TrollQuest

MileStone 1: Future Milestone Requirements
Team Trollolol

Team:
Taylor Berger
Ian Mallett
Stephen Patel
Martin Tice

Overview:

TrollQuest has been broken down into four main elements, each assigned to a team member. The four elements are: User Interface, Networking, Monster/NPC AI, and Game content. Each team member has defined their role in each of the milestone deliveries and what they expect to have done for each milestone.

Table of Contents:

Networking (Stephen Patel)

Server
The server for JWorld waits for clients to connect, and then hands off the responsibility of managing them to handlers. The server will run on it's own thread.

ClientHandler
The ClientHandler contains two Sockets, one for receiving, and one for sending. It runs on it's own thread, and perpetually "listens" for input from its Client. It is responsible for handling input from the Client, and sending data from the Server to the Client.

Client
The Client contains two Sockets, one for receiving, and one for sending. It runs on it's own thread, and perpetually "listens" for input from it's ClientHandler. It is responsible for handling input from it's ClientHandler, input from the User, and sending data to it's ClientHandler. Client shall be an Interface implemented by different concrete Clients (DebugClient, StandardClient, etc.).

Scheduler
The scheduler contains a Timer, and a PriorityBlockingQueue. It is responsible for executing scheduled commands every ts milliseconds. The queue orders it's commands by their prep time. When a Command is added, it's prepTime will be incremented by the current time, in order to "normalize" it to the rest of the commands. So let us suppose that the current time is 1000, and a MoveUp command is sent with a prepTime of 1. The commands prepTime will be incremented to 1001, and then added to the Queue. So when time 1001 comes around, the MoveUp command (and all commands with that time) will be pulled off the Queue and executed sequentially. This process is superseded by commands with 0 preptime. Such commands are executed IMMEDIATELY without being added to the Queue (such commands will be extremely rare). Commands are responsible for synchronizing on the specific part of the game state they access.

Communications Protocol
To minimize network traffic, Objects will be sent rarely. Most communication will be integers. This will be facilitated through a Commandler(CommandHandler) enum. Every Command type will be added to the enum, and it's .ordinal() will be sent across the network. The actual Command object will be instantiated on the server side, given reference to the game state and the sending player, and sent to the scheduler. If a command requires an object, then that will be handled by a special handler method.
Suppose that Client A wishes to send a message to all other connected Clients. A will send the Commandler.MESSAGEINC.ordinal() integer to it's ClientHandler, which will begin to wait for a String to be sent. A will then send the message and the ClientHandler will instantiate a new MessageCommand() with the game state, the Player tied to the ClientHandler, and the message, and then schedule it.

Testing Strategy
Server/Client interactions are difficult to test through JUnit, so JMock will be used. Testing will

also be performed by a Tester using a DebugClient, and a script of actions to perform, the Tester will record any bugs or enhancements he feels need to be made, and then route them to the Dev responsible for that section.

Threading Issues

On the Server side, there are several groups of threads. There is the Server thread, the Scheduler thread, and all the Handler threads. The game state is simply a data structure holding various aspects of the game (players, monsters, etc). The handlers only interact with the game state directly during player creation, so the register player method will synchronize on the registered players structure. After Player creation, the handlers all interact with the Scheduler only. So issues could arise with adding commands, and commands with 0 preptime (immediate execution without scheduling). The issues with adding are taken care of by the specific implementation of Queue, the PriorityBlockingQueue, which blocks it's methods. The issues with immediate execution are handled inside the commands themselves. All commands have a reference to the game state, and will synchronize on whatever they access. For example, the MoveUp command affects a player, and a tile. So it will synchronize on the list of characters, since we don't want them to be able to change their locations.

Main Process

Here I will describe the sequence of events from a player connecting to the game, to him being able to perform actions, to him disconnecting.

We will assume for this example, that the server is running.

Player will start the application, and be prompted for an IP address (optionally, a port as well). The Client will connect to the server (specified by this address) and a new ClientHandler will be instantiated with an input Socket, an output Socket, and a reference to the game state. This ClientHandlers run method will then begin executing on a separate thread, while the server resumes listening for Clients. The ClientHandler will send a name request command to the Client, which will open an option box to allow the Player to enter a name, a gender, and a class. This information will then be sent to the ClientHandler, which will check to see if a Player with that name exists in he game already. If such a player does exist, the ClientHandler will request another name from the Client. If such a player does NOT exist, then the ClientHandler will check a directory to see if such a player has a file. If it has, then the ClientHandler will read in the Player data from the file add it to the game state, and signal the client that his name has been registered. An update will then be sent to the Client, telling it where the player is, what the player can see, and what the player has on him. The player may then begin playing the game. If the player does not have a file (i.e. a player with the specified name has never been registered), then the Player will be registered, and the update package will be sent, at which time the Player will be able to play the game.

During game play, the Players commands will be sent through the Client to the server. For example, let us examine a Move Command. Let us assume that the player moves up. First, the integer value for the MoveUp command will be serialized from the Client to it's Handler. The Handler will instantiate a new MoveUp() command, that takes the game state, and the sending Player as parameters. This command will be sent to the Scheduler. The scheduler will normalize the commands prep time, and add it to the queue. When the appropriate time step has been reached, the scheduler will remove all commands to be executed and execute them. The scheduler will then

notify all the ClientHandlers that an update has happened, and they should decide what to send their Clients.

On disconnect, the player will be unregistered from the game, and a players data will be saved to a file with his name. If such a file already exists, it will be overwritten.

Milestone requirements:

Milestone 1 (Nov 14th, 5:00 pm): Initial Design, test plan, and development schedule. Interacting server/client demonstrating design feasibility.

Milestone 2 (Alpha release, Nov 20th, 11:59 pm): Working client/server supporting many user commands. Multiple users should be able to log in to the game and perform actions in the world. Monsters, and AI not necessarily supported. Integration tests shall be performed before submission by Testers. JMock tests for client/server interactions should exist, and pass.

Milestone 3 (Beta release, Dec 4th, 11:59 pm): All core functionality implemented and working. The game should be playable, that is to say, the Client/Server should support all required commands/functionality. Players should be able to log in, perform quests, interact with npc's, and fight monsters. All changes from this point forward shall be bug fixes and features not specified in the specification. JMock tests should be extended for more client/server interactions, and should pass. Integration testing will be performed before submission by Testers.

Rollout (Full Release, Dec 16th, 9:00 am): All functionality in the spec should be implemented and working properly, all JMock tests should exist and pass. Integration testing to be performed before submission by Testers.

==========

Monster/NPC AI (Martin Tice)

1. Lexing
2. Parsing to an Abstract Syntax Tree(AST)
3. Interpretation
4. Interaction with game state.

LEXING: due m2

Handled by the NanoLexer class.
Parts:
NanoToken enum: described by a set of tokens given on pages 37,38 of spec.
LexicalAnalyzer interface: from p1 support code.
HasPattern interface: from p1 support code.
NanoLexer class: implements LexicalAnalyzer.
Token interface: from p1 support code.

These classes operate in the same general way as in p1. A NanoC file is passed as a parameter into a NanoLexer constructor. A Pattern object is created that encompasses all of the individual regex strings that define all of the NanoC terminals. When the nextTok() method is called on a NanoLexer object, the pattern object cycles through its list of regex. Expressions and returns a match in the form of a Token() object.

PARSING TO AST: due m2

Handled by the NanoParser class.
Parts:
NanoParser class: implements Parser interface from p1.
ASTCell interface: defines the shell of all nodes on the AST.

The NanoParser holds a NanoLexer object. It cycles through the tokens returned in order and turns them into the non-terminals represented by concrete ASTCells. This is done through recursive decent parsing along the NanoC grammar. There are several concrete ASTCells that handle each individual
type of behavior and token type.

INTERPRETATION: due m3

Handled by the execute() methods inside each ASTCell, and the VM class that holds the characteristics of each non-terminals created.
Parts:
concrete ASTCell execute() methods: return an ASTCell consisting of different data types.
A VM object gets passed as a parameter through each execute method. In this way, one VM object holds

can hold all off the variable, name, data and define how each is created and accessed.

Each time an npc issues a command, their AST strategy tree is traversed. During this traversal of ASTCell nodes, a final decision is made and returned, concrete command is issued.

The relation to the parsed trees and the way they are handled can be describes as a command pattern with the following members:

ASTCell = command interface. has execute() method
concrete ASTCells = define execute() method, what to return.
AIHandler = receiver: action() command cycles through each npc/monster's AST to get a command to return to the game scheduler.

INTERACTION WITH GAME STATE: ongoing.

The AIHandler object is the first "player" in the game. It is initialized at
startup. Once initialized, a thread starts that cycles through all of the existing monster/npc.execute() methods and continues through duration of game play. This cycle can be considered the "action()" method in the above
command pattern structure.

The AIHandler has a reference to the VM, which holds a gameState reference. When an ASTCell needs to reference to any of the
gameState fields(closest pc, pc health.. etc.), it gets the info from methods in
the VM object it holds... e.g. vm.getHealth(pc1).

THREADING ISSUES: on all of the vm.get*() methods, the field being modified can potentially be referenced from multiple places. This can be handled by surrounding the field access section of each method with a synchronized(field){} block. Dead-lock is not an issue on these blocks because nothing will ever lock onto the AIHandler.

CYCLE: AIHandler cycles through every monster/npc object, calling the execute()
method in each. This triggers a call to ONE static AST at some index AST[i].

each monster/npc.execute() method will:

-traverse tree AST[i], calling vm.get*() methods to direct traversal.
-when leaf of AST[i] is reached, a command has been decided.
-command is passed to the Scheduler object.
-move on to next monster/npc and repeat.
-wait some unit of time (to be decided)
-repeat

TESTING STRATEGIES:

LEXING/PARSING:

Can solely be handled by Junit tests. Different cases can be generally separated by input files with different content. The following cases will all describe input files.

Individual
files will include:

Common cases -- ways you expect it to be used

- -list of all individual terminals.
- -one-deep declarations of each non-terminal.
- -different ordering of tokens.
- -simple example strategies with multiple non-terminal types.
- -whitespace inside string terminal

Uncommon cases -- things that you don't expect,
but are legal

- -repetition of non-terminals
.
- -deeply nested non-terminals.

Edge/Corner cases -- edge of what's legal

- -(Lexer)non NanoC script
- -(Lexer)random text
- --super long files/ tiny files (;)
- -multiple tests on same edge cases.
- -super long files/ tiny files (;)

Illegal cases! Make sure it fails the right way.

- -test all exceptions thrown correctly.
- -dependence on logical expression precedence.
- -(Parser)non NanoC script.

INTERPRETATION:
Junit Tests can be bulk of Interpretation. Creating a VM object with a set gameState object inside a Junit suite. The testing strategy involves modifying the gameState in different ways and testing different scripts on changes.

Common cases:

- -change one game state element at a time, test on each script for

correct return cell.
  -both inside and outside visibility limit checks.
Uncommon cases:
  -gameState data at limits of allowable fields.
Edge cases:
 -gameState data at limits of allowable fields.
Corner cases:
 -high number of players in visibility range.
Illegal cases:
 -test against all individual command() requirements.
 -test all exceptions thrown correctly
INTERACTION WITH GAME:

AIHandler tests will performed at later stages of project development. To Correctly test, multiple real time scenarios should be performed by actual humans(developers). Close relation between developers necessary.

Common cases:
 -single run of execute() methods on different AST strategies.
 -multiple runs
 -monsters/npcs exiting, entering correctly reflected
 -correctly responding to spawning/dying of a character.
Uncommon cases:
 -no monsters/npcs present
 -many in small area.
 -many players vs 1 monster
 -many monsters vs 1 player
 -many characters in response range of one npc.
Edge/Corner cases:
 -repetitive npc interaction.
 -unexpected npc interaction.
 -
Illegal cases:
-testing exceptions thrown correctly

DELIVERABLES:

M2: nov20th 11:59 pm

NanoLexer class, documentation, tests.
NanoParser class, documentation, tests.

The NanoLexer MUST be able to correctly turn any text into correct tokens.
The NanoParser MUST be able to correctly parse NanoC grammer into correct ASTCells. The execute methods of these cells does not have to be implemented.

M3: dec4th 11:59 pm

ASTCell hierarchy complete, all execute() methods.
AIHandler class.

All ASTCells MUST have correctly functioning execute() methods. Test suites and documentation MUST be complete.
AIHandler MUST correctly initialize and load script. Test suites and documentation MUST be complete.
Actual NanoC scripts do not need to be finalized.

ROLL: dec16th noon

All NanoC scripts complete.

MUST have 4 monster scripts.
Must have 5 npc scripts.
One of the npc scripts MUST be a merchant script.

==========

User Interface (Ian Mallett)

=== Graphics ===

The Graphics of the game are set up as a plugin to the Game Mechanics. When updated, the Graphics polls the Game Mechanics for information about the current game's state. The graphics then draw the game.

The Graphics are implemented as two to three major classes: an abstract graphics class that handles basic commands and overall structure, and one to two concrete classes that handle implementation details of rendering. A class implementing these details for CPU rendering is currently provided, while a class for a GPU implementation may or may not come to fruition.

All visible elements (with a few exceptions) are handled by the Graphics. The abstract graphics has a draw method, which invokes abstract methods, like drawHUD and drawTerrain, which are implemented by the concrete renderer(s).

=== GUI ===

the GUI is implemented primarily as a number of classes within the Graphics. These include the inventory pane, chat panes, etc. In some cases, the required functionality is too small to warrant an actual class. For example, the HUD bars simply poll the game state for the latest values, and then draw an image. These are implemented as a method in the concrete graphics.

The whole Graphics resides inside of a Swing context. Display updates are handled by a Timer that sends refresh commands in accordance with the frame rate.

Due to technical limitations with Swing, the way the command architecture will most likely be structured is as follows. User input will be caught by the Swing framework, and then processed into commands by Main. The commands will execute on their own thread, so as not to conflict with the the rendering. As before, the Graphics will poll the current state, using the newest available data. This allows commands to execute and be seen as they are executing, which is important for animation.

=== User Input ===

The exact mechanism of the game's controls are under revision, but it is likely that the player shall be controlled primarily with the mouse. Users will click on the square they would like to travel to, and the player will find its own way there, if possible. By holding down a given key while doing this, players will be able to attack or cast or otherwise interact with the object in the ending square.

For example, if "x" is held down, and the user clicks on a monster three squares away, the user's player will move until it is adjacent to the monster, and then attack it. If "z" is instead held down when the user clicks on the monster, the user's player will approach the monster and then cast the currently selected spell on it. For ranged spells and attacks, the user's player need only approach to within range, although ranged interaction may or may not be implemented.

Players can click on spaces that are vacant to simply move there.

==========

Game Content (Taylor Berger)

Definitions of Characters, Environment, Quest, Items, and testing strategies
* CHARACTERS
- Defined Character class structure with concrete classes defined. Class tree as follows:
  = CHARACTER - Overall Abstract Class that encapsulates the CREATURE from the spec. Contains a String name, Gender Enum, animation and an inventory.
  = NPC - Subclasses CHARACTER. These are simply instatiable versions of character that can be typed as an NPC. These will have an AI scripting behavior method that shouldn't contain much. These will also contain quest event objects that can be completed or obtained.
  = CREATURE - Another direct subclass from CHARACTER. These represent all intractable creatures in the game world. All CREATURES have base stats and current stats. Both stat lists, current and base, will be stored in two arrays. All access to the stats must be through the Stat enum set, as each stat's index is defined by the enum Stat's .ordinal(). They all can equip weapons and armor to modify their character. Each creature has a levelUp() command that will increase the CREATURE's base stats by a particular amount specified in lower classes. They also include a recovery function that will cause them to recover when not in combat. CREATURES also contain a set of SPELLS based on their current SPELL set and level. These are the spells CREATURES will be able to cast
  variables needed to implement many of the functions in the CREATURE class. Other than that, these will also have their own AST they can recurse down to for their decision making process
  = PLAYER - Abstract class that extends CREATURE. These will contain all default information for the three required classes as well as any other classes we may decide to create.
  = CONCRETE PLAYER CLASS - either Thief, Mage, or Fighter (for defaults) that contain the static constants for a newly created thief, mage, or fighter.
  = CONCRETE MONSTERS - can be any of the 5 monsters required for default plus any extra monsters. Defines all static constants for each CONCRETE MONSTER
- Proper maintenance of inventory and spell sets.
  = Each character will have accessor methods to both inventories to provide them an immutable view of what they currently have available. Loot functionality is defined below in the ITEMS SECTION
  = All Spell Inventories will have the ability to add spells to the list, but only if the spell's requirements have been met. Each spell will have an integer level requirement and a set of spell requirements that the character must have before the character can learn those spells.
- Level up functionality
  = Each CREATURE will have the ability to 'level up' based on a given formula defined by each concrete instantiation of a creature. Upon a level up, a creature gains stats in accordance to the formulas and their base/current stats are recalculated. This will allow us to have dynamic monsters that upon a player defeat, get stronger. Hopefully it will add extra flavor to the game.

*CHARACTER TESTING
- Character testing will be done through JUnit tests and some in-game play test. Most of the tests can be achieved through JUnit tests with the proper structure. Other effects that depend on the UI to give commands will have to be tested in the ALPHA/BETA version.
- Character testing will be done by testing the initial conditions upon creation of a new player character and all other concrete implementations of Monsters or NPC's. Testing will ensure that operations such as equipping weapons/armor, managing loot/inventory, correct spell lists are maintained during level up, all current stats are correctly calculated, base stats are only altered during level ups, recovery functions, and multiple characters can exist without interfering with each other's stats or inventories.
- NPC testing will involve testing to make sure merchants can purchase an infinite* amount of Swag. NPC's will also be tested to make sure a player interacting with them, properly polls the first quest event off of a player's quest queue if they match. NPC's should also be tested to see if a player or monster can attack an NPC though this would have to be done through the UI after the attack command has been laid implemented.
- Game Characters will be tested for the correct ability to die and drop some arbitrary loot along with all MOOLAH on the ground on the creature's current cell.

* ENVIRONMENT
- Tile sets defined and mapped to correct images. Game board will be instantiable but only as a temporary instantiation as tile will be random. Portal tile will exist but upon creation they will not necessarily point to another cell and therefore won't be in the random generation of maps.
- All overlays used by the GUI were developed as well, including arrows and a tile highlight for selection purposes
- Terrain included so far: DESERT, FOREST, MOUNTAIN, PLAIN, PORTAL, SWAMP, WASTELAND, and WATER.

* ENVIRONMENT TESTING
- Environment testing will have to be tested with both JUnit tests and ALPHA/BETA testing
- For JUnit testing, simple tests should include getting the correct movement values for every terrain type. The inability to move into illegal terrain types.
- For ALPHA/BETA testing, all terrain types should be drawn and any special terrain types (like the portal) should be tested with as many players as possible to ensure their special properties are working correctly. Also, each tile needs to ensure that the correct texture was drawn.

* QUEST
- Quests will simply be a list of QUEST EVENTS that have to occur in a specific order. The easiest way to do this is to implement the Quest log as a queue and each action a player takes should check to see if there is a quest event contained within that action. If there is there should be

another check to see if the quest event matches the quest event next in the player's queue and if it is, it will simply poll that quest event, thereby updating the player's quest.
- QUEST EVENTS are comparable, singleton object that can be owned by any CHARACTER in the game. For NPC's and MONSTERS, the quest events will be static and final to ensure they cannot be changed. The human PLAYER's quest events will simply have a method (@override equals) to check to see if their next QUEST EVENT matches the QUEST EVENT owned by the NPC or MONSTER. Because there is no reason to have more than one of a particular type of QUEST EVENT, they should be singleton objects.

* QUEST TESTING
- Quest testing will be tested through a mixture of JUnit tests and ALPHA/BETA testing.
- For the JUnit tests, a simple run through of an interaction with another CHARACTER that has the correct next step in the PLAYER's quest log should trigger the poll on the PLAYER's quest log.
- for ALPHA/BETA testing, the correct interactions with another CREATURE should trigger the poll of the PLAYER's quest log rather than just any interaction with the CREATURE.

* ITEMS
- Items will be managed by a class structure similar to the player structure. Items class structure as follows:
  = Loot - A single loot item that encapsulates a list of all SWAG and a monetary amount. Acts as a portable/drop-able inventory. Every CHARACTER as a loot object as their inventory.
  = Swag - A single item that can be added to a Loot object. Every SWAG has a value, and encumbrance and can be sold to merchant NPC's
  = Weapon - An abstract weapon class that boosts combat damage and alters stats. A specific type of swag that has special properties. These items are equip-able but only as weapons. The abstract class creates its own damage calculator method and has a way of returning the strength multiplier
  = Armor - an abstract armor class that boost physical defense. These items will be equip-able but only as armor. The abstract class manages the method that returns the defense value of each armor.
  = CONCRETE WEAPONS - these are the weapons with the specific static values defined in the system specification plus any other weapons we decide to define.
  = CONCRETE ARMORS - these are the armors with the specific static values defined in the system specifications plus any other armors we decide to define.
- Equipped items should properly affect the player's current stats and should be viewable by the UI for command calculations.

MILESTONE 2: ALPHA RELEASE
Sunday 20th, Nov.
11:59pm

* CHARACTERS
 - Characters will be fully developed from here. All functionality for players should be done at this point which will allow for players to move around the game map. All animations should be shown for movement purposes. NPC's will be developed at this point, if just as placeholders within the game. Merchants will have a defined roll and should function within the required specs. But the option to interact with them would be limited upon the development of the UI.
 - All character tests for JUnit testing should be completed at this point. All ALPHA/BETA tests as possible with the implemented command set should be completed and documented as well.
* ENVIRONMENT
 - Portals should be active tile options at this point. The development of the first game map should be finished as well. There possibly might be time for a Map Creator.
 - Environment at this point should be finished besides any other Maps we decide to create.
 - Environment testing should be completed at this point as well. All ALPHA tests should include the portal transportation and the uncrossable tiles.
 - If there is enough time, a Map Builder would be a useful tool to develop
* QUEST
 - The main should have a developed 'plot' to each one with a working queue of actions that need to be performed to advance in the quest.
 - A "random" sub-quest may be able to be implemented here as well. Something like a kill quest that would send you to kill a random monster or series of monsters would be sufficient.
 - All JUnit tests for quests should be complete and any ALPHA/BETA tests that are possible with the current command set should be thoroughly tested and documented.
* ITEMS
 - Full item set will be developed by this point. All items will have their unique stats set.
 - All JUnit test will include testing inventory merging for pickups. Adding items to a player's inventory. Correct calculation of the total encumbrance of an inventory. Correct removal of Swag from an inventory, as well as cases where the swag you are trying to remove is not in the inventory. Proper equip and unequipped of armor/weapons and that effect on inventory.
 - ALPHA release of Inventory tests will be any interactions possible with characters. Full documentation for these tests must be completed.

MILESTONE 3: BETA RELEASE
Sunday 4th, Dec.
11:59pm

* CHARACTER
 - Characters should be relatively static at this point forward as all functionality should be developed.

- NPC's will have properly defined quest structures that can be followed and completed.
- Merchant NPC's can be correctly interacted with and they do buy and sell items. Inventory management is handled correctly during these transactions.
- Upon level-up all spell checking should be handled and appropriate spells should be learned as necessary.
- Any extra classes we choose to add in and their proper starting statistics/ inventory.
- All ALPHA/BETA tests should be complete with full documentation.

* ENVIRONMENT
- All maps created by this time.
- No Changes except for any bug fixes not completed during last rollout
- Any extra tiles we choose to implement, with the appropriate tests to ensure they are working correctly.

* QUEST
- All quests should be defined.
- "Random" quests should spawn properly at this point and be able to be completed
- Documentation of the completion of the "main" quest as well as at least one "random" kill quest.

* ITEMS
- No changes except for any bug fixes not completed during last rollout
- Any extra swag we choose to implement, with the appropriate tests for each extra swag item.

ROLLOUT:
Friday 16th, Dec.
9:00 am
FEATURE FREEZE Tuesday 13th, Dec.

**All core feature sets should be done by Milestone3. For rollout, this should be the embellishment of the characters. Adding any other feature sets should be tested and documented with the appropriate documentation.

* CHARACTER
- Full development of all Characters and all implementation.

* ENVIRONMENT
- Full environment and all tests associated with it. Full documentation of any bugs or anomalies in the map.

* QUEST
- No significant changes from last milestone. There should be all documentation for any other quests added

* ITEMS
- No changes to item sets from last milestone.

*******NOTES ON JUNIT AND ALPHA/BETA TESTING**********
For all JUnit tests. The tests should be written to include all normal functions for the particular class, but extreme test cases MUST be added as well. Invalid operations for all classes should be tested as well. These test suites should conform to standard testing practices.

For ALPHA/BETA testing, Tests should be run 25 times and only when all 25 tests return no bugs should the ALPHA/BETA test pass. Documentation for each test should be as follows:

Date:
Time Started:
What feature you are testing:
Character you are playing as:
Level:
Base Stats:

[
Bug found:
What happened:
Time Found:
Conditions before:
Conditions after:
What were you trying to do:
Terrain type you were in:
Characters in viewable range:
Other interesting notes:
]*

==========

Signatures and Agreements

I agree to all above listed specifications and design. I agree to all deadlines requirements and understand there MUST be three day warning to any changes to the Milestone requirements.

Team Signatures:

X_____ X_____

X_____ X_____

Client Signature:

X_____