PROJECT PLAN

Overview
Our assignment is to implement a program which simulates data traffic in a network consisting of arbitrary set of nodes and connections (links) between them. The program should simulate and log data traffic during the given time interval.

The project will be split in few sprints. Before writing any code we should agree on the style, class structure and interaction. During the first iteration we should do a bare bones implementation focusing mainly on the core functionality, which will be written from the scratch. Core should be written as modularly as possible and it should provide interface for separate graphical user interface objects. Possible GUI will be implemented later during the final phase of the project if we have the time, and it be written using SFML.

Tooling and iterative work methods

We aim to ensure quality of the code by documenting it appropriately and unit testing it. For these there is a wide variety of tooling available of which we will figure out the best choices for this team. These would include test frameworks, documentation generation tooling and deciding on a code style that we intend to follow (for example https://google.github.io/styleguide/cppguide.html)

Examples of this tooling would be google-test and gcov for testing and test coverage. We might use something like doxygen to generate code from the documentation in the code. Versioning would be handled with git and most probably on gitlab. Communication will be handled by means like Slack that can integrate gitlabs issue tracking and other events straight to a centralized communication system.

Builds will be handled with make and makefiles.

Gitlab’s branching and merging should follow feature branch model (https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow), in which we have a centralized master and dev branches of which each developer branches out their separate feature branches to work on. Forking model came up in our discussions, but we thought it might complicate the process. There is only four of us which implies two subgroups of two so the unraveling of potential entanglement of merge issues should be manageable.

Agile work methods will be followed in a relaxed manner. This means that we intend to apply Scrumm sprint type of iterations - two or three in the time span of the course - and circulating roles. After group discussions we found out that most of us are quite new to all aforementioned toolings, so it is important to keep the overhead of new techniques and methods, including the agile ones, on an appropriate level. In otherwords, we try to keep in mind not to over engineer the development system and its cycle.

Along the lines of the Scrumm principles, we adopt the ideology of committing to the sprint's items. This means allowing to withtake only what we can deliver during the sprint and producing a "working product" by the end of the sprint. This will be integrated into the master branch in git to serve as the "release".

This type of commitment and iteration intends to avoid the problems found in most waterfall style working methods, which include last minute integration, focusing on secondary priorities, neglecting the development infrastructure hygiene etc.

The first iteration will be a bare bones application focusing on satisfying the minimum requirements of the core functionalities with some demonstrable user interface - probably text based.

The next iteration intends to build on the insights and work done in the first iteration bringing more features as well as grooming the sprint process. Grooming refers to enhancing the development infrastructure in ways we find adequate and necessary during the first sprint. These enhancement might include CI system, if found necessary. Gitlab seems to provide some sort of CI, so automated builds could be seen as a viable option.

The third sprint intends to deliver the actual end product once again building on everything we have conceived in the previous sprints.

Architecture and Mopflow

The applications implementation will follow roughly an MVC-model with these prioritizations:

Must

model group of network nodes, packets and links with characteristics

- nodes have address
- send and receive 'packets'
- routes/paths and end hosts that 'run' the packets
- multiple simultaneous packets
- customizable 'schemas' for
    - nodes
    - links
    - applications

Should

- be easy to extend (class structure)
- packets have content
- f.ex.
    - source
    - address
    - data
- packets have size

Could

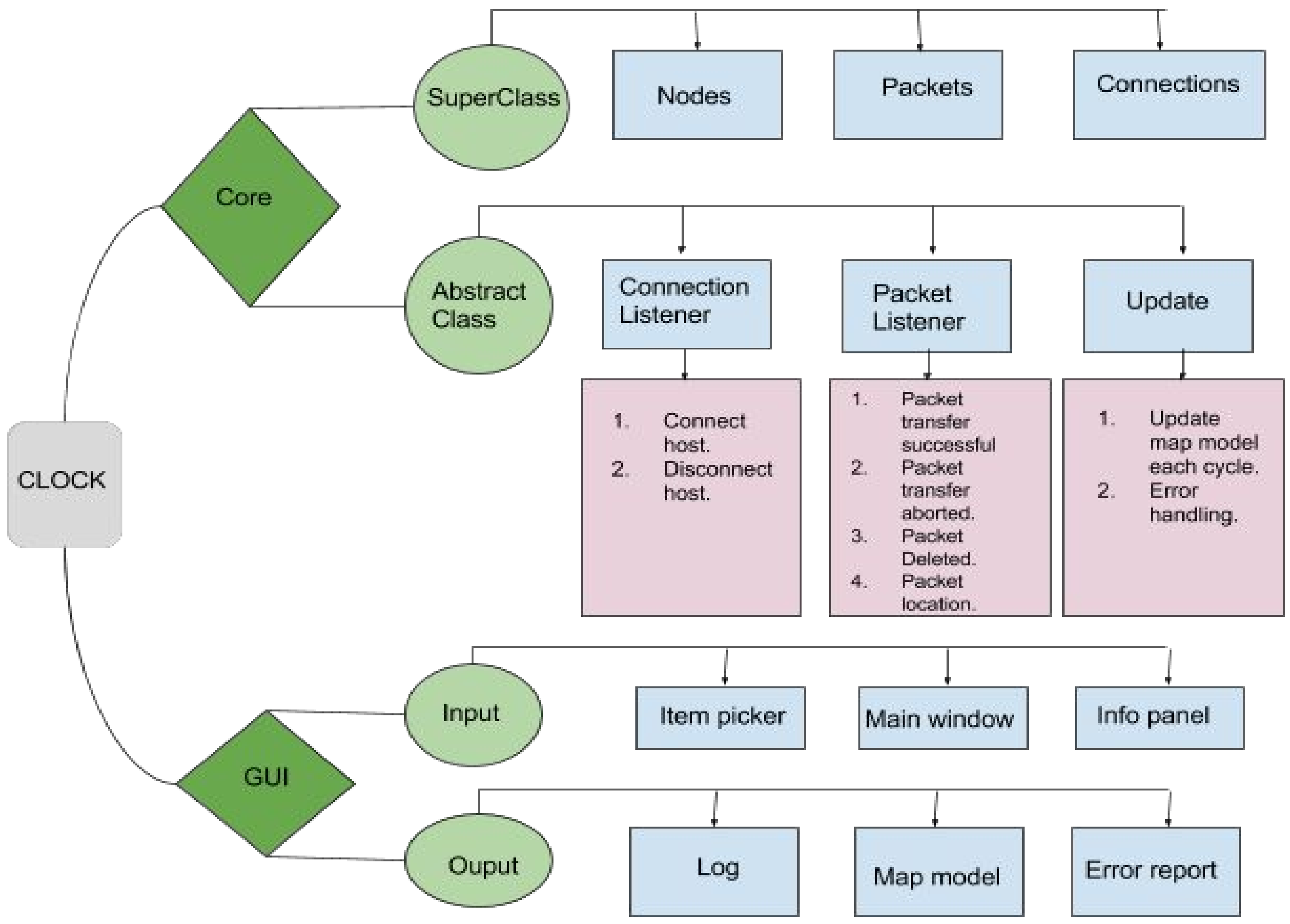
- addresses with ports for applications
- wireless links with data loss
- nodes that move
- animation of network traffic
- diverse queue behaviour

Won't

- use actual networking / as in purely a simulator
- be tightly coupled to a single user interface implementation

MVC

MVC could be described as diagram below:



Our Network simulator architecture is divided into tw3 main components
1: Clock
2: Core
3: GUI

1.Clock
We use this class to take care of all the timings of the events. We would be using threading in this module.

2.Core
Core would contain all the super classes and we would be using the superclass to design specific classes according to user need.
As every classes would need an Interface to perform all the actions we have considered some abstract classes where we will be defining all the actions performed by the superclasses and their subclasses. In the diagram we have mentioned some of the methods which we will be defining as the basic actions in the abstract interfaces.

3.GUI

GUI component will contain all the classes which are responsible for input and output graphical views.

students

Joni Turunen
Tommi Gröhn
Agrasagar Bhattacharyya
Chamran Moradi Ashour