Discussion Architecture Design (Group 7)

Our Design Choice

When looking at the type of game that we are gonna create, that being a detective simulation game-ish kind (I know loose with the terminology). We think the event driven architecture style would fit well, the purpose is that events drive the interaction between program components instead of individual requests. Which seem to fit the description of the user arriving to different scenes and interacting with it, causing different things to happen depending on interactions and situations, that is different events. We also believe this to be quite easy to scale later, when adding more features in future iterations, especially the multiplayer feature. This is because events triggered by one player can affect another's world.

When looking for downsides and researching this architecture type, we have reached the conclusion that the system may become complex and debugging will be difficult.

Our thoughts on how we designed the packages

> ***NOTE:***
>
> The class relations were taken from the interaction diagrams, therefore game has a bit to much responsibility and could be decoupled by delegating work to different handlers for example a GameHandler. Its role then would be the first interaction from UI requests, that is the controller, and only being the information expert on the fundamental game area (game loop).

- UI Package: All the UI logic exists in its own package with their responsibilities. That currently does not exist in this use case. Nonetheless it's there and has a relation to the overall system.
- Systems Package: A system package for containing system related aspects such as the game class, where its responsibility is dividing up tasks. In this package other classes that deal closely to the system exist such as potential game handler, thereby lessening the games responsibility by indirection.
- GameSystems Package: For the functionality of content with the game that is stories, interfaces, events and so on, we created a package called GameSystems, that contains all classes relating to how the game functions, to increase the modularity and cohesion of the program.
- Environment Package: This package contains the knowledge and logic behind different environments and how different environments within the game function, therefore separating it into its own package, we believe we achieve higher cohesion with this. In this package we placed the Scene class which is a class representing different environments within the game.
- Interface Package: We decided to separate everything that has to do with different interfaces into its own package for handling different interfaces and their logic and therefore organizing responsibilities. In this package we therefore placed the characterInterface which is a middleman component for interacting with characters. We believe this approach will make it easier to manage and implement more interfaces.

- Event Package: We created a package called Events that is separated handling of different events within the game and therefore handles how different game components work together. This aligns with our design choice.
- Entity Package: We have a package for entities containing sub-packages separating characters and other GameObjects. And containing their different inheritance classes that specify their different kinds of characters and gameobjects, then through the grasp pattern polymorphism effectively manage the different types, improving flexibility and maintainability.

> ***NOTE:***

> Was unsure if we were supposed to add inheriting character classes from the interaction diagram, but since there was nothing said we decided not to do it. But if we did it could looks something like this.

```
class Secretary
class Npc

Character <|-- Secretary
Character <|--  Npc
```

- Interaction Package: A package for interactions which separates interaction logic and management, therefore encapsulating handling of users interaction with characters and gameobjects. From the use case classes we added the interaction engine.
- Storage Package: Following the same pattern we gave storage related logic its own package, from the two use cases we added PhoneBook being the storage of contact information and CharacterRepository being the storage for characters.
- Utility Package: We added a package for utility functionality, where common utilities that could be useful in multiple parts of the system exist, also for other programs. In it we place the InputSanitiser class that deals with well.. "Sanitising" user input.
- Plot Package: The overall plot also got placed in its own package where other future classes dealing with plot reside, following the information expert pattern.

Reasoning behind the packages

Our thought when creating the packages was to separate the different package responsibilities and by grouping related classes from the domain model, as well as thinking of future iterations and improvements to a reasonable extent to avoid technical debt. Our decision on the packages and classes followed different GRASP principles, mainly information experts, thinking what the packages responsibilities would be and also enabling indirection between classes, that is outsource to someone else. This follows high cohesion and low coupling and follows our architecture style.

Because the event-driven architecture we choose improves maintainability and changeability by reducing coupling between components by bridging components communicating using events. This in turn makes it so changes to one part of the system do not affect other parts, because they don't need to know about one

another. This architecture style provides the flexibility to scale the system by adding more components or removing them without affecting the overall system.

Other alternatives

Other types we looked at where such as microservices and layered architecture types.

Microservices

We believe microservices could be useful as it looks more at the components and their individual relations/connections, making scalability easier. Downsides being increased complexity in communications amongst all services, and an increase in maintenance cost (not that we are getting payed or anything...).

Layered

And the last we will mention here, layered architecture (much like onions). With having layers such as the UI (presentation), the backend (logic) and a storage layer (eventual database). The benefit with this according to us is that every layer has its own specific responsibility and communication appears only with closely related layers. This causes the program to be more manageable because of the boundaries between layers. Testing can be done on isolated layers, making a test driven development efficient. Downsides is that it is hard to change requirements, as many layers might be affected. Since this for now is quite a small program, creating layers may cause it to become overly complex.