1. Document History

Rev 1.0 - February 13, 2007 - Initial revision
Rev 2.0 - February 15, 2007 - Flesh out document
Rev 3.0 - February 19, 2007 - Add Controller Package Class Diagram
Rev 3.1 - February 19, 2007 - Added Subsystem Diagram
Rev 3.2 - February 19, 2007 - Added Model(GamePlay) Package Class Diagram
Rev 3.3 - February 20, 2007 - Added View Package Class Diagram
Rev 3.4 - February 20, 2007 - Added Testing Section
Rev 3.5 - February 20, 2007 - Started Plan Section
Rev 3.6 - February 21, 2007 - Update Model and Controller Diagrams, and Planning Chart

2. System Architecture

1. The architecture of a system will be one of multiple packages and classes. We plan to have a Model-View-Controller Pattern implementation. Defined as: “The MVC pattern hinges on a clean separation of objects into one of three categories — models for maintaining data, views for displaying all or a portion of the data, and controllers for handling events that affect the model or view(s).” (http://www.enode.com/x/markup/tutorial/mvc.html). This will work well because we have our gameplay rules, which are already defined, as well as the view of our game, and will use the controller to be the relationship between the two.

2. Subsystems

   1. The subsystems of our Software will be represented as packages, containing classes. The main packages are GameSystem, Controller, Game Rules, and View.
   2. GameSystem Package: This will be fully global package of our system that has all of the changeable data in it. Whereas the game rules and events are handled in the other packages, the players current units, positions, and status will all be stored in the GameSystem package, to check/change the current state or status of the game. The model or controller can change these states/statuses if it needs to/is supposed to.
   3. Controller Package: The functionality of the Controller Package is to take input supplied by the user via mouse clicks, buttons presses, and text inputs, and interpret it as needed based upon where the events happened, and use the gamePlay layer to handle the data of the requests, and the View layer to decided what to be drawn for
the "player" to see.
4. Gameplay Package: The gameplay package is the Model part of the MVC pattern. Here, all of the game rules are defined. The game will revolve around the data in these classes, and as well as when any action happens, the rules will be checked to know what is to be done.
5. View Package: The View package will be the way that the user sees what is happening in the game. This will take care of drawing the Board, the Units, the player IDs, the pop-ups and menus, and whatever else is needed for the user to interact with the game.
6. Data Package: This is simply data objects that will have no functionality, but will be used by all of the other packages (mostly the GamePlay package).

3. Class Design

1. Controller

The Controller Package has an AbstractController class. It also has a defined class for each of the phases, as well as a Dispatcher to be the event handler, and the ControllerFactory class to easily use the different necessary controllers.
2. **GamePlay(Model)**

The Gameplay package has a ManagerFactory, which will know how to use the different Managers (each game phase has its own Manager) to operate the gameplay.

3. **View Package**

Like the other main packages, the View Package will have a Factory of the different Views (each game phase has its own view), where the Factory will be able to handle knowing which class to use.
4. Data Package

Abstract Unit

- String Name
- static final int kill_Troopers
- static final int kill_Gennies
- static final int kill_Battlers
- static final int kill_Satellites
- static final int kill_Transports
- static final int kill_Factories
- (static final int [][] kil_Matrix)
- int moveSpaces
- int movementConstraint
- int buildTime

Player

- String color
- int profile_id
  - (pointer to) leader
  - (pointer to) current Territories
- int turn_order_num
  - (pointer to) units

Leader

- String name
- int bonus_mod
- int [] bonus_unit

4. Detailed Design
1. Dice Roll: For the roll of the die we need to use a random number generator. We plan on using the srand() <ctime> built in function, and will seed with number generation with the system time, to hope to get as random of a number as possible.
2. For reading the text from file(profiles), we will use the built in C++ file reading functionality.
3. The view will be using OpenGL to draw all images to the screen for the user to see.

5. Testing Plan

1. Test will be performed both manually and automatically. The manual ones will be done by the developers as well as the customer, to ensure code efficiency and correctness, as well as use-case correctness.
2. Division of Testing
   1. Unit Testing
      We plan to make all subsystems divide neatly into corresponding layers which rely on loosely coupled interfaces, allowing us to easily supplement class stubs for ease of testing individual units without relying on the functionality of more complex and more specific units deeper in the system.
   2. Integration Testing
      We plan to use the Top-Down Integration strategy, where we will begin by providing simple stub implementations for all major interfaces and gradually add enhanced functionality as we progress. For instance, the View layer will initially contain rudimentary graphic functions, and develop simultaneously with the corresponding GamePlay layer.
   3. System Testing
      We plan to test the system through repeated QA-style manual testing of the entire game as needed. All developers and customers will take part in the manual system testing.
   4. Regression Testing
      We plan to implement a system for regression testing whereby after every major step in our plan, we will re-compile the system and run tests exhibiting previously found bugs. New bugs found after every major test will be added to the regression testing after short group discussion.

6. Plan

<table>
<thead>
<tr>
<th>Step Description</th>
<th>Time Unit</th>
<th>Responsible Party</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup makefile</td>
<td>1</td>
<td>All</td>
</tr>
<tr>
<td>Build initial startup to home menu screen</td>
<td>2</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Create new profile interaction</td>
<td>2</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Setup testing files</td>
<td>2</td>
<td>All</td>
</tr>
<tr>
<td>Determine storage/retrival of records and profiles</td>
<td>2</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Create new game menu and allow to select options</td>
<td>1</td>
<td>Available</td>
</tr>
<tr>
<td>Develop graphical layout of the game board</td>
<td>3</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Create functionality and menu for game board</td>
<td>2</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Implement well-randomized 10-sided die</td>
<td>1</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Develop graphics for each of the game units</td>
<td>3</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Implement the movement phase</td>
<td>2</td>
<td>JV</td>
</tr>
<tr>
<td>Create click-based movement for units</td>
<td>2</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Task</td>
<td>Time Units</td>
<td>Authors</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>Develop click-based loading onto transports</td>
<td>2</td>
<td>WW</td>
</tr>
<tr>
<td>Create live graphical movement updates</td>
<td>4</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Implement end-turn handler</td>
<td>1</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Implement the attack phase</td>
<td>3</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Create click-based attack for units</td>
<td>2</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Develop post-battle removal system for killed units</td>
<td>2</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Implement the building phase</td>
<td>2</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Create menu for building</td>
<td>1</td>
<td>Available</td>
</tr>
<tr>
<td>Create building manager to handle progress</td>
<td>3</td>
<td>JV, WW</td>
</tr>
<tr>
<td>Create graphical deployment of units</td>
<td>2</td>
<td>JF, WW</td>
</tr>
<tr>
<td>Implement the upkeep phase/end-game conditions</td>
<td>1</td>
<td>JV</td>
</tr>
<tr>
<td>Graphical movement of planets in the solar system display</td>
<td>2</td>
<td>JF, MC</td>
</tr>
<tr>
<td>Implement on-close prior to end-game handler.</td>
<td>1</td>
<td>WW</td>
</tr>
</tbody>
</table>

7. Checklist

Content

- Is the design complete, well-organized, and clear? Are the components and interfaces specified in enough detail that the design could be turned over to an independent group for implementation and still be understood?
- Is the design specified hierarchically?
- Are diagrams used effectively?
- Are design patterns used effectively?
- Is the level of abstraction appropriate and consistent?
- Is information hiding used effectively to isolate complexity?
- Is information hiding used effectively to isolate parts of the program that might change?
- Is the design minimal?
- Is coupling between components in the design minimal?
- Is the strategy for the user interface design covered?
- If needed, is the database design specified?
- Does the design include a coherent error-handling strategy?
- Are all the requirements in the requirements document covered by the design in a sensible way, by neither too many nor too few building blocks?

Testing

- Does the design avoid nondeterministic behavior?
- Does the design include self-checks?
- Is the design stateless wherever possible?
- Does the design include a test interface?
- Are all the testing stages (unit, integration, validation) covered?
- Is the regression test strategy covered?
- Are non-functional validation tests necessary and covered (stress, performance, security, etc.)?
- Does the testing plan include walkthroughs or inspections?
Planning

- Does the plan include multiple iterations, each adding functionality incrementally?
- Does the plan take dependencies among different components into account?
- As much as possible, does the plan avoid situations in which the entire team would have to wait if one task is not finished on time?
- Is the timeline realistic?