Implementation 2

Cohort 1, Team 4:

Robert Watts
Adam Wiegand
Josh Hall
Travis Gaffney
Xiaoyu Zou
Bogdan Lescinschi

To highlight the changes made to code we have done this using <change></change> tags, these are placed around a changed section of code (or in line if only one line has been changed) using comments. If a pre-existing piece of code has been edited the tag has a description of the change is included with the tags, if no description is included with the tags the code is purely new and has just been added into the old code.

To implement FR_POWER_UPS a new PowerUp class was created (as shown in architecture), this class extends GameEntity (and thus has a location and sprite).
The class includes an enum PowerUpEffect for storing the different effects, upon construction one of these effects is assigned to the powerup, it uses this to determine its sprite (the method spriteName returns this), and also, when the power up is triggered, to determine its effect.
PowerUp contains a method for each effect which applies it to the player when used.

Player was changed to include a boolean flag for some of the powerups (fast, invinc), and a variable storing the amount of shield the player has.
To facilitate the collection of powerups, the player's update method was changed, it now searches the player’s hitbox for powerups (activating all found).
Player was also given a damage method (this is not on the concrete architecture as we realised how useful this change would be while implementing PowerUps), which projectile now calls to damage the player, where shield and invinc are checked to see if the player's health should be affected.

Infiltrator’s handleTeleporterShot was changed so upon capture the infiltrator would create a new powerup (giving it a random effect).

World was given a series of new constants for the effects of powerups (buff duration, effect amount for each power up etc.)

To implement FR_DIFFICULTY the class World was changed, the previously final constants with values defined in code were changed to no longer be final (so they and thus the difficulty could be changed), and to no longer have their values defined in code, their values are now set using the method changeValues. These constants can be seen on the concrete architecture, only NPC_SPEED_VARIANCE is different as changeValues requires it to be the correct shape to set values.
changeValues uses difficultyData a JSONobject containing a value for each constant, difficultyData is set in the method changeDifficulty. changeDifficulty reads the JSONObject gameData (gameData’s information is the parsed json of difficulty.json, an internal file) and sets difficultyData to be one of the difficulty modes stored within gameData.
difficulty.json was created and placed in the asset fold of the game, it contains values for each difficulty-affecting “constant” (they are no longer actually constants, but are unchanged other than when changing difficulty) for each difficulty (easy, normal, and hard).

For the user to pick the difficulty MenuScreen was changed to include a difficulty button (which toggles between the available difficulties) and a variable which stores the current difficulty level, when the button is pressed World’s changeDifficulty method is used to change the world’s difficulty to the correct level.

No other classes needed to be changed as they already all referenced the “constants” in World.

To implement FR_SAVING a method to store each entity in the game and the general game state is required. A toJSON method was given to each important class (GameEntity, Npc, Infiltrator, Player, Projectile, PowerUp, World; as shown in the architecture) which produced an org.json (a library for handling JSON in java) JSONObject for that entity.

Each class in an inheritance tree was given its own method which called its parent class’s method, these toJSONs only added the details that its parent’s hadn’t (i.e. GameEntity.toJSON added position, Npc.toJSON added speed, state, and sprite name, Infiltrator.toJSON added the exposure state). This allowed the system to take advantage of inheritance to allow code reuse.

Player.toJSON stored the player's health and shield, as well as the status of its buffs/debuffs. In order to store the time remaining on some of these effects (confusion, slowed, blinded, fast, invinc) the task scheduling system was changed: tasks are now added to playerTimer via addTask. addTask adds it to playerTime and also adds the details of said task (its effect and time) to some ArrayLists.

For this change to the player's task scheduling system to work all places which use it (Projectile and PowerUp) must be changed to use the new system. filterTasks is called in update (thus in every render loop), it removes tasks which have been completed from the lists. The lists are read during toJSON, and the tasks time and effects are stored in player’s JSONObject

Infiltrator was given an ID system (id, curId, nextId, idCheck, entities) that gave each infiltrator a unique id which was stored in its JSONObject, this was used so the stored entities could be told apart when the save was loaded. This is because Projectiles need to know their originating entity so they hit them. Projectiles store the id of the infiltrator they originated from in order

World’s toJSON stored the systems details, the player’s JSON (generated via Player.toJSON), the difficulty, and the values depicting the current game state (e.g. infiltratorsAddedCount which keeps track of the number of infiltrators spawned, and thus when they should no longer be spawned)

GameScreen was given a saveGame method which creates a JSONObject “store” and stores the generated JSONObjects of Civilians, Infiltrators, PowerUps, and Projectiles (made via their toJSON methods) in JSONArrays which are put in “store”. World’s JSONObject is also put in store. saveGame then opens saves.json (a local file), parses it, adds “store” to it under the provided filename and then saves it.

GameScreen’s render method was changed to include a check for f5 being pressed, which would call saveGame with the file name being the time (as generated by java.util.calendar) with that time being saved in world.saveTime.
world.saveTime is used in GameUI’s new method drawSaveIndicator to display the save’s success to the user.

In order to load the games each important entity was given a new constructor method which used that classes JSONObject to create the object. Inheritance was similarly used as the JSONObject is passed into the equivalent constructor of the parent class.
Each of these constructors did the same as the previous ones for non-stored values (e.g. World's generates fleepoints like its original constructor), whilst also using the stored value to reconstruct each object's state.

Projectile's reads the stored id of its origin infiltrators id and uses Infiltrator.CheckId to fetch a reference to the newly created instance of its origin infiltrator

Player's reads its status effects and tasks, apply the effects of them and adding new tasks to unapply them (using the stored time remaining to give them the current duration)

GameScreen's reads saves.json, gets the JSONObject under the name given to it, and gives the various JSONObjects stored within to the constructors of said objects.
It constructs World first, calling World.addEntity on each of the subsequent objects constructed. It runs through the stored JSONArrays of Civilians, Infiltrators, PowerUps, and Projectiles, constructing and calling addEntity on each one (doing this for Infiltrators before Projectiles so Projectile can reference their origin infiltrator). Once the game state has been recreated it starts the game playing under that state.

GameScreen was given a new Load button which when pressed takes the user to LoadScreen. LoadScreen is a new class extending ScreenAdapter which allows the user to view, select, delete, and load saves.
LoadScreen reads saves.json, storing the stored filenames, when initialised and calls refreshText to generate a string representing the save menu.
It displays this string by calling com.badlogic.gdx.graphics.g2d.BitmapFont.draw (in its render method) to draw this string on the screen.
LoadScreen's render method has sections which check for various buttons being pressed:

- If "escape" is pressed it returns the user to MenuScreen.
- If "W" is pressed it scrolls the selected save up one (using a selected_line variable).
- If "S" is pressed it scrolls the selected save down one
- If "enter" is pressed it calls GameScreens new constructor, giving it the selected filename and changes the screen to that GameScreen (loading that save).
- If "backspace" is pressed it deletes the selected file from saves.json and its store of filenames.
- It then calls refreshText to reflect the changes on the menu.

To further satisfy NFR_ENJOYABLE sound effects and music were added to the game, each screen was given looping music using libGDX's audio.Music, selections on menus where given a sound effect (using libGDX's audio.Sound).
Various game entities were given sound effects for various actions (e.g. player being hit, teleporter being used, power up being collected etc.).
These sounds prevent the game from being silent, when the game was silent it was much less engaging and thus less enjoyable. The sounds also gave additional feedback to the users about the game, e.g. notifying them when a system was attacked.