POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Tesi di Laurea Magistrale

An open-source library for the visualization of complex graphs

Relatori
Prof. Stefano Di Carlo
Ing. Alessandro Savino

Candidato
Angelo Prudentino

Luglio 2017

Resume

The project has been developed with the purpose of building a powerful and dynamic tool to study complex graphs and networks. The idea was being able to display any kind of graph or network, in order to make the information carried by its elements and their relations clearly visible, and then, to edit such networks, to allow final users to easily enrich them.
All the software modules required to satisfy the specifications of the project have been developed using the C++ programming language; in fact, this project is mostly based on the primitive data structures and classes offered by the **Qt** development framework, which represents the only external dependency.

The final result of this project is composed by several interconnected pieces; the first one in order of importance is the **QtNets** framework, meaning the shared library that implements all the algorithms and data structures needed to satisfy the requirements briefly mentioned above. This framework is divided into four blocks, each one with its own specific task and a different level of importance in the overall structure.

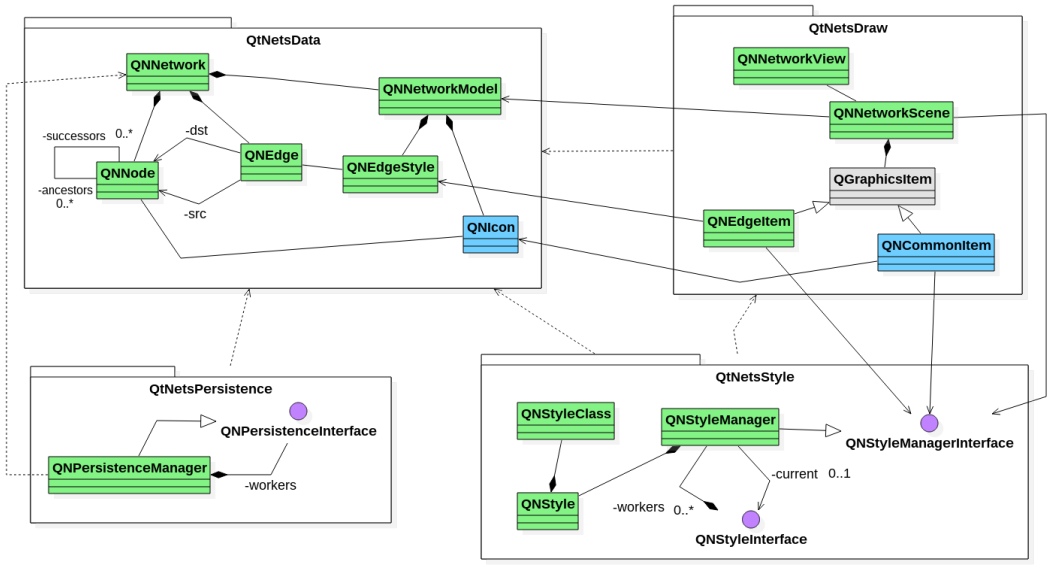

Figure 1: Structure of the QtNets framework

The first and most important module is named **QtNetsData** and represents the very core of the entire framework; it defines all the classes necessary to model a generic graph and all its

components, along with all the methods to retrieve and manipulate the information they store. The fundamental concepts at this level are modeled by few classes: the first one is named **QNNetwork** and models the whole graph and its generic properties, like its unique identifier or version; in addition to that, this class acts as a container for all the elements that constitute the network and manages their life cycles. Those elements could be nodes, modeled by the **QNNode** class, or edges between two different nodes, modeled by a specific class named **QNEdge**. For convenience, every node's instance has direct access to all the other nodes to which it is connected, as well as all QNEdge's instances have direct access to the two nodes involved.
All the classes mentioned so far are designed to model the core information associated to a generic network, but since the main goal of the whole project is being able to draw a network on the screen, it is necessary to model also a set of properties to define the appearance of each category of elements on the screen; in order to achieve that, a parallel data structure to manage such properties is necessary.
The most important classes of this "second-level" data structure are **QNNetworkModel**, equivalent to QNNetwork but focused on graphical properties, the **QNIcon** class, which is the base class to model every element rendered on the screen, and **QNEdgeStyle**, which models the graphical properties that might be associated to an edge.

The actual rendering of the network on the screen is managed by the classes included in the **QtNetsDraw** module, which implement all the algorithms and procedures required to paint the current scene and manage the user's interaction with it.
Everything that is drawn on the screen is organized after a precise hierarchy: the first layer is represented by the view, a scroll area provided with both horizontal and vertical scroll bars that acts as top level container for the scene to be displayed and offers native support to the most common transformations of the contained scene, like zooming requests and rotations. In the context of the QtNets framework, the view is implemented by the **QNNetworkView** class extending the Qt base view class named **QGraphicsView**.
The view contains in turn the scene, which can be considered the workspace where the user directly interacts with the network he is building. More specifically, the scene contains and controls all the items that populate it, but its role also involves the management of the external events determined by the user's interaction, like mouse clicks or context menus; the events related to the keyboard are also intercepted by the scene and propagated to the contained items to which they were initially directed.
As it was in the case of the view, also the implementation of the scene extends the native functionalities offered by the Qt development framework; in fact it makes available the **QGraphicsScene** class offering all the native support needed to implement complicated algorithms to respond to equally complicated interaction patterns, like, for example, drag and drop of different items. This class is used as a base for the **QNNetworkScene** class provided by the QtNetsDraw module to implement all the extensions or limitations necessary to be able to manage complex and generic networks.
At the bottom of the hierarchy, there are the single items modeling the elements of a network, meaning nodes, edges, text boxes, to enrich the pure structure with descriptive annotations, and models, to group different elements with common properties in sub-networks. Every item is implemented by a different and specialized class, but all of them have a common ancestor, offered by the **QGraphicsItem** class; its most important features are related to the management of the external events dispatched by the containing scene, grouping policies for items, like parent-child relationships, and standard procedures to enforce collision detection.

Once a network has been created and configured to fit the specific requirements of the case to study, it has to be possible to save it on a persistent memory device, in order to reuse it in the future; this is the concern of the **QtNetsPersistence** module, which offers all the instruments necessary to transfer the information from and to a memory device. This module has been designed and developed using a different approach from the previous two: it offers an interface that describes how information transfers from and to persistent memory has to be done, rather than a class that does it, following an approach that could be defined *"plug-in based"*.
In this way, several plug-ins implementing the common interface, named **QNPersistenceInterface**, can be developed to support different formalisms or different memory devices, leaving the specific characteristics of each one of them completely separated from the rest of the framework. The QtNetsPersistence module provides then a class, named **QNPersistenceManager**, specifically designed to manage the runtime loading of generic persistence plug-ins and the communication between them and the rest of the framework.

The QtNetsDraw module offers all the rendering utilities required to represent a network on the screen, but that representation could result incomplete or inappropriate in some specific situations; for this reason, the QtNets framework offers the possibility to customize networks using external modules. Those modules are also plug-ins and are managed by a further component of this framework named **QtNetsStyle**. Similarly to the QtNetsPersistence module, it offers an interface that describes how an external module can interact with the default rendering procedure in order to customize it and produce the required outcome. Also in this case, several plug-ins could be provided to address different kind of situations and all of them are managed by a concrete implementation of **QNStyleManagerInterface**, that coordinates the runtime loading of such modules and their interaction with the default graphic engine.

The QtNets framework, given its complexity, requires an effective testing and profiling system, therefore, a significant part of the project is represented by two analytic console applications: the first, named **QtNetsTest**, is a collection of unit tests designed to cover all the functionalities provided by the different modules of the framework and validate their correctness, while the other, named **QtNetsBenchmark**, executes a performance evaluation of every feature of the framework in order to roughly determine their time complexity.

Finally, since the QtNets framework is not directly executable, a traditional window-based desktop application, named **QtNetsEditor**, is provided; it simply offers an intuitive but powerful environment in which the final user can work on his networks interacting directly with the instruments provided by the underlying framework.
The QtNetsEditor application is composed of the following parts:

- a title bar on which it is reported the full path of the loaded network;
- a menu bar, where are exposed all the available actions, conveniently grouped according to their area of interest;
- a dock panel on the left side of the application through which it is possible to see and modify the properties of the currently selected items in the scene;
- a dock panel on the right side of the application through which it is possible to customize the currently loaded network using external style plug-ins;
- a central widget that contains the view on which the user can directly work.