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

Summary

The project discussed in this document has been developed with the purpose of building a powerful and dynamic tool to study complex graphs and networks, that was able to overcome some of the major limitations of other available tools and, at the same time, to offer an easy and intuitive framework to other developers who want to build their own custom application to fit very specific needs.

Networks of every kind, from social to economical ones, are constantly growing in size and importance during these years, so the capability of understanding the information contained in the relationships exposed in those networks becomes even more crucial; for instance, it is very common today that marketing campaigns of big companies and multinationals are driven by the results of complex studies about social networks and how their members interact among them and react to certain events.
It is easy to understand that, the more precise are those studies, the more effective will be the resulting strategy.
Another important application could be the analysis of the business relationships that exist between the members of a certain company and its partners, suppliers and customers; a graph modeling such connections between the different parties of a complex business process might be useful for example to determine the existence of weak links in the organization so to improve the overall productivity.

However graphs and networks are very important also in other fields, like the medical one; in this scenario, a graph structure could be used to represent the interactions between cells, or neurons, or several other parts of the human body and could become an essential mean to reach new and fundamental discoveries.

Those are just few of the many possible examples that could be proposed to convince of the importance of the topic and the necessity of a powerful set of tools to overcome all the related issues.
Therefore the aim of this project is to create a tool that is able, first of all, 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.

Acknowledgements

Un sincero ringraziamento va a tutti coloro i quali hanno permesso che questo importante lavoro potesse prendere vita, in particolare i miei due relatori, il prof. Stefano Di Carlo e l'ing. Alessandro Savino, che durante questi mesi mi hanno guidato e aiutato a raggiungere tale grande risultato.
Tengo tuttavia a ringraziare tutti i professori e i docenti che durante questi anni di formazione e di crescita tra le mura del Politecnico hanno contribuito a rendermi il professionista che oggi sono.

Il piú sentito dei ringraziamenti non posso che riservarlo alla mia famiglia, a mia madre Chiara, a mio padre Rocco e a mia sorella Irene, i quali mi hanno costantemente spronato a dare il massimo e mi hanno aiutato a credere sempre nelle mie possibilitá.

In ultimo, ma non certo per importanza, voglio ringraziare i miei compagni di corso Giuseppe, Daniele, Roberta, Francesca, Lucia, Omar, Enrico e Luca, che negli anni sono diventati degli amici per me insostituibili e che mi hanno aiutato a superare i momenti di maggiore pressione; grazie alla loro vicinanza e al costante confronto con persone brillanti e preparate sono riuscito a raggiungere risultati che da solo non avrei potuto sperare di raggiungere.

Grazie di cuore.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this project was to create a powerful and dynamic instrument to display and edit complex graphs and, at the same time, an easy and intuitive framework that other developers could use to build their own custom applications.
The resulting framework goes under the name **QtNets** and it is fully developed using C++ programming language and Qt libraries; for this reason the QtNets framework is fully portable on every platform and completely open source.
All the features implemented for this sake are ideally divided into two categories:

- core functionalities;
- optional functionalities;

There are many differences between the two categories, and those are going to be discussed in appropriate details in the next chapters, but the fundamental one is that core functionalities are essential for the whole infrastructure to work properly, while optional functionalities could be excluded, or simply bypassed, still guaranteeing the correct behavior of the other components.
The fact that some components of the framework are not strictly necessary to the others, allows to make them dynamically loadable as needed and other developers to produce their own version of such optional functionalities, without affecting the core; in fact, they do not even need to know exactly how the core works, it is enough to know how it interfaces optional modules.

There are four main topics addressed by the framework, and they can be summarized as follows:

- data representation;

- scene painting and user events management;
- data persistence;
- items graphical style.

A generic graph has been modeled in the most abstract way possible so to enable every kind of network to be represented using the provided structures; the set of classes implementing the data representation part are the very heart of the project.

Scene management, although very important, has been separated by the core, basically to make the data representation lighter and easy to be used in all those scenarios in which data computation is fundamental and the graphical representation is not even needed.

The persistence of the information associated to the graph is managed by a different module, optional in theory, but essential for obvious reasons.
What is important to say about this module is that it can support different data encodings and formats for the information to be stored, as long as the provided software component implements a common interface. In this way, different encodings and/or storage devices can be supported during time without affecting any other section of the framework.

The style module is very similar to the persistence one, in the approach at least, given the fact that concerning look and feel, customizable software is preferable; the module basically offers the possibility to customize items in the scene associating them style classes, while leaving the meaning of each class completely separated from the semantic of the graph, similarly to what happens with CSS classes in web pages.

Lastly, the framework is accompanied by a graphs editor application, that is built upon it and uses all the functionalities it offers. This application has actually a dual purpose: on one side it provides a complete and working editor application that can be used to evaluate the available features, or in those cases in which the network to study is quite common and a generic tool is enough; on the other side, it provides a valid usage example of the framework to all those developers who intend to build their own tool upon it.

Chapter 2

State of the art

This chapter will be dedicated to a brief overview of the most popular applications and libraries for network analysis available nowadays with the aim of pointing out their strengths and weaknesses; in addition to that, it will illustrate similarities and differences between them and the developed framework and editor, focusing the attention on those aspects they try to improve.

2.1 Cytoscape

The most powerful and spread networks analysis tool is certainly Cytoscape, distributed by the Cytoscape Consortium.

> *"Cytoscape is an open source software platform for visualizing molecular interaction networks and biological pathways and integrating these networks with annotations, gene expression profiles and other state data. Although Cytoscape was originally designed for biological research, now it is a general platform for complex network analysis and visualization."*

Cytoscape is an open source network analyzer, completely implemented in Java™, which offers both an open source developing framework and a desktop editor application.
The fact that it is a Java product represents the first remark: although the Java language is very easy to learn and use, also portable on every platform, it is undeniably much more inefficient than a compiled language such as C++; this is generally not a problem when using modern laptops or analyzing medium-sized networks, but it could become a serious obstacle when the graph to be studied is very large or the available computer is not very performing.

This is the reason behind the choice of the C++ language, along with the Qt framework, to develop all the software modules required for the project: being able to realize the most performing tool possible, capable to show without any problem networks with several thousands of elements.

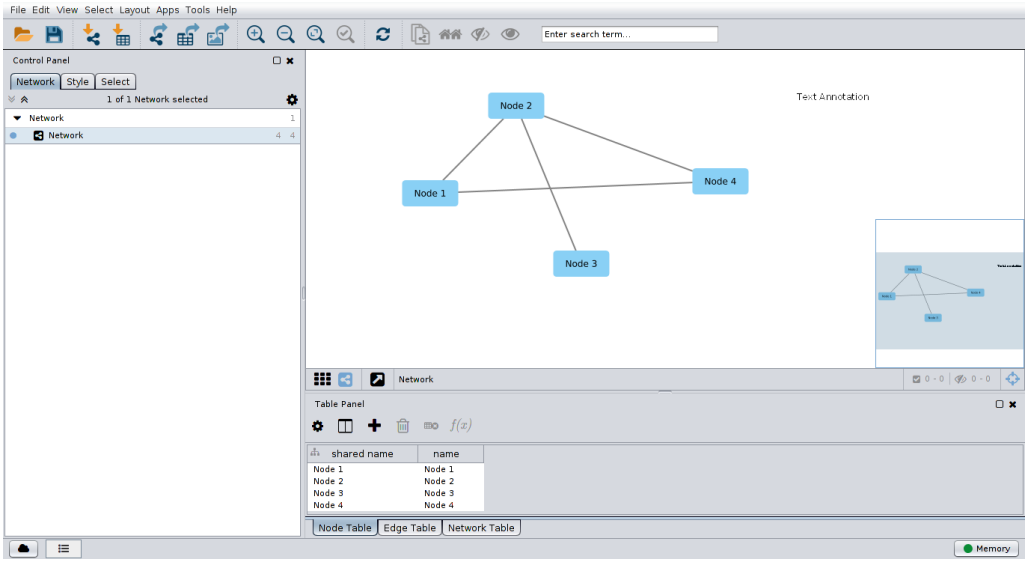


Figure 2.1: Cytoscape editor application

The editor is very complete and, to be fair, it has been a reference for many of the features introduced in the user interface of my own editor application; it offers the possibility to add nodes, edges between nodes and text annotations directly from the tools' panel and allows to analyze the attributes associated to each active element in the scene.
In the editor application developed for this project, I implemented all the functionalities described above, since they are the very basic features for an application of this kind to be effective, and also added the possibility to define sub-networks, so to allow different level of details for the same network.
How that has been accomplished will be discussed later.

Another very interesting feature offered by the Cytoscape platform is the possibility to customize the appearance of the items in the scene defining a style for the current working session; this means that is possible to define the look and feel of every element in the network, or networks, loaded in the current session and the style remains bound to the session rather than to the network.
I took inspiration from this feature and designed a slightly different styling system, in which there is no session, but the separation between the information carried by the network and its visual representation is still maintained. The session will be replaced by the association between items to be customized and classes that defines style properties for those items.

It must be said that Cytoscape is a very mature software platform, with a lot of extra functionalities with respect to the one I have designed, for example the layout functionality which allows to automatically rearrange items in the scene according

to a certain algorithm.
However this topic will be analyzed in further details in the last chapter of this document, where possible improvements for the framework and the editor application will be discussed.

2.2 GeNIe

The second application that is worth to be analyzed is the GeNIe modeler distributed by BayesFusion, LLC.

> *"GeNIe is a graphical user interface (GUI) to SMILE and allows for interactive model building and learning. It is written for the Windows environment but can be also used on Mac OS and Linux under Windows emulators."*

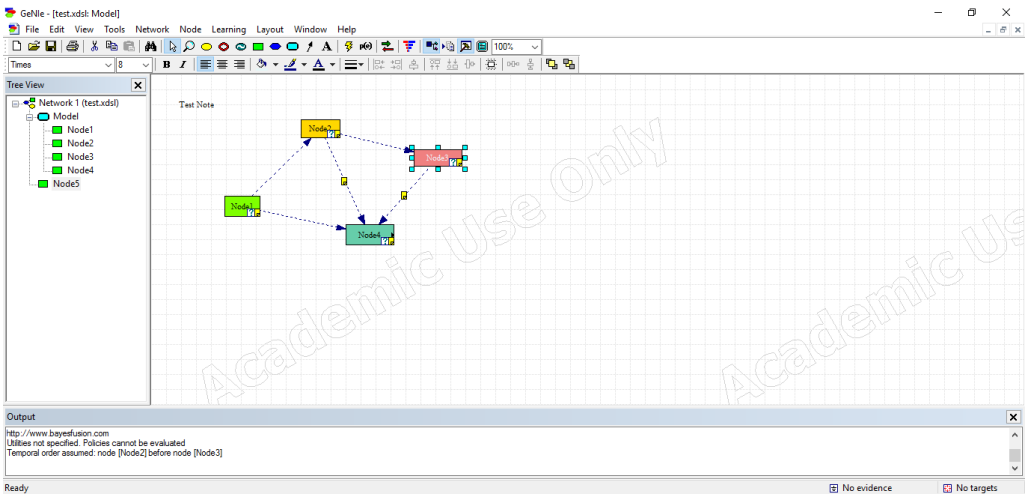


Figure 2.2: GeNIe editor application

The GeNIe application is quite intuitive to use, as it offers the possibility to draw or edit a network using drag and drop and other active elements in the user interface, but it is distributed for free only for academic purposes.
Furthermore, as already said, GeNIe is native only for Windows environment and this makes it very difficult to use in those cases in which such an environment is not available, basically because Windows emulators typically requires much more computational power than a native environment, making almost impossible to work properly on large networks.

The modeler is built upon a library provided by the same BayesFusion, named **SMILE**; it stands for *Structural Modeling, Inference, and Learning Engine* and is basically an engine for graphical models, such as Bayesian networks, influence diagrams, and structural equation models.
The SMILE engine is actually very much used in the academic community and it is portable and available for most computing platforms, also offering wrappers that makes it possible to use SMILE from Java, .NET, and other development platforms.

For this reason, my tutors and I, considered very useful to realize a module for our application that enables to edit SMILE networks so to allow those users who work on SMILE networks to easily move to our platform one day. This idea was then elevated to a more general level and leads to the dynamic persistence module that will be explained later.

As the previous tool, also in this case an automatic layout adjustment functionality is available, but it is way less sophisticated and precise than the other.

The SMILE engine, as well as the GeNIe modeler, also supports other more specific functionalities related to the kind of networks it has been primarily developed for: Bayesian networks, influence diagrams, and structural equation models; those functionalities will not be discussed here because the objective of this project is to realize a general purpose framework, that final users can eventually tune to make it fit their very specific case of study.

2.3 Graphviz

Graphviz is another very important and very much used graph visualization software distributed for free and completely open source.
Differently from the previous cases, graphviz is a developing framework only and does not provide any ready to use editor application, even though there are several tools distributed by other developers that use it library to display networks. Unfortunately, those tools are too simple and, most of the times, are just visualizers, without any other possibility to edit the network but the direct editing of the source file in which it is stored.

The following is an example of one of those tools, named xDot, that can only display and print out networks, as the picture clearly shows.

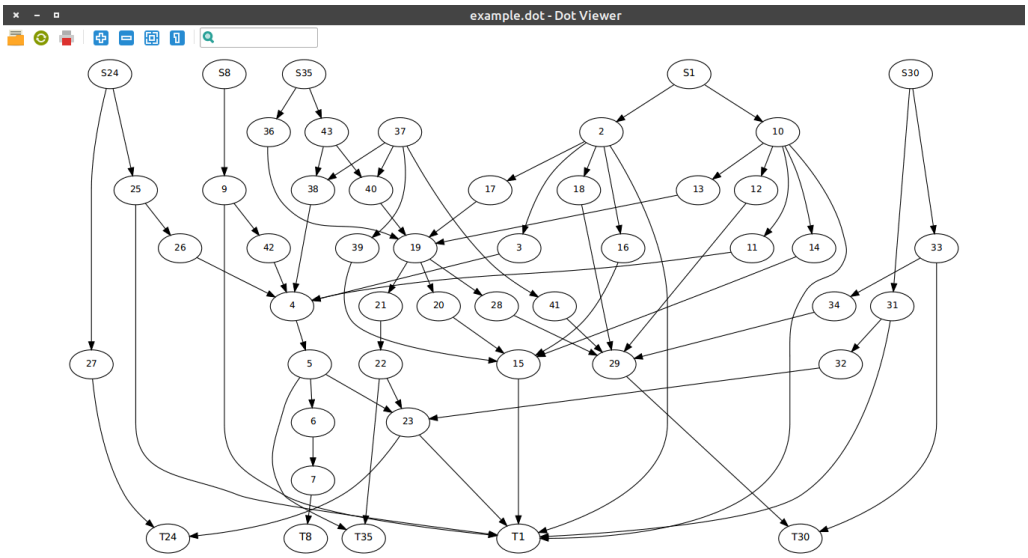


Figure 2.3: xDot visualizer

Given the limitations of the tools currently available, the best strategy would be to design one's custom application using the API (*Application Programming Interface*) provided by the graphviz framework, but this requires a great amount of time to prepare the tool before the needed network analysis could actually start, and some, at least basic, software developing skills, that makes the library, although powerful, unusable for a great deal of potential users that are just looking for a ready-made instrument.

Beside these problems, it must be said that the library is rich of functionalities and there are many tools, designed for very specific needs, that use graphviz to perform drawing operations. It provides a large number of possible layouts and a

large number of different elements, each one with its shape and proper attributes, so that is possible to display also very complicated graph structures; moreover it allows some customizations for elements such as nodes or edges, even if does not support a dynamic styling sub-system that allows to separate network's style from network's core.

Given the importance of the graphviz framework, it could be very useful to design, some day, a plug-in module for the QtNets framework to support its file formats and networks, so to increase the interoperability of the whole product.

Chapter 3

Development tools

This chapter has the purpose of presenting a brief overview of all the tools and platforms that have been used during the development of the QtNets framework and all the related modules and applications.

The first tool to mention is **StarUML**, a modeling tool that I massively used during the design phase of this project.
Then, when the code writing phase actually started, I used the **Qt** development framework as a base upon which building the entire project, and the **QtCreator** IDE (*Integrated Development Environment*), which is the official developing environment for Qt based applications.
Even if I was the only developer involved in this project, I decided to use a version control system anyway, in order to better monitor the evolution of the source code and to make it easy to go back in its history to find the cause of possible regressions and bugs. I decided to use git as version control system along with an integrated client application, named **SmartGit**.
Finally, during testing phases, I used a profiling tool to analyze the resource consumption of the developed code; in particular I used **Valgrind**, one of the most popular memory analyzers available.

3.1 StarUML

StarUML is a very complete and easy to use modeling tool, which supports a very large number of different diagrams for specific purposes; just to give some examples, this tool supports UML (*Unified Modeling Language*) class diagrams, activity diagrams, use-case diagrams, ER (*Entity Relation*) diagrams and many more.
The tool is protected by a commercial license but a free evaluation version is also available.
StarUML offers the possibility to automatically generate code starting from models, provided that the model is well formed, or to generate models directly from source code to perform some sort of reverse engineering.
Finally, it allows to share models with other people in several ways, by converting them into PDF files, by exporting them as pictures, that can be easily inserted into digital documents, or by publishing them on the Internet as HTML pages, automatically generated by the tool itself.

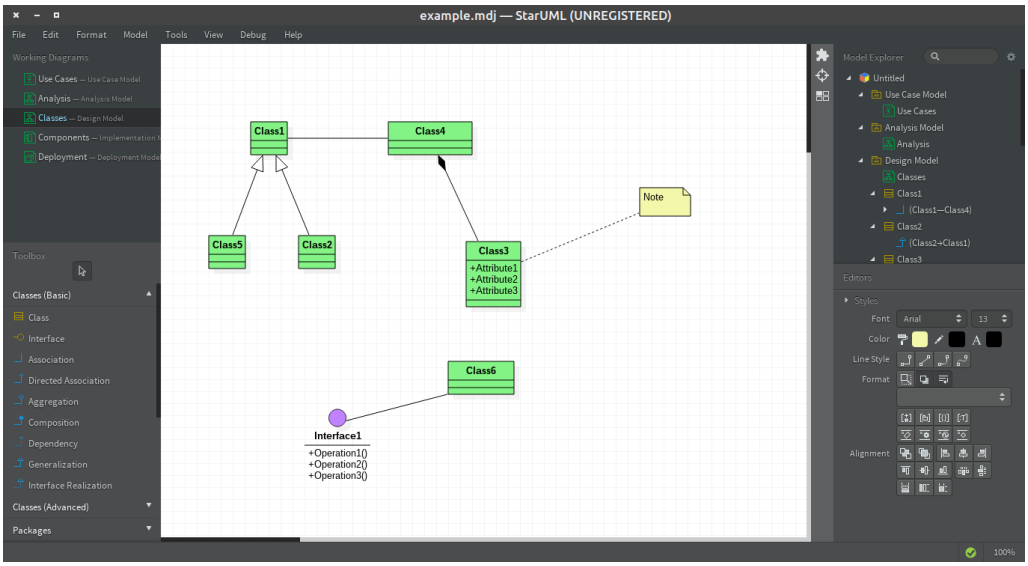


Figure 3.1: StarUML editor application

Concerning this project, I used the tool mostly to produce UML class diagrams modeling the properties and the interactions among all classes. Those diagrams have been very useful during the design phase because they allow to have a general picture of all the software infrastructure and makes it easier to find potential issues soon enough to avoid them becoming critical during the subsequent code writing phase.
For this reason, those diagrams are also very useful to explain the implementation details of the project, therefore they will be widely used in the following chapters.

3.2 Qt framework and IDE

Qt is a cross-platform application development framework that supports several different platforms, from desktop to mobile and even embedded systems; it is used by a good and constantly increasing number of companies all around the world, such as AMD, Siemens and Panasonic, just to mention some of the most famous ones.
Qt is distributed under various licenses, both commercial and free. The license agreement to be followed depends basically on the nature of the project to be developed: commercial license is mandatory for commercial projects, while open-source projects, as this one, can freely use the framework under the terms of one of the available GPL (*General Public License*) licenses.

> *"Qt is not a programming language on its own. It is a framework written in C++. A preprocessor, the MOC (Meta Object Compiler), is used to extend the C++ language with features like signals and slots. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler"*

That being said, I prefer not to go too much in detail about the Qt framework here, which wants to be just an introductory section, deferring the moment when I actually explain how I used it to develop this project.

The Qt framework is accompanied by a complete IDE, specifically designed to manage Qt based projects; its name is **QtCreator**.
It is not mandatory at all to use this specific IDE to develop a project based on the Qt framework, but it offers several useful functionalities, specifically designed to make the coding process easier, that really help to focus on important aspects and save quite some time. Valid examples are intelligent code completion, syntax highlighting, integrated help system, complete debugger and support to the most common version control systems.

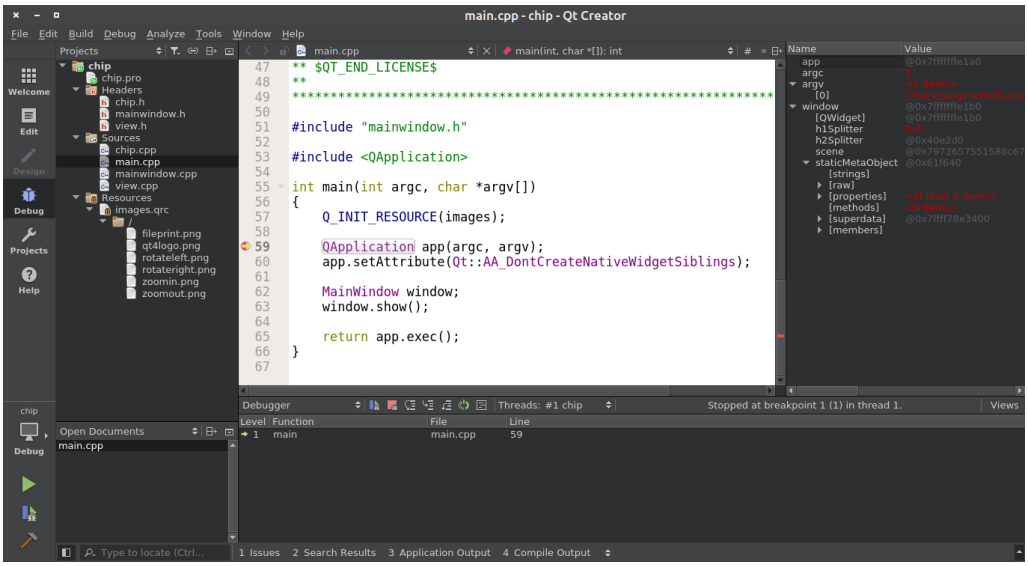


Figure 3.2: QtCreator IDE application

QtCreator also offers great support for the development of effective user interfaces; as a matter of fact, the IDE includes a section, named **QtDesigner**, that allows to create GUI (Graphical User Interfaces) without writing any line of C++ code but simply using drag and drop.
Thank to that, it is possible for user experience experts, which often do not have deep code writing skills, to design rich and effective interfaces without worrying about the underlying details.

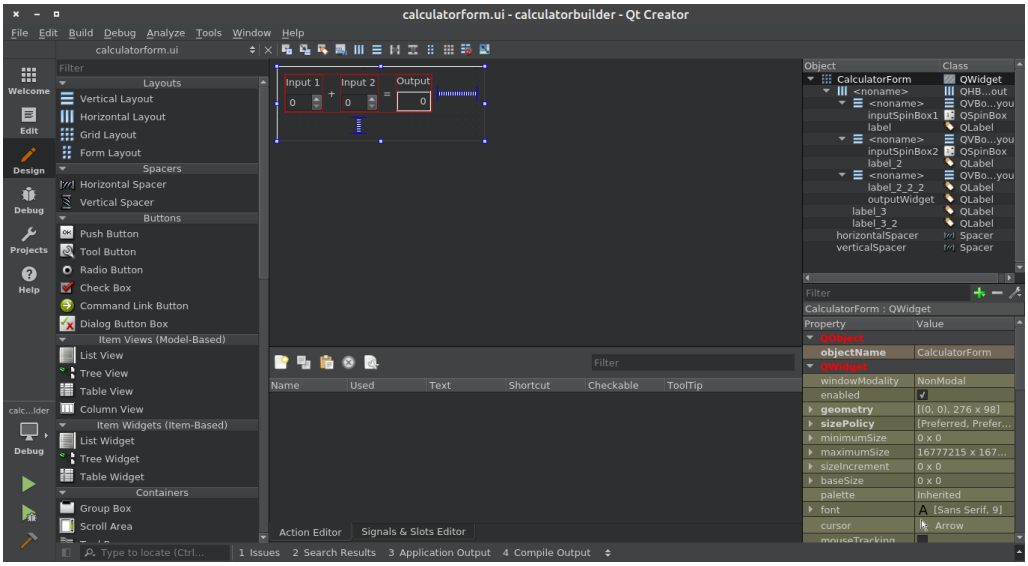


Figure 3.3: QtDesigner IDE application

Using this tool it is also possible to simply check the properties of all the available widgets, in order to decide which of them works better for the user experience to be realized, and to directly see the appearance of the interface while designing it; in this way the GUI design process becomes a sort of "trial and error" process in which possible inconsistencies emerge immediately, and not after a long and time consuming coding activity.
Personally, I used this tool in the first phase of the GUI design process of the desktop editor application developed for this project; it helped me to analyze strengths and weaknesses of the different choices so to reach the final balance. Only then I deeply studied the characteristics of the set of widgets I decided to use and coded the final solution, minimizing the chance of wasting great amounts of time pursuing dead-end roads.

3.3 SmartGit

SmartGit is a very convenient cross-platform git client that offers a simple integrated user interface to best exploit all git functionalities; it is also distributed for free for non-commercial uses.

Git is an open source version control system for tracking changes in source files and coordinating work among multiple people. It is primarily used for software development, but it can also be used to keep track of changes in any files.
As a distributed revision control system, its best features are speed, data integrity, and support for distributed, non-linear work-flows.

The SmartGit tool basically allows to execute every available git operation without actually need to write any complex command in the prompt, thus making the use of git much more intuitive.

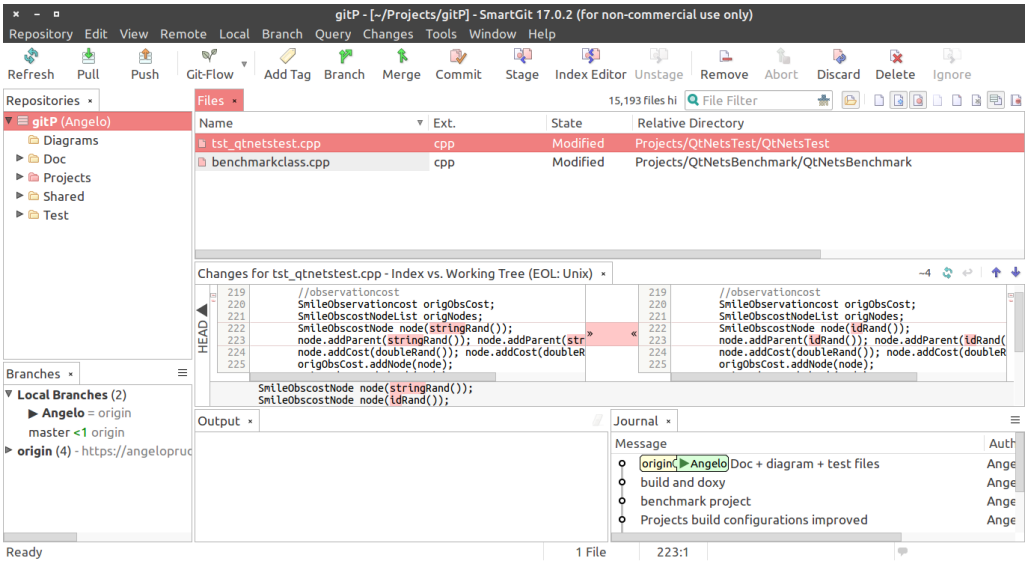


Figure 3.4: SmartGit client application

As shown in Figure 3.4, SmartGit allows to immediately see the list of modified files since the last stored version, with the possibility of checking the exact differences line by line.
The journal functionality allows to check the history of code revisions and for every stored version it is possible to see the difference with the previous one, and even isolate the evolution of one single file.
It also offers a customizable tool bar, right under the menu bar, in which there could be buttons and shortcuts for the most frequently used operations, thus making the interaction quite straight forward.

Finally, SmartGit fully supports GIT flow methodology, that here I just mention because it is too sophisticated for a single-developer project like this one, so I never used it.

3.4 Valgrind

Valgrind is a development tool for memory debugging, memory leak detection, and profiling.

It is basically a virtual machine using JIT (*Just In Time*) compilation techniques, including dynamic recompilation.
The original program never gets run directly on the host processor; instead, Valgrind first translates the program into a temporary and simpler form, called IR (*Intermediate Representation*), that is a processor-neutral form, and then, after the conversion, the chosen tool (one of those available along with Valgrind standard distribution or purposely made) is free to do whatever transformations it would like on the IR, before Valgrind translates it back into machine code and lets the host processor run it.
It must be noticed that a considerable amount of performance is lost in these transformations; It has been experimentally proved that a generic program executed through Valgrind, without any intermediate profiling tool, runs at 1/4 to 1/5 of the speed in normal situation.

There are multiple tools included in Valgrind distribution and many more are available as external packages, but the one that I used more often is **Memcheck**.
Memcheck inserts extra instrumentation code around almost all instructions, which keeps track of the validity and addressability of every allocated memory slot. The list of problems Memcheck can detect and warn about include the following:

- use of uninitialized memory;
- reading/writing memory after it has been de-allocated;
- reading/writing outside of the allocated memory block;
- memory leaks.

The price of this is a dramatic lost of performance: programs running under Memcheck usually run from 20 to 30 times slower than running outside Valgrind and use more memory.

Chapter 4

The QtNets framework

This chapter is focused on the architecture of the QtNets framework; in particular design choices and implementation details of all the modules composing the whole framework will be discussed.
In the introduction of this paper, I anticipated that the topics addressed by the framework are basically four and that there is a specialized module for each one of them.

The first and most important issue to solve concerns data representation, meaning all the centralized data structures needed to represent any imaginable network, with every kind of properties and attribute associated, in the most general and concise way possible; the solution to that form the module named **QtNetsData**, which is the very core of the entire framework.

Since the final goal of this project is to accomplish a user-friendly tool to display and edit networks, the second, yet not by importance, issue is related to the actual rendering of the elements of a graph and to the effective management of the events generated by the interaction of the user with the instrument itself; the module addressing those problems is named **QtNetsDraw**.

Another very crucial point, actually for every kind of software product, is related to the persistence of the information kept an processed by the user during the everyday usage of the product. The name of the module designed to face this very critical point is **QtNetsPersistence** and, just like the QtNetsDraw module, it is optional. The word optional associated to those modules has a slightly different meaning and the reason is simply explained: on a strictly technical base, QtNetsData is completely self-sufficient and can be used as it is, so none of the two modules is mandatory to the mere execution; but, while it makes complete sense to have an application using only the core module to represent a certain graph or network in

a common form and perhaps operate some background computation on it, which does not need screen rendering, it is very difficult to find an application field where it is not necessary to store the managed information onto some persistent storage device, is this a simple file rather than a complex database, or vice versa to load some information from a persistent source. For this reason, I think the QtNetsPersistence module could be considered anyway part of the core of the QtNets framework, given the importance of the functions it implements.

The last module developed for this project is related to the customization of all the elements that could be represented using the data structures provided by the QtNetsData module and rendered by the QtNetsDraw module; the name of this module is **QtNetsStyle**.
The QtNetsStyle module has been designed to offer the possibility to define the appearance of each and every element rendered in the screen separating the meaning of those elements from the definition of the look and feel associated to them; in this way the maximum re-usability is guaranteed.

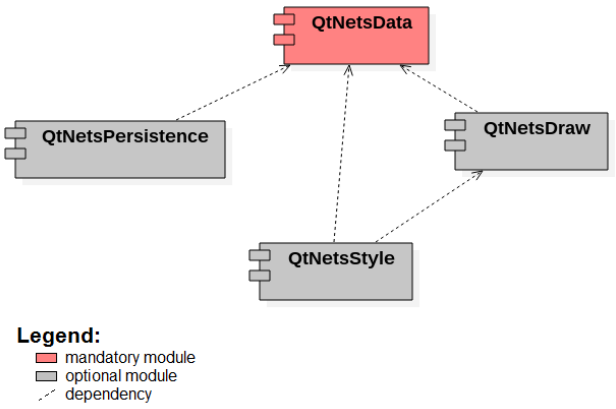


Figure 4.1: Dependencies between QtNets's modules

The diagram in Figure 4.1 clearly highlights the difference between QtNetsData and the others, as it is the only one without dependencies; it also shows how QtNetsPersistence and QtNetsDraw are at the same hierarchical level, both depending only from QtNetsData. Finally it shows that QtNetsStyle is also depending on QtNetsDraw, which might be not immediately deductible from the brief description provided above, given the fact it interacts directly with the rendering procedures defined in the QtNetsDraw module.
Of course this diagram is quite abstract and shows only the global relations between the different modules, but is enough to offer a very high-level description of the scenario.
In order to describe precisely how these four modules are related with each other, it

is first necessary to understand their structure, therefore I prefer to delay it to the moment in which all the needed background information is available.

In addition to that, the QtNets framework depends on the Qt development framework, which means, from a strictly practical point of view, that it is necessary to have the Qt environment installed to use any software module released as part of this project.
In any case this is definitely a price worth to be paid, considered all the advantages offered by this powerful platform.

4.1 QtNetsData

This module of the framework defines which are the fundamental entities that every graph must have to be meaningful and therefore processed; for each one of these entities it has been defined a class to model its attributes and links between classes to represent actual relations between the modeled entities.

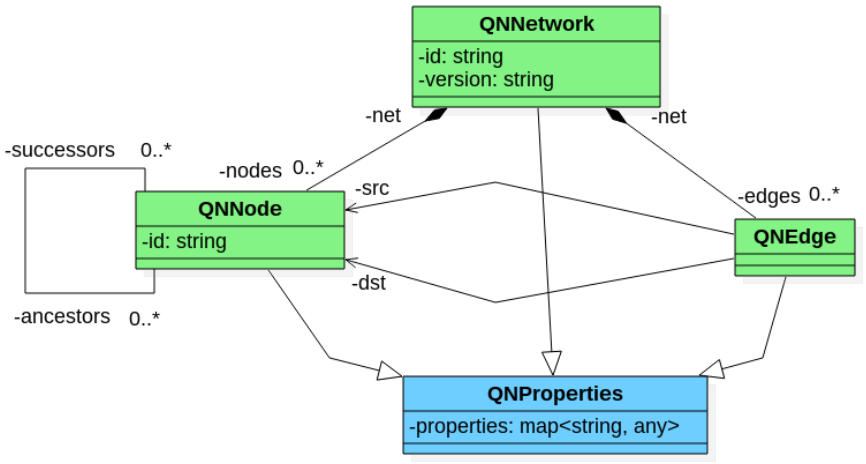


Figure 4.2: UML class diagram of the data-oriented section of the QtNetsData module

At the very beginning it is necessary to define what a network or graph is and this duty is up to the **QNNetwork** class; the UML class diagram in Figure 4.2 shows that the only two attributes needed to characterize a network are its **id**, which has to be unique in order to identify the network itself, and **version**. Both of them are strings, meaning that can contain any sequence of characters that makes it possible to satisfy their reason to be: uniquely identify the network, concerning the id attribute, and identify the evolution of the network during time, concerning the version.

The diagram does not strictly specify the type of strings that has been used, because it was meant to be independent of the programming language, so it is necessary to spend some time discussing which kind of strings have been used; the choice here was between the traditional C++ strings, the ones provided along with the Standard Template Library (STL), and the **QString** class provided by the Qt framework.
After some research and experiments, the decision was to use the QString class as they offer some very interesting advantages. First of all, it provides a *Unicode* (an international standard that supports most of the writing systems in use today)

characters string, thus making possible to deal with basically all the known characters, and offers predefined methods to convert such a string to other encodings, like for instance *US-ASCII* (*ANSI X3.4-1986*) or *Latin-1* (*ISO 8859-1*). It also offers very high performances thanks to the implicit sharing (*copy-on-write*) policy implemented to reduce memory usage and to avoid the needless copying of data.
In addition to this very important technical reasons, there is one more thing that makes QString preferable over C++ standard strings: QString offers a long series of predefined methods to support almost any kind of string computation that prevents the developer from losing time writing its own version of standard and well known algorithms, thus reducing the chance of introducing bugs. In this way the developer has to deal only with the implementation of the algorithms needed for his specific goal.

A network, to be useful, must contain nodes and edges to connect such nodes; in this framework they are represented respectively by the **QNNode** and **QNEdge** classes.

The QNNode class has only one mandatory attribute, which is the **id** of the node to be modeled; as for the QNNetwork class, the id must be unique inside the same network and could be any Unicode string.
Since it is not uncommon to have networks containing a large number of nodes, the QNNode class has been provided with a method to randomly generate strings to be used as identifiers. The random generator provided is actually simple but powerful at the same time, and it is also fully customizable; in particular the length of the produced random string and the set of characters to be used for its generation can be decided by the invoker, even if they have default values. In the case of the QNNode class, after some analysis, I considered acceptable to have identifier strings made up of twenty characters randomly chosen among all alpha-numeric signs, both lowercase and uppercase, because it guarantees a reasonably low probability of collisions.

The QNNode class does not represent only the characteristics of one node, but also its connections with other nodes, through an "*ancestor-successor*" relation; in other words, every node takes track of all the other nodes it is linked to, differentiating those for which it is source of the binding (successors) from those for which it represents the destination (ancestors). In this way, given every node, it is always possible to reach every single node it is linked to, is this link direct or indirect, simply navigating this ancestor-successor chain.
Now it is necessary to answer a question: why introduce the QNEdge class to represent links between nodes if those connections can be represented by the ancestor-successor relation just explained?

An edge between two nodes keeps a piece of information that is intrinsic in its nature: it informs of the fact that two nodes are in some way related to each other; this information is modeled perfectly by the QNEdge class as well as the ancestor-successor relation between QNNode objects. Nevertheless there are cases in which an edge needs to represent much more than the simple binding, as for instance its importance or "weight" inside the whole network, and in those situations only the QNEdge class could be used to model such extra information.
Secondly, the ancestor-successor relation implicitly assumes nodes, modeled as QNNode instances, to be predominant compared to edges, in this case not modeled by a class but only by the relation itself; this might be true or convenient in some scenarios, but there are other situations in which the edge itself is predominant, or at least of equal value, compared to the nodes it connects, so the QNEdge class is also meant to be an alternative, and sometimes more convenient, way of reaching the same information.

Figure 4.2 also highlights the relation between the network and the nodes and edges it contains; in particular the QNNetwork class is a container for QNNode and QNEdge objects, which means that the life cycle of those objects is subordinate to their containing network and that the network can directly reach every node or edge it owns.
For the sake of completeness, it is also possible from nodes and edges to reach the network that contains them, and from QNEdge objects to reach both the QNNode object source of the edge and the one that is the destination.

It is important to notice that the Figure 4.2 also shows another class that I did not mention yet but is equally remarkable; this is the **QNProperties** class, the common parent of all the others.
During the introduction of this chapter I stressed the attention in particular on one aspect of the QtNetsData module, that is it is meant to be general; the QNProperties class is the very instrument to reach that goal. It is possible to see it only has one attribute, **properties**, which is a generic map able to contain any possible kind of object identified by a string key; the type of the properties attribute is defined in Figure 4.2 using a general and intuitive notation, so to leave the diagram completely separated from the specific rules of the programming language used to develop the model, but it encloses a great deal of information that needs to be discussed in depth.
The following is the exact type definition of the properties attribute:

```
typedef QMap<QString, QVariant> QNVariantMap;
```

Three classes participates to the above definition, all of them part of the Qt framework and demanding a detailed explanation.
In order of appearance, there is the **QMap** class, which is one of Qt's generic container classes; it stores key-value pairs, where both the key and the value could be of any type, and provides fast lookup of the value associated to a key. Of course, as always happens with template classes, there are some conditions to be satisfied by other classes to be used as concrete instance of the template, and in this case the condition is that they must provide the implementation of a predefined comparison method, so to be uniquely identified and ordered.
Also this time it has been necessary to decide which class to use as map container and, just like the strings' case, the choice was between the map class offered by the STL and the one offered by the Qt framework, and the final decision in favor of the latter.
The reason behind this decision is almost the same as the one concerning strings: the two implementation are honestly very similar, but QMap offers a more intuitive and clean interface and a predefined version of a large number of standard algorithms, much higher than its opponent, to operate on maps and their content.

There is not much to add about the class used as key, QString, since I already explained the choice describing the QNNetwork class, but there is a relevant number of notions worth to be examined about the class used as value: **QVariant**. This is a sort of generic container able to store any value without knowing its class or type at compile-time but only at run-time, acting like a traditional C union for the most common predefined data types.
In order to achieve that goal, QVariant strongly depends on the Qt's meta-object system, which is the part of the Qt framework responsible of providing the signals and slots mechanism for inter-object communication, run-time type information, and the dynamic property system.
The meta-object system is based on the following things:

- the **QObject** class, which provides a base class for any object that can take advantage of the meta-object system;
- the **Q_OBJECT** macro, which has to be defined inside the private section of a class to enable meta-object features, such as dynamic properties, signals, and slots;
- the **Meta-Object Compiler** (moc), which supplies each QObject subclass with the code necessary to implement all meta-object features.

The moc tool acts as a sort of pre-compiler, as it reads any C++ source file and, if it finds one or more class declarations that contain the Q_OBJECT macro, produces

another C++ source file which contains all the code necessary to implement the meta-object features for each of those classes; the resulting source code is then automatically linked to the project.
Thanks to this system it is possible to use the QVariant class as a generic container, not only for predefined Qt's data type, for which meta-object features are already enabled, but also for user defined classes, after providing them with the necessary code to enable meta-object functionalities.
The following is an example of the definition of a custom class that enables the run-time type information functionality provided by the meta-object system:

```
class QNExample{
public:
    QNExample();
    QNExample(const QNExample& aOther);
    ~QNExample();
};
Q_DECLARE_METATYPE(QNExample)
```

The **Q_DECLARE_METATYPE** macro is needed to inform the run-time type information subsystem of the existence of the QNExample class and to enable the moc tool to generate the needed code; the only restriction for the class to be added to the meta-object system is that it must define a default constructor, a public copy constructor and a public destructor, as shown in the example.
To be noticed that the derivation from QObject and the Q_OBJECT macro are not mandatory if the class is meant to use only the run-time typing system.

This is the very essential code required to use a custom class associated to a QVariant container; but in this case only a subset of the functionalities provided by the generic container could be exploited, which means that the QVariant container can only store instances of the specified custom class and convert them into other classes, when possible. In order to enable more sophisticated features, like ordering for instance, it is required to instruct the meta-object system properly and provide the custom class with the implementation of the basic predefined comparison procedures.

This is the line of code needed to instruct the meta-object system about the fact that the QNExample class defines the comparison methods required, meaning the standard C++ **operator==** and **operator<**:

```
QMetaType::registerComparators<QNExample>();
```

It is clear now that the properties map contained in the QNProperties class can be used to store any kind of property, given that it is an instance of a class satisfying the conditions explained above.

For the sake of simplicity, the diagram proposed in Figure 4.2 shows only attributes and relations for each class, completely ignoring all methods exposed by them; it is necessary therefore to spend some time discussing them. All the classes mentioned by now (QNNetwork, QNNode, QNEdge and QNProperties) offer methods to read the current value of each of their attributes and to eventually update them; so for example the QNProperties class will offer a method to add a new property with its proper key and one to get a property given its key or to read all properties, as well as the QNNetwork one will offer methods to retrieve one or more of the contained nodes or edges, and so on.
I consider not very useful to enumerate all the methods that appear obvious by simply looking at the diagram in Figure 4.2, because that family of methods is only meant to manage the information stored in the class it belongs, which means they are just fetching and updating utilities.
For this reason, I will focus the attention only on those methods that are more interesting. There are situations in which it is necessary to search nodes or edges inside a network, filtering them based on the existence or value of some properties associated to them; to address those scenarios, the QNNetwork class exposes two search methods, one focused on nodes and one on edges, capable to manage also complicated filtering patterns.

The following are the prototypes of such methods:

```
QNNodeList getNodesByProperties(
       const QNPropertyList& aProperties = QNPropertyList(),
       const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNEdgeList getEdgesByProperties(
       const QNPropertyList& aProperties = QNPropertyList(),
       const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);
```

Those two methods accepts as parameters a list of properties, where a property is a *QString-QVariant* pair exactly like the one's accepted by the QNProperties class, and a filtering strategy, which is a constant that defines how properties in the list should be treated; the various combinations of those two parameters allow to define a quite large set of filtering rules.
First, all objects could be retrieved without any filtering just invoking one of those methods with no parameters; if it is necessary to use some properties to choose which node or edge to select, they can be checked for existence, if only the key is given, or by value, if that is also provided.
There is no doubt when the property to be matched is just one, but when a list of properties has to be considered as selection criterion, at least two options are available:

- all the required properties must correspond exactly to make the item eligible;
- at least one of the required properties must correspond in order to pick the current item.

The **QNFilterStrategy** enumeration is provided to distinguish between those two cases: the first scenario is labeled as **AND** strategy while the second as **OR**.

All the classes described so far constitute what can be defined the "*data-oriented*" part of the QtNetsData module, meaning that those are the classes developed to managed all the essential information associated to a certain graph; however those are not the only classes that are part the QtNetsData module.
Since, once again, the objective is to be able to render a graph with all its elements on the screen, it is necessary to design a set of classes to support such feature by managing all the data related to the appearance of each element of the network. This set of classes is organized as a sort of parallel structure focused on the graphical characteristics of the elements already modeled by the classes belonging to the

data-oriented section; it also provide some classes modeling entities without a data-oriented counterpart that are only necessary to improve rendering capabilities.
The reasons behind this architectural choice are basically two: first this approach guarantees an appropriate separation of concerns, as each section is focused only on one specific topic; secondly this approach allows to spare memory resources in those situations in which rendering is not required.

The classes that will be analyzed now can be considered as part of the "*graphic-oriented*" section of the QtNetsData module.
Also in this case I will start the analysis proposing a UML class diagram to provide a very high-level description of the topic and then go in depth explaining details.

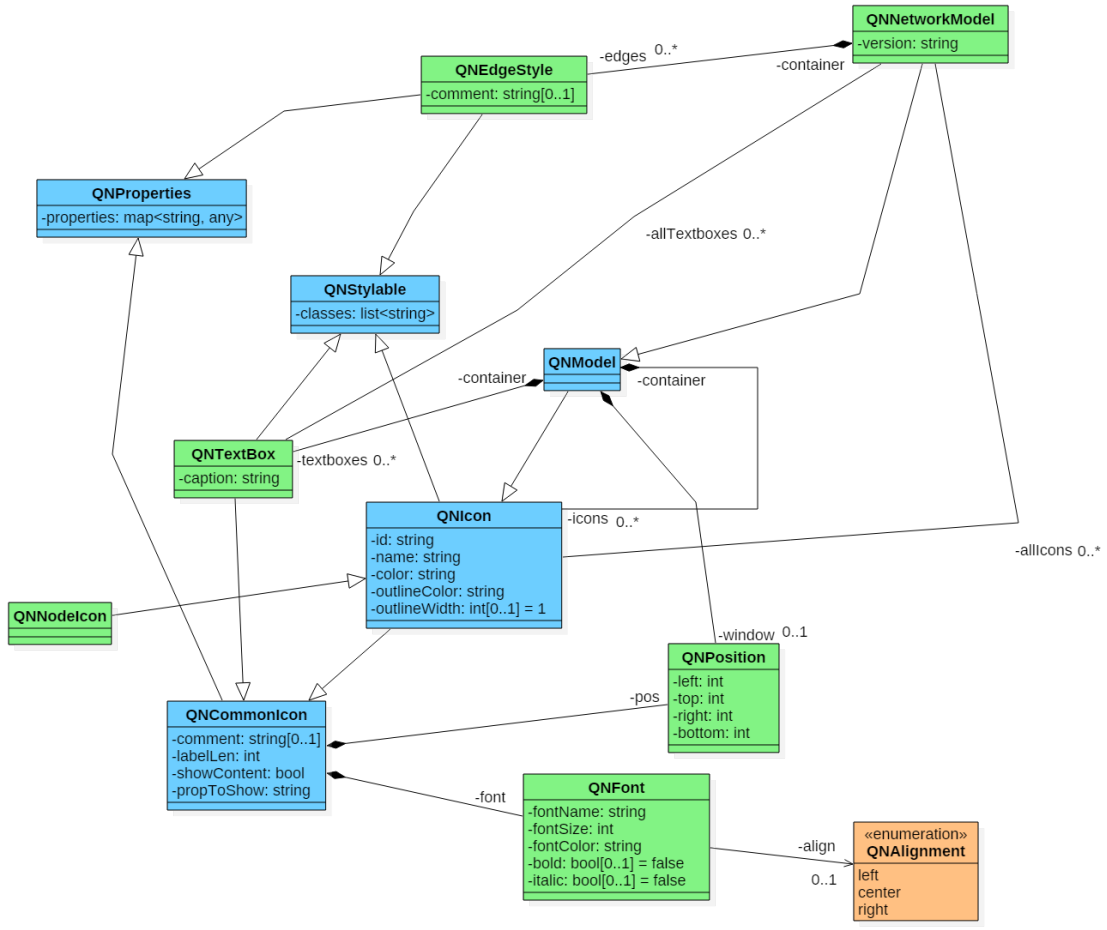


Figure 4.3: UML class diagram of the graphic-oriented section of the QtNetsData module

First of all I want to clarify that the diagram shown in Figure 4.3 is intentionally not showing the connections between the graphic-oriented classes and the data-oriented ones; those connections will be analyzed later.

Whichever object could be rendered on the screen is generally called icon, and it is modeled through the **QNCommonIcon** class. This class collects all the attributes that are required to any icon and represents the base for more precise classes that model further details, specific of the kind of icon they refer to. To be noticed that also the QNCommonIcon class extends QNProperties, so that it could be possible to associate different attributes to each icon depending on several different circumstances, just as it was possible with every data-oriented class.
Among the attributes of the QNCommonIcon class there is one that is fundamental to actually draw the icon on the screen: the **pos** attribute, which models the position of the icon in the scene. More specifically, it is a reference to an instance of the **QNPosition** class, which is the class designed to model a generic position in the scene to be displayed.
The QNPosition class collects four attributes, the exact number of coordinates needed to identify two specific points in a bi-dimensional Cartesian coordinate system: the top left and bottom right corners of the icon.

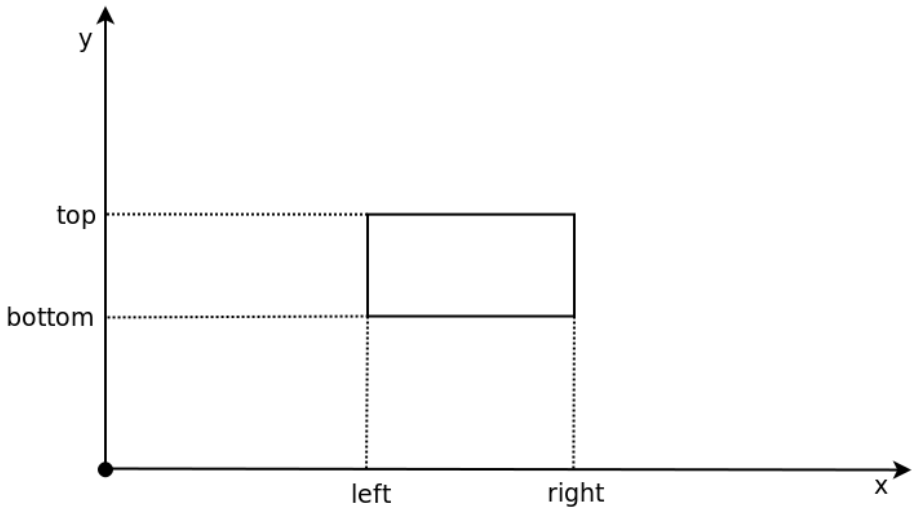


Figure 4.4: Icons' positioning system

The advantage of using this system to store the current position of an icon is that those two points does not only keep the information about where the icon should be placed in the scene, but also about the size of it, thus making the notation very compact.

Another important feature, common to all icons, is the possibility to print some information as icon's content, therefore it is necessary to model the visual properties

of the text to be printed; these properties are commonly referred to as **font**, so it will also be the name used for this attribute of the QNCommonIcon class. As in the icon's position case, the font attribute is just a reference to an instance of the **QNFont** class, purposely designed to model the graphical property of a text.
The attributes I decided to use to model a font are the most typical ones, those usually available in almost every text editing tools:

- **fontName**, the name of the font's family on which the global appearance of the text mostly depends;
- **fontSize**, the size of the font;
- **fontColor**, the color used to render the font;
- **bold**, a boolean value used to set or unset the bold property for the text;
- **italic**, a boolean value used to set or unset the italic property for the text;
- **align**, the alignment of the text, which can be left, right or center, in this case modeled by the **QNAlignment** custom enumeration.

Given an instance of the QNCommonIcon class, it is possible to decide which property should be displayed when the icon it models is actually rendered on the screen; the **propToShow** attribute is the one that stores the name of the textual property which value will be then rendered inside on the screen together with the icon.
Since the property to be shown could be a text of any size, it is possible that its length make difficult to render the icon in a reasonable amount of space; if, for instance, the user wants to show a complex mathematical expression, the space needed to display it entirely could be too much. For this reason it is possible to decide the maximum size of the displayed property through the **labelLen** attribute, which contains an integer value representing the exact number of characters to be displayed.
The showContent attribute of the QNCommonIcon class is a simple boolean variable used to store in a centralized way the visibility status of the icon; at the moment, this represents just a simple binary information about the icon showing or not its content, but the meaning of that concept is different for different kind of icons, as I will make clear in the next section.
Finally the QNCommonIcon allows to associate a generic description to each icon using the optional comment attribute, which can contain any sequence of characters being a Unicode string (QString).

The QNCommonIcon is extended by another general class which models more complicated elements that can be rendered on the screen; this class is named **QNIcon**.
The QNIcon class is actually very simple to understand, for it adds just few other attributes related to the appearance of the icon compared to its base class, but also crucial: as a matter of fact it adds the **id** attribute that makes icons uniquely identifiable. This is very important because the presence of an id attribute is the base to build a class able to model the icon associated to a complex element like a node.

Until this moment only internal classes have been analyzed, meaning only those classes that do not directly model elements that can be displayed and the final user can interact with; it is now time to tell exactly which entities can be seen on the screen and with which characteristics.
The QtNets framework is able to render and properly manage the following entities:

- nodes (modeled at this level by the **QNNodeIcon** class, which extends directly the QNIcon class);
- edges (edges between nodes are not properly considered icons, therefore the class modeling their graphical attributes, the **QNEdgeStyle** class, does not extend any icon class);
- text boxes (those are a simpler kind of icons and for this reason the **QNTextBox** class that models them just extends the QNCommonIcon class);
- models or sub-networks (modeled by the **QNModel** class extending the QNIcon class).

The QNNodeIcon, as the diagram in Figure 4.3 suggests, does not add any attribute to its base class, but it is not to be considered wasteful; as a matter of fact, its purpose is to centralize the management of the unique identifier associated to the corresponding node.
As explained while analyzing the QNNode class, any node is identified, in the scope of its container network, by a unique id, so even the icon associated to it must have the same id and propagate every id's change, so that the two related object could remain synchronized. The QNNodeIcon is just responsible for ensuring this synchronization bond.

In order to keep the structure as simple as possible, during the design phase of this module, I decided to model an edge between two nodes not as an icon, but using a simpler class; this class is QNEdgeStyle. The reason for this choice is that an icon, due to its proper nature, requires some mandatory attributes, for example

the position in the scene or the unique id, that are meaningless when referred to an edge. As a matter of fact, the position inside the scene and the length of an edge depends only on the two nodes it connects, so no more information is needed to render the edge but its source and destination nodes. It is easy to understand that also in the case of the id, even assuming that it might be necessary in some circumstances, the information required is deducible by the two nodes involved.
The UML class diagram in Figure 4.3, to be fair, does not show any direct connection between the QNEdgeStyle class and the QNNodeIcon one, because that connection is actually indirect and passes through the respective data-oriented classes.

The next two classes I am about to describe are different compared to the QNNodeIcon and QNEdgeStyle classes since they model entities that does not have an equivalent data-oriented class; in other words the entities they model does not have an independent information content but are just meant to enhance the graphical representation of the network.
The first of these classes is the QNTextBox class; it models a simple textual annotation that can be associated to the network to explain its meaning or to attach some extensive detail to one or more of its elements. This is a simple icon and for this reason the QNTextBox class simply extends QNCommonIcon.
The second, but actually more important, class of this kind is QNModel; it models the concept of a sub-network, which is a set of zero ore more other elements of the scene that are put together for having some common properties or to reach a better separation of concepts.Whichever is that these elements have in common, what matters is that the sub-network, or simply the model, containing them also controls their life-cycle.
This kind of hierarchical relation is expressed in a very elegant way by the UML class diagram shown in Figure 4.3: a model is basically an icon (QNModel extends QNIcon) an it can be the container of other icons that are actually owned by it.

Now that I have introduced all the required concepts, it is possible to analyze the top-level class of what I previously refer to as the graphic-oriented sub-module: the **QNNetworkModel** class.
The very name of this class already suggests two of its most important characteristics: first of all it is a model, as it extends the QNModel class, but it is the most important among models, for it represents the entire network and so controls all of its elements. In other words, the QNNetworkModel class represents what the QNNetwork represents for the previous group of classes: they are both a centralized management point for the specific information stored by their children.
Being an extension of QNModel, the QNNetworkModel class has already access to all the contained icons, as well as all the contained text boxes, but it has also access to all the QNEdgeStyle instances associated to this network.

It is perfectly reasonable to ask why edges are associated exclusively to the QNNetworkModel class and not to the single model that might contain them: as a matter of fact, it is possible for an edge to be completely contained (graphically speaking) into a model, so it would make sense if the QNEdgeStyle instance that represents it was also contained into the corresponding QNModel one. However this is not strictly necessary, so avoided for simplicity, because the only circumstance in which this scenario can take place is that both the source and the destination nodes are already contained into the model. This fact implies that, once again, the specific properties of the edge are determined by its nodes, therefore it is only necessary to manage them properly.

The diagram in Figure 4.3 also shows that the QNNetworkModel class has a further direct relation both with the QNIcon and QNTextBox classes with respect to the ones already guaranteed by its parent class; these are labeled as **allIcons** and **allTextboxes** respectively. They actually do not add any useful information to the whole structure, but exist only due to performances reason.
I did not stress the attention very much on the fact that also every class belonging to the graphic-oriented section of the QtNetsData module extend, directly or indirectly, the QNProperties class, because I already told everything it needed to be said about the topic; however, this implies that the QNNetworkModel class has to provide methods similar to those provided by the QNNetwork class to search and filter the contained elements based on the value of some of their properties. For this reason it is better to have direct access to all the elements to be filtered instead of fetching them every time navigating the whole model hierarchy, which in theory might be deep; besides, the cost in terms of time of this parallel structure is not so high, as it can be easily updated at the same time of the hierarchical one, while the return is remarkable.
For the sake of completeness, I will report the prototypes of the filtering methods provided by the QNNetworkModel class, without any further comment as their behavior can be easily inferred.

```
QNIconList getIconsByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNTextBoxList getTextboxesByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNEdgeStyleList getEdgesByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);
```

The last class to be analyzed is strictly related to the style customization system proposed by the QtNetsStyle module and actually represents the connection point between the two modules; this class is **QNStylable**.
As the name might suggest, this class models the prerequisite for an icon to take advantage of all the functionalities provided by the styling subsystem; in particular, what it is necessary to know at this level is only the list of the style classes, identified by their name, associated to a given icon, because the exact meaning of each class is managed by the QtNetsStyle module.
The QNStylable class only has one attribute, named **classes**, to keep the icon-class association just mentioned; it is, technically speaking, a list of QString objects. The UML class diagram in Figure 4.3 shows that all the classes that correspond to a displayable item, in this case the QNNodeIcon, QNModel, QNTextBox and QNEdgeStyle classes, extends QNStylable, so they are all potentially able to exploit the style subsystem.

It is important to underline the fact that the list of classes associated to potentially every element in the scene can also be used for purposes different from pure style; as a matter of fact, it could be used as a generic way to categorize elements and associate them some sort of labels, disregarding the fact that those correspond to particular graphical properties or not. To support this kind of usage, the QNNetworkModel class also offers the possibility to search and filter objects instantiating the QNStylable, or one of its derived class, in a way similar to the one proposed for properties:

```
QNStylableList getStylablesByClass(
        const QStringList& aClasses = QStringList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);
```

Looking at the definition of the proposed method, it is possible to notice only one detail concerning the first parameter, which is the list of classes to be filter by: in this case a simple list of QString objects is allowed because the only selection criterion applicable is the existence of the association between the class and the current object.

To conclude the analysis of the QtNetsData module, I want to propose one final diagram which gives a global picture of the module and also highlights the connections between the classes belonging to the so called data-oriented section with those belonging to the graphic-oriented one.

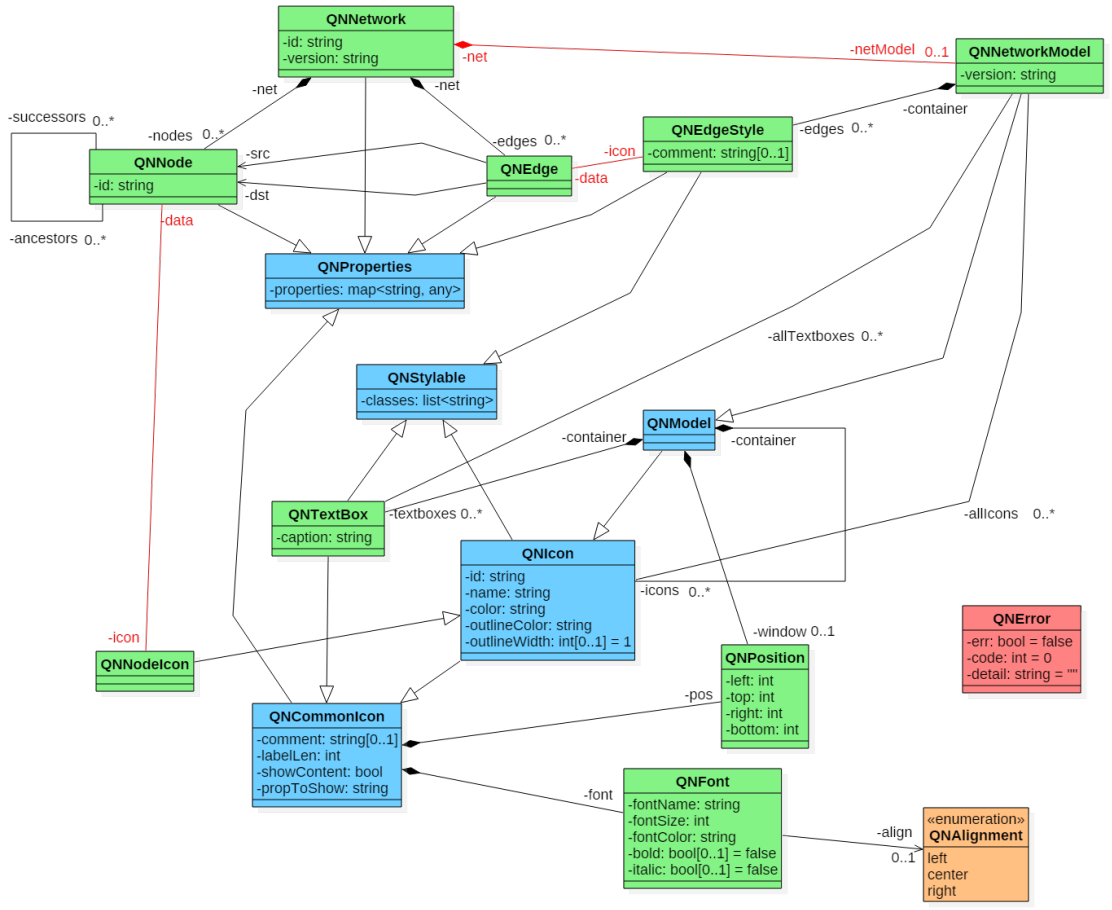


Figure 4.5: Complete UML class diagram of the QtNetsData module

The only thing needed to be highlighted about the UML class diagram in Figure 4.5 is the fact that the QNNetwork class is the owner of its corresponding QNNetworkModel class and indirectly of all the classes dedicated to model the graphical properties of the allowed elements; this once again confirm the optionality of the graphic-oriented classes.

In conclusion few words about the **QNError** class shown for the first time in Figure 3; this is a simple class designed to unify the error propagation interface exposed by all methods and functions belonging to the QtNets framework. This means that, each and every method that can incur into an error condition, must return an instance of the QNError class, properly populated to signal the specific error.
In particular, an error is characterized by a unique code, which identify the error's category, and a textual detail, which provide all the information needed to understand the problem. The only reason why this class is part of the QtNetsData module is that it is already required by all the other modules so they automatically have access also to the QNError class with no extra effort.

4.2 QtNetsDraw

The QtNetsDraw module is dedicated to the implementation of all the functionalities and algorithms necessary to display all the entities that might be part of a graph. Since all those entities have already been modeled by one or more classes belonging to the QtNetsData module, QtNetsDraw has to communicate with it in order to extract all the information needed to physically draw each item on the screen and to allow the final user to navigate all the properties and attributes associated to each item.

The graphic engine contained in the QtNetsDraw module is based on the Qt development framework, because it is simple to understand and use, as well as powerful; as a matter of fact, it already solves a large number of the most common 2D rendering issues and offers great support to the implementation of the basic features required to a graphic engine.
The Qt distribution provides a specific framework to deal with graphics, the **Graphics View Framework**. It provides a surface for managing and interacting with a large number of custom-made 2D graphical items, and a view widget for visualizing the items, with support for zooming and rotation.
The framework includes an event propagation architecture that allows double-precision interaction capabilities for the items on the scene; those can track mouse movement and can also handle key events, mouse press, move, release and double click events. Graphics View uses a Binary Space Partitioning (BSP) tree to provide very fast item discovery, and as a result of this, it can visualize large scenes in real-time, even with millions of items.

The most important classes of this framework, the ones that represent the very core and the base for every custom graphics view system are:

- QGraphicsView;
- QGraphicsScene;
- QGraphicsItem.

The **QGraphicsView** class provides the view widget, which is a scroll area provided with both horizontal and vertical scroll bars, necessary to visualize the content of a scene.
It is possible to attach several views to the same scene, to provide several viewports into the same data set, and it is also possible to take advantage of the OpenGL support calling a simple configuration method to enable it.

The view is able to manage input events of different kinds: when it receives input events from the keyboard or the mouse, it translates these to scene events before sending them to the visualized scene to be further processed.
The QGraphicsView class uses a transformation matrix, with which it can transform the scene's coordinate system to allow advanced navigation features such as zooming and rotation.

The **QGraphicsScene** class provides the Graphics View scene and has the following responsibilities:

- providing a fast interface to manage a large number of items;
- propagating events to each item;
- managing item state, such as selection and focus handling;
- providing native rendering functionalities.

The scene serves as a container for all the items belonging to the scene itself. Items are added to the scene calling a simple method and can be retrieved later invoking one of the many items discovery functions made available.
The QGraphicsScene's event propagation architecture schedules scene events for delivery to items, and manages propagation between them; if the scene receives a mouse press event from its containing view at a certain position, the scene passes the event onto whichever item is at that position.
It manages certain item states, such as item selection, focus and item's keyboard input focus and allows to render portions of the scene, or even all of it, into a paint device, which could be a printer interface object or simply an image file to export the content of the scene.

QGraphicsItem is the base class for all graphical items in the scene. The Graphics View Framework provides several classes extending QGraphicsItem to manage common shapes, like rectangles (QGraphicsRectItem), ellipses (QGraphicsEllipseItem) and text items (QgraphicsTextItem); those can be used as more specific base classes for custom items, in order to exploit the most powerful features made available by the framework.
Among other things, QGraphicsItem supports the following features:

- mouse press, move, release and double click events, as well as mouse hover events, wheel events, and context menus;
- keyboard input focus and key events;

- drag and drop;
- grouping policies, like parent-child relationships;
- collision detection.

Items live in a local coordinate system and the QGraphicsItem class provides many functions to map coordinates between the item and the scene, and from item to item. Besides, like the QGraphicsView class, it can transform its coordinate system using a transformation matrix, which is useful for rotating and scaling individual items.
Items can contain other items, which are called children; unless the item has no parent, its position is in parent coordinates, meaning the local coordinate system of the containing item.

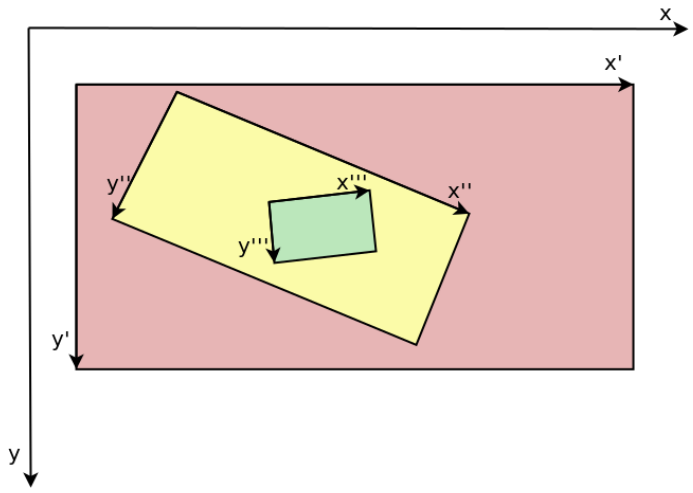


Figure 4.6: Items' positioning in parent's coordinates

Parent items' transformations are inherited by all its children, however, regardless of an item's accumulated transformation, all its functions still operate in local coordinates.

This is a quite fast description of how the Qt's Graphics View Framework is structured and works, but it is necessary to understand the architecture of the QtNetsDraw module, because it offers nothing more than an extension of the Graphics View Framework, properly customized to fit the specific goals of the project.

The following UML class diagram provides an overview of the structure of the QtNetsDraw module.

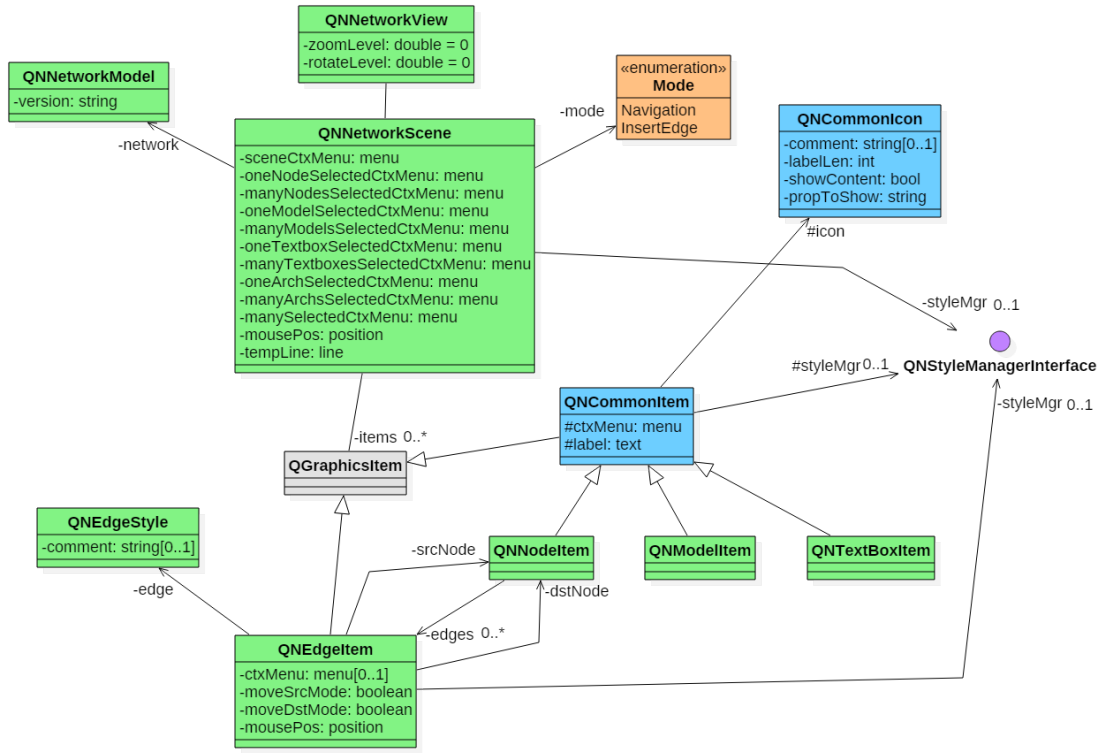


Figure 4.7: UML class diagram of the QtNetsDraw module

The very first thing that comes to the attention simply looking at Figure 4.7 is that the class hierarchy proposed by the Qt's Graphics View Framework is preserved; as a matter of fact the fundamental classes are:

- QNNetworkView extending QGraphicsView (omitted in Figure 2 for simplicity);
- QNNetworkScene extending QGraphicsScene (same as above);
- QNCommonItem and QNEdgeItem, both extending QGraphicsItem.

The **QNNetworkView** class is responsible of enriching its base class with a proper implementation of the zooming and rotation functionalities; as a matter of fact, the QGraphicsView class does not provide default methods to directly manage scene's zoom and rotation, but only the basic instrument to develop them, that is the already mentioned transformation matrix. Thanks to this matrix, it is possible

to apply several kinds of transformations on a scene, the entity of which can be determined using double-precision numbers, thus making the whole implementation quite intuitive.
It is therefore clear the purpose of the only two attributes associated to the QNNetworkView class, the **zoomLevel** and **rotateLevel**: they store the entity of the currently applied transformation.

The QNNetworkView class does not override any of the QGraphicsView's functionality related to scene management or input events propagation because they already meet all the requirements of the QtNets project; it only configures some properties of the underlying view structure to tune its behavior.
First of all the QNNetworkView configures the rendering properties of the view to optimize the rendering of items' edges and text:

```
this->setRenderHint(QPainter::Antialiasing, true);
this->setRenderHint(QPainter::TextAntialiasing, true);
```

The QGraphicsView class also offers the possibility to improve the rendering time of particularly dense scenes by caching some of its elements like textures, gradients and alpha blended backgrounds. After some experiments, I actually noticed an interesting rendering time reduction thanks to those caching policies, so I decided to exploit them to cache the background of the scene:

```
this->setCacheMode(QGraphicsView::CacheBackground);
```

More specifically, QGraphicsView can cache pre-rendered content in a pixmap, which is then drawn onto the viewport. The cache is invalidated every time the view is transformed, but only partially when the transformation is due to scrolling.

In addition to that, the QGraphicsView class allows to configure how to update areas of the scene that have been re-exposed or changed in order to further improve performances; among all the available configurations, after several experiments, I found that the following is the best choice considered the requirements of the project:

```
this->setViewportUpdateMode(QGraphicsView::SmartViewportUpdate);
```

In this working mode, the QGraphicsView class will attempt to find an optimal update mode by analyzing the areas that require a redraw.
Another aspect that is necessary to handle is how the view has to react when the user clicks on the scene's background and drags the cursor around: the available options are to scroll the viewport's content using a pointing hand cursor or to select multiple items with a rubber band. Given the nature of the tool to be designed, I considered the possibility to select multiple items at the same time, simply dragging the mouse pointer over them, more useful than simple scene scrolling, so I decided to use the rubber band mode:

```
this->setDragMode(QGraphicsView::RubberBandDrag);
```

Thanks to this configuration, clicking on the scene's background, a rubber band will appear and its geometry can be modified dragging the cursor; all the items covered by the rubber band are then selected.

The last configuration command is related to the way in which the view positions the scene during transformations; in particular it is possible to decide which point should be used as scene's anchor during transformations.

```
this->setTransformationAnchor(QGraphicsView::AnchorUnderMouse);
```

The above command sets the current mouse position as anchor of the scene during transformations, to ensure that the focus is always where the user wants it; however, the effect of this property is noticeable only when a portion of the scene is visible, otherwise the problem does not even exist.
Finally, it is important to notice that the QNNetworkView class, given the fact it extends a general purpose class (QGraphicsView), to which only adds general functionalities, is able to manage every kind of scene that at least extends the QGraphicsScene class.

The **QNNetworkScene** class provides a scene able to contain every possible item; it extends the QGraphicsScene class so it inherits most of the general features previously described.
The major task of the QNNetworkScene class is to be the container object of all the displayed items, in order to manage their life-cycle, dispatch them user events and coordinate rendering and updating procedures. This is what in principle QGraphicsScene already offers, so the custom class I designed extending it only provides its own implementation of some methods and procedures in order to satisfy the specific requirements of the QtNets framework.

The UML class diagram in Figure 4.7 shows that the QNNetworkScene class is able to contain generic items, instances of whatever class extends the standard QGraphicsItem one; this is true only in theory because, even if instances of every class extending QGraphicsItem can be inserted into the list that physically implements the **items** relation, actually only the instances of the items' classes provided by the QtNetsDraw module are allowed to be rendered onto the screen and therefore accepted as children of the QNNetworkScene class.
This is achieved introducing appropriate filtering strategies into those methods provided to add new items to the scene: whenever it is required to add a new item to the scene, this one is checked for actual type and properties and it is accepted only if satisfies the imposed restrictions.

It is important to highlight the relation between the QNNetworkScene class and the QNNetworkModel one; as a matter of fact, the QtNets graphical engine is designed to render and manage only items which correspond to entities already

stored into the data structure provided by the QtNetsData module. This is the fundamental condition to always guarantee the coherence of the data structure when the user interacts with it through the editor interface or directly through the scene. To show the importance of the matter even more clearly, I propose a brief example: It is reasonable to think that a typical use-case scenario might be the one in which the final user wants to edit a network because he needs to remove a node that is no longer required and has to add a new one with different properties and attributes; to do this he simply interacts with the QtNets graphical engine using the editor application provided with this project. When the user hits the delete button on the selected node, the QNNetworkScene class receives the node elimination event and, not only it has to remove the item from the scene so it is no longer visible, but it also has to communicate with the QNNetworkModel class in order to have the corresponding node removed from the persistent data structure managed by the QtNetsData module. It is straightforward that every edge associated to the selected node has to be removed too, because an edge without one of its terminal nodes does not make any sense.

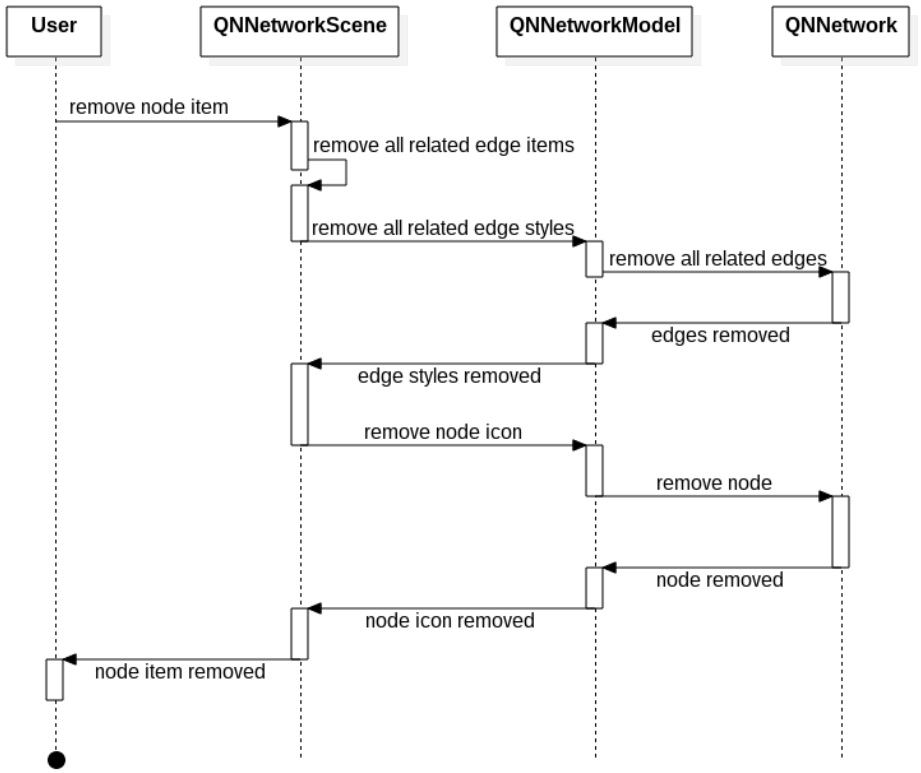


Figure 4.8: UML sequence diagram of the node removal process

The same of course applies when a new node is added to the scene or a property

of every of the active items in the scene is updated.
Since the displayed network is modified by the user interacting directly with the scene, the QNNetworkScene class is responsible to translate the inputs coming from the user into requests for the underlying data model in order to always keep coherent what is visible and what is stored.

With that being said, it seems clear that the QNNetworkScene class is also designed to be a central manager for the external events that the containing view receives and dispatches, in charge of deciding whether they have to be propagated directly to one or more of the contained items or they first need to be preprocessed. A particularly useful family of events is the one related to the contextual menu that typically shows up in almost every application when the user clicks the right button of the mouse. The importance of these events is in the nature of the contextual menu itself: it is highly customizable, therefore it can expose different commands in different situations depending on its configuration and can be used to address almost every use-case scenario.
In the specific case of the QNNetworkScene class, it is possible to provide several different menu pointers, each one with its own actions, in order to address different situations; those are differentiated based on the position where the contextual menu is invoked and on the number and type of the items currently selected.

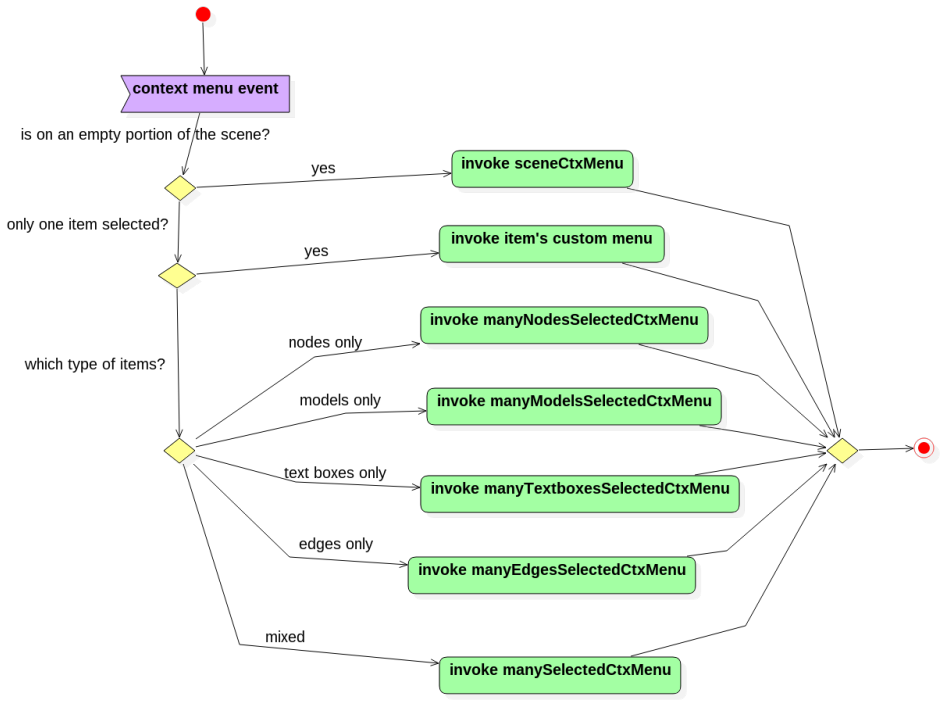


Figure 4.9: UML activity diagram of context menu's management

The UML activity diagram in Figure 4.9 summarizes all the possible scenarios originating from a mouse right-click event and how the QNNetworkScene elaborate it to decide which contextual menu to provide in response.
The first situation takes place when the user right-clicks on an empty portion of the scene: in this case the QNNetworkScene class invokes the menu object pointed by the **sceneMenu** attribute, which is meant to contain general-purpose actions.
When the user right-clicks on a single item, with no other items currently selected, the QNNetworkScene class invokes the contextual menu associated to that specific item; in this way it is possible to associate different actions to different items based on their characteristics. Moreover, it provides common menus to be used by the different items' families as a base upon which building their own custom ones; that is the task of the **oneNodeSelectedCtxMenu**, **oneModelSelectedCtxMenu**, **oneTextboxSelectedCtxMenu** and **oneEdgeSelectedCtxMenu** attributes.
The last possible scenario takes place when the user invokes the contextual menu while multiple items are selected; actually this is the most complicated one to manage because the menu to be shown depends on which kinds of items are selected.
The QNNetworkScene class allows to distinguish the following cases:

- if only nodes are selected, it is invoked the menu pointed by the **manyNodesSelectedCtxMenu** attribute, which is designed to contain actions that only nodes supports;
- if only models are selected, the menu pointed by the **manyModelsSelectedCtxMenu** attribute is invoked;
- if only text boxes are selected, the menu pointed by the **manyTextboxesSelectedCtxMenu** attribute is invoked;
- if only edges are selected, the menu pointed by the **manyEdgesSelectedCtxMenu** attribute is invoked;
- if many items of different kinds are selected, it is invoked the menu pointed by the manySelectedCtxMenu attribute, which should contain only actions common to every family of items.

The Qt's Graphics View Framework offers a very powerful set of classes that already manage all the common issues related to the appearance and event dispatching of the contextual menu, leaving the developer anything but its configuration.
One of those classes is **QMenu**, a class that provides a menu widget to be used in menu bars, context menus, and other pop-up menus. A menu widget is a selection menu and can be either a pull-down menu in a menu bar or a standalone context

menu usually invoked by some special keyboard key or by mouse's right-click.
A menu consists of a list of action items, which are instances of the **QAction** class; this class provides an abstract interface for actions that can be inserted into widgets. In applications, many common commands can be invoked via menu, tool-bar buttons and keyboard shortcuts so, since the user expects each command to be performed in the same way, regardless of the user interface, it is useful to represent each command as an action. A QAction's instance may contain an icon, a text, a keyboard shortcut, a status text, and even a tool-tip to provide some extra information; on the other hand, to specify the actual behavior of the action, it is common to define a receiver to be notified whenever the action in the menu is triggered.
To achieve that, the signals and slots mechanism provided by the Qt framework is used.

In GUI programming, when a widget changes, it might be necessary that other widgets get notified; more generally, it is useful that objects of any kind could be able to communicate with one another. The signals and slots mechanism is the recommended instrument to support such a communication in Qt-based applications.
A **signal** is emitted when a particular event occurs. Qt's widgets have many predefined signals, but it is always possible to subclass them to add custom signals. A **slot** is a function that is called in response to a particular signal. Qt's widgets have many predefined slots, but it is common practice to subclass them to add custom slots too.
The signals and slots mechanism is type safe and loosely coupled: as a matter of fact, the signature of a signal must match the signature of the receiving slot but, when a class emits a signal, neither knows nor cares which slots receive it. That allows to enforce true information encapsulation, and ensure that the object can be used as a pure software component.
Signals are usually emitted by an object when its internal state has changed in some way that might be interesting to other objects.
Slots can be used to receive signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.
It is possible to connect as many signals as needed to a single slot, and a signal can be connected to many slots as well; it is even possible to connect a signal directly to another signal.
When a signal is emitted, the slots connected to it are usually executed immediately, just like a normal function call; when this happens, the signals and slots mechanism is totally independent of any GUI event loop. The execution of the code following the emit statement will occur once all slots have returned and, if several slots are

connected to one signal, those will be executed one after the other, in the order they have been connected.
As already said, slots are normal C++ functions with the special characteristic they can be connected to signals. This means that, being normal member functions, they follow the standard C++ rules when called directly, but, as slots, they can be invoked by any component, regardless of its access level, via a signal-slot connection.

This generic and very dynamic mechanism offered by the Qt framework is used in QtNets, not only to connect menu actions to slot members implementing the required commands, but also to support many other features that will be described later; this brief overview is therefore necessary to understand their behavior.
Exploiting signals and slots, the QNNetworkScene class is able to offer some built-in utilities to perform basic operations on the network displayed in the scene, in order to avoid other developers to design and implement several times methods that are common to every possible application field. It is also possible to use those utility methods as a reference in case it is necessary to re-implement them to fit the characteristics of a very sophisticated scenario.
More specifically, the QNNetworkScene class offers a series of slot methods covering all the basic interaction with the scene, that can be used as generic actions in one of the contextual menus designed for the editor application.
First of all, there are some methods designed to create and add new items to the scene:

```
void newNode();
void newModel();
void newTextbox();
void newEdgeMode();
```

Each one of those methods creates a new item with default properties and takes care of the most delicate issue related to item insertion into the scene: positioning. As a matter of fact, if they are associated to a contextual menu, they are able to recognize the position in the scene where the mouse click took place, so they can put the new item exactly where the user expects it; if they are invoked directly, for instance at the final step of a user-defined algorithm, they simply put the new item in the center of the scene.
It is worth to spend some time analyzing in detail the last method of the list, because it is slightly different from the others, as the signature might suggest. The UML class diagram in Figure 4.7 shows the existence of a very important enumeration related to the QNNetworkScene class, named **Mode**, which determines its current

working mode. At the moment, the only two available values are **Navigation** and **InsertEdge**.
The Navigation mode is the default choice and allows every possible interaction with the network, from inserting new nodes to deleting one or more of them, from selecting items to invoking a context menu; every action or procedure described so far is intended to be executed in this working mode.
On the other hand, the InsertEdge mode allows only one action: connecting two nodes through an edge. The newEdgeMode method is only meant to switch the operational mode of the QNNetworkScene class to the second value.
When the InsertEdge mode is active, the response to some input events is different from the standard behavior because it has been designed to let the user draw the required edge directly in the scene. First of all, when the user left-clicks in a certain position of the scene, that is stored as the source point of the candidate edge; the **mousePos** attribute of the QNNetworkScene class acts as temporary memory slot for the last clicked position. After that, but only if the mouse's left button is still pressed, a line is drawn from the previously selected point to the current cursor position, as the user freely moves it around the scene; the line is updated in real-time for as long as the user keeps moving the cursor, using the **tempLine** attribute of the QNNetworkScene class to keep track of the temporary line's state.
In the moment in which the user releases the button, the last cursor position is stored and the line is fixed: if the source and the destination points of the line just drawn intersects two distinct and valid nodes, than the line is converted into a real edge, with all its properties set to default values, otherwise it is simply removed from the scene; In both cases the working mode is automatically restored to Navigation. Any other interaction with the scene different from the one just described during the InsertEdge mode causes the immediate return to the default working mode, where the input can be processed as usual.

The second category of methods I want to analyze contains pure utilities designed to facilitate the usage of the framework; those are not necessarily similar to each other concerning the task they perform, so I prefer to briefly describe them one at a time.

```
void deleteItems();
```

As the signature clearly suggests, this method's task is to trigger the elimination of items; there is no more to add but the fact that it is a general method that can be invoked to delete every kind or number of items, because, using an algorithm very similar to the one already explained speaking of the management of contextual

menu events, it is able to recognize and invoke the delete routine specific for each item.

```
void bringToFront();
void sendToBack();
```

Those two utilities are meant to allow the user to modify the items' stack order, so to bring back and front specific items when the scene is quite crowded.

```
void newCycle();
```

This method is proposed as a shortcut for all those situations in which is necessary to build a complete coverage between a certain number of nodes; for this reason, when the newCycle method is invoked, it automatically inserts all the edges necessary to connect each of the currently selected nodes to all the others in both directions. Of course, this is a routine designed to work only with nodes, so it discards every other kind of items that might be selected too.
The last category of methods is related to item's content visibility; since the meaning of content visibility depends on the type of item it is referred to, here I just mention the concept with the purpose of giving more details during the analysis of the different items.

```
void toggleShowContent();
void hideAllContents();
void showAllContents();
```

Those are simply shortcuts provided by the QNNetworkScene class to invoke the corresponding methods of all the selected items at one time, automatically forwarding the request to the right method based on the actual type of each item.

Typically, in an editor application of every kind, it is necessary for the scene to have an asynchronous communication channel opened with the other components of the same application, in order to promptly inform them about any change of the contained items or of the scene itself. The QtNets framework, which is meant to be used as a building block for networks editor applications, must address this requirement.
For this reason the QNNetworkScene class implements an asynchronous communication system based on the already known Qt's signals and slots mechanism.
The objective is to obtain a system able to update in real-time every other component or module that is interested in being informed about the current status of the scene and its content. After several analysis and experiments, I found that two signals, if properly managed, are enough to reach the goal.
The following are the prototypes of those two signals:

```
void modified();
void itemsSelected(QNNodeItemList aNodeList,
                   QNModelItemList aModelList,
                   QNTextBoxItemList aTextboxList,
                   QNEdgeItemList aEdgeList);
```

The first signal is general and therefore used to notify a generic event, while the second is detailed and specify exactly which items are involved.
To be more specific, the modified signal is emitted every time the scene, or any of its items, changes, no matter the nature of the change; for instance, this signal is emitted when an item is moved inside the scene, inserted or even deleted, when a certain property of one or more items changes and so on. In other words, every event that causes any evolution of the data structure representing the network is signaled through the modified signal.
The itemsSelected signal, on the other hand, is used to notify events specifically

related to one or more items; the items involved are provided as arguments of the signal itself, in order to allow the listeners to execute the necessary actions on them. I do not intentionally provide here any usage example of this signal, because there will be plenty of them in the next chapter, where the editor application I developed for this project will be presented, but I want to add a clarification about this signal and the fact that it is enough to cover all possible events: this is true because every action that can cause a change in the network requires that the item on which it has to be executed is selected; if no items are selected, no changes can happen and so no signals are emitted.
Of course the itemsSelected signal is emitted also every time one or more items are simply selected.

As already said in many occasions, the purpose of this project is to realize a framework able to render and display a generic graph structure, so the QNNetworkScene class has been intentionally developed to support only the items provided by QtNetsDraw module, as they are the only ones modeling the required entities.
I listed all the supported elements, or icons, when I described the graphic-oriented section of the QtNetsData module, but I briefly recall them here because they are strictly related to the items offered by the QtNetsDraw module. First of all there is the QNNodeIcon class that models the graphical properties of a node; the class that actually implements all the procedures and algorithms to physically render it on the screen is the **QNNodeItem** class. A similar relation exists between the QNModel and the **QNModelItem** classes, concerning models, and between QNTextBox and **QNTextBoxItem**, concerning text boxes. Finally, the graphical properties associated to an edge, modeled through the QNEdgeStyle class, are physically managed and rendered by the **QNEdgeItem** class.

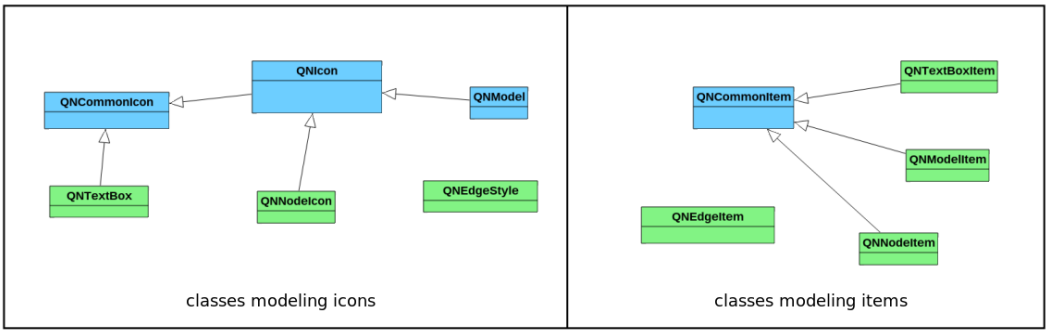


Figure 4.10: Icons compared to items

Figure 4.10 proposes a simplified version of the QtNetsData class model compared to an equally simplified version of the QtNetsDraw class model, in order to

analyze the similarities between the classes modeling the supported icons and the ones that model the corresponding items.
The first thing that can be noticed is that both QNEdgeStyle and QNEdgeItem have no common parents with other classes, because they both model a concept that is completely different from the others. Concerning the remaining classes, speaking of icons as well as items, it can be observed that the concepts of node, model and text box are always related to each other and have many things in common; for this reason, in both cases it is possible to extract all the common aspects in a parent class to generalize the structure. The only difference is that, in the case of icons, the abstract base classes required are two, QNCommonIcon and QNIcon, to properly model the slight differences between nodes and models, on one side, and text boxes on the other; in the case of items this distinction is not necessary, because already implied by the underlying icons.

With that being said, it is possible to start a detailed analysis of the classes implementing the available items, starting with QNCommonItem. This class is designed to manage in a centralized way all the issues related to rendering and events processing of the complex items supported by this platform.
The UML class diagram in Figure 4.7 shows that the QNCommonItem class extends the standard QGraphicsItem class, but this is a simplification to make the diagram more readable; as a matter of fact, the QNCommonItem class extends the **QGraphicsTextItem** class, which is a specialization of the base QGraphicsItem. The reason of this choice lays in the fact that QGraphicsTextItem offers great support to the development of items showing text inside their boundaries, thus making reasonably easy to realize an item able to show dynamic textual properties directly in its body.
The specific property that has to be displayed is derived directly from the instance of the QNCommonIcon class to which it is connected and the **label** attribute of the QNCommonItem class represents its container. The icon associated to each item also provides the exact number of characters that have to be printed inside the bounding rectangle of the item: if the text is longer than specified, the rest of it is truncated and replaced by an ellipsis, while, if the number of characters to show is negative, then the entire content of the property is shown, regardless of its length.

The procedure to determine the textual content of a generic item could be used as a reference to explain a concept that is true for every item modeled in the QtNetsDraw module: no core information is stored in any of those classes; all the information required to populate the items rendered in the scene is taken directly from the corresponding icon classes.
This optimizes memory consumption as it avoids wastes due to data duplications.

As explained during the analysis of the QNNetworkScene class, the QNCommonItem class is able to manage its own contextual menu, making possible to define a different set of actions for every item, if this could be useful for the current use-case; the instance of this contextual menu is stored in the **ctxMenu** property.

The QNCommonItem class allows to hide every detail of the corresponding item, if necessary; the current content visibility status of the item is stored in the showContent attribute of the corresponding icon instance.
Unfortunately, the content visibility of an item is a complex concept, which strongly depends on the nature of the item itself; for this reason, QNCommonItem only offers an abstract interface to manage this feature, leaving to its descendant classes the burden of the implementation of all the required algorithms. This interface is actually very simple as it only requires a method that, given a single boolean attribute representing the new content visibility status, updates the entire item showing or hiding its content.
The following is the signature of such method:

```
virtual void setShowContent(const bool aValue) = 0;
```

This concept will return later during the analysis of the specialized items, when I will explain its meaning and the algorithm designed to enforce it for each family of items.

The management of all the external events dispatched by the containing scene is also a concern of the QNCommonItem class; in particular it manages all the mouse-related events like mouse click, double-click, move, release, hover and the ones related to capture and loss of focus. Through the proper combination of all this basic events, the QNCommonItem class is able to manage complex user interactions like drag and drop, resize and in-line editing of the text displayed by the item.
The drag and drop functionality is managed exploiting the following three events:

- mouse click event;
- mouse move event;
- mouse release event.

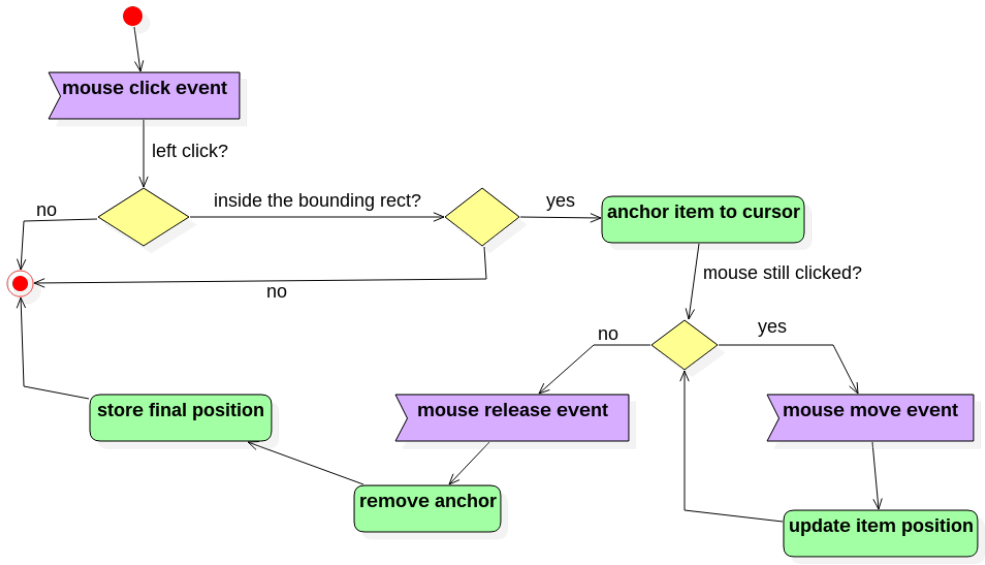


Figure 4.11: UML activity diagram of item's Drag&Drop functionality

When a mouse left-click is recognized inside the bounding rectangle associated to the item, this is immediately anchored to the mouse cursor; to signal this situation the traditional cursor icon is changed with a closed hand. For as long as the user keeps the mouse clicked, a series of mouse move events are delivered in real-time to the item and those can be exploited to continuously refresh the item and make it move around the scene with the cursor. When the user finally releases the click, the cursor icon is restored to the standard arrow icon and the position of the item in the scene is blocked and propagated to the corresponding icon to be stored.

The item's resize functionality is more complicated and requires the combination of more events to be managed; the required events are the following:

- mouse hover enter event;
- mouse hover move event;
- mouse hover leave event;
- mouse click event;
- mouse move event;
- mouse release event.

Mouse hover events are only used to update the cursor icon when it moves over one of the active corners of the item; when the cursor enters into the active area around

one of the four corners of the bounding rectangle, the icon is changed to the resize one, so to make immediately clear what kind of interaction could be started just clicking into those areas.

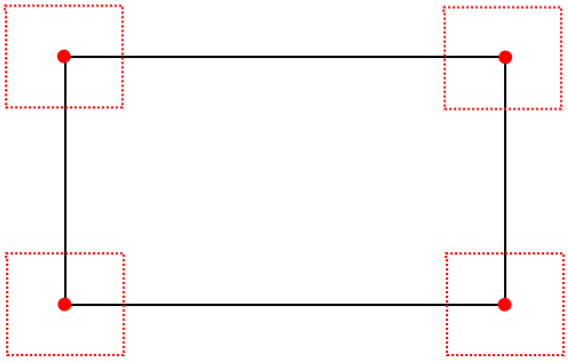


Figure 4.12: Item's active corners

The actual resize functionality is implemented exploiting mouse click, move and release events.

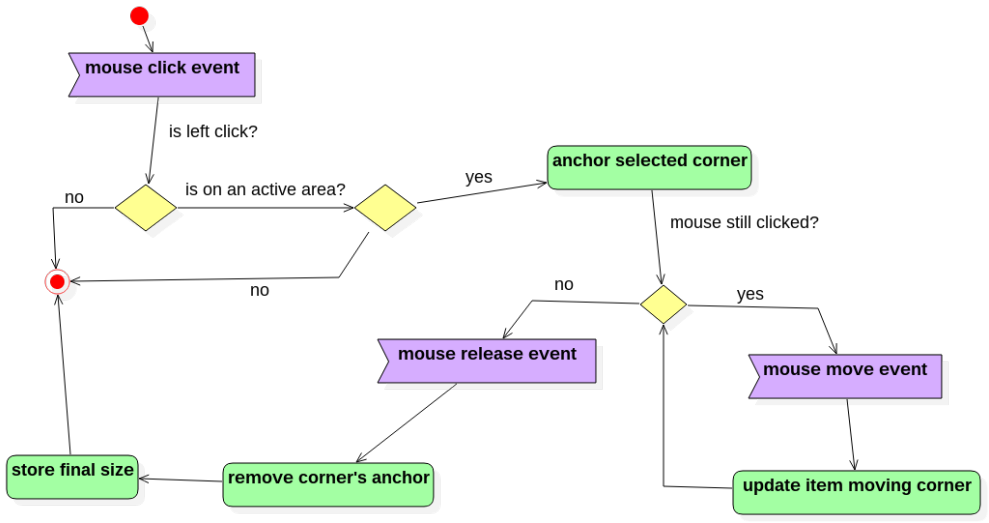


Figure 4.13: UML activity diagram of item's Resize functionality

Once a mouse click event is received and it is on the active area of one of the item's corners, the resize actually starts and the item is once again anchored to the cursor; however this time only the selected corner is anchored. The cursor icon is updated to a proper resize icon to signal the anchor. When the user moves the mouse without releasing the left-click, the selected corner will follow the movement

causing the item to be stretched or shrunken based on its direction.
As in the case of the drag and drop functionality, the continuous series of mouse move events received is used to update in real-time the size of the item and give the user the feeling that something is actually happening.
The item's size is fixed and propagated to the corresponding icon to be stored only when the mouse click is finally released.
In this case, as in the drag and drop one, an intermediate step is required to convert the position and the size of the item according to the coordinates system used by the QNCommonIcon class to store both the new position and size of the icon.

The last supported user interaction, the item's text editing, is actually the simplest to deal with, given the fact that it is almost entirely provided by the QGraphicsTextItem base class; as a matter of fact, this class offers the possibility to write a text directly in the body of the item, so the only thing that is up to the developer is the activation and deactivation of the editing mode.
In the case of the QNCommonItem, I decided to use the mouse double-click event to activate the text editing mode and the focus out event, which is dispatched every time the user "leaves" the current item by clicking somewhere else, to deactivate it; this seemed to be the best choice to offer the most intuitive interaction possible.

QNCommonItem is a base class itself, that provides the basic features common to all the QtNets items; the actual items, the ones that are rendered in the scene, are modeled by the **QNNodeItem**, **QNModelItem** and **QNTextBoxItem** classes. They represent respectively nodes, models and text boxes in the scene and are responsible to implement all the functionalities that are exclusive of the modeled item's family.
The two most important aspects handled by those specialized classes are the rendering of the item and the content visibility management.
The QGraphicsItem base class exposes a pure virtual method, named **paint**, that is called every time the portion of the scene containing the item is invalidated and requires a repainting; every item class must provide its own implementation of this method in order to define its appearance in the scene.
The following is the prototype of the paint method:

```
virtual void paint(QPainter *painter,
                   const QStyleOptionGraphicsItem *option,
                   QWidget *widget = Q_NULLPTR) = 0;
```

This procedure, usually invoked by a QGraphicsView or derived instance, paints the content of an item in local coordinates.
Custom classes extending QGraphicsItem must implement it to provide the item's painting algorithm, that has to use the provided **painter** to perform all the primitive operations required to draw the item on the screen. The painter argument is an instance of the **QPainter** class, the standard Qt class to perform low-level painting operations on widgets and other paint devices. It provides highly optimized functions to do most of the drawing GUI programs require: it can draw everything from simple lines to complex shapes, like pies and chords, to even aligned text and pixmaps.
The **option** argument, instance of the **QStyleOptionGraphicsItem** class, provides style options for the item to be painted, such as its state or its exposed area. The last argument, **widget**, is optional; when provided, it points to the widget that is being painted on, otherwise is null. In case of cached painting, which is the of the QtNets framework, this is always null.

Exploiting the objects provided as arguments of the paint method, each item class is able to render all the graphical properties stored in the corresponding icon, like the background color, the outline thickness or color and all the properties related to the font used to draw the text inside its body.
It is necessary to highlight an important detail about the icon associated to each item; the UML class diagram in Figure 4.7 shows that every instance of the QNCommonItem class is associated to an instance of the QNCommonIcon one. This is true only on an abstract level because, thanks to the polymorphic nature of the C++ programming language, it is possible that every instance of a specific item is associated to an instance of its corresponding specific icon class.
For this reason, at run-time, every instance of the QNNodeItem class is associated to an instance of the QNNodeIcon class, every QNModelItem instance to a QNModel one and every QNTextBoxItem object is linked to a QNTextBox object.

The content visibility management algorithm is implemented by all the three item classes respecting a common and fundamental principle: an item is editable only if it is in the show content state; if the item is hiding its content, than no action is allowed on it, not even resizing or movement around the scene.
In the specific case of the QNNodeItem class, when the hide content mode is selected, it is not possible to add new edges but only remove one or more of those already defined.
Even more complicated is the management of the content visibility of models; as a matter of fact, they are complex items that can contain other items, so, when the containing model is in hide mode, every contained item must be completely hidden and every edge between any of the contained node and any other outside the model

must be originated from the center of the hidden model. In this way it is easy to use model's hierarchy as a way to separate items into different levels of detail that can be shown or hidden depending on the situation.
The following figures represent a comparison between how each of the items discussed so far appears when is in show mode and how it looks when is hiding its content:

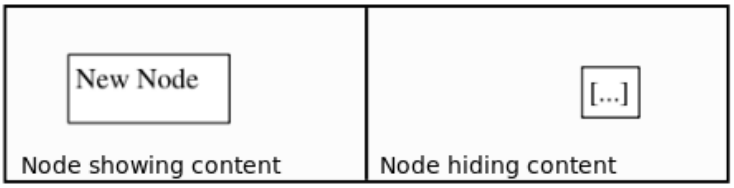


Figure 4.14: Output of the node's default rendering process

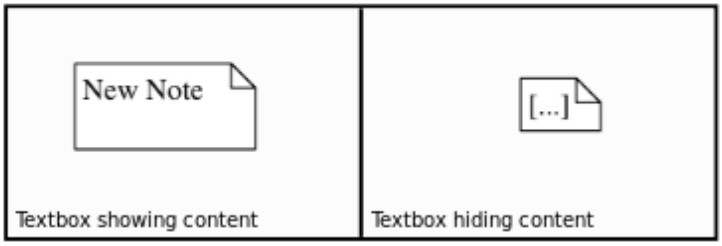


Figure 4.15: Output of the text box's default rendering process

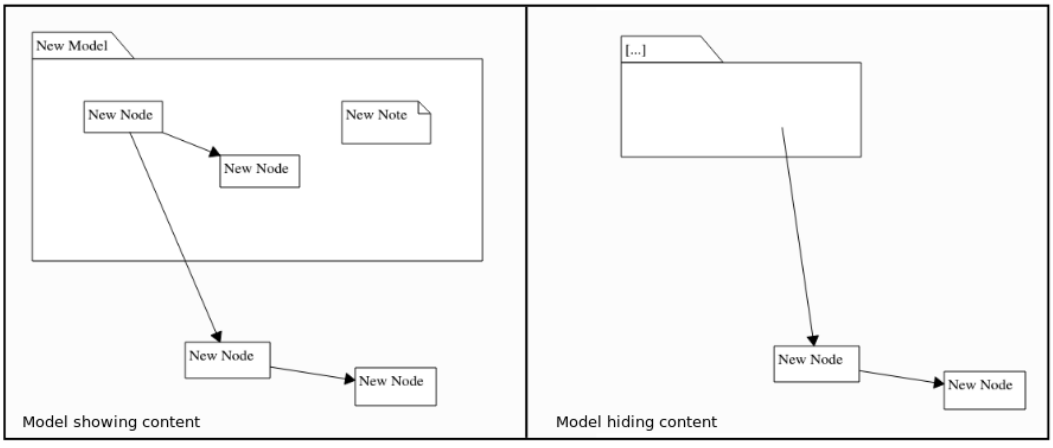


Figure 4.16: Output of the model's default rendering process

A completely different entity is the the link that can be established between two nodes to connect them; that is commonly named edge and is modeled in the QtNetsDraw module by the **QNEdgeItem** class. This class does not extend QNCommonItem like the other ones discussed so far, because there are no common points between the items they model; for this reason, it has to implement all the custom events' management algorithms needed to manage the required features.
The QNEdgeItem class extends the standard **QGraphicsLineItem** class, differently from what is shown in the UML class diagram in Figure 4.7, where only the common ancestor is pointed out; that in turn extends QGraphicsItem, in order to provide a specialized class to manage lines.

The item class provided to model edges in the QtNetsDraw module implements only the algorithms needed to support rendering procedures and external events' management, leaving the control of the information needed to populate the item to its corresponding graphic-oriented class defined in the QtNetsData module; this is the nature of the **edge** relation between the QNEdgeItem and QNEdgeStyle classes.
The QNEdgeItem class is linked to the QNNodeItem class in both directions through 3 different relations:

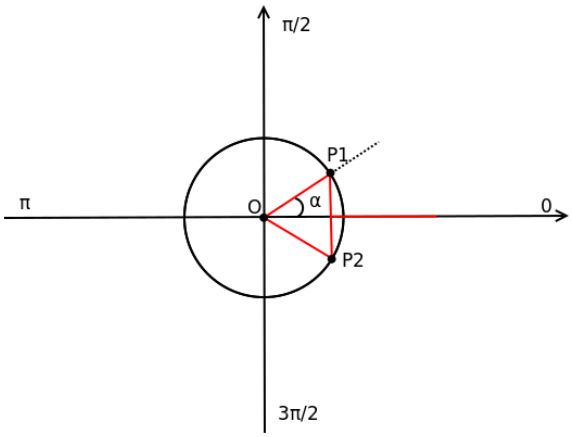
- **srcNode**, which connects the edge to the item representing its source node;
- **dstNode**, which connects the edge to the item representing its destination node;
- **edges**, which connects a node to every edge that involves it, either as source or destination.

To be noticed that none of those relations is strictly necessary because none of them really adds new information; as a matter of fact, the multiple relations between nodes and edges are already modeled in the QtNetsData module, therefore the same amount of information could be retrieved simply navigating the connection to the associated icon classes.
The reason why these new direct relations have been added anyways is exclusively related to a possible performance's leak during scene's updating procedures. More specifically, it is necessary to guarantee that the updating process of each item in the scene is as fast as possible, to avoid the application to get stuck, and the user annoyed, when the scene is very crowded; those direct relations are useful to speed up the information collection phase of every re-painting process.

Being an extension of the QGraphicsItem class, QNEdgeItem has to provide its own custom implementation of the paint method too. The concept behind the

implementation of the paint method is the same of the other items, so I am going to focus on the aspects specifically related to edges managed by the designed algorithm.
The edge is ideally originated from the center of the source node and terminated in the center of its destination; however, simply drawing a line from those two points would be ugly and a little confusing for the user when a great number of nodes and edges are simultaneously rendered in the scene. In addition to that, it could be possible to have undesired overlapping of edges and descriptive text inside the body of the node.
For these reasons, the line representing the edge must start from a point on the source's outline and terminate on the outline of its destination node. The union of those two concepts bring to the algorithm actually used: the two edge's ends are determined by the intersection of the fictitious "center-to-center" line with the outline of the bounding rectangles of the two nodes involved.
Using this approach also an interesting visual effect could be obtained: since the two ends of the edge are dynamically determined at every position change of the two nodes involved, it seems like the edge "follows" that movement.
Nevertheless, this is not the complicated part of the edge's painting algorithm; as a matter of fact, in order to make the direction of the edge clear, it is necessary to draw a specific symbol in correspondence of the destination node. The symbol that I found to be the most appropriate in this situation is the arrow head, because it is the one commonly used to point at things.
Since the base line which constitute the edge adapts itself to the relative position of the source and destination nodes, so has to do the termination sign. Theoretically it should be simple to draw, as it is nothing more than triangle, but the three points to be concatenated in order to draw it must be dynamically determined in a way such that the arrow head is always pointing to the destination node, no matter where it is in scene.

One of the three points needed is already available and it is the end point of the line, while the remaining two are calculated using trigonometric expressions:



$$P_1 = O + Point(sin(\beta + \alpha) * AS, cos(\beta + \alpha) * AS)$$

$$P_2 = O + Point(sin(\beta + \pi - \alpha) * AS, cos(\beta + \pi - \alpha) * AS)$$

Figure 4.17: Arrowhead determination's algorithm

Where $\boldsymbol{\alpha}$, which default value is $\pi/3$, is the angle used to determine the stretch of the arrow and $\boldsymbol{\beta}$ is the angle between the edge's line and the horizontal axis of the scene; it has to be considered to normalize the coordinate system and make the arrow always pointing to the destination node.
AS is a constant used to determine the width of the arrow head; its default value, experimentally determined, is 12.
The following is an example of the final result:

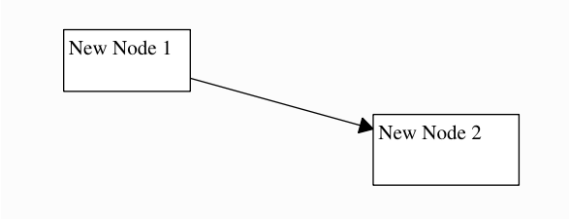


Figure 4.18: Output of the edge's default rendering process

Beyond the obvious functionalities of adding and removing edges, the QtNets graphic engine offers another very useful function: moving the source or the destination of an already defined edge from a node to another, given that the new node is allowed to be involved in an edge.

This is actually very similar to the resize functionality provided by the QNCommonItem class, mostly because it exploits the same combination of events and the concept of active areas.

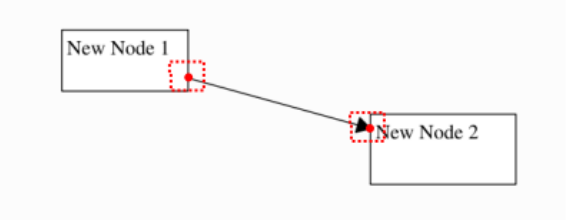


Figure 4.19: Edge's active areas

Also in this case, mouse hover events are used to update the cursor's icon when it moves across the active areas to inform the user that an interaction could be started just clicking there.

Figure 4.20: UML activity diagram of the edge's movement process

When a left-click event is received and it is located inside an active area, the corresponding end is anchored to the mouse cursor and can be moved around the scene along with it; the cursor's icon is set to a closed hand for as long as the anchor is held. The value of **moveSrcMode** and **moveDstMode** boolean attributes of the QNEdgeItem class is determined depending on which end of the edge is currently moving; the default condition corresponds to both of them set to false.
As usual, mouse move events are dispatched to allow real-time update of the item. When the mouse cursor is finally released, it is checked whether the edge's end intersects a node, because only in that situation the edge can actually be modified; if that is not the case, it is restored to the initial condition, the one before the interaction started. However, the selected node must be in a valid state to be associated to the edge; this is a general concept that can be extended in the future, but for now it correspond to the show detail state of the node item.
In conclusion, if the new node is confirmed, the replaced one is updated to remove the current edge from its list, the new one is updated to add the current edge to its list and the edge itself is finally updated; at this point, all the changes are propagated to the specific classes of the QtNetsData module to be persisted.

The final thing to be said about the QNEdgeItem class is that it does not support a dynamic content visibility management similar to the one offered by the QNCommonItem class, simply because it does not make sense hiding an edge; on the contrary, that may be deceptive for the user.

There is another very important class shown in the UML class diagram in Figure 4.7, that is linked to the QNNetworkScene class, as well as the QNCommonItem and QNEdgeItem classes, and that I never mentioned until this moment: the **QNStyleManager** class.
This class represents the connection between the graphic engine provided by the QtNetsDraw module and the styling system provided by the QtNetsStyle module. All the required details will be provided later on during the analysis of that module; for now it is possible to say that the QNStyleManager instance, if available, interacts with the painting procedures of all the items in the scene in order to apply the custom rules defined by the user.

4.3 QtNetsPersistence

The QtNetsPersistence module is somehow different from the modules analyzed so far, as it is focused on interfaces rather then classes.
The task of this module is to manage the persistence of the information spread among all the classes belonging to the QtNetsData module, that, I remember, is the only one in charge of managing core data related to the current network.
In order to make the entire framework as dynamic as possible, the QtNetsPersistence module provides indeed a general and abstract development interface, thanks to which it is possible for any developer all around the world to add sub-modules, called **plug-ins**, to support whatever format or storage device they need. n other words, the QtNetsPersistence module does not define a rigid storage format to be used to persist the network's information, but the basic instruments to allow everyone to do so.

The Qt framework offers great support to the plug-in oriented development, making relatively simple to obtain a dynamic system able to load software modules of different kinds directly at run-time.
More specifically, Qt provides two APIs to create plug-in modules:

- a high-level API to write extensions to the framework itself: custom database drivers, image formats, text codecs, custom styles, and so on;
- a low-level API to extend Qt applications.

In the specific case of the QtNetsPersistence module, the second approach has been used, so I am going to describe that one in detail; after all, the high-level API is based on it too.
Qt applications can be extended through plug-ins but this requires the application to detect and load them using a class specifically designed for the purpose; in this context, plug-in modules may provide arbitrary functionality and are not limited to database drivers, image formats, text codecs, styles, and the other types of plug-in that extend basic Qt's functionalities.

The **QPluginLoader** class is the standard class provided to load a generic plug-in at run-time. It provides full access to what is called a Qt plug-in, meaning a software module of any kind stored in a shared library.
An instance of the QPluginLoader class operates on a single shared library file and provides access to the contained functionalities in a platform-independent way.
To specify which plug-in to load, it is necessary to pass the complete path of the file containing it. Once loaded, all the plug-in modules remain in memory until all

instances of QPluginLoader has been unloaded, or until the application terminates. If the unload method is directly invoked on a plug-in that other instances of QPluginLoader are still using, the call will fail, and the actual unloading will only happen when every instance has invoked it. Right before the unloading happen, the root component will also be deleted.

Making an application extensible through plug-in modules involves the following steps:

- define a set of interfaces used to interact with the plug-in modules;
- use the **Q_DECLARE_INTERFACE()** macro to inform the Qt's meta-object system about the existence of those interfaces;
- use the already mentioned QPluginLoader class to physically load the plug-ins in the application;
- use the **qobject_cast()** function to test whether a plug-in implements the given interface.

On the other hand, writing a plug-in module requires these steps:

- declare a class that inherits from QObject and from the interfaces that the plug-in wants to provide;
- export the plug-in using the **Q_PLUGIN_METADATA()** macro;
- build the plug-in and use it.

The interface proposed by this module is actually very simple and is designed to guarantee that every plug-in for the QtNets persistence system has a set of fundamental functionalities, without which the correct behavior of the whole framework would be compromised.

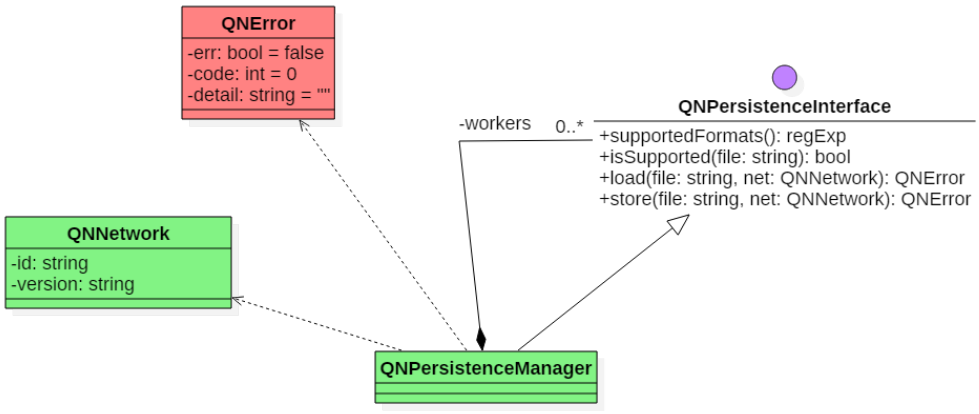


Figure 4.21: UML class diagram of the QtNetsPersistence module

It is possible to ignore the QNerror and QNNetwork classes, because they have been already described in detail during the analysis of the QtNetsData module, to which actually belong; they are reported in the UML class diagram in Figure 4.21 only because they are referenced by other components of this module. However, I take this opportunity to make an important note about the QNNetwork class: it is fundamental for those developers who want to build their own persistence plug-ins to understand and use this class properly, as well as all the related ones also belonging to the QtNetsData module; as a matter of fact, the information to be stored would be provided to them according to that formalism, as well as the information loaded from whatever supported storage device has to be delivered to other modules properly converted to suit the QtNetsData's formalism.
This will be clearer later, when I will present the one persistence plug-in I realized for this project.

The main component of this module is without any doubt **QNPersistenceInterface**; it is the general interface that has to be supported by any software module candidate to be a persistence plug-in.
Since this project has been entirely developed using the Qt-extended C++ programming language, the QNPersistenceInterface is, technically speaking, a class exposing only pure virtual methods, but in order to make the analysis as general as possible, I will consider it just as an interface to be satisfied, regardless of the implementation details; the UML class diagram in Figure 4.21 is also realized with this purpose in

mind, so the symbol used to represent QNPersistenceInterface is specific for interfaces rather than classes.
The methods that the interface exposes can be divided into two different categories:

- inquiry methods;
- action methods.

The first category contains all the methods that can be used only to know the public characteristics of the plug-in, while the second one contains all the methods designed to invoke specific actions on the plug-in module.

QNPersistenceInterface defines exactly four methods and for each one of them it is quite easy to guess the category it belongs to just looking at the signature. The first method in order of appearance is named **supportedFormats** and it is used to inform all the other modules about the storage formats that are supported by the plug-in module itself.
A single plug-in is free to support as many formats as it wants; for this reason, the method returns a list of regular expressions, one for each supported format, that is generically denoted in Figure 4.21 as **regExpList**. Every regular expression in that list has to follow the well-known filename extension convention.
For example, if it is necessary to design a brand new storage format, it is possible to associate a new non-standard extension to it, for example ".xyz", and the corresponding regular expression would be "*.xyz". Following the same rule, it is obviously possible to use also standard file extensions, as ".txt" or ".csv".

The **isSupported** method is used to check whether a given file, of which it is known the complete name, is supported or not by the current plug-in. This method is actually related to the first one, as it provides the same kind of information, just focused on a single file; as a matter of fact, the filename given as argument should be considered supported only if it matches at least one of the regular expressions that would be returned by the supportedFormats method.
It could be argued that the isSupported method is not strictly necessary to the QNPersistenceInterface, since it does not add any information, and that would be true: I decided to make this method mandatory anyways to make the interface easier to use in many different scenarios and to concentrate all the boilerplate code needed to determine the compatibility of a certain file in one point only.

The two methods just described constitute what I defined the inquiry section of the interface; the remaining two are part of the action section. Since they represent a pair of opposite actions that present many common aspects, I prefer to analyze them at the same time focusing on their differences, so I will start presenting their full signatures:

```
virtual QNError load(const QString& aFile,
                     ScopedQNNetwork& aNet) = 0;
virtual QNError store(const QString& aFile,
                      const ScopedQNNetwork& aNet) = 0;
```

The first method is in charge of loading all the data saved in the file pointed by the **aFile** argument and properly populate the network instance available through the **aNet** parameter, while the second method is designed to do quite the opposite, meaning that it stores all the information contained in the network pointed by aNet into the given file exploiting the formalism internally defined by the plug-in itself; if the file does not exist, it is simply created and used.
Simply looking at the signatures, it is clear the difference concerning the second argument: as a matter of fact, the store method requires that the given network is not modified at all by its execution, because that would cause inconsistencies between the information stored and the one already available to the user.
One final note is required about the returned object of both the load and store methods: they both return as a result an instance of the QNError class, which is, I remember, the default error class for the QtNets framework. It is important, as much as obvious, that this error is coherent with the results of the inquiry methods, in order to guarantee the correct behavior of the module; for this reason, the implementation provided for both the load and store method have to check whether the given file is supported before operating on it and return an appropriate error in case it is not.

Every persistence plug-in must implement the interface exposed by QNPersistenceInterface and an application using the QtNetsPersistence module can use how many plug-in modules it needs. However, those are not accessed directly, for it does not scale, but through a class of this module specifically designed for the purpose: the **QNPersistenceManager** class.
This class implements QNPersistenceInterface but it is not a plug-in, as it lacks the necessary configurations required by the Qt framework that I mentioned in the initial part of this section; it is instead a container, or manager, for all the possible persistence plug-ins that the application might use.

First of all, the QNPersistenceManager instance has to load all the plug-ins that are made available for the specific application: to do so, during its boot phase, it scans the content of a directory specified as only argument to its constructor method. For every file in such directory that is also a compatible shared library, it checks whether the QNPersistenceInterface is properly implemented and, in case it is, loads the plug-in into an internal data structure designed to contain all the plug-ins managed during the current session. This structure is technically a list of generic objects implementing QNPersistenceInterface and is named **workers**.
Moreover, since the QNPersistenceManager class in turns implements the persistence interface, it has to provide a valid and coherent implementation of the four methods previously discussed; given the fact this is a manager class, it does not directly provide any persistence functionality but acts as a centralized requests dispatcher for all the plug-in modules it manages.
In order to clarify this concept, I provide, as an example, the detailed analysis of the implementation strategy used for the mandatory load method.

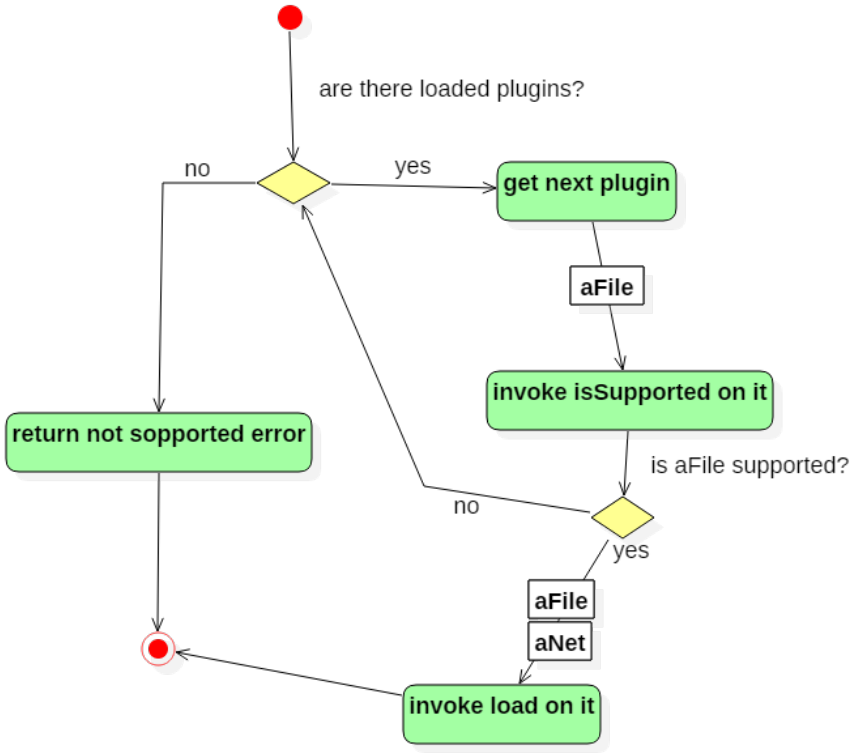


Figure 4.22: QNPersistenceManager's implementation of the load method

When the load method is invoked, QNPersistenceManager scans all the plug-in modules active during the current session in order to find at least one of them

supporting the required format. If there are no plug-ins supporting the format of the file specified as first argument, it directly returns an error object signaling that the required format is not managed at the moment, otherwise it invokes the load method of the first compatible plug-in found, forwarding the received parameters and returning the result directly to its caller.
In this way the QNPersistenceManager class acts as a sort of requests dispatcher for the different plug-ins it manages, offering at the same time an independent entry point to all the other modules that need to use the persistence subsystem.
The remaining methods are implemented following the same principle.

Thanks to this approach, it is possible to expand indefinitely the persistence subsystem without affecting the interface or any other part of the application; all it is needed is to design and develop a persistence plug-in that fits the specific application scenario.

It is clear that the QtNetsPersistence module is useless without at least one plug-in module that physically implements the algorithms required to manage the persistence of a generic graph.
At this point, during the design phase of the project, an important decision had to be taken: the choice was between designing a new custom file format to store the content of a network and supporting an existing one to make the framework usable also to edit networks previously built with other tools.
Although the first option was the cleanest and the most dynamic one, it would have made the framework accessible only to those who wants to build a complex graph from scratch, but useless for all the already existing networks; for this reason, I decided to realize a plug-in supporting the storage format used by one of the most popular graphs editors available today, thus making possible to create and import networks at the same time.
Among the most commonly used graphs editor applications already mentioned in the second chapter of this document, I chose to support the **SMILE** format, the one used by the GeNIe editor. The reasons that led to this choice are basically two: this format is widely spread in the academic environment, as a matter of fact the research group I collaborate with for the realization of this project uses it too, and it is actually quite simple and complete.
The SMILE format is an XML application fully defined by two schema files: one is focused on all the elements and attributes necessary to describe the core characteristics of the network, while the second defines all the instruments available to customize its graphical properties. The complete definition of both schema files can be found in the "Appendices" section at the end of this document.
The SMILE XML application also defines a custom file extension to conventionally identify its source files: that is **.xdsl**. Based on what have been said so far, the

regular expression representing this category of files is quite straightforward.

In order to load all the information contained in a SMILE source file, it is necessary to develop an XML parser that is able to validate the input file against the standard schema and convert the contained data from the SMILE format to the QtNets one. The opposite conversion must be performed when it is necessary to write all the data saved in the QtNetsData module's classes on an output file respecting the standard SMILE format.
The Qt development framework, as usual, comes to the aid of the developer offering support for the basic parsing and writing operations on a generic XML file; in particular two base classes are provided: the **QXmlStreamReader** class to support parsing and the **QXmlStreamWriter** one to perform writing operations.

The QXmlStreamReader class provides a fast parser for reading well-formed XML files via a simple streaming API.
The basic concept of a stream reader is to report an XML document as a stream of tokens, similarly to what happens when exploiting the well-known SAX (Simple API for XML). The main difference between the QXmlStreamReader class and the traditional SAX is how XML tokens are reported: with SAX, the application must provide handlers, meaning callback functions, to receive the so-called XML events from the parser at the parser's convenience; on the other hand, using QXmlStreamReader the application code itself drives the loop and pulls tokens from the reader, one after another, as it needs them.
A set of convenient functions can then be used to examine the token and obtain any kind of information about it.
The big advantage of this pulling approach is the possibility to build recursive descent parsers, meaning it is possible to split the XML parsing code easily into different methods or classes; moreover it is possible to easily keep track of the application's own state during parsing execution.

On the other side, the QXmlStreamWriter class provides an XML writer with a simple streaming API very similar to that offered by QXmlStreamReader; in other words, QXmlStreamWriter is the counterpart to QXmlStreamReader for XML writing. Like its related class, it operates on a **QIODevice**, a base and generic interface representing all I/O devices in a Qt based environment; this means that the output of the XML writing process could be any, from a traditional file to a connected TCP socket.
The API is simple and straightforward: for every XML token or event to write, the writer provides a specialized function. In addition to that, the stream writer can automatically format the generated XML data adding line-breaks and indentation to empty sections between elements; in this way the resulting code is more readable

for humans and easier to work with for most source code management systems.

The plug-in module realized to manage the SMILE format is named **SmilePersistencyPlugin**; it exploits the available standard classes I just described to implement the mandatory QNPersistenceInterface and support the basic load and store functionalities.
It must be said that the SMILE data format is absolutely complete but specialized in modeling a very particular family of networks, that are Bayesian networks; for this reason it offers a great number of specialized elements and attributes that are totally meaningful, and some times essential, in the context they are designed for, but can create confusion when applied to other different types of networks.
As a result, both the parser and the writer, allow the user to choose between two different working modes:

- strict mode;
- lenient mode.

The **strict mode** is designed to enforce exactly all the restrictions imposed by the SMILE schema file and is configured to immediately terminate the parsing or writing process whenever a mistake of any kind is found; on the other side, the **lenient mode** is less rigid concerning schema rules and is configured to try to correct all the possible discrepancies that could affect the compliance to the standard schema. The strict mode has been designed to address all those situations in which the user is actually dealing with Bayesian networks and needs a rigid control over the schema compliance; working in this mode guarantees that the resulting network is fully compatible with the native GeNIe editor application, so, in case an export to that environment is necessary, the network results identical to the one displayed by the QtNets framework.
On the other hand, the lenient mode has been designed to support all the users who need to study other kind of networks but still want to keep compliance with the standard. In this scenario, the user might be not interested in knowing the details of all the data structures made available by the SMILE standard to model a network, because they can result too much complicated or even unnecessary for the specific matter; for this reason, both the parser and the writer, try to provide a default definition of all the data structures that are required to keep the compliance with the standard but are not meaningful to final users.

Supporting the SMILE file format does not mean just provide an XML parser able to understand the standard notation and a writer able to write it somewhere, it also means providing the algorithms to convert all the data structures defined in

the SMILE standard into instances of the classes defined in the QtNetsData module and vice-versa.
For instance, the SMILE schema defines seven different kind of nodes, each of them with its own peculiar properties; they are all mapped to the QNNode class but are differentiated by a specific property reporting the name of the corresponding SMILE node. Thanks to that property, it is possible to perform the reverse conversion when it is required to save a QNNetwork instance, with all its children, into a smile file.
This is just a simple example to show how it is possible to manage the conversion between structures representing shared concepts, as is the concept of node; however, there are several other situations in which this is not possible because the very specific SMILE structures does not have a direct equivalent in the QtNets environment, which is designed to be as abstract and general as possible. In those situations, a custom class is defined and exported by the SmilePersistencyPlugin to physically implement the entity formally described in the SMILE schema.
Those custom classes have to support all the functionalities required by the Qt's meta-object system to allow their instances to be inserted into a QVariant container; in this way they can be associated to any class of the QtNetsData module that supports generic properties extending the QNProperties class.

The concepts and methodologies at the base of the SmilePersistencyPlugin that I explained in this section are actually very important because they should represent a sort of solid guide line for other plug-in modules that might be added in the future: they are the very few rules to be satisfied in order to ensure that a certain plug-in is functionally compatible with the rest of the framework, not just because it implements the required interface.

4.4 QtNetsStyle

The graphic engine offered by the QtNetsDraw module is theoretically enough to provide the rendering utilities needed to display a network; however, the very basic functionalities provided by that module can certainly not be enough to represent all the possible meanings that a network may have based on its context, because they are too much general.
For instance, a single element, like a node, could have different meanings based on its role inside the network, or based on its siblings; those different roles might be represented using different shapes, colors, or even a combination of the two. The list of examples could be much longer, but the idea is quite clear: it is needed, maybe required, a dynamic module able to set additional attributes and properties to elements in order to allow them to assume different roles in different occasions.
This is the main reason why the QtNetsStyle module has been designed and developed.

The inspiration on how to implement the style system came from the approach typically used to develop web applications: in that scenario, an abstract system that completely separates the structure of the web page from the detail of its appearance is obtained using Cascading Style Sheets (CSS) to define and apply style rules on the elements of the web page. The structural definition of an element, typically contained in an HTML source file, is linked to its style definition through a unique id or a class name, making the whole system incredibly dynamic: it is enough, for instance, to provide a different style sheet using the same identifiers and classes to provide a completely new look and feel to the application without changing any structural aspect.
Since the advantages related to the use of external style sheets are exactly those the QtNets framework needed, I based the design of this style engine on it, with the important difference that I did not define a custom notation similar to the CSS language, but only the guidelines and instruments to manage styles.

Mindful of the experience matured during the development of the QtNetsPersistence module, I decided to use a very similar plug-in based approach, in order to make it possible to define plenty of plug-in modules to implement whatever style feature can be imagined.
To enforce that, the QtNetsStyle module must provide the basic instruments, meaning classes and interfaces, to manage the relation between a style rule and the item to which it is applied and to manage the proper rendering of the expected graphical properties.

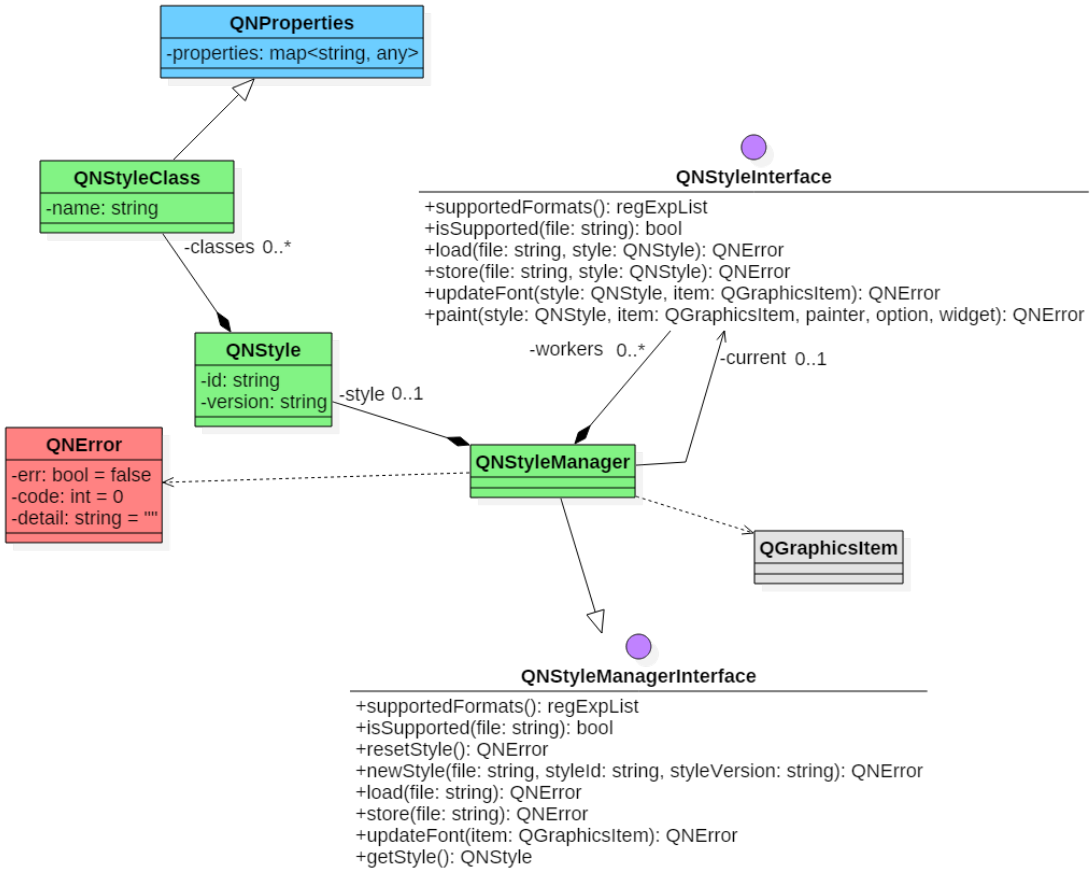


Figure 4.23: UML class diagram of the QtNetsStyle module

The UML class diagram in Figure 4.23 transpires the fact that the module can be ideally divided into two sections: the first one defines what a style rule is and how it can be related to the item it has been attached to; the other is strictly focused on how to enforce those dynamic rules during the scene's rendering process.

The base concept is this: it is not possible to associate scattered style rules to some of the items of the current network, but it is necessary to associate a properly

configured style object to the whole network; the characteristics of such object are modeled by the **QNStyle** class.
There could be only one instance of this class active in the context of the application at a given time and it represents the set of all the style rules associated to the current graph.
The UML class diagram in Figure 4.23 shows that the QNStyle class has two string attributes, **id** and **version**, meant to uniquely identify the style's instance and keep track of its evolution during time respectively; however, they are just additional information to better classify the instance itself: the primary role of the QNStyle class is to act as a centralized container for the style rules to be applied, so the fundamental attribute is **classes**, the one that connects the instance to all the rules defining it. To be noticed that the composite symbol (the black diamond on the container side) has been used in Figure 4.23 to represent that relation, in order to remark, once again, that a style rule cannot live outside the context of the style object containing it.
Given its role inside the module, the QNStyle class offers a series of methods to add, modify, delete and retrieve one or more style rules, concentrating all the complexity related to those actions in one single point. For instance, it guarantees that no duplicated rules are ever defined by checking them for uniqueness whenever a new one is about to be added: if there is already a rule identified with the same name of the one to be added, the two are merged together enforcing a merge criterion such that any possible conflict is solved in favor of the youngest rule.

The style rules I mentioned so far are modeled by **QNStyleClass**; the "class" suffix in its name is a sort of tribute to CSS classes from which I took most of the inspiration. For this reason, it is correct to call them style classes as well.
A QNStyleClass instance only needs a **name** to be identified; as already said, it is suggested to choose a unique name, but in case it is not, the related issue is entirely managed by the QNStyle container. What is really important to notice about this class is that it extends the already known QNProperties class of the QtNetsData module, thus taking all the benefits; this is the first structural dependency of the QtNetsStyle module over QtNetsData.
The fact that a style class is defined in such an abstract way as a simple map of generic properties offers a series of advantages, the most important of which is its intrinsic dynamicity: using this approach, it is neither necessary nor required to define a fixed notation for style rules, leaving that design choice entirely on the shoulders of the developers of plug-in modules. In other words, thanks to this approach, a generic plug-in module has the possibility to enforce any kind of graphical property, as long as it guarantees three fundamental things:

- the definition of a proper notation to define custom style classes;
- a set of utilities to load style rules from a source file and to store them;
- a set of algorithms to enforce the supported graphical properties during the standard rendering process.

Properly populating a QNStyle object with a certain number of QNStyleClass instances is the right way to provide the network with an external custom style, but those two classes have no information at all about which rule has to be applied to which item; this information is totally managed by the QtNetsData module and represents the second dependency between the two modules; this time a logical one. The conjunction ring is represented by the QNStylable class of the QtNetsData module; as a matter of fact, every icon class extends it, directly or indirectly, and it is responsible to store the relations between the current icon and all the style classes associated to it.

The relation between an icon and a style class has to be explicitly created by the user in order to enhance the property of that icon through a certain class; as it is, the style system does not allow to customize an entire category of icons at one time because it does not support selection by element. To overcome such limitation, every icon, when it is first created, automatically associates to itself a predefined class name to identify the family it belongs to; in this way, those particular class names can be used to define global style rules. They can be therefore considered as a sort of keywords for the QtNets style system and have to be used as such.
The currently supported keywords are:

- **node**, the class name representing all the nodes in the scene;
- **edge**, representing all the edges in the scene;
- **textbox**, implicitly associated to every text box in the scene;
- **model**, representing every sub-network or model in the current scene.

The **QNStyleInterface** pure virtual class models the set of functionalities that a generic software module has to provide in order to be considered a compatible style plug-in. A brief analysis of its definition reported in Figure 4.23 reveals an interesting number of similarities with the QNPersistenceInterface one analyzed in the previous section: in a sense, QNStyleInterface could be considered as a sort of specialized extension of QNPersistenceInterface, for it manages style's persistence but also provides a set of methods to actually apply its rules.

For this reason, there is not much to add about the nature of the first four methods of the interface, the ones dedicated to style's persistence management, because they are absolutely equivalent to those specialized in managing network's persistence; the only difference is they manage different data structures.
The remaining two methods, on the other side, are specifically designed to enforce the style rules supported by the plug-in itself interacting with the standard painting process managed by the QtNetsDraw module, therefore represent the specialized section of the interface. The two methods **updateFont** and **paint**, are, all things considered, quite similar to each other concerning the actions they perform; the only difference is the target of those actions: the first method is specialized in managing all the graphical properties referred to fonts and texts in general, while the second deals with every other graphical aspect of the item to be rendered.
It is natural, at this point, wondering why the two methods have been split, if they really do the same job: the reason for that separation is exclusively related to performances. As a matter of fact, in order to guarantee responsiveness and real-time update of the scene, the containing view continuously dispatches re-painting requests to the portion of the scene the user is currently modifying, so it is possible that the paint method of each item involved is called a significant number of times per second. If the time necessary to update all the items in the active portion of the scene is higher than the repainting requests rate required to manage the particular interaction the user is performing, the application is likely to crash due to overlapping update requests.
This is generally not a problem in most cases, since the time needed to configure the QPainter's instance provided to render the item is usually very short, but when the properties of the text to be displayed inside the bounding rectangle of the item is continuously updated, even if nothing has changed in the meanwhile, the time needed to update the scene grows so much to make a crash possible, if not likely. Even in the lucky circumstance that a crush is avoided, stretching too much the scene's update process will cause a tangible loss of reactivity that might lead the user to think the application is not working properly; it has to be prevented in any case.
Therefore, in order to avoid the materialization of that risky situation, all the procedures to update text properties have been separated from the rest and put in a specialized method to be called only when something really changes concerning fonts and similar.

As already known, it is highly discouraged to invoke QNStyleInterface's methods directly on the instance of the plug-in implementing it, because this approach is not abstract enough and generally does not scale. It is necessary to have a manager class that can be used by other modules to request the execution of actions and, at

the same time, manages entirely the complexity of having dynamic plug-in modules; this role belongs to the **QNStyleManager** class.
Similarly to the QNPersistenceManager class, also QNStyleManager has to deal with the runtime loading of the different plug-in modules made available for the current application and exploits the benefits of the QPluginLoader class to solve the problem. Both the implementation and the general behavior of this feature are equivalent to those offered by QNPersistenceManager, so there is not much to add; the principles and guide lines followed in this case are exactly the same.
Differently from the QNPersistenceManager class, QNStyleManager does not implement the common interface as all the style plug-in modules have to, because it rather provides a different and easier one to the other framework's components; this is named **QNStyleManagerInterface**. Looking at the UML class diagram in Figure 4.23, it can be noticed that the only QNStyle's instance allowed is actually owned by the QNStyleManager class, which means that the clients of the manager class do not have to worry about the life cycle of the style's instance or about its correctness. For this reason, the QNStyleManagerInterface is identical to the one exposed by QNStyleInterface with the only difference that the style parameter is not required anymore; in addition to that, it also offers some other useful methods to create from scratch and reset the current style object.
The second fundamental purpose of the QNStyleManager class is to manage all the plug-ins currently instantiated and dispatch requests to the right module as they are received. This task is not to be neglected because it is not guaranteed that all the style plug-ins are functionally equivalent to each other as it was in the case of network's persistence management: as a matter of fact, it is likely more than possible that different style plug-ins support different sets of graphical properties, which might be incompatible to each other; for this reason, a style object populated using a specific plug-in module can be managed only by the same plug-in until it is reseted. In order to remember which plug-in module initialized the current QNStyle's instance, the QNStyleManager class stores its reference in the **current** attribute.

The last piece of information needed to understand the behavior of the style sub-system is related to how the QNStyleManager class is involved in the standard rendering process managed by the QtNetsDraw module. To make that clear, I propose as an example the UML activity diagram describing the steps required to manage the paint method; it is valid to understand the execution flow of the updateFont method too, as it is more or less the same.

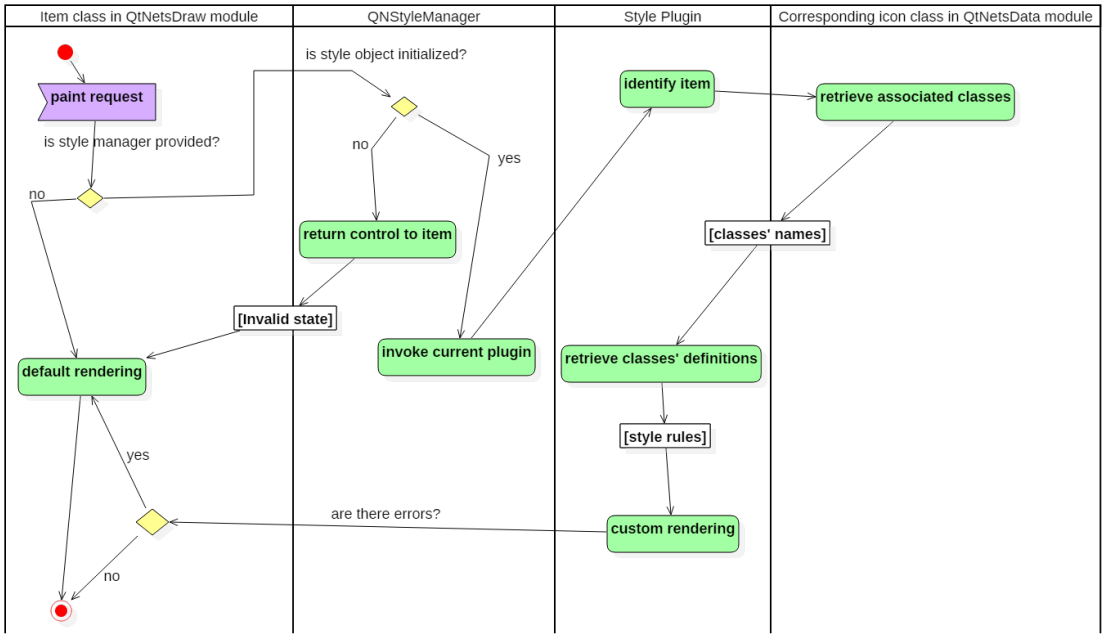


Figure 4.24: UML activity diagram of the custom paint procedure

Every item class in the QtNetsDraw module (QNNodeItem, QNEdgeItem, QNModelItem, and QNTextBoxItem) have access to the active QNStyleManager's instance, therefore, at the very beginning of every paint cycle, they test the validity of that manager in order to know if a custom procedure is available; in that case, control is transferred to the style manager, otherwise the default paint algorithm is executed.
When QNStyleManager is involved in the painting procedure of every item, first of all, it has to verify if the style object is properly initialized, in order to yield in turn control to the plug-in module that actually manages that specific kind of style; if this condition is not verified, or the plug-in module in charge is not available for any reason, QNStyleManager informs the calling item about the invalid state returning an appropriate error object, so the item can provide the default paint implementation.

From now on, it is only possible to provide general guidelines about the main steps that should be followed by plug-in modules, because the actual implementation

depends on their own internal conventions. However, the following rules should be kept into serious consideration in order to develop an effective style plug-in:

- the first thing to do should be identify the item to be painted, in order to know the kind of element it represents; this is important because it might be possible that some rules are applicable only to certain items. For instance an item representing an edge may accept properties related to its arrow that would make no sense if associated to a node or a model.
- When the item is identified, it is necessary to retrieve all the associated style classes from the icon class associated.
- When all the class names are available, the plug-in module has to translate them into graphical rules using the specification of each style class internally defined.
- The last step consist in the execution of all the algorithms needed to enforce the style rules just determined.
- Every error that might occur during the execution of any procedure supplied by a style plug-in should be returned directly to the calling item, exploiting the common QNError class; in this way, the item can still provide its default implementation to recover the error.
- The implementations provided by every style plug-in for the paint and updateFont methods must respect the same principles of the default ones provided by the QtNetsDraw module: no matter the graphical customizations supported, those update procedures must always respect quite rigid time limits otherwise the whole application will be affected; if a certain customization requires too much time to be performed, it should not be supported.

In order to make all this system work it is necessary to provide at least one style plug-in, so I decided to develop a style module in the context of this project named **BaseStylePlugin**.
This time I designed not only the set of style rules to be supported, but also the encoding to be used to store them on file system. I decided to develop an XML application for the definition of the style properties associated to each item on the screen for a number of reasons: first of all, the XML notation is compact and easy to read for human beings, thus making a style sheet defined using that notation understandable simply looking at the source file. Secondly, it is not only easy to understand source files, but also to develop applications to automatically process

them, especially in this scenario where the Qt framework offers great support. Finally, since a very similar notation has been already used to develop another module of this project (the SmilePersistencyPlugin module), it is well tested and properly controlled, not to mention the fact that all its strengths and weaknesses are already known.
The base style module allows to define rules to customize the following areas of interest:

- background;
- outline;
- font;
- item shape;
- arrow shape.

The background property of every item can be customized in several ways exploiting the combination of **color** and **brush style**; it is quite straightforward the meaning of the color property, while as for brush style I refer to the filling pattern used to apply the color property.

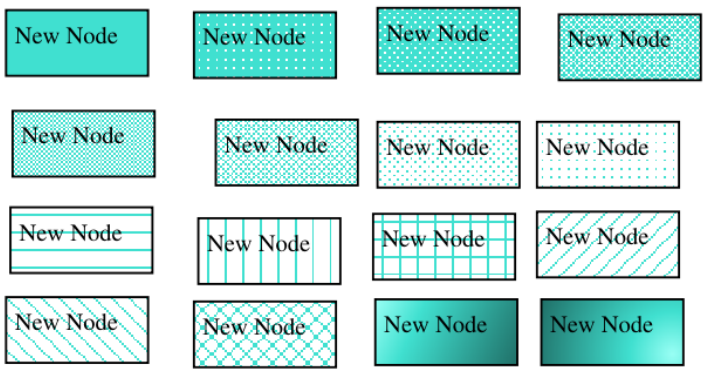


Figure 4.25: Background's available configurations

Figure 4.25 shows all brush styles supported by the BaseStylePlugin, that are from top left to bottom right:

- solid (default choice);
- dense 1;

- dense 2;
- dense 3;
- dense 4;
- dense 5;
- dense 6;
- dense 7;
- horizontal lines;
- vertical lines;
- crossed lines;
- forward diagonal;
- back diagonal;
- crossed diagonal;
- linear gradient;
- radial gradient.

Since Figure 4.25 already speaks for itself, I will focus only on the last two options, because they are definitely the most interesting. Linear gradient allows to obtain a smooth transition between two colors, which represent the start and end points respectively; since the user has the possibility to specify only one color, the start and end points are derived from that color by fading and darkening it respectively of an experimentally determined value.
On the other side, radial gradients interpolate colors between a focal point and several end points on a circle surrounding it. In the case of this style module, the focal point of the radial gradient is always located in the bottom right corner of the item, so the graphical effect obtained is clearly visible; concerning the two colors to be interpolated, it has been used the same approach of the linear gradient.

Qt of course offers native classes to support the realization and configuration of both linear and radial gradients: **QLinearGradient** and **QRadialGradient**. Thanks to those classes, it is possible to define a complex pattern like a gradient using a simple piece of code like the following:

```
QLinearGradient linearGradient(0, 0, width, height);
linearGradient.setColorAt(0.0, fillColor.light(150));
linearGradient.setColorAt(0.3, fillColor);
linearGradient.setColorAt(1.0, fillColor.dark(200));
```

To conclude, few words about the color: it is simply the concatenation of three hexadecimal digits, one for each RGB component, preceded by an hash symbol; each component can assume value between 00 and FF, which correspond to 0 and 256 respectively using the traditional decimal system. For instance the encoding associated to white, that is also the default background color, would be #FFFFFF.

The outline customization is a little more complicated, because it depends on an higher number of factors compared to background; those factors are:

- line width;
- line color;
- line style;
- cap style;
- join style.

Line width is just an integer number stating the thickness of the item's outline and the line color property is identical to the background one, both in the meaning and in the encoding being used; default values are 1 for the outline's thickness and black (#000000) for its color.
The line style property can be used to define the pattern for the outline; the following choices are available:

- no line at all;
- solid line (default choice);
- dashed line;

- dotted line;
- dash-dot line;
- dash-dot-dot line.

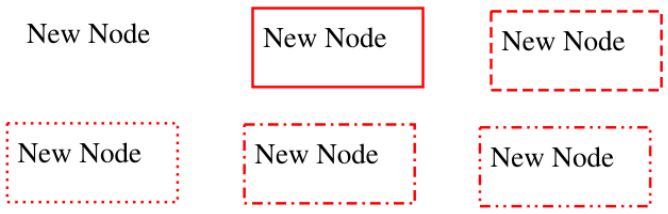


Figure 4.26: Outline's available configurations

The cap style defines how the end points of lines are drawn; this option only apply to wide lines, meaning lines for which the width is one or greater. Supported values are:

- flat (default choice);
- square;
- round.

The flat cap style is a square line end that does not cover the end point of the line, the square cap style is a square line end that covers the end point and extends beyond it by half the line width, while the round cap style is a rounded line end covering the end point.

Figure 4.27: Outline cap's available configurations

The join style defines how joins between two connected lines can be drawn and it also apply to wide lines only. The following options are available:

- miter join (default choice);
- bevel join;
- round join.

The miter join style extends the lines to meet at an angle, the bevel join style fills the triangular notch between the two lines and the round join style fills a circular arch between the two lines.

Figure 4.28: Outline join's available configurations

The font configuration is actually very easy to understand and common to a great number of text editors available today; to be specific, the BaseStylePlugin module allows to configure the font family, its size and color, the italic and bold flags and finally the alignment of the text, that can be left-aligned, centered or right-aligned. An example of font configuration, that is also the predefined one, is the following:

- font family: Times;
- font size: 8;
- font color: black (#000000);
- bold: false;
- italic: false;
- align: left.

The item's shape is actually considered valid only if associated to a node item, therefore it would be more precise to refer to it as node's shape. The reason for this choice is quite simple: the BaseStylePlugin module is not meant to address any particular application field; on the contrary it has been designed to be a sort of demo module, a module that is able to trace the guidelines for any specific style module to be developed in the future.
Besides that, the node shape attribute simply states the name of the geometric figure used to draw the nodes to which it is applied.

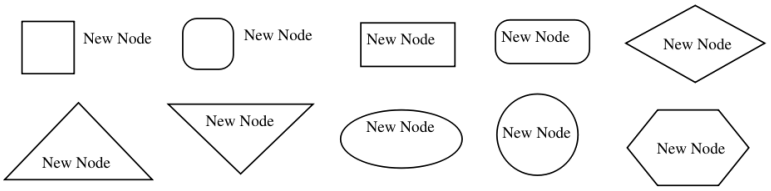


Figure 4.29: Available node's shapes

Considering the shapes in Figure 4.29 from top left to bottom right, the supported values for this attribute are:

- square;
- square with rounded corners;
- rectangle (default choice);
- rectangle with rounded corners;
- rhombus;
- triangle;
- inverse triangle;
- ellipse;
- circle;
- hexagon.

If those shapes are not enough, or the application field requires more specific ones to represent a node, it is possible to load a custom image to be used as background of the node.

The last property that is possible to customize using the BaseStylePlugin module is meaningful only if applied to edges; this is the shape of the terminal arrow used to signal its direction.
Valid arrow shapes are:

- no arrow;
- empty arrow;

- full arrow (default choice);
- circular arrowhead;
- diamond arrowhead.

The following figure shows the result of all the supported values in the same order they have been presented:

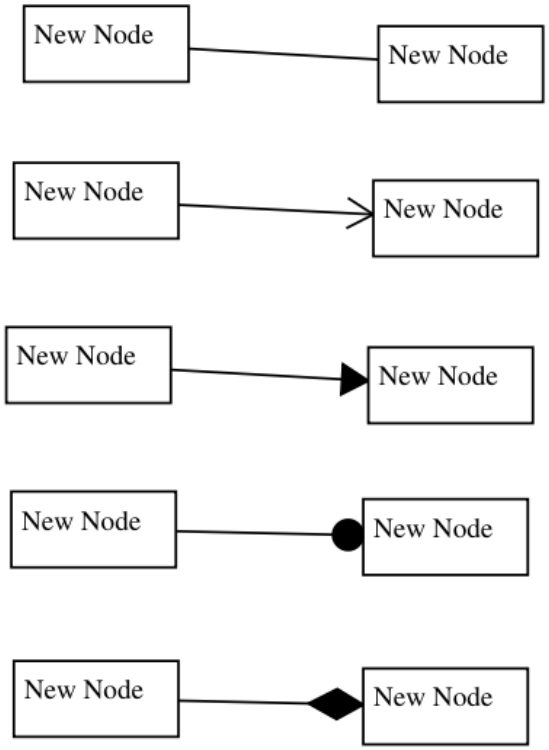


Figure 4.30: Available arrowheads

Chapter 5

Test and performance analysis

The QtNets framework is accompanied by a set of test applications to validate the final result under several points of view; in particular, the applications designed for the purpose are two: **QtNetsTest** and **QtNetsBenchmark**. The first one is a console application that executes a series of unit tests to check the correctness of the various software modules that compose the framework, while the second is a console application designed to analyze its performances.

Unit testing is in general very important in every modern software development chain, because it helps incredibly to automate a number of tedious and error prone testing procedures and makes it easier to find potentially critical bugs in the early phases of the development process.
The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the rest of the code, and determine whether it is consistent with the specifications or not; each unit is tested separately before integrating them into modules to verify interfaces and connections between each other. The most common approach to unit testing, which is also the one followed in this specific case, requires drivers and stubs to be written: drivers are necessary to simulate a calling unit, while stubs simulate the called ones. However, it is very common that the investment in terms of time required by this activity demotes unit testing to a lower level of priority; that is almost always a mistake because, even though drivers and stubs cost time and money, unit testing provides much more valuable advantages. For instance, it allows for automation of the testing process, reduces difficulties in discovering errors contained in more complex pieces of the application and test coverage is often enhanced because attention is given to each unit.
In addition to that, unit testing has the advantage of transforming the pure testing phase, which is traditionally one of the last phases of the chain, into a new, and also challenging, development activity. This advantage is not to be neglected because, as

well as making testing fun, it allows to discover not only functional bugs, the ones that are immediately visible to final users because linked to a misbehavior of the application, but also structural ones, which are the most insidious and hard to find. In order to explain the concept of structural bugs, I propose a practical example: consider the scenario in which a developer has to design, and then implement, a software module to be used by its colleagues, or team mates, in the various projects they are involved into; the first step is to design the interface of such module, trying to be as general as possible in order to make it easy to use in several different situations. It is common that, even when an interface seems well designed on paper, it eventually results not so effective when implemented; moreover this kind of problems is typically discovered after the product has been delivered, by the same users who have to exploit its functionalities to develop their own products.
If unit testing is used during the development process of the single software module, it is very likely the developer discovers those issues himself, way before the product is ready to be delivered, so he has the possibility to fix them right away, saving precious time to other members of the team. In addition to that, the developer has the possibility to strengthen its own best practices, which will help him not to make again the same mistakes in the future tasks to perform.
This example actually reflects very much the lessons I learned myself during the development of this project: unit testing really spared me quite some time and allowed me to find silent structural bugs that would have become critical if found in the late stages of the development chain.

The QtNetsTest application has been developed exploiting the instruments provided by the Qt framework, which offers great support to all the phases of a typical software development process. In this specific case, it offers a specialized framework, named **Qt Test**, to support unit testing of Qt based applications and libraries.
Qt Test provides all the functionalities commonly found in unit testing frameworks, as well as powerful extensions to test graphical user interfaces; most importantly, it is designed to ease the writing of unit tests: it is lightweight and self-contained, in fact it requires only few symbols from other Qt modules, especially in those situations in which GUI testing is not required; it offers an interface designed to support rapid testing, as it does not need any special test-runners or registration procedures, and data-driven testing, as it makes it easy to execute a test multiple times with different sets of inputs. It guarantees also a basic GUI testing environment, which offers functions for mouse and keyboard simulation to trigger specific reactions of the user interface to be tested. Finally, it is also IDE friendly, since it outputs detailed messages about the results of each test case and especially about the error occurred in case of a failed test.
Despite all of that, the most important characteristic of the Qt Test framework is

its ease of use: all it is required to create a test application is to develop a new class extending QObject to which add a number of private slots; each private slot will be a test function. Besides, the QtCreator IDE offers a very intuitive configuration widget for unit testing projects, thanks to which it is possible to initialize the test class with all the required boilerplate code simply interacting with the interface of such a widget.

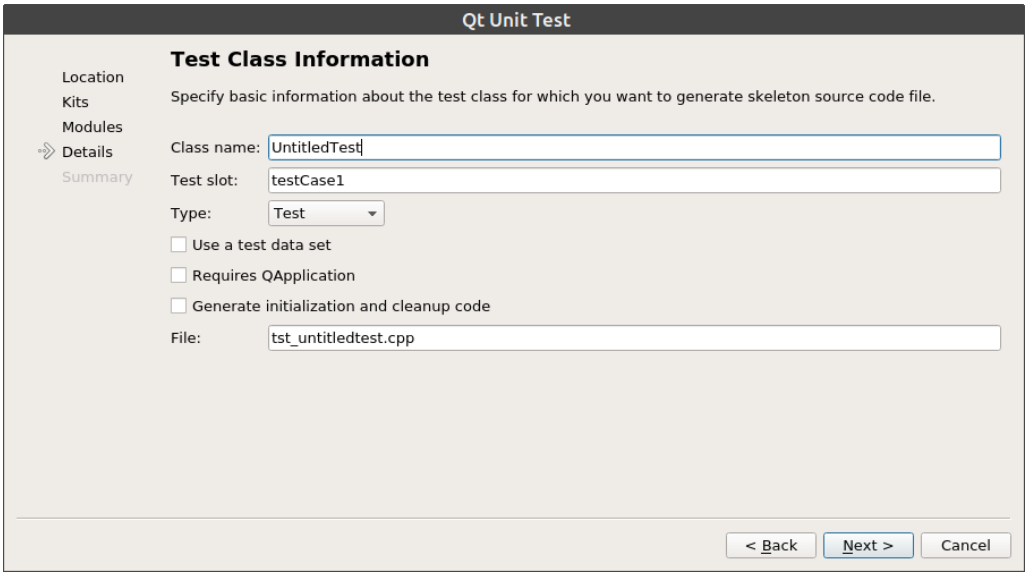


Figure 5.1: Configuration widget for test projects

Among other things, it is possible to configure the IDE so it automatically adds four private slots to the test class that are not treated as traditional test functions: they will be executed by the testing framework in specific moments and can be used to initialize and clean up either the entire test session or the current test function. Those slots are:

- **initTestCase**, executed before the first test function is executed;
- **cleanupTestCase**, executed after the last test function is executed;
- **init**, executed before each test function is executed;
- **cleanup**, executed after every test function.

To be noticed that, if initTestCase fails, no test function is executed, while, if init fails, the current test function is not executed but the flow continues to the next slot.

5.1 QtNetsTest

In order to analyze in detail all the test functions that have been developed, I start providing an example of the output returned by the QtNetsTest application.

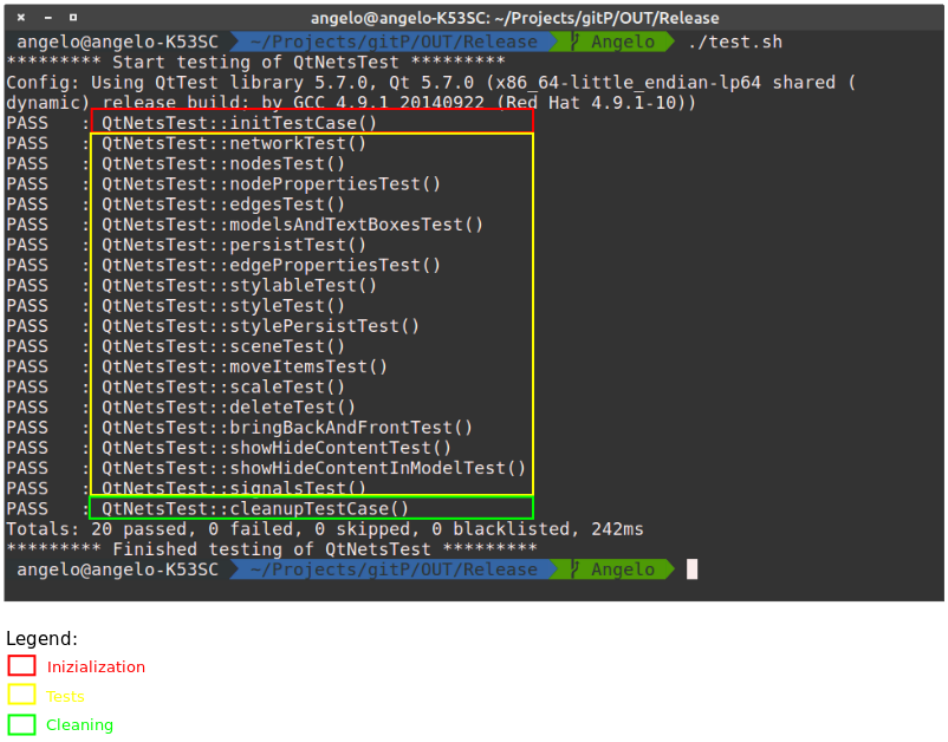


Figure 5.2: Output produced by QtNetsTest

First of all, I want to focus the attention on the classification proposed in Figure 5.2, because it is useful to distinguish real test cases from the ones provided by the testing environment. In particular, it should be noticed that only global initialization and cleaning functions have been exploited, leaving aside the init and cleanup procedures associated to every single test; the reason for that is simple: given the differences between all the test cases, it is not possible to identify common initialization and finalization procedures to be collected in those two methods, so the approach that have been used consists in including initialization and finalization procedures, specific of each test case, directly in the corresponding slot.
Concerning the initTestCase and cleanupTestCase methods, the default implementation did not need to be changed as they only take care of managing all the structures and tools used by the testing environment.

The section identified by the yellow rectangle in Figure 5.2 actually contains all

the test cases designed to check every feature of the QtNets framework. However, before analyzing the purpose of each test function, it is necessary to spend some time listing the common classes that are required by every test case, since they represent the very core of the whole framework; those are allocated by the constructor of the QtNetsTest class, shared among all slots, modified as required by the test case they implement and finally freed by the QtNetsTest's destructor method.
To be specific, the entities without which the entire test application could not work are:

- a shared instance of the QNNetwork class, that represents the randomly populated network on which all test cases applies;
- an instance of the QNPersistenceManager class, to be used during the validation of the persistence system;
- an instance of the SmilePersistencyPlugin to provide the actual implementation of the QNPersistenceInterface exposed by QNPersistenceManager;
- an instance of the QNStyleManager class, along with one of the BaseStylePlugin, to be able to check all the style related features;
- shared instances of the QNNetworkView and QNNetworkScene classes to verify all the GUI related features offered by the QtNets graphical engine.

The first test case on the list is named **networkTest** and is dedicated to verify all the methods and functions provided to initialize a network structure: it basically invokes all the setter methods offered by the network and, once all the initialization phase is complete, it invokes all the corresponding getter methods with the aim of checking whether the returned values are exactly the same that have been previously set. The properties that can be associated to a network could be of any kind, since every data-oriented class of the QtNetsData module extends QNProperties, so this test case is also useful to check the basic features offered by that class.

The **nodesTest** slot is designed to check all the features offered by a node structure, represented by the cooperation of the QNNode and QNNodeIcon classes. Also in this case, the great part of the test procedure is about setting the value of properties and then retrieving such values to verify they are equals, but the slot is also meant to verify the relation between nodes and network. In the sake of this purpose, a randomly determined number of nodes is attached to the shared network instance and it is then verified whether every one of them has actually been updated accordingly.

The **edgesTest** method, one step ahead of nodesTest in Figure 5.2, is designed to perform the same kind of validations on edge structures, meaning the related instances of QNEdge and QNEdgeStyle classes; it has to be executed necessarily after the success of the nodesTest slot, because the network must have valid nodes to be used as sources and destinations of the edges to be tested.

With the same spirit, the **modelsAndTextBoxesTest** slot verifies all the getters and setters methods of the QNModel and QNTextBox classes respectively and their relation with the current network entity.

Considering this four test cases it is quite evident how the stub and drivers concept I presented at the beginning of this chapter can be put into practice: before any feature of the framework could be validated, it is necessary to randomly generate values to be then used to check the coherence between the setter and the corresponding getter methods for every supported property and attribute. This of course might require a significant effort.

The **nodePropertiesTest** and **edgePropertiesTest** slots are in some way linked to each other, as their names suggest; in particular, they share the purpose of checking the correctness of the set of methods provided by the QNNetwork and QNNetworkModel classes to retrieve objects based on a given combination of properties. Those methods have already been analyzed in detail, but I briefly recall them just for the sake of clarity:

```
QNNodeList getNodesByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNEdgeList getEdgesByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNIconList getIconsByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);

QNEdgeStyleList getEdgesByProperties(
        const QNPropertyList& aProperties = QNPropertyList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);
```

The first two methods are provided by the QNNetwork class while the second couple belongs to QNNetworkModel.

The guidelines followed to develop these two test slots are more or less the same as before: the nodes and edges allocated by the previous test cases (nodesTest and edgesTest) are updated in order to associate them a specific set of properties, then the methods under analysis are repeatedly invoked using different combinations of input parameters and the result of every call is verified. If the set of nodes, or edges depending on the test case considered, is not the expected one, a detailed error message is displayed to explain exactly how the actual result is different compared to the expected one.

The **persistTest** method is designed to check the persistence engine provided by the QtNets framework; the strategy used to pursue the objective is the simplest that can be imagined: the QNNetwork instance that have been used so far is stored into a temporary file and subsequently reloaded from that file into a temporary network structure; at that point, the two networks are simply compared to each other and an error is signaled if they are not identical.
There is an interesting detail about the error management technique used in this case: as I mentioned before, every class of the QtNets framework is provided with a proper implementation of the operator== standard C++ method, which allows to compare instances of the same complex class using the standard and intuitive "==" notation; in the specific case of the QNNetwork class, this comparison method analyzes every component of the two given networks, including nodes, edges and properties, which are in turn compared to each other exploiting the same principle. Anyway, since this is a testing utility also meant to help the developer finding the reason of errors, it cannot be enough to simply inform that the two networks are different; it is necessary to be more specific. Therefore, when an error occurs, the two network instances are explicitly re-compared in every aspect by the test procedure itself, until the exact difference, or at least the first one, is found. This allows the test application to display a very specific message about the location of the detected error.
At the end of the persistTest method, both the temporary network and file used during the test are deleted and the allocated memory freed.

The next set of test cases is dedicated to the validation of every aspect of the style subsystem offered by the QtNets framework; the following is the list of methods required for the purpose, in the same order they are executed:

- stylableTest;
- styleTest;
- stylePersistTest.

The first test case on the list, **stylableTest**, is designed to verify the correctness of the getStylablesByClass method of the QNNetworkModel class, which should return the list of objects to which are associated the given style classes; for the sake of completeness I recall its prototype:

```
QNStylableList getStylablesByClass(
        const QStringList& aClasses = QStringList(),
        const QNFilterStrategy& aStrategy = QNFilterStrategy::AND);
```

The approach used to implement this test case is equivalent to the one used for the nodePropertiesTest and edgePropertiesTest methods, therefore there is not much to add.
The **styleTest** slot, on the other side, is in charge of verifying the correctness of the QNStyleClass and QNStyle classes, focusing especially on their strict relation. The following is the strategy used to validate them: first a number of QNStyleClass instances is initialized with random values, determined through a proper algorithm to ensure they are always inside the range of valid values for each property of the class; then an instance of the QNStyle class is initialized using those classes and other random values for its identifier and version. The final step of the algorithm consists in extracting and validating id, version and all the style classes that have been previously attached to the current QNStyle instance. As usual, if an error is detected, the user is informed with a detailed message reporting all the available information about the cause of the error.
The last method designed to validate the style engine, **stylePersistTest**, checks whether load and store of a generic stylesheet work properly; similarly to the persistTest method already discussed, the QNStyle's instance, populated by the previous two test cases, is stored into a temporary file and immediately reloaded into a temporary memory location. The two style objects are then compared exploiting the operator== standard method and, if a difference is found, every property and attribute is double-checked to find the exact location of the divergence and inform the user.

The QtNetsTest application is also able to validate most of the features related to the user interface. However, some approximations are mandatory, given the fact that QtNetsTest is a console application with no GUI associated; those approximations will be exposed and discussed in detail during the analysis of the test cases requiring them.
The first GUI related test case is named **sceneTest** and has a dual objective: the first one is to initialize all the structures needed to simulate a scene containing

items and most of the supported user interactions, as they are needed also in the subsequent test cases; the second is to verify the correctness of the methods and procedures offered to populate a QtNets scene. In particular, the sceneTest method initializes an instance of the QNNetworkView class and of the QNNetworkScene one, then configures the context menu to be associated to the scene and all its elements, so to simulate user interactions; once all those basic initialization procedures are terminated, it populates the newly created scene using the QNNetwork instance that have been incrementally populated by all the previous test cases and checks if all the entities modeled by that instance are also represented in the scene.
Here comes the first mandatory approximation: since the scene is only simulated and not actually painted, all the items it contains, from nodes to edges, to even models and text boxes, are simulated as well, so they are equivalent to simple points; however, that is not a great limitation because all the information related to position and size of each element is still correctly updated, only not painted on the screen.
The sceneTest slot also checks the correct behavior of the actions provided to add new nodes, models, text boxes and edges by triggering the proper action in the scene's context menu; in this way it simulates what the user should do to add a new item to the scene in a realistic scenario. When the required element has been added, sceneTest checks if it has been correctly positioned in the scene and whether all the required properties are properly configured.

The **moveItemsTest** slot has the purpose of simulating and validating drag and drop of any item in the scene; even in this case, the typical interaction between the user and the scene required to move an item from its current position to another is simulated exploiting the instruments offered by the Qt Test environment. The results of the algorithms designed to manage all the external events involved are checked to guarantee they are coherent with the specifications.
All the possible scenarios have to be verified, especially those involving models, because they can have children elements; the most significant situations to be considered are:

- a node with no edges that is moved inside and outside a model;
- a node with edges that is moved inside and outside a model;
- a model without children that is moved inside and outside another model;
- a model having children that is moved inside and outside another model;
- a text box that is moved inside and outside a model;
- two nodes linked through an edge that are moved inside and outside a model;

- every supported item that is moved to a new position in the scene without intersecting any model.

Every item in the scene, except for edges, could be resized and scaled, therefore the **scaleTest** slot verifies that the size of every element is always updated correctly after every possible user interaction meant to change the height and width of an item.
The two kind of operations the user can perform to modify the size of an item are:

- proportionally scale the size of an item using a scale factor, like 2.0 to double the original size or 0.5 to halve it;
- freely modify the size of an item dragging one of its active corners.

The **deleteTest** method verifies the correct elimination of all the supported items from the scene, considering both the case in which they are deleted one by one and that in which they are removed all at once.
The delete operation is complex by definition, especially in scenarios in which elements might be related to each other and the elimination of one of them must be propagated to the others according to some predefined rules, as the one under analysis actually is; for this reason, there are many possible environmental conditions that have to be considered before deleting an element.
This test case has to verify all the situations in which a delete request can be raised; they can be summarized as follows:

- a node, a model or a text box without relations with other items is deleted;
- many nodes, models and text boxes without relations with other items are deleted;
- one or many edges between nodes are deleted;
- a node with associated edges is removed propagating the elimination to every edge in which it is involved as source or destination;
- a model having children is eliminated propagating the elimination to every item it contains;
- any combination of the previous cases.

The deleteTest slot invokes the delete procedure for each of the listed scenarios and verifies that no item is deleted by mistake or not deleted when it had to.

The **bringBackAndFrontTest** method is designed to verify the correctness of the bringToFront and sendToBack slots of the QNNetworkScene class, that manipulate the stack order of the items in the scene in order to bring to front or send to back the selected items. This test case just verifies that the actual stack order is correct after every invocation of each of the two slots.

The **showHideContentTest** and **showHideContentInModelTest** are very similar to each other and both checks the correctness of the content visibility management algorithms offered by the different items supported by this framework. The first one validates the show and hide primitive functions when applied to simple items, like nodes or models with no children; the second is specialized in the validation of such functions when applied to models with a complex hierarchy of children.
The reason for spitting the two test cases that seem so much similar is actually very simple: I did not mention it explicitly during the analysis of every test case, but each one of them executes some initialization steps to configure the environmental conditions in which the test should take place, as well as finalization procedures after the test is completed to restore the previous stable situation; since the environmental conditions required are quite different, I decided to split the test case into two distinct methods to emphasize the different behavior expected when the element to be shown or hidden is actually a complex hierarchy of items.
Both slots invoke the show and hide functions of the different kind of items and check whether their visibility status is correct after each invocation; in the specific case of models with child items, specifically managed by the second test slot, it is necessary to check also the content visibility status of all the contained items, because it is related to the one of the parent item.

The last test case of the list, **signalsTest**, is actually performed in parallel to all the other GUI related test cases and verifies the correct signal emission of the QNNetworkScene class. The QNNetworkScene class emits two different signals, modified and itemsSelected, in response to every action that determines any sort of change to the network or to the currently selected items; therefore, after a proper initialization performed in the first part of the sceneTest method, signalsTest intersects every signal emitted by the QNNetworkScene class during the execution of every other test case and, at the very end, compares the number of intersected signals with the expected one.
In this case, given the asynchronous nature of the signals and slots mechanism, it is only possible to inform the user about the fact that some signals have not been emitted, but there is no way to automatically discover which operation caused the error.

To conclude, one final note about this collection of unit tests: they are very important to automate and speed up the test phase of the QtNets framework but also as a fundamental development tool to be used during every future integration; as a matter of fact, they can, and should, be used whenever any new feature is added to guarantee that no regressions have been introduced. That is one of the most valuable advantages of using unit testing tools: once the initial effort required to write all the test cases is done, they can be used as long as the product lives to continuously validate it without additional costs.

5.2 QtNetsBenchmark

The QtNetsBenchmark application has been designed to evaluate the performance of the different modules of the QtNets framework. It could be considered in a sense dependent on QtNetsTest: given the fact that QtNetsBenchmark executes one after the other all the methods under analysis, disregarding any possible error they might encounter, it is highly recommended to run the performance evaluation only after all test cases have been passed, so to guarantee reliable results.
The benchmark application proposed simply initializes a completely random network and executes a series of methods and functions on it measuring the time they need to complete the specific task; all things considered, it is reasonable to say that QtNetsTest and QtNetsBenchmark share a very similar execution flow, with the only difference that the first one is focused on the correctness of the data structures and methods it analyzes, while the second considers only the time needed to complete the required actions.

The following is a snippet of the output produced by QtNetsBenchmark during the performance evaluation of the framework:

```
# **********************
# **********************
# ** Nodes: 70000
# ** Edges: 70000
# ** Models: 10000
# ** Textboxes: 10000
# ** Style classes: 10000
# **********************
# **
# ** network initialization: 2s 56ms
# ** store: 9s 111ms
# ** load: 15s 99ms
# ** extraction of all nodes (one by one): 37ms
# ** extraction of all nodes (by empty properties): 9ms
# ** extraction of all nodes with comment "something": 71ms
# ** extraction of all edges (one by one): 35ms
# ** extraction of all edges (by empty properties): 7ms
# ** extraction of all edges with comment "something": 26ms
# ** extraction of all icons (one by one): 51ms
# ** extraction of all icons (by empty properties): 11ms
# ** extraction of all icons with comment "something": 140ms
# ** extraction of all textboxes (one by one): 0ms
# ** extraction of all textboxes (by empty properties): 0ms
# ** extraction of all textboxes with comment "something": 8ms
# ** extraction of all edgeStyles (one by one): 35ms
# ** extraction of all edgeStyles (by empty properties): 7ms
# ** extraction of all edgeStyles with comment "something": 51ms
# ** extraction of all stylables (one by one): 22ms
# ** extraction of all stylables (by empty properties): 20ms
# ** extraction of all stylables with class "node": 52ms
# ** style classes initialization: 90ms
# ** store style: 324ms
# ** load style: 525ms
# **********************
# **********************
```

Figure 5.3: Output produced by QtNetsBenchmark

The first thing to be noticed is that the output of the performance evaluation is divided into different sections, each one of them introduced by the configuration of the currently tested network in terms of number of nodes, edges, models and text boxes; since the benchmark application also evaluate the style subsystem, the number of style classes used is also reported.
Figure 5.3 shows only one of the sections that form the final output, but the performance evaluation is conducted on different kind of networks, each one approximately one order of magnitude bigger than the previous, in order to provide a global picture of how execution times grow with the size of the network.
After the analysis of each section of the benchmark's output, reported as appendix for the sake of completeness, it is possible to state that the performances of the QtNets framework are linearly dependent on the size of the network, as the time required to perform each operation grows more or less with the same rate of the network's size.

The first operation that the QtNetsBenchmark application executes at the beginning of each section is the initialization of the random network to be used during the current performance evaluation; the configuration of that network is then printed out. Even if this operation might be considered just introductory for the real tests, I decided to report its execution time anyway, because the initialization procedure reveals some important pieces of information as well: for instance the time required to allocate memory space and initialize a given number of the different elements composing the network.
When the test environment has been set up, the first feature evaluated by the QtNetsBenchmark application is persistence management: first the random network is saved on a temporary file and the time required to do so is printed out, then the same network is reloaded from that file and the time required is printed out as well; at the end, the temporary file is deleted. The only comment to add is related to the time gap that exists between store and load: the second is generally almost two times slower than the first. This fact is not related to the specific operations performed by the QtNets framework, but rather to the characteristics of the persistent device used to store the data: as a matter of fact, both hard disks and solid state drives execute reading operations measurably faster than writing ones.

The next tests are dedicated to evaluate the time needed to extract a certain number of elements, of every kind, from the collections that respectively contain them. The reason why I focus great attention on this aspect, is that it represent the base of every other more sophisticated action that can be performed on a node, an edge, or everything else; in order to operate on items, first it is necessary to get access to them.
The QtNets framework provides different ways to get access to elements, each one

designed for a specific purpose and with its own characteristics; disregarding for a moment the obvious differences that exist between the supported elements, it is possible to summarize the basic procedures available to retrieve one or more elements as follows:

- all elements of the same kind can be retrieved at the same time getting access to the list or map structure that contains them; this action is by itself constant concerning time complexity, but it is actually not very interesting as is, because it requires the collection to be scanned in order to get access to every single element.
- A single element can be retrieved given its unique id or key.
- Many elements can be retrieved based on the fact they have or not a given combination of properties; if an empty set of properties is specified, this procedure is equivalent to the first one.

Analyzing in further details the performance evaluation's output shown in Figure 5.3, it is possible to notice that there are three extraction tests for each kind of element, but they do not exactly match the procedures listed above.
The first test extracts all the elements one by one and correspond to both the first two extraction strategies listed above; as a matter of fact, first it gets access to the whole collection of elements, then it extracts them one after the other using the key that uniquely identifies them inside the containing collection.
The remaining two tests are dedicated to evaluate the time needed to extract elements based on a list of properties: first, all elements are extracted passing an empty list of properties to the dedicated methods, then only those with a very specific comment property. In the first case, execution times are generally very short, because, once the method recognizes the fact that no properties to filter elements are given, it simply returns the whole collection, without any further computation; in the second case, a list of property is actually given, so the involved collection has to be scanned and every element it contains has to be checked in order to decide whether it satisfies the filtering rules or not. In this case, the time required to scan all the elements is generally higher due to the filtering procedures performed at every iteration, but the order of magnitude is always the same as expected.
In conclusion, considering all the elements involved in this performance evaluation, meaning nodes, edges, icons, text boxes and models, it is correct to state that all the procedures and function offered by the framework to retrieve elements of the managed network are linearly dependent on its size; the execution times of the different retrieving techniques just confirm that theory.

The last part of the execution flow of the QtNetsBenchmark application is dedicated to evaluate the persistence management of style sheets offered by the QtNetsStyle module. First of all a style object is initialized and populated with randomly defined classes; after that, the style object is stored on a temporary file and immediately reloaded from it, the exact same way it was for the evaluation of the network's persistence management. The relation between the execution times of the store and load methods is once again related to the characteristics of the memory device, therefore what have been said before perfectly applies in this case as well.

To conclude, it must be said that QtNetsBenchmark is very useful to analyze all the components and features of the QtNets framework that do not require a user interface, but, being a console application, it is practically useless to measure the time required by all the scene's update utilities; despite all of that, it is very important to give at least an overview of the performances this framework is capable to offer, because it evaluates the very basic features that deeply affects the performances of every other part of the library.

Chapter 6

The QtNetsEditor application

QtNets is a library and can be used to build applications through which it is possible to graphically edit complex networks; however, being a library, it is not executable as it is, therefore is useless until a traditional executable application is built upon it. For this reason, a very important part of this project is represented by a desktop application that actually uses the framework to implement a basic graphs editor: its name is **QtNetsEditor**.

The QtNetsEditor application is actually quite simple and offers no more than an intuitive interface through which the user can trigger the several actions offered by the underling QtNets framework: all the complexity related to the management of a network is in fact demanded to the framework. Nevertheless this application is of paramount importance: it offers a ready to use instrument to try all the features made available by the framework, or to study simple networks with no special requirements in terms of user interface; in other words, the QtNetsEditor application is perfect in all those situations in which the case to study is not particularly sophisticated and a general purpose tool is enough. This of course does not mean that QtNetsEditor is completely useless in more complicated scenarios; as a matter of fact, it also represents a valid example to start from in all those situations in which a particular user interface has to be designed to address the specific application field.

Technically speaking, QtNetsEditor extends the standard Qt class for traditional window-based desktop applications: **QMainWindow**; this class offers a base layout that can be exploited to implement several kinds of user interface, depending on the requirements and the characteristics of the application field.

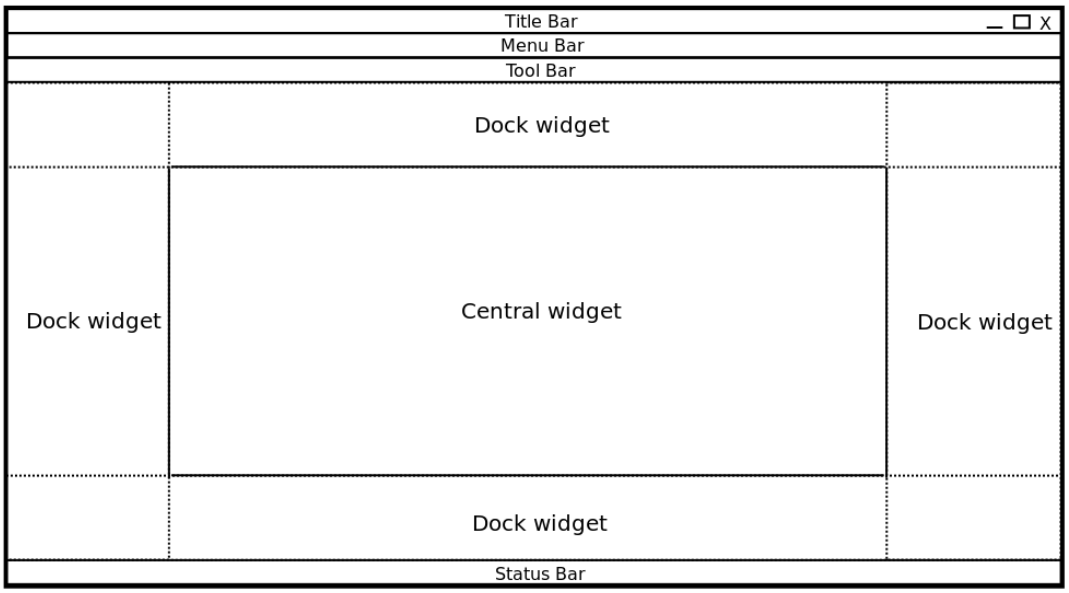


Figure 6.1: Layout configurations supported by QMainWindow

The title and status bars have similar roles: they both can be used to show pieces of information about the current status of the application, but, while the first one can show only a textual content, the second can be used to present data in more sophisticated forms, like through loading bars or small icons.
The menu bar can be exploited to group all the actions and commands the application supports in simple and intuitive drop down menus where the user can find everything he needs; the tool bar, in a sense, is strictly related to the menu one, since it is typically exploited to show the most frequently used commands, so that the user can invoke them directly. In other words, it could be considered as a sort of quick menu designed to ease the interaction.
The dock widgets are meant to host complex menus with sophisticated layouts and typically are used to implement tool palettes or configuration panels to facilitate the user in the interaction with the content of the main view; in a Qt based application, there are four available locations for a dock widget, as shown in Figure 6.1, and it is possible to define as many dock widgets as needed, as well as no one at all.
Finally the central widget represents the very core of the application, where the scene containing the actual workspace is typically hosted.

The QtNetsEditor application exploits only a subset of all the components offered by the base QMainWindow class to implement the desired user interface; basically it consists of the following components:

- a title bar;
- a menu bar;
- an item's dock;
- a main view;
- a style's dock.

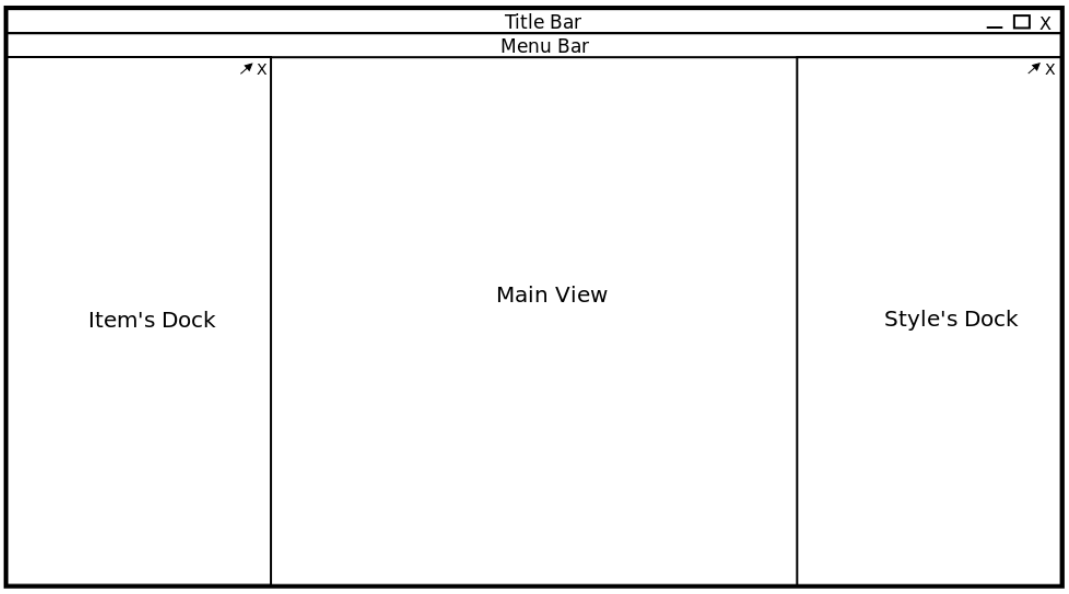


Figure 6.2: QtNetsEditor's layout

The title bar, besides containing the standard buttons to close, enlarge and reduce to icon, is designed to display the full path of the network currently opened in the editor; if the network is not loaded from an existing file, the title bar simply displays the name of the application, which is also the default behavior. In addition to that, the title bar reports another very useful information: it signals the existence of unsaved modifications of the currently loaded network through a star symbol (*) immediately after the name of the network; in this way, the user is always able to know whether his work is in a volatile state or not.

The menu bar is the place in which an application traditionally exposes all the provided functionalities, properly grouped based on their area of interest; in the case of the QtNetsEditor application, all the provided functions are divided into the following groups:

- File;
- View;
- Items;
- Style;
- Settings;
- Help.

The **File** group contains all the actions typically related to how data could be imported from a file to the application's workspace and exported the other way around; in particular, it exposes an action to **open** a network and load its content from a given file, with the opportunity to find it navigating the file system through a convenient widget.
Then it offers a specialized action to close the loaded network when the working session is completed; if the close action is invoked when the current network presents unsaved modifications, QtNetsEditor notifies that condition to the user asking whether to save the network before closing it or not. After the network has been successfully closed, the workspace is cleared and the user can start all over designing a new network from scratch or loading it from another file.
The QtNetsEditor application helps the user in all those situations in which he might have made a mistake and wants to roll back to the last stable situation: the reload action just have the purpose to reload the whole network from persistent memory and discard all the unsaved modifications.
It is quite obvious that the File group in the menu bar has to offer actions to save the current network onto a file: if the network is saved for the first time, the user must decide its location in the local file system, therefore a widget, very similar to the one described speaking of the open action, appears to help the user choose the location. If the network is already associated to a valid file, the user is offered the choice either to maintain the same file name or to change it using the **save** or **saveAs** action respectively.
It is also possible to export the currently loaded graph to an image using the standard and most supported formats, as PNG, JPEG and XPM. This function basically creates an image containing all the elements of the presented scene exploiting the combination of three very important Qt standard classes:

- QGraphicsScene;
- QPainter;

- QImage.

The **QGraphicsScene** class, that I remind is the base class for QNNetworkScene, offers a method named render, through which is possible to populate a paint device with its content; the paint device is modeled by the **QPainter** class, already mentioned as well speaking of item's rendering procedures, that in turn allows to actually render its content to a physical media. The **QImage** class is an example of physical media, as it represents generic images. Once an instance of the QImage class has been populated with the network contained in the QNNetworkScene's instance, it can be customized setting several properties, like the encoding or the background color, and then saved to be used as any other image file.
Another example of physical media that could be used as a destination of the QPainter class is the printer, modeled in the Qt environment by the **QPrinter** class. This class basically offers an interface layer that allows the application to directly use one of the printers globally configured for the host computer to print the content of the scene; moreover, it manages the case in which no printers are configured or supported by the hosting operative system, showing a detailed error pop-up to the user. All of that has been exploited to implement an action of the File group that enables the user to also print on paper the network he designed.
Finally, there is an action to close the whole application that is identified by the **exit** keyword; if this action is invoked while a network is open with unsaved modifications, the user is once again asked to save them before leaving.

The **View** group contains all the actions designed to customize the way in which the network is presented; none of the actions in this group alters the current state of the graph but only how the application displays it.
The first actions with those characteristics are related to the **zoom** level of the main view: as a matter of fact, there is an action to increase the zoom level, one to decrease it and one to restore the default zoom level.
The second common interaction to modify the presentation of the scene is related to the angle between the scene and the ground; in other words, to the possibility to **rotate** the scene. Also in this case, multiple actions are offered to manage that interaction and they allow to achieve the following results:

- slight rotation to the left;
- slight rotation to the right;
- 90 degrees rotation to the left;
- 90 degrees rotation to the right;

- default rotation restored.

It is worth to underline the fact that those actions does not directly implement the necessary procedures to manage zooming and rotation, but simply trigger the corresponding methods of the QNNetworkView class that manages those particular aspects.
Finally, the View group contains two actions to hide or show the item's dock and the style's dock respectively.

The next group of actions is probably the most important of all, as it contains the actions that allow the user to interact with the scene and modify it or its contained items.
The following are the actions contained in the **Items** group:

- **select All** to select all the items in the scene at once;
- **show/hide details** to toggle the content visibility of all the selected items;
- **bring to front** that brings all the selected items in front of all the others;
- **bring to back** that is the exact opposite of the previous one;
- **delete** to definitely remove all the selected items from the scene;
- **new node** that inserts a new node in the center of the scene;
- **new model** that inserts a new model in the center of the scene;
- **new text box** that inserts a new text box in the center of the scene;

The last four actions are also available in the context menu associated to the scene, with the fundamental difference that, if invoked through the right click of the mouse, those actions will add the new item in the exact position the mouse click took place.
To make the application more intuitive and easy to use, the actions that can be invoked exclusively from the menu bar, also belonging to other groups, have been associated to keyboard shortcuts, respecting the common conventions of other very popular and widely used applications; for example, the open action of the File group is associated to the standard *CTRL+O* keyboard shortcut, the save action is associated to *CTRL+S* and the delete action of the Item group is invoked simply pressing the *DEL* key.

The **Style** group contains all the actions related to the possibility to customize the current network using an external plug-in; in the case of the QtNetsEditor application, the only style plug-in available is the BaseStylePlugin one but the structure

of the application is general enough to manage also other plug-ins.
The actions available in this group are the following:

- open style;
- reload style;
- save style;
- saveAs style;
- close style.

It is quite clear the similarity between those actions a the equivalent ones in the File group: not only their names are very similar, but also their purpose; the only difference is that the actions included in the Style group work with style sheets, while the previous ones are dedicated to networks.

The **Settings** group contains at the moment only one action, which is named "**do not ask for confirm on delete**"; since it is possible to select more than one item in the scene and delete them simply pressing the *DEL* key on the keyboard, it would be annoying for the user if he mistakenly pressed the delete key while many items were selected because all his work would be lost. For this reason, QtNetsEditor refuses to directly delete multiple items all at once and always asks a confirm to the user. This procedure applies also when the item to be deleted is just one but it is a model with at least one child.
If the user finds the confirm request annoying, he has the possibility to accept the risk and disable that functionality through the specific action under the Settings voice.

The last group of actions in the menu bar is **Help** and contains just one action named "**about**". The only task associated to this action is to show a small information pop-up with a brief description of the purpose of the application.

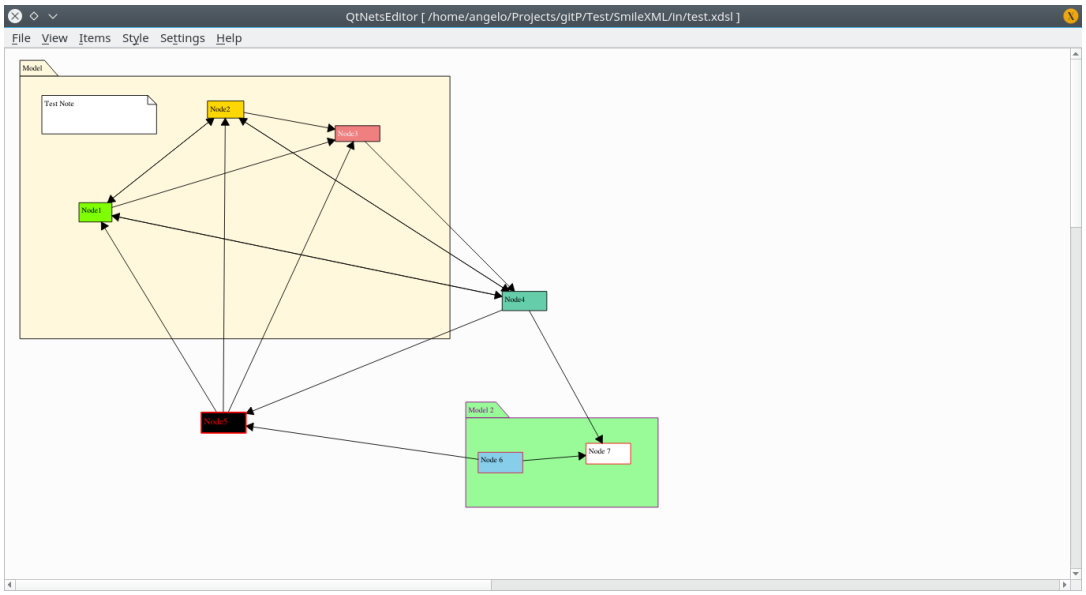


Figure 6.3: QtNetsEditor's base user interface

Figure 6.3 shows how the QtNetsEditor application looks like when a simple network is loaded, but omits a very important part of the user interface: the dock widgets. As a matter of fact, QtNetsEditor offers two dock widgets: the first, on the left side, is designed to allow the user to easily modify the properties of the elements in the scene, while the second, on the right side, is designed to help the user in the customization of network's style.
Both the dock widgets are specializations of the **QDockWidget** class of the standard Qt distribution; it basically provides a widget that can be docked inside a QMainWindow or floated as a top-level window on the desktop. In addition to that, dock windows can be stacked in their current area, moved to new ones and even floated by the end-user; however, the QDockWidget"s API allows the programmer to restrict their ability to move, float and close, as well as the portions of the dock area around the main widget in which they can be placed.
As far as appearance is concerned, a QDockWidget roughly consists of a title bar and a body; the title bar includes the title assigned to the dock window, a float button, through which the user can undock the widget, and a close button. Depending on the state of the QDockWidget, the float and close buttons may be either disabled or not shown at all. Lastly, the visual appearance of the title bar and its buttons depends on the global style currently in use on the host computer.
A QDockWidget's instance acts as a wrapper for its child widgets, therefore custom size hints, limits in terms of size and policies should be implemented by the children in order to have the QDockWidget container respect them and adjust its own constraints to include frame and title. Furthermore, size constraints should not be set

on the QDockWidget itself, because they change depending on whether it is docked or not: for instance, a docked QDockWidget has no frame and a smaller title bar, while a floating one has no limitations and can even equals the size of the whole screen.

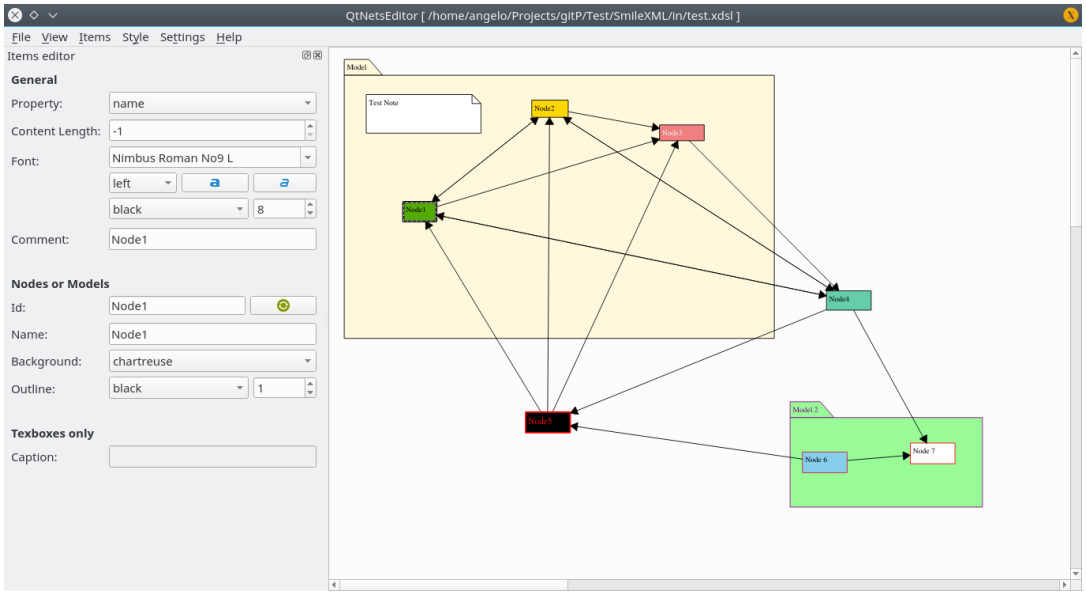


Figure 6.4: QtNetsEditor showing item's dock widget

The item's dock on the left side allows to modify the properties and attributes of the selected item in the scene. In order to recognize which and how many elements are currently selected, it intercepts the itemsSelected signal emitted by the scene and populate the all the contained widgets with the data extracted from the selected items.
Observing Figure 6.4, it is possible to notice that the active elements of the layout are organized in rows, where the first part is a description of the widget, or widgets, on the right. Those rows are divided into three groups:

- general;
- nodes and models;
- textboxes only.

This separation is meant to underline the fact that some properties could be associated to every kind of item while other are more specific; as a matter of fact, in the example showed in Figure 6.4 a single node is selected and all the elements in the "textboxes only" section are disabled.

A very similar behavior can be noticed in those situations in which many different items are selected simultaneously: only the properties that are shared among all items can be modified, while the others are disabled, so to avoid undesired modifications; for example, if ten nodes and five models are selected together, all the properties under the first two groups would be editable except for the id of the item, since that property is unique by definition and cannot be shared among different items. If also some text boxes were selected, then the set of shared properties would be further reduced to only those labeled as general and only the elements in the first group would be actually enabled.

The layout of the item's dock is designed to be as simple as possible, therefore it is not necessary to explicitly describe every single element it contains, because most of the times the purpose is quite clear; for this reason, I am going to focus the attention only on the most important components.
The first row of the general group is the one that determines the textual content of the different elements in the scene that actually support this feature; when an element is selected, the combo box on the right is populated with the names of all the textual properties associated to that element, so the user can choose which one of them wants to see displayed inside the body of the item. In the example in Figure 6.4, the chosen property is the name of the item, so the selected node shows its name inside its bounding rectangle.
The following row is also related to the content of each item in the scene, as it defines the number of characters to be actually shown: if the property to be displayed inside the item is longer than the specified number of characters, it is truncated and an ellipsis is postponed; if, as in Figure 6.4, the specified number of characters is negative, then the full property is displayed inside the limits of the item, no matter its length.
The meaning of the remaining elements of the general section is quite straightforward, so I am going to move directly to the first row of the following section, the one meant to edit the unique identifier associated to each node and model. Given the nature of the attribute it manages, this row is very important, therefore an error could compromise the integrity of the whole network; in particular the uniqueness of every identifier inside the network has to be guaranteed, so the user must be stopped and properly warned whenever he tries to use the same value to identify more than one item. However, since assigning unique values can be complicated or tedious, it is possible to demand the generation of a random identifier to the QtNetsEditor application using the small button on the right end of the row.

The following figure shows the style editor dock on the right side of the user interface;

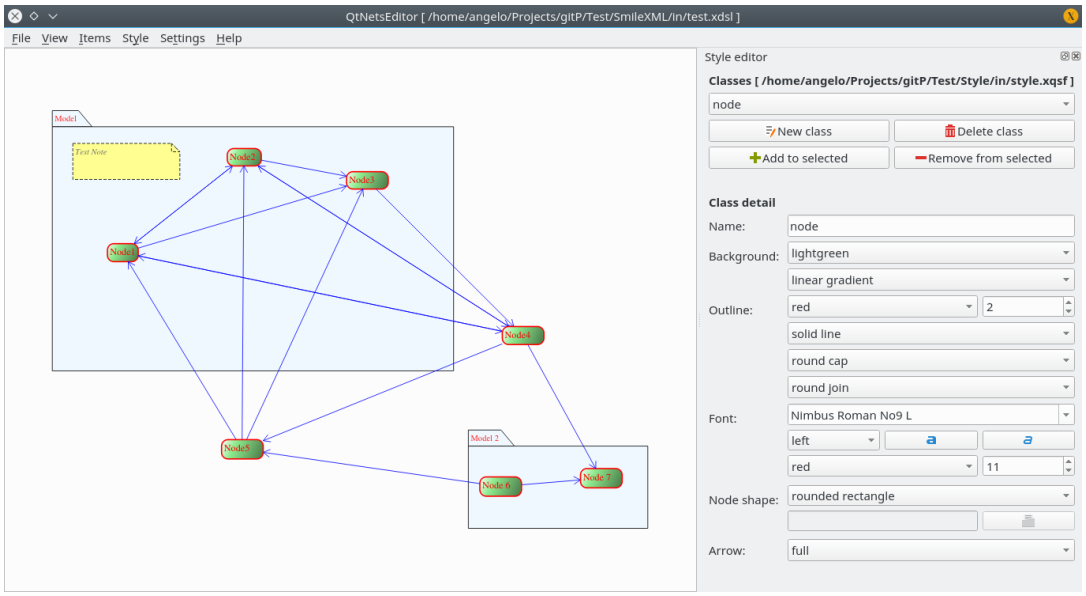


Figure 6.5: QtNetsEditor showing style's dock widget

It is designed to allow the user to simply generate a style sheet and apply it to the currently loaded network; in addition to that, once the style sheet has been saved, it can be used again associated to other networks, thanks to the generality of the style sheets mechanism.
This dock is also divided into two sections: one contains all the buttons to manage the style sheet, while the other shows all the detail of the selected style class and offers the possibility to modify its attributes. The first section, simply named **Classes**, contains a combo box that lists all the style classes defined for the current style instance and a series of buttons to manage it; the actions allowed in the specific case are:

- create a new class;
- delete an existing class;
- attach a style class to all the selected items in the scene;
- remove the association to a class from all the selected items in the scene.

As it can be noticed in Figure 6.5, the label that identifies the first section of the style editor dock contains an extra piece of information: if the current style instance

is loaded from a persistent file, then its full path is reported inside square brackets; the presence of unsaved modifications is signaled using a star symbol after the name of the file, according to the exact same strategy used to manage the QtNetsEditor's title bar.
On the other hand, the **Class detail** section is dedicated to the inspection of a generic style class and allows to modify all of its properties; furthermore, whenever a property of a style class attached to at least one item in the scene is modified, that is immediately applied on the scene with no need to reload, neither the network nor the style.
With that being said, it is easy understand the meaning of the rows in the class detail section, since they are made following the same logic used to design the item's dock: each row is made of a label to describe the property, or category of properties, that is possible to edit through the widgets on its right. In order to further facilitate the user, when the range of values a certain property can assume is clearly defined, the widget proposed is a combo box populated with all the possible values for that property; in this way it is almost impossible to poorly configure a style sheet and it is guaranteed that every property assumes a valid value.
To conclude the analysis of the two dock widgets, it is worth to spend few words about their position constraints: both the item's and the style's docks can only be located either on the left or right side of the QtNetsEditor application, but it is also possible to locate them on the same side. In that case, only one dock window at a time is visible and the user can switch between them using a navigation bar purposely added at the very bottom.

The real core of the QtNetsEditor application is the central widget: as a matter of fact, it contains an instance of the QNNetworkView class which in turn includes the scene modeled by the QNNetworkScene class. In other words, the central widget represents a sort of canvas where the user can draw his network and interact with it exploiting all the interaction patterns that have been analyzed in detail in the previous chapters.

Chapter 7

Achieved results and future improvements

The results obtained with this project are, all things considered, very satisfying; first of all at the human level, since this project represented to me the first time in which I really had the chance to develop a complex project starting from nothing and face most of the critical moments that characterize, more or less, any software development process. All of that represents to me a precious training for what I will face soon in my professional career.
The results obtained at the technical level are no less; after a great deal of work and effort, I managed to build a framework that is enough dynamic and abstract to be effectively exploited in many different application scenarios, but also easy to understand and use. The QtNets framework is also quite stable, enough to allow anybody to work safely on large networks, with hundreds, maybe thousands, of items; in addition to that, performances are good enough to make the navigation between the different elements in the scene always fluid.

Results like these obviously lead to great satisfaction and pride, but represent also a motivation to do something more, and in this case much can still be done.
The first area that definitely demands to be improved is the persistence management of a network: at the moment, the QtNets framework comes with only one persistence plug-in module to actually move network's data in and out a persistent memory device, the SmilePersistencyPlugin. However, this module implements all the functions and algorithms required to manage a storage format that has been designed by another team of developers to address a very specific application field; for this reason, a great number of the structures and functionalities it offers are useless in many cases, given the generalist nature of the library.
I was absolutely aware of those limitations when I decided to follow this path, but

I decided to take it anyway in order to guarantee some interoperability with other more popular tools, at least in the early stages of life of the product; on the other hand, this is not the final and stable situation I planned at the beginning: all the possible plug-in modules that can be developed by third parties, like the SmilePersistencyPlugin one, have to be considered as bridges to other storage formats, while the QtNets framework proposes its own custom one. In this way, it is possible to save a considerable amount of memory space whenever a network structure needs to be stored, because only its useful and necessary attributes would be actually memorized.
Clearly, this task is not as simple as it might seems, for it requires a huge design effort to guarantee that the final storage format is compatible with the plug-in based principles that characterize the QtNets's persistence system.

The second improvement that could be made is related to the edge's rendering procedure implemented by the QNEdgeItem class of the QtNetsDraw module.
The current implementation only allows straight edges, which connect the center of the source node to the center of the destination one; while this is generally not a problem when the network to be studied is small, it may lead to inconvenient overlapping of elements when the scene is particularly crowded. Therefore, the simplest, but at the same time most effective, strategy to improve that situation could be to extend the QNEdgeItem class to make it support zig-zag lines too; in this way, the proposed edge between two nodes would remain the shortest path between them, as it is now, but the user would have the possibility to change the recommended route for the edge simply adding one or more intermediate points between its two ends. In other words, the user would have the possibility to draw itself the path between nodes.

The QNNetworkView class, the one that provides the view structure, is designed to support only one scene at a time; clearly this is the most basic, and perhaps the most common, working situation for a framework like this, since most of the times the user is likely to be focused on a single network.
However, extending the view class so it can support multiple scenes, and therefore multiple networks, might represent a great improvement of the whole user experience, as it might lead to new forms of communication between different networks; for instance, it would be possible to move elements, such as nodes or text boxes, from one network to another, or even use multiple network instances to split a very complex graph into many smaller and related networks.
As already said, one of the most useful characteristic of models is that they can have children elements, therefore can be used to represent subnetworks, or at least groups of elements sharing properties and meaning; at this moment, all the elements

contained in a model are still part of the same network, but if the rest of the environment supported multiple scenes, they could be moved in a completely separated network instance capable to live its own life. In this way, much more complicated cases of study could be easily addressed using this very framework, because the complexity of the matter could be spit into several smaller and connected pieces.

The next improvement I am about to propose is the one I consider an authentic breakthrough for the entire project.
A plug-in based development approach has already been used to design some components of the QtNets framework, but so far it has been limited only to some specific duties. Just to make it clear, a persistence plug-in can only implement a specific interface and is not allowed to provide any additional feature, since nothing but the functions registered in that interface can be invoked inside the QtNets runtime environment.
It is reasonable to think that this plug-in based architecture can be made much more general than it is now, up to the moment in which any generic plug-in module is supported; however, to reach this ambitious goal, it is necessary to design an interface general enough to allow any sort of algorithm or feature to be executed and a series of strict rules to guarantee a correct and safe interaction between the QtNets environment and the plug-in module. Once again, this might seem an easy activity but it actually hides many traps; in fact, the more complex is the system, in this case the QtNets framework, the highest is the number of factors that have to be considered in order to design a generic entry point that could be used in thousands different ways, some of them not even imagined yet.
Anyways, the return of investment in this case would be incredible: the framework would become universal, meaning that anyone, all around the world, would have the possibility to easily build his own plug-in to solve a specific problem; if then a web portal, designed to share such plug-ins, is also developed, it would make it possible that a specialized community grows and the QtNets framework becomes the base on which anybody can build his own modules, or exploits some of those developed by other members, to solve concrete problems.

Bibliography

[1] Cytoscape, http://www.cytoscape.org/what_is_cytoscape.html

[2] GeNIe, https://www.bayesfusion.com/genie-modeler

[3] SMILE, https://www.bayesfusion.com/smile-engine

[4] Graphviz, http://www.graphviz.org/About.php

[5] xDot, https://github.com/jrfonseca/xdot.py

[6] StarUML, http://staruml.io

[7] Qt Wiki, http://wiki.qt.io/About_Qt

[8] Qt Creator, https://www.qt.io/ide

[9] Git, https://git-scm.com/about

[10] SmartGit, http://www.syntevo.com/smartgit

[11] Valgrind, http://valgrind.org

[12] Valgrind - Wikipedia,
https://en.wikipedia.org/wiki/Valgrind

[13] STL string class reference,
http://www.cplusplus.com/reference/string/string

[14] QString class reference, http://doc.qt.io/qt-5/qstring.html

[15] STL map class reference,
http://www.cplusplus.com/reference/map/map/?kw=map

[16] QMap class reference, http://doc.qt.io/qt-5/qmap.html

[17] QVariant class reference, http://doc.qt.io/qt-5/qvariant.html

[18] Qt's meta-objects, http://doc.qt.io/qt-5/metaobjects.html

[19] Qt's meta-type, http://doc.qt.io/qt-5/qmetatype.html

[20] Graphics View Framework, http://doc.qt.io/qt-5/graphicsview.html

[21] QGraphicsView class reference,
http://doc.qt.io/qt-5/qgraphicsview.html

[22] QGraphicsScene class reference,
http://doc.qt.io/qt-5/qgraphicsscene.html

[23] QGraphicsItem class reference,
http://doc.qt.io/qt-5/qgraphicsitem.html

[24] QMenu class reference, http://doc.qt.io/qt-5/qmenu.html

[25] QAction class reference, http://doc.qt.io/qt-5/qaction.html

[26] Qt's signals and slots, http://doc.qt.io/qt-5/signalsandslots.html

[27] QGraphicsTextItem class reference,
http://doc.qt.io/qt-5/qgraphicstextitem.html

[28] QPainter class reference, http://doc.qt.io/qt-5/qpainter.html

[29] QStyleOptionGraphicsItem class reference,
http://doc.qt.io/qt-5/qstyleoptiongraphicsitem.html

[30] QGraphicsLineItem class reference,
http://doc.qt.io/qt-5/qgraphicslineitem.html

[31] Qt's plug-ins reference, http://doc.qt.io/qt-5/plugins-howto.html

[32] QPluginLoader class reference,
http://doc.qt.io/qt-5/qpluginloader.html

[33] QXmlStreamReader class reference,
http://doc.qt.io/qt-5/qxmlstreamreader.html

[34] QXmlStreamWriter class reference,
http://doc.qt.io/qt-5/qxmlstreamwriter.html

[35] QIODevice class reference, http://doc.qt.io/qt-5/qiodevice.html

[36] CSS introduction, https://www.w3schools.com/css/css_intro.asp

[37] BrushStyle reference, http://doc.qt.io/qt-5/qt.html#BrushStyle-enum

[38] QLinearGradient class reference,
http://doc.qt.io/qt-5/qlineargradient.html

[39] QRadialGradient class reference,
http://doc.qt.io/qt-5/qradialgradient.html

[40] PenStyle reference,
http://doc.qt.io/qt-5/qt.html#PenStyle-enum

[41] PenCapStyle reference,
http://doc.qt.io/qt-5/qt.html#PenCapStyle-enum

[42] PenJoinStyle reference,
http://doc.qt.io/qt-5/qt.html#PenJoinStyle-enum

[43] Unit Testing Overview,
https://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx

[44] Qt Test Overview, http://doc.qt.io/qt-5/qtest-overview.html

[45] QMainWindow class reference, http://doc.qt.io/qt-5/qmainwindow.html

[46] QImage class reference, http://doc.qt.io/qt-5/qimage.html

[47] QPrinter class reference, http://doc.qt.io/qt-5/qprinter.html

[48] QDockWidget class reference, http://doc.qt.io/qt-5/qdockwidget.html

Appendices

Appendix A

SMILE schema file

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:include schemaLocation="genie.xsd" />
5
6  <xs:element name="smile">
7    <xs:complexType>
8      <xs:sequence>
9        <xs:element ref="properties" minOccurs="0" maxOccurs="1" />
10       <xs:element ref="algparams" minOccurs="0" maxOccurs="1" />
11       <xs:element ref="nodes" minOccurs="0" maxOccurs="1" />
12       <xs:element ref="observationcost" minOccurs="0"
             maxOccurs="1" />
13       <xs:element ref="extensions" minOccurs="0" maxOccurs="1" />
14     </xs:sequence>
15     <xs:attribute name="version" use="required" />
16     <xs:attribute name="id" type="xs:token" use="required" />
17     <xs:attribute name="numsamples" type="xs:integer"
           default="1000" />
18   </xs:complexType>
19 </xs:element>
20
21 <xs:attributeGroup name="nodeAttributes">
22   <xs:attribute name="id" type="xs:token" use="required" />
23   <xs:attribute name="target" type="xs:boolean" use="optional" />
```

```
24    <xs:attribute name="ranked" type="xs:boolean" use="optional"
         default="false" />
25    <xs:attribute name="mandatory" type="xs:boolean" use="optional"
         default="false" />
26    <xs:attribute name="diagtype" default="auxiliary"
         use="optional">
27      <xs:simpleType>
28        <xs:restriction base="xs:string">
29          <xs:enumeration value="target" />
30          <xs:enumeration value="observation" />
31          <xs:enumeration value="auxiliary" />
32        </xs:restriction>
33      </xs:simpleType>
34    </xs:attribute>
35  </xs:attributeGroup>
36
37  <xs:attributeGroup name="stateAttributes">
38    <xs:attribute name="id" type="xs:token" use="required" />
39    <xs:attribute name="label" type="xs:token" use="optional" />
40    <xs:attribute name="target" type="xs:boolean" use="optional" />
41    <xs:attribute name="default" type="xs:boolean" use="optional" />
42    <xs:attribute name="fault" type="xs:boolean" use="optional" />
43  </xs:attributeGroup>
44
45  <xs:element name="property">
46    <xs:complexType>
47      <xs:simpleContent>
48        <xs:extension base="xs:string">
49          <xs:attribute name="id" type="xs:token" use="required" />
50        </xs:extension>
51      </xs:simpleContent>
52    </xs:complexType>
53  </xs:element>
54
55  <xs:simpleType name="intList">
56    <xs:list itemType="xs:integer" />
57  </xs:simpleType>
58
59  <xs:simpleType name="doubleList">
```

```
60    <xs:list itemType="xs:double" />
61  </xs:simpleType>
62
63  <xs:simpleType name="idList">
64    <xs:list itemType="xs:token" />
65  </xs:simpleType>
66
67  <xs:element name="state">
68    <xs:complexType>
69      <xs:attributeGroup ref="stateAttributes" />
70    </xs:complexType>
71  </xs:element>
72
73  <xs:element name="parents" type="idList" />
74
75  <xs:element name="algparams">
76    <xs:complexType>
77      <xs:all>
78        <xs:element name="epis" minOccurs="0">
79          <xs:complexType>
80            <xs:attribute name="proplen" type="xs:integer"
                 use="required" />
81            <xs:attribute name="numstates1" type="xs:integer"
                 use="required" />
82            <xs:attribute name="numstates2" type="xs:integer"
                 use="required" />
83            <xs:attribute name="numstates3" type="xs:integer"
                 use="required" />
84            <xs:attribute name="eps1" type="xs:double"
                 use="required" />
85            <xs:attribute name="eps2" type="xs:double"
                 use="required" />
86            <xs:attribute name="eps3" type="xs:double"
                 use="required" />
87            <xs:attribute name="eps4" type="xs:double"
                 use="required" />
88          </xs:complexType>
89        </xs:element>
90      </xs:all>
```

```
91    </xs:complexType>
92  </xs:element>
93
94  <xs:element name="observationcost">
95    <xs:complexType>
96      <xs:sequence>
97        <xs:element name="node" minOccurs="0" maxOccurs="unbounded">
98          <xs:complexType>
99            <xs:sequence>
100             <xs:element ref="parents" minOccurs="0" maxOccurs="1"
                    />
101             <xs:element name="cost" type="doubleList" />
102           </xs:sequence>
103           <xs:attribute name="id" type="xs:token" use="required" />
104         </xs:complexType>
105       </xs:element>
106     </xs:sequence>
107   </xs:complexType>
108 </xs:element>
109
110 <xs:element name="properties">
111   <xs:complexType>
112     <xs:sequence>
113       <xs:element ref="property" minOccurs="0"
             maxOccurs="unbounded" />
114     </xs:sequence>
115   </xs:complexType>
116 </xs:element>
117
118 <xs:element name="nodes">
119   <xs:complexType>
120     <xs:choice minOccurs="0" maxOccurs="unbounded" >
121       <xs:element ref="cpt" />
122       <xs:element ref="noisymax" />
123       <xs:element ref="noisyadder" />
124       <xs:element ref="deterministic" />
125       <xs:element ref="decision" />
126       <xs:element ref="utility" />
127       <xs:element ref="mau" />
```

```
128     </xs:choice>
129   </xs:complexType>
130 </xs:element>
131
132 <xs:element name="cpt">
133   <xs:complexType>
134     <xs:sequence>
135       <xs:element ref="state" minOccurs="2" maxOccurs="unbounded"
            />
136       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
137       <xs:element name="probabilities" type="doubleList" />
138       <xs:element ref="property" minOccurs="0"
            maxOccurs="unbounded" />
139     </xs:sequence>
140     <xs:attributeGroup ref="nodeAttributes" />
141   </xs:complexType>
142 </xs:element>
143
144 <xs:element name="noisymax">
145   <xs:complexType>
146     <xs:sequence>
147       <xs:element ref="state" minOccurs="2" maxOccurs="unbounded"
            />
148       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
149       <xs:element name="strengths" type="intList" minOccurs="0"
            maxOccurs="1" />
150       <xs:element name="parameters" type="doubleList" />
151       <xs:element ref="property" minOccurs="0"
            maxOccurs="unbounded" />
152     </xs:sequence>
153     <xs:attributeGroup ref="nodeAttributes" />
154   </xs:complexType>
155 </xs:element>
156
157 <xs:element name="noisyadder">
158   <xs:complexType>
159     <xs:sequence>
160       <xs:element ref="state" minOccurs="2" maxOccurs="unbounded"
            />
```

```
161       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
162       <xs:element name="dstates" type="intList" />
163       <xs:element name="weights" type="doubleList" />
164       <xs:element name="parameters" type="doubleList" />
165       <xs:element ref="property" minOccurs="0"
             maxOccurs="unbounded" />
166     </xs:sequence>
167     <xs:attributeGroup ref="nodeAttributes" />
168   </xs:complexType>
169 </xs:element>
170
171 <xs:element name="deterministic">
172   <xs:complexType>
173     <xs:sequence>
174       <xs:element ref="state" minOccurs="2" maxOccurs="unbounded"
             />
175       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
176       <xs:element name="resultingstates" type="idList" />
177       <xs:element ref="property" minOccurs="0"
             maxOccurs="unbounded" />
178     </xs:sequence>
179     <xs:attributeGroup ref="nodeAttributes" />
180   </xs:complexType>
181 </xs:element>
182
183 <xs:element name="decision">
184   <xs:complexType>
185     <xs:sequence>
186       <xs:element ref="state" minOccurs="2" maxOccurs="unbounded"
             />
187       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
188       <xs:element ref="property" minOccurs="0"
             maxOccurs="unbounded" />
189     </xs:sequence>
190     <xs:attributeGroup ref="nodeAttributes" />
191   </xs:complexType>
192 </xs:element>
193
194 <xs:element name="utility">
```

```
195   <xs:complexType>
196     <xs:sequence>
197       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
198       <xs:element name="utilities" type="doubleList" />
199       <xs:element ref="property" minOccurs="0"
              maxOccurs="unbounded" />
200     </xs:sequence>
201     <xs:attributeGroup ref="nodeAttributes" />
202   </xs:complexType>
203 </xs:element>
204
205 <xs:element name="mau">
206   <xs:complexType>
207     <xs:sequence>
208       <xs:element ref="parents" minOccurs="0" maxOccurs="1" />
209       <xs:element name="weights" type="doubleList" />
210       <xs:element ref="property" minOccurs="0"
              maxOccurs="unbounded" />
211     </xs:sequence>
212     <xs:attributeGroup ref="nodeAttributes" />
213   </xs:complexType>
214 </xs:element>
215
216 </xs:schema>
```

Appendix B

GeNIe schema file

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:element name="extensions_any">
5    <xs:complexType mixed="true">
6      <xs:sequence>
7        <xs:any namespace="##any" processContents="skip"
           minOccurs="0" maxOccurs="unbounded" />
8      </xs:sequence>
9    </xs:complexType>
10 </xs:element>
11
12 <xs:element name="extensions">
13   <xs:complexType>
14     <xs:sequence>
15       <xs:element name="genie">
16         <xs:complexType>
17           <xs:sequence>
18             <xs:element name="comment" type="xs:string"
                 minOccurs="0" maxOccurs="1" />
19             <xs:element name="diagautoformat" minOccurs="0"
                 maxOccurs="1">
20               <xs:complexType>
21                 <xs:attribute name="target" type="colorType"
                     use="required" />
```

```
22              <xs:attribute name="targetnr" type="colorType"
                   use="required" />
23              <xs:attribute name="observation" type="colorType"
                   use="required" />
24              <xs:attribute name="observationnr" type="colorType"
                   use="required" />
25              <xs:attribute name="auxiliary" type="colorType"
                   use="required" />
26            </xs:complexType>
27          </xs:element>
28          <xs:choice minOccurs="0" maxOccurs="unbounded">
29            <xs:element ref="node" />
30            <xs:element ref="submodel" />
31            <xs:element ref="textbox" />
32          </xs:choice>
33          <xs:element name="arccomment" minOccurs="0"
               maxOccurs="unbounded">
34            <xs:complexType mixed="true">
35              <xs:attribute name="parent" type="xs:token"
                   use="required" />
36              <xs:attribute name="child" type="xs:token"
                   use="required" />
37              <xs:attribute name="cost" type="xs:boolean"
                   use="optional" default="false" />
38            </xs:complexType>
39          </xs:element>
40        </xs:sequence>
41        <xs:attribute name="version" type="xs:string"
             use="required" />
42        <xs:attribute name="name" type="xs:string"
             use="required" />
43        <xs:attribute ref="faultnameformat" use="optional"
             default="user" />
44      </xs:complexType>
45    </xs:element>
46  </xs:sequence>
47 </xs:complexType>
48 </xs:element>
49
```

```
50 <xs:simpleType name="colorType">
51   <xs:restriction base="xs:string">
52     <xs:pattern value="[0-9|a-f|A-F]{6}" />
53   </xs:restriction>
54 </xs:simpleType>
55 <xs:attribute name="color" type="colorType" />
56
57 <xs:complexType name="fontType">
58   <xs:attribute ref="color" use="required" />
59   <xs:attribute name="name" type="xs:string" use="required" />
60   <xs:attribute name="size" type="xs:integer" use="required" />
61   <xs:attribute name="bold" type="xs:boolean" use="optional"
        default="false" />
62   <xs:attribute name="italic" type="xs:boolean" use="optional"
        default="false" />
63 </xs:complexType>
64 <xs:element name="font" type="fontType" />
65
66 <xs:attribute name="faultnameformat">
67   <xs:simpleType>
68     <xs:restriction base="xs:string">
69       <xs:enumeration value="node" />
70       <xs:enumeration value="nodestate" />
71       <xs:enumeration value="user" />
72       <xs:enumeration value="inherit" />
73     </xs:restriction>
74   </xs:simpleType>
75 </xs:attribute>
76
77 <xs:complexType name="alignedfont">
78   <xs:complexContent>
79     <xs:extension base="fontType">
80       <xs:attribute name="align" use="optional" default="left">
81         <xs:simpleType>
82           <xs:restriction base="xs:string">
83             <xs:enumeration value="left" />
84             <xs:enumeration value="right" />
85             <xs:enumeration value="center" />
86           </xs:restriction>
```

```
87          </xs:simpleType>
88        </xs:attribute>
89      </xs:extension>
90    </xs:complexContent>
91  </xs:complexType>
92
93  <xs:complexType name="tableComment" mixed="true">
94    <xs:attribute name="col" type="xs:integer" use="required" />
95    <xs:attribute name="row" type="xs:integer" use="required" />
96  </xs:complexType>
97
98  <xs:element name="link">
99    <xs:complexType>
100     <xs:attribute name="title" type="xs:string" use="required" />
101     <xs:attribute name="path" type="xs:string" use="required" />
102   </xs:complexType>
103 </xs:element>
104
105 <xs:simpleType name="position">
106   <xs:restriction base="intList">
107     <xs:length value="4" />
108   </xs:restriction>
109 </xs:simpleType>
110 <xs:element name="position" type="position" />
111
112 <xs:complexType name="icon">
113   <xs:sequence>
114     <xs:element name="name" type="xs:string" />
115     <xs:element name="interior">
116       <xs:complexType>
117         <xs:attribute ref="color" use="required" />
118       </xs:complexType>
119     </xs:element>
120     <xs:element name="outline">
121       <xs:complexType>
122         <xs:attribute ref="color" use="required" />
123         <xs:attribute name="width" type="xs:integer"
              use="optional" default="1" />
124       </xs:complexType>
```

```
125     </xs:element>
126     <xs:element ref="font" />
127     <xs:element ref="position" />
128     <xs:element name="comment" type="xs:string" minOccurs="0"
           maxOccurs="1" />
129   </xs:sequence>
130   <xs:attribute name="id" type="xs:token" use="required" />
131 </xs:complexType>
132
133 <xs:element name="textbox">
134   <xs:complexType>
135     <xs:sequence>
136       <xs:element name="caption" type="xs:string" />
137       <xs:element name="font" type="alignedfont" />
138       <xs:element ref="position" />
139       <xs:element name="comment" type="xs:string" minOccurs="0"
             maxOccurs="1" />
140     </xs:sequence>
141   </xs:complexType>
142 </xs:element>
143
144 <xs:element name="node">
145   <xs:complexType>
146     <xs:complexContent>
147       <xs:extension base="icon">
148         <xs:sequence>
149           <xs:element name="state" minOccurs="0"
                maxOccurs="unbounded">
150             <xs:complexType>
151               <xs:sequence>
152                 <xs:element name="fix" type="xs:string"
                       minOccurs="0" maxOccurs="1" />
153                 <xs:element name="comment" type="xs:string"
                       minOccurs="0" maxOccurs="1" />
154                 <xs:element ref="link" minOccurs="0"
                       maxOccurs="unbounded" />
155               </xs:sequence>
156               <xs:attribute name="id" type="xs:token"
                     use="required" />
```

```
157             <xs:attribute name="faultname" type="xs:string"
                   use="optional" />
158           </xs:complexType>
159         </xs:element>
160         <xs:element name="barchart" minOccurs="0" maxOccurs="1" >
161           <xs:complexType>
162             <xs:attribute name="active" type="xs:boolean"
                   use="required" />
163             <xs:attribute name="width" type="xs:integer"
                   use="required" />
164             <xs:attribute name="height" type="xs:integer"
                   use="required" />
165           </xs:complexType>
166         </xs:element>
167         <xs:element name="question" type="xs:string"
              minOccurs="0" maxOccurs="1" />
168         <xs:element ref="link" minOccurs="0"
              maxOccurs="unbounded" />
169         <xs:element name="defcomment" type="tableComment"
              minOccurs="0" maxOccurs="unbounded" />
170         <xs:element name="costcomment" type="tableComment"
              minOccurs="0" maxOccurs="unbounded" />
171       </xs:sequence>
172       <xs:attribute ref="faultnameformat" use="optional"
            default="inherit" />
173     </xs:extension>
174   </xs:complexContent>
175 </xs:complexType>
176 </xs:element>
177
178 <xs:element name="submodel">
179   <xs:complexType>
180     <xs:complexContent>
181       <xs:extension base="icon">
182         <xs:sequence>
183           <xs:element name="window" type="position" minOccurs="0"
                maxOccurs="1"/>
184           <xs:choice minOccurs="0" maxOccurs="unbounded">
185             <xs:element ref="node" />
```

```
186             <xs:element ref="submodel" />
187             <xs:element ref="textbox" />
188           </xs:choice>
189         </xs:sequence>
190       </xs:extension>
191     </xs:complexContent>
192   </xs:complexType>
193 </xs:element>
194
195 </xs:schema>
```

Appendix C

BaseStyle schema file

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:element name="style">
5    <xs:complexType>
6      <xs:element name="class" minOccurs="0" maxOccurs="unbounded">
7        <xs:complexType>
8          <xs:sequence>
9            <xs:element name="background" type="backgroundType"
                minOccurs="0"/>
10           <xs:element name="outline" type="outlineType"
                minOccurs="0"/>
11           <xs:element name="font" type="fontType" minOccurs="0"/>
12           <xs:element name="shape" type="shapeType" minOccurs="0"/>
13           <xs:element name="arrow" type="arrowType" minOccurs="0"/>
14         </xs:sequence>
15         <xs:attribute name="name" type="xs:token" use="required" />
16       </xs:complexType>
17     </xs:element>
18     <xs:attribute name="id" type="xs:token" use="required" />
19     <xs:attribute name="version" use="required" />
20   </xs:complexType>
21 </xs:element>
22
23 <xs:simpleType name="colorType">
24   <xs:restriction base="xs:string">
```

```
25      <xs:pattern value="[0-9|a-f|A-F]{6}" />
26    </xs:restriction>
27  </xs:simpleType>
28
29  <xs:complexType name="fontType">
30    <xs:attribute name="name" type="xs:string" use="optional"
          default="Times"/>
31    <xs:attribute name="size" type="xs:integer" use="optional"
          default="8"/>
32    <xs:attribute name="color" type="colorType" use="optional"
          default="000000"/>
33    <xs:attribute name="bold" type="xs:boolean" use="optional"
          default="false"/>
34    <xs:attribute name="italic" type="xs:boolean" use="optional"
          default="false"/>
35    <xs:attribute name="align" use="optional" default="left">
36      <xs:simpleType>
37        <xs:restriction base="xs:string">
38          <xs:enumeration value="left" />
39          <xs:enumeration value="right" />
40          <xs:enumeration value="center" />
41        </xs:restriction>
42      </xs:simpleType>
43    </xs:attribute>
44  </xs:complexType>
45
46  <xs:simpleType name="brushType">
47    <xs:restriction base="xs:string">
48       <xs:enumeration value="transparent"/>
49       <xs:enumeration value="solid"/>
50       <xs:enumeration value="dense1"/>
51       <xs:enumeration value="dense2"/>
52       <xs:enumeration value="dense3"/>
53       <xs:enumeration value="dense4"/>
54       <xs:enumeration value="dense5"/>
55       <xs:enumeration value="dense6"/>
56       <xs:enumeration value="dense7"/>
57       <xs:enumeration value="horizontal"/>
58       <xs:enumeration value="vertiacal"/>
```

```
59      <xs:enumeration value="cross"/>
60      <xs:enumeration value="backDiagonal"/>
61      <xs:enumeration value="forwardDiagonal"/>
62      <xs:enumeration value="crossDiagonal"/>
63      <xs:enumeration value="linearGradient"/>
64      <xs:enumeration value="radialGradient"/>
65    </xs:restriction>
66  </xs:simpleType>
67
68  <xs:complexType name="outlineType">
69    <xs:attribute name="color" type="colorType" use="optional"
          default="000000"/>
70    <xs:attribute name="width" type="xs:integer" use="optional"
          default="1" />
71    <xs:attribute name="line" use="optional" default="solidLine">
72      <xs:simpleType>
73        <xs:restriction base="xs:string">
74          <xs:enumeration value="noLine"/>
75          <xs:enumeration value="solidLine"/>
76          <xs:enumeration value="dashLine"/>
77          <xs:enumeration value="dotLine"/>
78          <xs:enumeration value="dashDotLine"/>
79          <xs:enumeration value="dashDotDotLine"/>
80        </xs:restriction>
81      </xs:simpleType>
82    </xs:attribute>
83    <xs:attribute name="cap" use="optional" default="flatCap">
84      <xs:simpleType>
85        <xs:restriction base="xs:string">
86          <xs:enumeration value="flatCap"/>
87          <xs:enumeration value="squareCap"/>
88          <xs:enumeration value="roundCap"/>
89        </xs:restriction>
90      </xs:simpleType>
91    </xs:attribute>
92    <xs:attribute name="join" use="optional" default="miterJoin">
93      <xs:simpleType>
94        <xs:restriction base="xs:string">
95          <xs:enumeration value="miterJoin"/>
```

```
96          <xs:enumeration value="bevelJoin"/>
97          <xs:enumeration value="roundJoin"/>
98        </xs:restriction>
99      </xs:simpleType>
100   </xs:attribute>
101 </xs:complexType>
102
103 <xs:complexType name="backgroundType">
104   <xs:attribute name="color" type="colorType" use="optional"
        default="ffffff"/>
105   <xs:attribute name="brush" type="brushType" use="optional"
        default="solid"/>
106 </xs:complexType>
107
108 <xs:complexType name="shapeType">
109   <xs:attribute name="name" use="optional" default="rectagle">
110     <xs:simpleType>
111       <xs:restriction base="xs:string">
112         <xs:enumeration value="square"/>
113         <xs:enumeration value="roundedSquare"/>
114         <xs:enumeration value="rectagle"/>
115         <xs:enumeration value="roundedRect"/>
116         <xs:enumeration value="rhombus"/>
117         <xs:enumeration value="triangle"/>
118         <xs:enumeration value="reverseTriangle"/>
119         <xs:enumeration value="ellipse"/>
120         <xs:enumeration value="circle"/>
121         <xs:enumeration value="hexagon"/>
122         <xs:enumeration value="image"/>
123       </xs:restriction>
124     </xs:simpleType>
125   </xs:attribute>
126   <xs:attribute name="path" type="xs:token" use="optional"
        default="">
127 </xs:complexType>
128
129 <xs:complexType name="arrowType">
130   <xs:attribute name="name" use="optional" default="full">
131     <xs:simpleType>
```

```
132       <xs:restriction base="xs:string">
133          <xs:enumeration value="none"/>
134          <xs:enumeration value="empty"/>
135          <xs:enumeration value="full"/>
136          <xs:enumeration value="circular"/>
137          <xs:enumeration value="diamond"/>
138       </xs:restriction>
139     </xs:simpleType>
140   </xs:attribute>
141 </xs:complexType>
```

Appendix D

Performance analysis results

```
1  ################################
2  # 2017-05-01 12:27:40
3  ################################
4  #
5  # **********************
6  # **********************
7  # ** Nodes: 70
8  # ** Edges: 70
9  # ** Models: 10
10 # ** Textboxes: 10
11 # ** Style classes: 10
12 # **********************
13 # **
14 # ** network initialization: 5ms
15 # ** store: 15ms
16 # ** load: 18ms
17 # ** extraction of all nodes (one by one): 0ms
18 # ** extraction of all nodes (by empty properties): 0ms
19 # ** extraction of all nodes with comment "something": 0ms
20 # ** extraction of all edges (one by one): 0ms
21 # ** extraction of all edges (by empty properties): 0ms
22 # ** extraction of all edges with comment "something": 0ms
23 # ** extraction of all icons (one by one): 0ms
24 # ** extraction of all icons (by empty properties): 0ms
25 # ** extraction of all icons with comment "something": 0ms
26 # ** extraction of all textboxes (one by one): 0ms
```

```
27 # ** extraction of all textboxes (by empty properties): 0ms
28 # ** extraction of all textboxes with comment "something": 0ms
29 # ** extraction of all edgeStyles (one by one): 0ms
30 # ** extraction of all edgeStyles (by empty properties): 0ms
31 # ** extraction of all edgeStyles with comment "something": 0ms
32 # ** extraction of all stylables (one by one): 0ms
33 # ** extraction of all stylables (by empty properties): 0ms
34 # ** extraction of all stylables with class "node": 0ms
35 # ** style classes initialization: 0ms
36 # ** store style: 0ms
37 # ** load style: 1ms
38 # **********************
39 # **********************
40 #
41 # **********************
42 # **********************
43 # ** Nodes: 700
44 # ** Edges: 700
45 # ** Models: 100
46 # ** Textboxes: 100
47 # ** Style classes: 100
48 # **********************
49 # **
50 # ** network initialization: 18ms
51 # ** store: 97ms
52 # ** load: 156ms
53 # ** extraction of all nodes (one by one): 0ms
54 # ** extraction of all nodes (by empty properties): 0ms
55 # ** extraction of all nodes with comment "something": 0ms
56 # ** extraction of all edges (one by one): 1ms
57 # ** extraction of all edges (by empty properties): 0ms
58 # ** extraction of all edges with comment "something": 0ms
59 # ** extraction of all icons (one by one): 1ms
60 # ** extraction of all icons (by empty properties): 0ms
61 # ** extraction of all icons with comment "something": 1ms
62 # ** extraction of all textboxes (one by one): 0ms
63 # ** extraction of all textboxes (by empty properties): 0ms
64 # ** extraction of all textboxes with comment "something": 0ms
65 # ** extraction of all edgeStyles (one by one): 0ms
```

```
66  # ** extraction of all edgeStyles (by empty properties): 0ms
67  # ** extraction of all edgeStyles with comment "something": 0ms
68  # ** extraction of all stylables (one by one): 1ms
69  # ** extraction of all stylables (by empty properties): 0ms
70  # ** extraction of all stylables with class "node": 0ms
71  # ** style classes initialization: 1ms
72  # ** store style: 4ms
73  # ** load style: 6ms
74  # **********************
75  # **********************
76  #
77  # **********************
78  # **********************
79  # ** Nodes: 7000
80  # ** Edges: 7000
81  # ** Models: 1000
82  # ** Textboxes: 1000
83  # ** Style classes: 1000
84  # **********************
85  # **
86  # ** network initialization: 183ms
87  # ** store: 867ms
88  # ** load: 1s 534ms
89  # ** extraction of all nodes (one by one): 3ms
90  # ** extraction of all nodes (by empty properties): 1ms
91  # ** extraction of all nodes with comment "something": 6ms
92  # ** extraction of all edges (one by one): 3ms
93  # ** extraction of all edges (by empty properties): 2ms
94  # ** extraction of all edges with comment "something": 3ms
95  # ** extraction of all icons (one by one): 5ms
96  # ** extraction of all icons (by empty properties): 1ms
97  # ** extraction of all icons with comment "something": 14ms
98  # ** extraction of all textboxes (one by one): 0ms
99  # ** extraction of all textboxes (by empty properties): 0ms
100 # ** extraction of all textboxes with comment "something": 1ms
101 # ** extraction of all edgeStyles (one by one): 3ms
102 # ** extraction of all edgeStyles (by empty properties): 0ms
103 # ** extraction of all edgeStyles with comment "something": 4ms
104 # ** extraction of all stylables (one by one): 1ms
```

```
105 # ** extraction of all stylables (by empty properties): 1ms
106 # ** extraction of all stylables with class "node": 5ms
107 # ** style classes initialization: 9ms
108 # ** store style: 34ms
109 # ** load style: 53ms
110 # **********************
111 # **********************
112 #
113 # **********************
114 # **********************
115 # ** Nodes: 70000
116 # ** Edges: 70000
117 # ** Models: 10000
118 # ** Textboxes: 10000
119 # ** Style classes: 10000
120 # **********************
121 # **
122 # ** network initialization: 2s 56ms
123 # ** store: 9s 111ms
124 # ** load: 15s 99ms
125 # ** extraction of all nodes (one by one): 37ms
126 # ** extraction of all nodes (by empty properties): 9ms
127 # ** extraction of all nodes with comment "something": 71ms
128 # ** extraction of all edges (one by one): 35ms
129 # ** extraction of all edges (by empty properties): 7ms
130 # ** extraction of all edges with comment "something": 26ms
131 # ** extraction of all icons (one by one): 51ms
132 # ** extraction of all icons (by empty properties): 11ms
133 # ** extraction of all icons with comment "something": 140ms
134 # ** extraction of all textboxes (one by one): 0ms
135 # ** extraction of all textboxes (by empty properties): 0ms
136 # ** extraction of all textboxes with comment "something": 8ms
137 # ** extraction of all edgeStyles (one by one): 35ms
138 # ** extraction of all edgeStyles (by empty properties): 7ms
139 # ** extraction of all edgeStyles with comment "something": 51ms
140 # ** extraction of all stylables (one by one): 22ms
141 # ** extraction of all stylables (by empty properties): 20ms
142 # ** extraction of all stylables with class "node": 52ms
143 # ** style classes initialization: 90ms
```

```
144 # ** store style: 324ms
145 # ** load style: 525ms
146 # **********************
147 # **********************
148 #
149 # **********************
150 # **********************
151 # ** Nodes: 700000
152 # ** Edges: 700000
153 # ** Models: 100000
154 # ** Textboxes: 100000
155 # ** Style classes: 100000
156 # **********************
157 # **
158 # ** network initialization: 23s 296ms
159 # ** store: 1m 30s 498ms
160 # ** load: 2m 31s 227ms
161 # ** extraction of all nodes (one by one): 477ms
162 # ** extraction of all nodes (by empty properties): 116ms
163 # ** extraction of all nodes with comment "something": 50s 612ms
164 # ** extraction of all edges (one by one): 497ms
165 # ** extraction of all edges (by empty properties): 75ms
166 # ** extraction of all edges with comment "something": 252ms
167 # ** extraction of all icons (one by one): 568ms
168 # ** extraction of all icons (by empty properties): 149ms
169 # ** extraction of all icons with comment "something": 1s 402ms
170 # ** extraction of all textboxes (one by one): 13ms
171 # ** extraction of all textboxes (by empty properties): 14ms
172 # ** extraction of all textboxes with comment "something": 892ms
173 # ** extraction of all edgeStyles (one by one): 373ms
174 # ** extraction of all edgeStyles (by empty properties): 71ms
175 # ** extraction of all edgeStyles with comment "something": 476ms
176 # ** extraction of all stylables (one by one): 243ms
177 # ** extraction of all stylables (by empty properties): 240ms
178 # ** extraction of all stylables with class "node": 586ms
179 # ** style classes initialization: 1s 795ms
180 # ** store style: 3s 369ms
181 # ** load style: 5s 556ms
182 # **********************
```

```
183 # **********************
184 #
185 ################################
```