TNM094 — Software Engineering
2. Design Modelling

January 7, 2019

1 Introduction

There are many ways to convert an idea into a final implementation of code. One important and powerful, in its full or by part, is modelling, be it Model Driven Development, Agile modelling or in other forms. With this approach a graphical model is used to design and define the software prior to implementation. This becomes a means for communicating ideas, testing concepts and protocols, exploring use of interfaces, defining parts and classes, and even generating code.

1.1 Purpose

In this exercise you will work with extensive UML modelling to structure and understand your program before writing a single line of code. The purpose of experimenting with this kind of tool is to see how you can explore programming concepts, through UML models at differently detailed levels, and create a natural progression into details, and even analyze possible behaviour already at an abstract level. This will allow you to familiarize yourself with the different diagrams and their strengths and weaknesses.

This exercise also requires programming and program design skills acquired in earlier courses. Therefore, it can be regarded as a test that you can fully benefit from what you have learned so far.

1.2 Prerequisites

It is necessary to have good understanding of the basic aspects of object oriented programming and object oriented design, to be able to perform and understand all part of this exercise. Please, before taking this exercise, review earlier material on abstraction, interfaces, polymorphism, how to extract classes representing real world objects, aspects and behaviour, how to select between inheritance and composition, and how to select and define object relationships.

1.3 UML Editor

There are many different UML editors available for various platforms. They provide different features and different levels of strictness. Some are even capable of exporting the final diagram as source code. The more advanced UML editors can export the structure to at least one programming language, many support a multitude. In our environment we have installed Astah-community, however you are welcome to use any tool of your liking as long as you can complete all tasks.

1.4 Documentation

This is not a manual so you are expected to search for information in manuals, tool documentation and command-line help. Useful sources for this exercise are

- The Unified Modeling Language Reference Manual
  https://www.utdallas.edu/ chung/Fujitsu/UML_2.0/Rumbaugh--UML_2.0_Reference_CD.pdf
- Astah guide at http://astah.net/resources/documents/astah-basic-operation.pdf
- Tutorial on Object Oriented Analysis and Design
  http://www.tutorialspoint.com/object_oriented_analysis_design/

1.5 Examination

After finishing all tasks in this exercise, show your results to a lab assistant and explain what you have done and what conclusions you have come to.

For the successful examination, please make sure that you fully understand what you have done and how the different diagrams relate to each other. Also, during the tasks, save intermediate steps so that you can show and explain the result from one individual task.

2 System to Implement

For this exercise you will design a basic library (API) for a 3D Particle System and possibly also provide implementations of some functionality. These types of systems are often used in games because of their impressive effect with relatively small amount of code. For an example of a 2D particle system, see http://jsoverson.github.io/JavaScript-Particle-System/. This example includes apart from the particle simulation itself also interaction, a graphical user interface for adjusting properties and pretty nice graphics. It is, however, common to divide a program into front-end (graphical interface and visualization) and back-end (doing computations). The library to design in this exercise will only handle the particles for some unknown application.

The library must

- be designed as an API to be included in OpenGL-based software, such as a game
- have an object oriented design
- allow the creation and configuration of several independent particle systems
- handle the motion of several particles in 3D (per system)
- let the particles die after their life-time has expired
- provide sources that generate particles, e.g. explosion source and cone source
- provide force effects that pushes them around, e.g. gravity effect and wind effect

Design your system to have two sources and two force effects, but through abstraction open for the future inclusion of more sources and effects.

3 Design Tasks

An incomplete model of the particle system API is provided for you. The only concepts already explored in the model are those for rendering particles. You will need to add support for sources and effects.

Task 1 — Understand the Current Diagrams:
Examine the already provided diagrams. Make sure that you understand the relationship between the diagrams at the different levels of abstraction.

There is no explicit relationship between the diagrams indicated in the UML editor, but one way to express *order* is by the level of abstraction, from high to low: Use case diagram → Activity diagram → Activity diagram with swimlanes → Component diagram. The rest of the diagrams in the current model, Class diagrams and Sequence diagrams, provide descriptions that can be translated into actual code and share thus the lowest abstraction level.

Task 2 — Use Case Diagram:
Complement the Use case diagram to make it as complete as possible, based on the requirements specified above. The typical *actor* for a library is the *client software*, the code using and calling the library. It is fine if this diagram starts of fairly plain, however do mark relations between use cases where appropriate.
Hint: Think, for example: "What would I, if I was *to use* this library, want to *control* and have the library *do for me*?", to explore the different use cases.

The Use case diagram models *what* can be done with the system. Based on this modelled information we can now explore *how*, first at an abstract level using Activity diagrams. You already have a basic Activity diagram over the typical execution of a use of your system, however it does not include your newly added use cases. In the next task you will complement the current or make new Activity diagrams to explore the general steps that the software will take to create and handle a complete particle system.

This activity modelling can optionally be divided into one diagram describing the initialization of the system, and one diagram describing the continuous execution. Include adding one source and one effect in the initialization, and updating particle positions, and drawing the particles in the continuous execution. Include at least one branch of exception, responding to an error, for example what will happen if the client software tries to create a particle with negative life time. Go back and supplement the Use case diagram if you realize that you've missed cases or associations.

Task 3 — Activity Diagram:
Add one or more Activity diagrams to describe the steps that the particle system library must perform to support the specified use cases. Break down the behavior of the system into steps, just like you would do when writing the actual code, programming. Each such step may be represented by a class method in the final implementation, but that is not known or of importance at this point.
[Note: Do *not* add swimlanes yet!]

The Activity diagram helps us visualize what needs to be done and in which order. We can now start reasoning about how the activities can be divided between different coherent units that each is concerned with a specific aspect of the system. To do this we add *swimlanes* that divide the Activity diagram into parts that represent these aspects. By moving activities between swimlanes, we visualize both what part or aspect of the system that will take care of each activity, and also transitions between parts or aspects of the system. These transitions may be represented by method calls in the final implementation, but that is not known or of importance at this point.

Task 4 — Lanes:
Adjust your activity diagram by adding *swimlanes* that mark which parts the system will have and which activities will be handled by each such part. You must also adjust the activities (split and merge where necessary) so that the activities in each swimlane are well associated with that aspect of the system.
[Note: This is where "separation of concerns" begins so be careful to divide the concerns into coherent units with well fitting activities.]

The lanes in your diagram should each have activities that are well related to the aspect they represent. If done well, the flow between the lanes can be identified as calls through interfaces and it should be possible to create a first preliminary Class diagrams, modelling the parts and associations. Observe that your design must be *object oriented*, which means that behaviour should be *encapsulated* together with associated data into objects with clear *interfaces*.

Task 5 — The Basic Class Diagram:
Create a basic Class diagram, based on the information in the lanes, containing the classes that will be involved in the final system and specify associations.

Now that we have an idea of which parts our system should include, we can start reasoning about *how* they will communicate through their interfaces. As we explore communication, the interfaces are further refined. Also, potential issues can be identified, which helps us adjust and refine the model.

Task 6 — Communication:
Use one or two sequence diagrams (or collaboration diagrams) to explore aspects of how the *client software* will call your library and how the library internally will handle calls to set up, modify and use the internal structure to draw particles and other data. Go back and adjust your previous diagrams when necessary.
Hint: Prefer Sequence diagrams, especially when creating and/or destroying objects is involved in the protocol.
Hint: Prefer Communication diagrams when many objects are involved.
[Note: To pass this task you have to show how the diagram can help to reason about the structure of the system.]
[Note: The client software will have to be included in your diagram, making calls to your API.]

If the communication between parts of the system seems to work fine, then we can start adding details supporting this communication. Observe that in Astah UML editor, if you added a method to a class in a Class diagram, this method can be selected when you specify a call in a Sequence diagram. This does not work the other way around.

Task 7 — Class Contents:
Add at least the most important members to the classes — both functions, to allow the classes to communicate, and attributes, that control their internal states and aggregations. Go back and adjust your earlier diagrams when necessary.

Actually implementing a model can take much longer than modelling it in a UML editor. Nevertheless, it is hard to know if a model even *can* be implemented without writing the code. You are not required to implement the full system, but you need to try out a part of the model to explore the connection between model and code.

Task 8 — Implementation:
Write code (in C++, Java or pseudocode) to demonstrate how the API can be used (the `main` function) and also code to show how at least one part of your API would be implemented. Use the class diagrams as blueprints. You will have to describe for the supervisor how the code relates to the diagrams.