Peer review for OOPP WITH THE BOIS

1 Design principles and design patterns

"OOPP WITH THE BOIS" project's design and implementation display several uses of design principles. It's clear to see that the group has worked with the **Single Responsibility Principle** in mind. The group has done a good job of achieving this for most classes. The class **Colony** (com/thebois/models/beings/Colony.java) serves as a collection of player controlled pawns, it has methods for allocating and deallocating rolls to the pawns and finding idle beings for instance, but the class also manages the colony's inventory and has several methods related to that. The Colony class has more than one responsibility and a suggestion would be creating a brand new class "ColonyInventory" that has the sole responsibility of managing the colony's inventory.

The group has done an excellent job of following the **Open/Closed principle**, they have in many cases tried to predict further possible extensions for both methods and classes by writing their code in a manner that would support this without requiring them to change a lot of their previously written code.

The **Liskov Substitution Principle** does not seem to be broken anywhere.

While the **Interface Segregation Principle** is followed in most cases, there are some violations to the principle as of now, for example StructureController and RoleController depend on a method update() even though update in both cases has an empty method body. If these update methods don't have any planned further usage, a suggestion would be to split the interface that forces these unnecessary dependencies into smaller interfaces so that the classes implementing them aren't forced to depend on unused methods.

The group has done a good job of following the **Dependency Inversion Principle**. By using and depending on abstractions they have decoupled their high level modules from their low level modules. Their abstractions don't depend on details which further goes to show that they have followed the principle.

The project uses some design patterns including the **MVC pattern** which is good as the model is separated from the rest of the application. By using this pattern the project has resulted by becoming well structured and more understandable. In the few cases where the model needs information from either the view or controller the **observer pattern** has been used which allows the model to access said information without having to rely on unnecessary dependencies. There is one instance of the **singleton pattern** being used to create a single EventBus object, this same object is then used by several classes. The **Marker interface pattern** has been implemented by using the empty interface IEvent to act as a tag. **Factory pattern** is used in the code to assign roles, this allows for easier creation of objects as you hide unnecessary complexity behind the creation which simplifies the assignment of roles in the program. There are good uses of **Adapter pattern**, one example is the ViewportWrapper which transforms screen coordinates to world coordinates for the StructureController without creating dependencies and giving access to all of ViewPorts other methods.

2 Documentation and proper names

Lots of fairly simple methods lack documentation, it seems like they are thought to be self explanatory since they are so simple. But in a lot of cases a small comment describing conditions, arguments etc. would be nice, for example there are some methods named "tryXxxXxx" that dont specify when the try succeeds or fails. When looking at the code to try and understand the method the if-statement simply calls another method. This makes it quite complicated to get an overview for when the method should be used. Also the pawntastic which is the application and perhaps "face" of the code lacks a lot of comments so it's hard to get an overview of the functionality. Classes and bigger methods are fairly well documented, and methods and functions in general have clear, understandable names.

3 Modular design

→ com/thebois/controllers/game/ColonyController.java
The method update() only calls for an update()-method in ColonyView which creates an extra unnecessary step and dependency. The method has usages in WorldController and InfoController in their own update()-methods which in turn has usages in Pawntastic. Consequently, Pawntastic has created a dependency for WorldController/InfoController, ColonyController and ColonyView for the sake of one method. It should be considered to loosen up these dependencies by another solution.

→ com/thebois/models/beings/Pawn.java
The code is in general very modular. The classes have been created with modular thinking and with a clear purpose in mind. However the class Pawn has hardcoded the WORLD_SIZE variable. If the program were to be changed for a different world size, this variable would need to be updated as well. There is a very high risk that this would be forgotten and hard to find the cause of error when they occur. It is commented that it will be removed when the pathfinding is implemented and since A* is implemented and a confirmed implemented user story, here is a little reminder.

4 Other

The code uses a good amount of abstractions, there are many parts which adhere to the open-closed principle allowing for extension if necessary. However at times the code uses what might be considered an excessive amount. For example there is an interface IBeing as well as an Abstract class which implements the interface. It's understandable to have the option to extend the program to use more than pawns to make use of the abstract being class but it's hard to picture the use of both interface and abstract class. The same can be found at other areas in the code as well.

There doesn't seem to be any major issues regarding tests,security or performance.

5 Additional suggestions for improvements

→ com/thebois/views/game/ColonyView.java
In the method draw there's an if-statement checking if colony is not null. As of now colony is only null once, which isn't a big problem as of now, however in the future depending on further implementations and extensions to the code there might be more cases where colony is null for whatever reason. By making it so that the method throws an exception whenever colony is null instead of relying on an if-statement checking the condition, it would be easier to go back and handle errors related to this. This would result in making the code a bit more maintainable.

→ com/thebois/models/beings/AbstractBeing.java
In the method move() there's a lot of things going on, it checks if the path is empty, calculates the distance between the current position and the destination, total distance is calculated using pythagorean theorem and it updates the current position etcetera. One method should preferably do one thing and do it well. A suggestion would be to extract things from the move method into other methods until the statement is true. There is an expression called "Extract till you drop", meaning that you should extract things from your method until you can't extract anything anymore. This term was founded by Uncle Bob and this (https://sites.google.com/site/unclebobconsultingllc/one-thing-extract-till-you-drop) blog post of his gives a clear example of what it really means. There are some other methods that can benefit from "Extracting till you drop", however most of the methods in the project are very well written and this method of extracting isn't always necessary, just a nice suggestion for further improvement.