically, provide the ability to extend our winning criteria and define your own.

8) 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!

Model-view-controller is an architecture pattern where the application data and information are separated from the representation of that information and the user's control over that information. Typically a view will be some representation of the data such as the display portion of a GUI, graph, or table, and a controller will mediate input to handle modifying the data or displaying the view. Our implementation of MVC is described in question 9.

9) 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 will be organizing the views and controls using the observer pattern. When the user changes something using a controller class it will update the model, and then notify the view that the model has changed, at this point the view will update the UI.

The composite pattern will likely show up in a large number of places within the UI coding. There will be a series of containers each of which will need to follow the generic container interface, at the same time each of these containers can and *probably* will contain other containers and components which inherit from the same interface.

I suspect that we will also be using the strategy pattern, but not as part of our MVC framework it will mostly be used when defining the behaviour of our AI, in order to allow it to be fully and easily extendable.

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

Yes...ish. Namespaces, like properly designed objects are never really “required,” however in order to make clean easily maintained code we will likely have different namespaces for each of the different categories of the MVC design. Eg. There will be both a Model::Board class and a View::Board class, they will deal with the behaviour of the board, and the code required to draw it to the screen respectively.

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

Yes, you will need to iterate over the “board state” of your game to see if a win condition has been met.

12) 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.

BouncerPattern: possibly to check if the correct player is making the move.

Exceptions cancel transactions: in case some exception is thrown, the game must be allowed to continue in a valid working state.