**Name of the game: Cold Boot (Custom Game)**

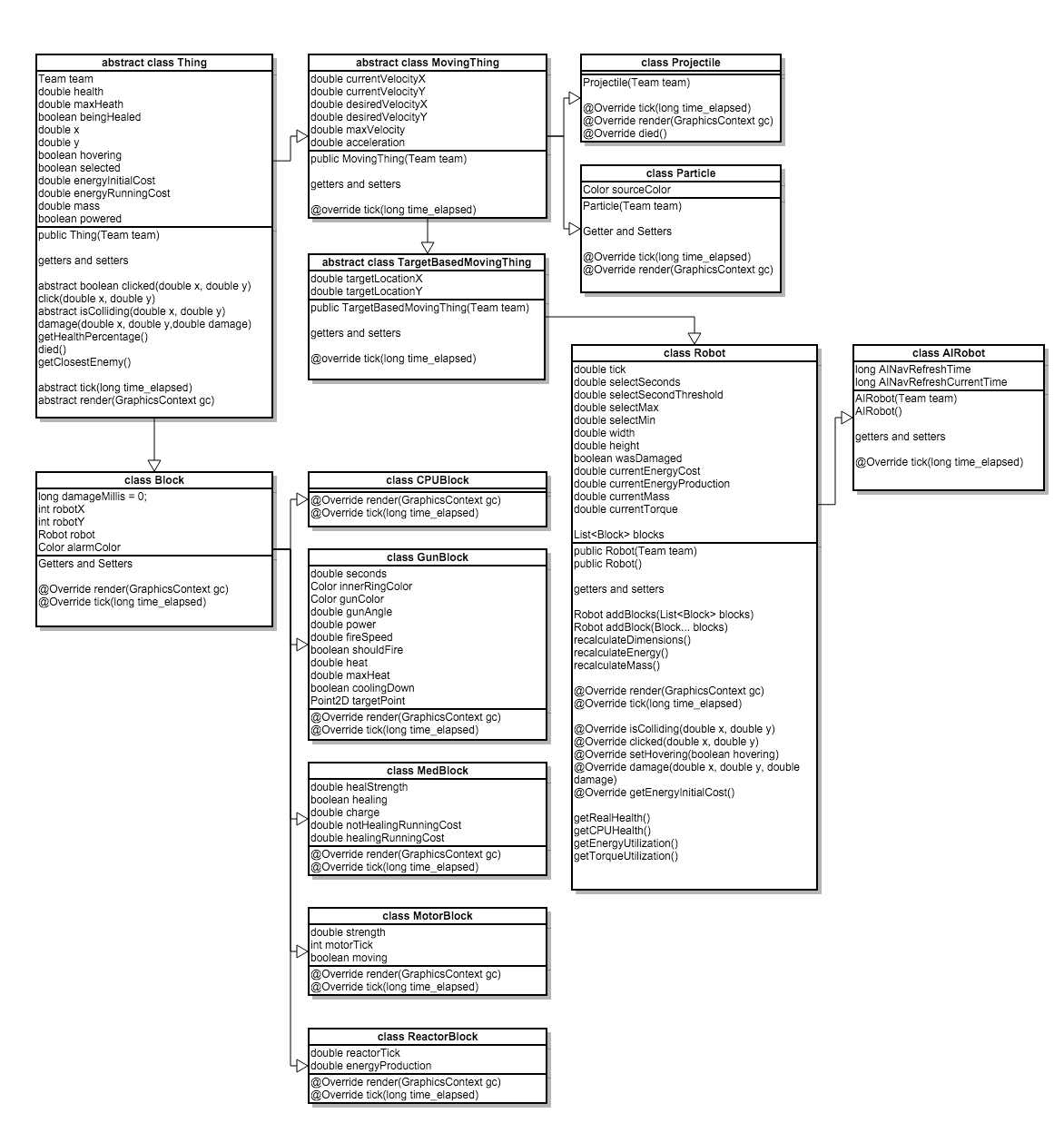
**1. Program Concept**

Our game is a real time strategy game the uses the design of the ships as the main strategic element. There are two teams, and in the gui, the player controls one of them. Each team has a base that must survive, otherwise you lose. There are also several factories that create robots. These robots are an amalgamation of different blocks. The different blocks include Guns, Reactors, Medical, Motor, and CPU Blocks. Every robot must have a CPU otherwise it instantly dies. Every block has different amounts of energy cost, running energy cost, and weight. And, these properties can change as the game progresses.

|  |  |
| --- | --- |
| Functionality Name | Description |
| Move Robots | In the GUI, the player can move their robots around the map |
| Spawn a Robot | In the GUI, the user can request a factory to start building a robot |
| Move around the screen | In the GUI, the user can move the screen around the world |
| Select Robots | In the GUI, the player can select multiple robots and tell them where to go |
| Inspect a Thing | In the GUI, the player should be able to view information on <? extends Thing> |
| View Game Status | In the Command Line watch a game progress. |
|  |  |
|  |  |
|  |  |

**2. Code structure**

**2.1 Class diagram**



The above is a snippit of the Thing class hierarchy.

**2.2 Briefly describe the overall structure of your code**

Every Thing in our game have a tick and a render. The tick is used to tell the object how much time has elapsed since the last tick (in millis). This is used for updating the position and action that any Thing is doing in the game. The render will ask the Thing to draw it’s self on the screen.

All of the Things are stored in an List<Thing> in the Game. The Game hase all of the generic game logic and is responsible for ticking all of the Things, but not rendering them. It also keeps track of if any player has won. We have two other classes ColdBootGui and ColdBootRepl. ColdBootGui is the graphical version of the game, and it is much more complex than the repl, but if fully playable. The Repl is a command line version, and only allows the user to view game stats, as the game is not optimal for a repl.

Thing is the class the represents everything that is in play. Concrete classes that extend this are Building, RobotFactory, Particle, Projectile, Block (all its subtypes), and Robot. There are many in-between classes that allow a simpler code flow, and less hacking. For example, projectile extends movingThing and so particle. Robot extends targetBasedMovingThing which extends movingThing.

Robot is the main Thing the player will operate on. It can be controlled, and send to destroy the other players base. It is an amalgamation of other Things. These Things are Blocks, they are independent entities that behave the rules of Things, but are ticked and moved by the Robot. This allows our robots to be fully destructible, which is where the strategy of the game comes from.

For our Repl, we have a Command interface that all Commands must implement, this allows us to call commands in a common and programmatic way.

**2.3 What are the data structures you used? Why did you used that specific data structure?**

We use List<Thing> and Stream<Thing> almost exclusively though our application as it allows us very fast filtering of information. Our project would have been simplified a great deal if we used a 2D grid as a backing, but instead we have look to see if any Things are around a given area. This slows us down greatly where there are a great deal of Projectiles or Particles. That said, Lists and Streams allow us to quickly filter and sort so in normal operation it turns out to be pretty favorable.

**2.4 What is the major algorithms of your program?**

Below is a piece of code that would feel at home in our program. It gets all of the Things from the current game. It then filters for only robots and casts all of the results to robots, then it makes sure that all of the robots are the Players team. Finally, for each robot, it instructs the robot to go to 0,0.

Game.game.getThings().parallelStream()

.filter(thing -> thing instanceOf Robot)

.map(thing -> (Robot)thing)

.filter(robot -> robot.getTeam().equals(Game.getPlayer()))

.forEach(robot -> {robot.setTargetLocationX(0); robot.setTargetLocationY(0);});

We use code like this almost exclusively as it is fast, and easy to write. This code is almost equivalent to a N pass over Game.game.getThings, but is a constant faster as it is running on every processor concurrently.

Thus code like this is used throughout, and is our main algorithm of finding and updating Things.

**3. Test**

**3.1 Unit tests**

|  |  |
| --- | --- |
| Test Case Name | Purpose |
| Game.testConfigureWorld() | Ensure that the game is capable of adding Things to the game |
| TupleTest.tupleTest() | Ensure that the tuple class correctly holds our tuples |
| ThingTest.testDamage() | Ensure that when damage is given to a thing it reduces it’s health. |
| ThingTest.testDied() | Ensure that when the health of a thing is reduced past 0 that it calls died |
| ThingTest.testGetPoint2D() | Check to make sure that the point we get back has the correct coordinates |
| ThingTest.testGetTeam() | Check to see if the team in the thing is the same as we told it to be |
| MovingThingTest.testTick() | Ensure that moving things move as expected. |
|  |  |
|  |  |

**3.2 System tests**

|  |  |
| --- | --- |
| Test Case Name | Purpose |
| GameTest.tickTest() | Checks to see if objects add to the system are added and ticked. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |