This document is an excerpt from our development documentation, discussing the decisions we made during development.

Retrospective

In this chapter we will discuss the various decisions we did or forgot to do during during the development, we will explain why did we do them and what do we think about these decisions today. We will divide this chapter into three sections: Organizational Decisions, Development Decisions and Changes From Specification. The first two sections just talk about different decisions we made, the third lists things in which we diverted from the original specification.

Organizational Decisions

- We had meetings with our supervisor every two weeks, discussing our progress
  - Reason: Because he told us to.
  - Looking back: This was definitely a good decision, even if we were not the ones who made it. Regular meetings definitely helped us with motivation, since we did not want to go to these meetings empty handed.
- We decided to have sprint plannings every two weeks, using Trello board to organize our work and track our progress.
  - Reason: We needed to organize our progress somehow and from our experience, agile approach works best for small teams like this.
  - Looking back: This was definitely a good idea, though in practice there were setbacks. As we all had personal lives, jobs and other courses, we were not able to schedule regular meetings, so we had to schedule the meetings ad hoc. For the same reasons the amount of time we could dedicate to the project varied wildly, so in some sprints almost nothing was done and in other sprints we did great progres. Still, overall, sprint plannings worked out great, as when there was little done in one sprint, we had extra incentive to work in the next one, else we would fall behind. So if nothing else, this helped us track our overall progress.
- We failed to create a proper prioritized backlog of tasks in Trello. We had an Excel table with features we needed to implement and updated from time to time.
  - Reason: No idea. We already had the Excel sheet, creating the appropriate tasks would have been easy. But none of us did that.
  - Looking back: This was definitely a bad idea. The Excel sheet was not updated often enough and since Trello did not have all the tasks in the backlog, we had no idea how many tasks were still left. Also, there were several times when we thought we had all the features, someone checked the specification and found something we did not do yet. This would not have happened if we took the time to fill the Trello board with tasks.
- We used Slack for communication.
  - Reason: We needed a place to communicate regularly and we were all familiar with this one from our previous projects.
  - Looking back: This was a great idea. Not only does Slack help organize the conversations by channels, but since we use Slack only for projects and work related things, we can easily separate work/school life from our personal one by ignoring Slack during free time and by being available on Slack when working.
- In the last month and a half we failed to keep the Trello board up to date and we failed to have regular sprint plannings. We planned 50 days in advance what we need to do

before release and who would do that, making planning decisions ad hoc using Slack, basically having a 50 day long sprint.
    - Reason: As everyone was focused mainly on their work and there was no reason to plan, this happened naturally.
    - Looking back: Not maintaining Trello Board was definitely a bad idea. Someone would write a bug in Slack, then it would get lost in a stream of messages discussing some problems and we would forget about it. However, extending the last sprint was beneficial. There was really no need to plan, priorities were set, tasks were clear, we just had to work on them and make sure we do them on time and let others know if we need help right away.
- We had a fallback plan in case our artists failed to deliver the level we wanted to present on time.
    - Reason: The artists are working on the project in free time with no outside incentives, therefore if something unexpected happens, this project is a low priority to them.
    - Looking back: Definitely a good idea. Our designer had a lot of valid personal issues preventing him from working on the game and our artist could not work on anything without the designs. So once we were sure 2 months in advance that there is no way they would be able to finish the demo on time, we started working on the demo using the assets from the original Time Lapsus
- We did not work on any documentation during the project.
    - Reason: The project kept changing and we deemed it more important to work on functionality instead of documentation.
    - Looking back: We think this was not as bad of a decision as we feared. While yes, creating all of the documentation in the last month was unpleasant, if we were working on it all the time we probably would have some obsolete information here, because we would forget to delete it or update it. But we do think that we should have been writing the javadoc in code continuously, as while yes, the methods and classes might change, when changing them you immediately see the documentation right above it, so it is easier to keep them up do date. Unlike external documents like this one which are likely to be forgotten.
- We met in person for analytical meetings, i.e. when we were making architectural decisions.
    - Reasons: Analytical meetings were only about talking and drawing things, which is better to do it person.
    - Looking back: This was definitely a good thing, the analytical meetings were very productive and we usually managed to decide on a lot of things.
- We worked from home, we met only for consultations with supervisors and for analytical meetings, rarely just for programming in the same room.
    - Reason: We were unable to schedule a day when we would all have time every week.
    - Looking back: While we definitely would have been more productive if we were meeting regularly, working from home was not the catastrophe we thought it would be.
- We cut a lot of features from the specification.
    - Reason: As there were still many unknowns, we chose to be conservative in our estimates and cut things which were not completely necessary to the project.
    - Looking back: This was definitely one of the best decisions we made. Because of this we were able to complete the project on time without being overly stressed and without rushing things. And we also had plenty of time to polish the project.
- We did not agree on coding standards when starting the project.

    - Reason: We thought we were in a prototyping phase, that lots of the classes we were developing would be thrown away and that forcing some standards and reviewing them would take a lot of time. We were also not aware of Unreal Engine 4 coding standards.
    - Looking back: This was a mistake, as when we finally decided that we should apply the Unreal Engine 4 coding standard, we had cca 200 classes to go through and review, which was a very unpleasant process.
- We used Github as source control for both the plugin and the demo projects, which were in two separate repositories.
    - Reason: We all have good experience with Github. We split the projects because we wanted the demo and plugin to be completely separate things, like it would be for people using our plugin in real life.
    - Looking back: Github for our C++ plugin was definitely great. For the demo, not so much, as working in parallel was difficult. Unreal Engine blueprints are binary files that cannot be merged. So we had to manually synchronize ourselves using Slack so we would avoid conflicts. It would have been better to use Perforce or SVN, as they support locking of files, which is a better way of handling this problem.
- We did not do code review.
    - Reason: We did not find it relevant at the time, as we had no coding standards and trusted each other to make quality code.
    - Looking back: It was not the worst decision we ever made, because we had no rules regarding coding standards and documentation. If we had, code review would have been necessary.

Development Decisions

- We decided to prioritize simplicity and easy extensibility over maintainable and testable architecture, creating larger classes which are easier to understand and use.
    - Reason: We suspected that people extending our plugin would not bother reading the documentation and would expect to do some easy modifications directly from code.
    - Looking back: We will discuss each instance where we applied this rule separately, but overall we do think that based on the requirements we set for our projects this really was the best way to go forward.
- We chose to use a Game Context as a form of dependency injection.
    - Reason: Originally we wanted to access the plugin classes through the Game Instance, forcing the designer to use our Game Instance. However, we then found out that our classes could not access the Game Instance correctly, because they were not actors. Our solution for this problem was the creation of the Game Context class, which we started passing to all of our methods.
    - Looking back: Architecturally this was a good decision. Designers can create their game context any way they want and there is an easy default way to get one. This means that the designers can use the context in a simple way or a complex way if they require dependency injection. Although it would have been better for all classes on the Game Context to be interfaces. That would make it easier to create mock implementations for testing. However, we did not do that. Instead we forced people extending our plugin to inherit from our classes, so they

always have to first see the original implementation before implementing their own. There were also technical issues - properties and event dispatchers cannot be directly exposed through interfaces, so we would need to create a lot of extra code, making the interfaces unnecessarily large. Though this could have been solved by abstract classes. But using abstract classes would prevent a single class from having multiple interfaces.
    But even if architecturally speaking this was a good idea, getting a game context for every function call is a bit bothersome for the designers. But we did not find a good way to solve this issue, other than providing the Get Current Game Context method.

- We did not create any unit tests or any other automated tests.
    - Reason: We did not see them as necessary for a project of this scope and they were not required by specification.
    - Looking back: This was a good decision for a project of this scope. We never faced an issue unit tests would help us avoid - the original developers were always around and problems with existing code were usually resolved swiftly. The tests might be useful if we continue to work on the project for a longer time, if the project expands or if the developers change.
- We decided to use the Generic Graph Editor as a basis for our project.
    - Reason: It had the basic functionality for creating graphs.
    - Looking back: Great idea, finding this plugin was the thing that kickstarted the development, as we finally had a class we could use for reference that was not as complex as e.g. Behavior Tree Editor from Unreal Engine 4.
- We decided to create the demo in blueprints only.
    - Reason: We wanted to prove that it could be done.
    - Looking back: It worked, we proved we set out to do, so it was a good choice.
- We decided to localize only text, not audio.
    - Reason: While we found an easy way to localize text assets, we found no easy way to localize audio assets.
    - Looking back: We still cannot localize the audio, the only way we know of is quite complicated, so this was probably a good call.
- We decided to lock the Unreal Engine version to 4.19.2 when we started developing
    - Reason: While changes between engine versions are not that significant, there might be some changes that could break some feature of our plugin. So it is smarter to use one version only.
    - Looking back: We did not need any features from newer versions, so this was probably worth it, as we avoided potential problems with upgrading.
- We decided to make UAdventureCharacter and UInventoryItem classes into blueprints instead of data assets.
    - Reason: Data assets do not support behaviors, which both of these classes needed to support, e.g. what will happen when using the item.
    - Looking back: It was probably a good decision, but we are not entirely sure. For data assets Unreal Engine 4 already manages their instances and provides pickers with thumbnails. And there was a way of implementing behaviors. Each data asset could have a class that implements the behavior as a property. Basically, blueprints for behaviors, data assets for data. We could also add methods on the UInventoryItem class that would just call the methods on the behavior. However, in the end we decided against this approach for following reasons:
        - It would hurt extensibility, as you cannot inherit from Data Assets properly. So it would be impossible to add new field to the item class. This

        issue could be solved by having some general UObject as a User Data property of the asset that could contain anything, but that brings us to the other point.
    - This approach would have been much more complicated for designers to understand. Create one object for item definition, one for its behavior, one for the actor in the scene. Explaining to them that usually, using properties and methods on the item is enough, but sometimes you might need to access properties and methods through behaviors… This approach would definitely not be friendly to the designers.

Changes From Specification

- We decided to allow the designer to use a different game instance.
    - Reason: Once we decided to use the Game Context, there was no reason to use our game instance. We just left it as an option.
    - Looking back: There were no problems with this decision and it increased designer flexibility, so that is a good thing from our perspective.
- Whenever specification says that something is accessible through Game Instance, it is now accessible only through Game Context.
    - Reason: We started using Game Context.
    - Looking back: This caused no problems.
- Dialog Controller sends itself to the presenter, which calls the callback method on the controller. The original version expected that the presenter would expose an event dispatcher that would be fired when a long running operation would complete and the controller would bind to the event.
    - Reason: Interfaces do not support event dispatcher properties.
    - Looking back: This increased coupling slightly, as the Dialog Presenter now knows about the Dialog Controller. But other than that we do not see any issues.
- We decided to allow the user to install the plugin both as an editor plugin and as a project plugin.
    - Reason: There was no reason not to.
    - Looking back: This caused no problems.
- We removed the Generate Plugin Files menu item, as there was no reason to have it there.
    - Reason: We no longer use a custom Game Instance.
    - Looking back: This caused no problem.
- Item tags now use the Unreal Engine Gameplay Tags system, which means that the tags are not specified in the plugin configuration
    - Reason: We did not know about Gameplay Tags before and they fit our use case perfectly, so why reinvent the wheel.
    - Looking back: This caused no problems.
- We decided not to implement the Quest manager class.
    - Reason: All planned functionality was implemented on the Quest Graph in a more user friendly way.
    - Looking back: This caused no problems.
- The example implementations of presenters are not included in the plugin, they are a part of the template.
    - Reason: If not using our examples, they would clutter the project needlessly. Right now they are available only in a template if a user needs them.

    - Looking back: This caused no problems and it should make the plugin easier to use, as it does not have superfluous blueprint assets when not using a template.