Project Final Report

AUTOMATIC REFINEMENT OF EQUILIBRIA IN GAME THEORY

JF10002

King Fahd University of Petroleum and Minerals
Dhahran
Saudi Arabia

Slim Belhaiza, Ph.D.

Assistant Professor

Department of Mathematics and Statistics

Abstract

This document is the final report to our project entitled *"Automatic Refinement of Equilibria in Game Theory"* realised between January 2010 and December 2010 under the DSR Junior Faculty Grant JF100002.

This document mainly presents the architecture of the software *XGame-Solver* we have designed and improved for this purpose.

Keywords: Game Theory, Refinement, Bimatrix, Sequential, Polymatrix, XGame-Solver.

Table of Contents

Introduction

This document provides a comprehensive architectural overview of the XGame Solver© application, using a number of different architectural views to depict different aspects of the application. It is intended to capture and convey all significant architectural choices.

XGame Solver© is an intuitive, extensible, modern cross-platform (Windows, Linux, Mac OS) Qt-based application developed by Alexandre Dzimi Mvé (*Software Developer at CMLabs Simulation, Montreal, Canada*) and Slim Belhaiza which aims to provide a very powerful tool for constructing and solving *Bimatrix*, *Sequential* and *Polymatrix* games.

Unlike its previous versions, XGame Solver 2.5 can be run in GUI mode (Editor) or Console mode. This has been made possible by decoupling the *Solver Engine* from the application presentation layer. In a distributed system, the *Solver Engine* could be run in a separate machine as a backend service to serve processing requests of applications over the network. This future architecture is not covered in this document though. The picture below shows the actual *Control Flow* of the system:

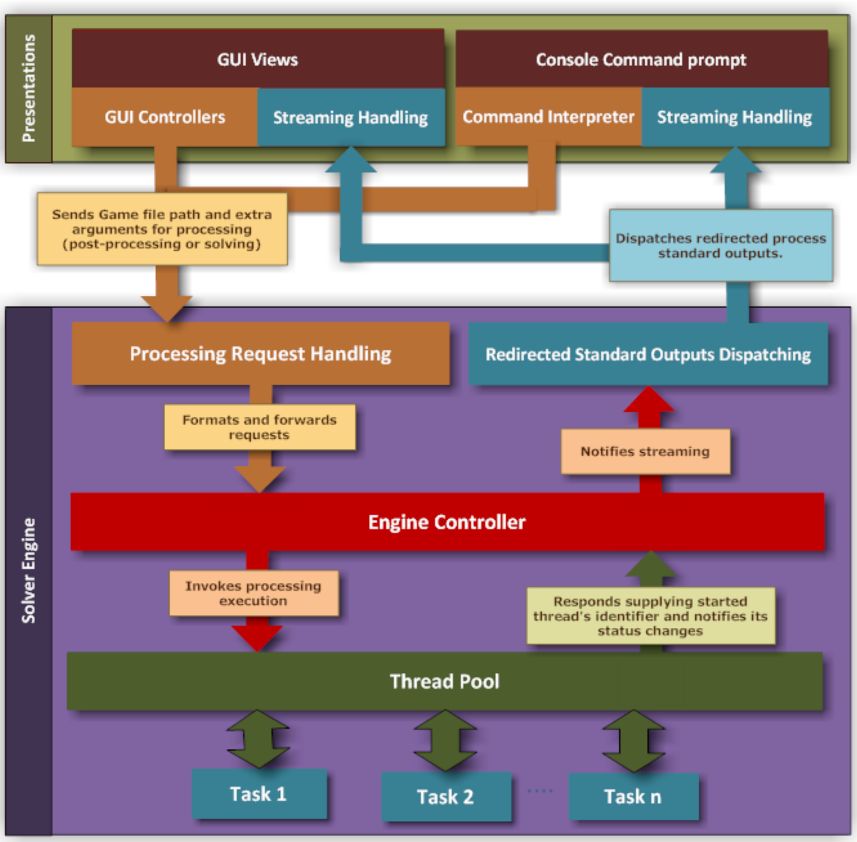

Figure 1: Control Flow diagram of the system

All diagrams throughout this document have been generated using *SmartDraw 2010*.

Definitions, Acronyms and Abbreviations

Bimatrix game	In Game Theory, a Bimatrix game represents a confrontation of two players in normal form. In a Bimatrix game, there are two players who effectively make their moves simultaneously without knowing the other player's action.
Sequential game	A Sequential game represents a sequential form of two person extensive form. In a sequential game, there are two players who effectively make their moves sequentially basing on the other player's action.
Polymatrix game	A Polymatrix game represents a confrontation of *n* players as a collection of *n(n-1)/2* Bimatrix games.
UML	Unified Modeling Language
SAD	Software Architecture Document

1. Architectural Goals and Constraints

This section describes the software requirements and objectives having significant impact on its architecture.

1.1. Technical Platform

The target operating systems are *Windows XP*, *Windows Vista*,*Windows 7*, *Mac OS X Tiger*, *Mac OS X Leopard* and *Linux*.

1.2. Concurrency

Multiple solvers can be run simultaneously. However, only one task can be run on a game at once. This is to prevent generated output files from corruption since solvers may generate the same output files of which names are based on the source game's file name. In such case, concurrent solvers may edit the same files at the same time, which induces data corruptions. Qt Concurrency and Thread Support will be used to fulfill this requirement efficiently. As XGame Solver 2.5 is not (yet) a distributed system, no other complex mechanisms will be used to manage concurrency.

1.3. Security

Currently, there is no way to efficiently secure plugin support of the system (user plugins have read/write permissions on disk). User has thus to ensure that unofficial plugins are safe. Any contributor who wants to make their plugins official has to provide us with sources.

1.4. Performance

Solvers (or post-processings) must complete within reasonable delays depending on complexity of the algorithm used. Complexity of each algorithm must be estimated and then be used as metric to define what its reasonable execution time must be. Memory usage of solver should not exceed 70 MB.

2. Use Cases

This section presents the significant use cases of the system. All actions performed on games are available in GUI mode only. **Figure 2** shows the *Use Case* diagram of *XGame Solver*.

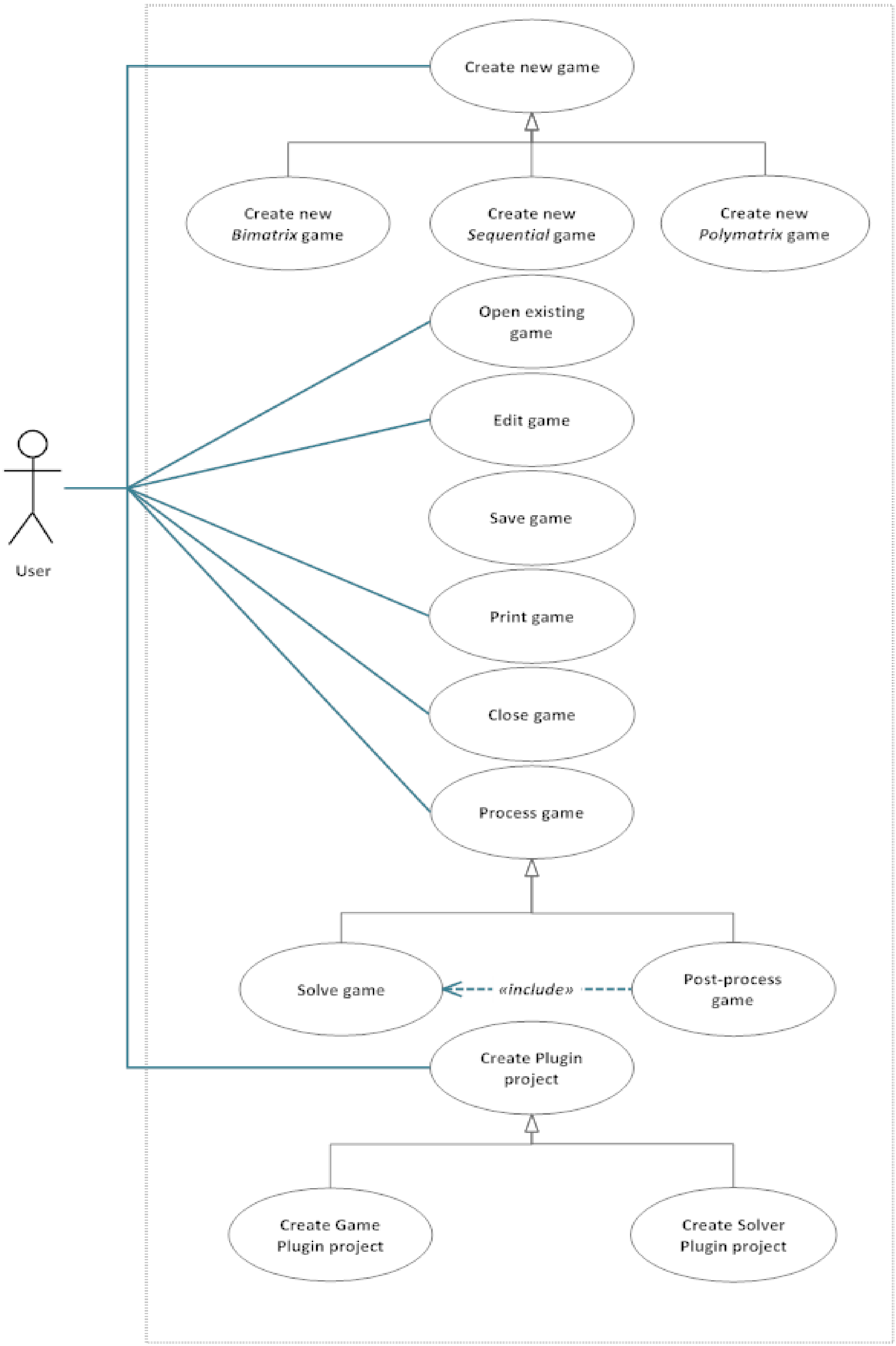

Figure 2: User Class diagram

- Create new game

Use case	Create new game
Actor:	User
Type	Primary
Description	Create a new strategic game.

- Create new *Bimatrix* game

Use case	Create new *Bimatrix* game
Actor:	User
Type	Primary
Description	Create a new *Bimatrix* game.

- Create new *Sequential* game

Use case	Create new *Sequential* game
Actor:	User
Type	Primary
Description	Create a new *Sequential* game.

- Create new *Polymatrix* game

Use case	Create new *Polymatrix* game
Actor:	User
Type	Primary
Description	Create a new *Polymatrix* game.

- Open existing game

Use case	Open existing game
Actor:	User
Type	Primary
Description	Open an existing game loading a file from the local machine.

- Edit game

Use case	Edit game
Actor:	User
Type	Primary
Description	Edit an opened game.

- Save game

Use case	Save game
Actor:	User
Type	Primary
Description	Save a modified game.

- Print game

Use case	Print game
Actor:	User
Type	Primary
Description	Print opened game.

- Close game

Use case	Close game
Actor:	User
Type	Primary
Description	Close an opened game.

- Process game

Use case	Process game
Actor:	User
Type	Primary
Description	Solve or post-process a game.

- Solve game

Use case	Solve game
Actor:	User
Type	Primary
Description	Solve a game.

- Post-process game

Use case	Post-process game
Actor:	User
Type	Primary
Description	Post-Process a game. User must run solver on the target game beforehand.

- Create Plugin project

Use case	Create Plugin project
Actor:	User
Type	Secondary
Description	Create a Qt/C++ project for creating Solver or Game plugins.

- Create Game Plugin project

Use case	Create Game Plugin project
Actor:	User
Type	Secondary
Description	Create a Qt/C++ project for creating Game plugins.

- Create Solver Plugin project

Use case	Create new game
Actor:	User
Type	Secondary
Description	Create a Qt/C++ project for creating Solver plugins.

3. Packages

This section presents the layered *Package* diagram of the system in **figure 3**. It is used as a central view to represent the software system compile-time logical architecture.

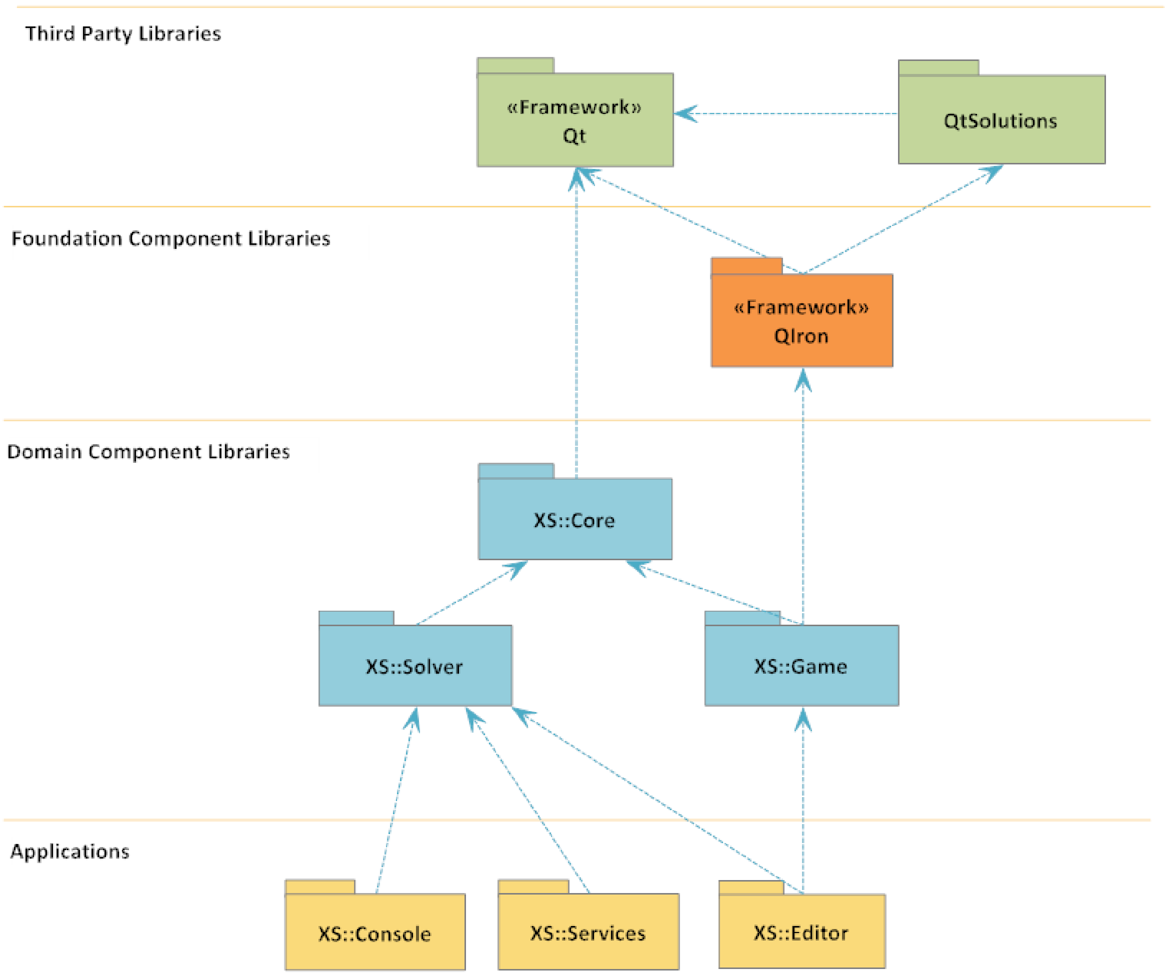

Figure 3: Layered Package diagram of the system

3.1. *Qt* package

Qt package is a cross-platform application development framework developed by Nokia, widely used for the development of GUI programs, and also used for developing non-GUI programs such as console tools and servers. It contains the *QtCore*, *QtGui*, *QtXml* and *QtNetwork* modules which provide most of GUI, XML, Network components and classes to *XGame Solver.*

3.2. *QtSolutions* package

QtSolutions is a catalogue of add-on components and tools to make development with Qt more efficient. It is also developed by Nokia as merged package to Qt and mainly used to make *XGame Solver* a single instance application by inheriting the domain Application class from the *QtSingleApplication* class provided by *QtSolutions*.

3.3. *QIron* package

QIron is a framework developed by *Alexandre Dzimi Mvé* that provides a set of stunning and very customizable Qt-based widgets for creating professional user interface with ease.

3.4. *XS::Core* package

The *XS::Core* package is a domain component package providing reusable Core components and model objects to *XGame Solver* such as matrices, transformations, value types (Big Integers), utilities. It also provides plugins support for user plugins.

3.5. *XS::Solver* package

The *XS::Solver* package is a domain component package that provides reusable Solver-related components to *XGame Solver* such as abstract *Processing* models and abstract *Solver* factory.

3.6. *XS::Game* package

The *XS::Game* package is a domain component package providing reusable Game-related components to *XGame Solver* such as abstract *Game* model and view, *Game* editor interface and abstract *Game* factory.

3.7. *XS::Services* package

The *XS::Services* package provides services used by the applications. These services include the *Solver Engine* and the *Code Generator*.

3.8. *XS::Editor* package

As the name suggests, the *XS::Editor* (*XGame Solver Editor Application*) package represents the *XGame Solver* application in GUI mode. It contains views of games, solvers, and matrix transformations; and their corresponding controllers (MVC architecture).

3.9. *XS::Console* package

As the name suggest, the *XS::Console* (*XGame Solver Console Application*) represents the *XGame Solver* application in Console mode. It mainly contains a command interpreter (which interprets user entries and then invokes the *Solver Engine* with the appropriate arguments) and a *Redirected Standard OutputsHandler*.

4. Components

This section presents the *Component* diagrams of the system. The diagram in **figure 4** presents the implied top-level components connected with association relationships or interfaces.

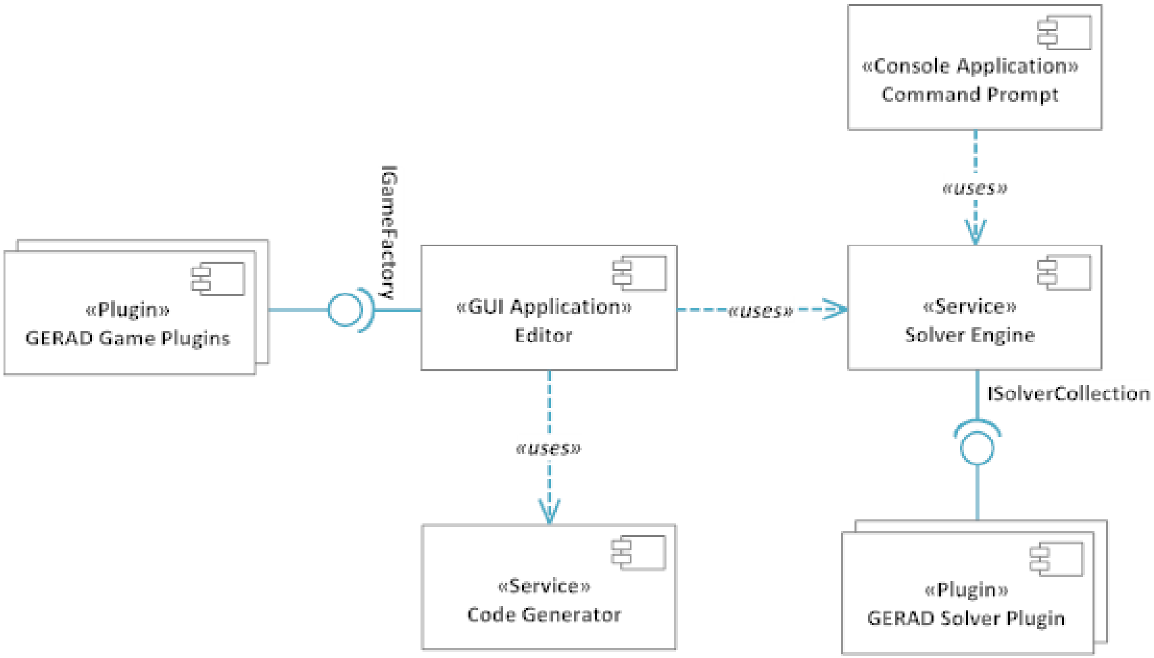

Figure 4: Top-level Component Diagram

4.1. *Editor* component

The *Editor* component represents the entire *Editor* application or the GUI application. It contains several sub-components which are the *Game Manager*, the *Streaming Handler*, the *Solver Manager* and the *Main Window* components is illustrated below in **figure** 5.

4.1.1. *Game Management* component

As its name suggests, the *Game Management* component manages all opened games and loaded game plugins. It gets notified of any changes of game states and then dispatches these events to all game observers to get them up to date. For example, when user modifies a game using the corresponding view, the *Game Management* component gets notified first and then forwards this modification event to all game observers including the *Main Window*'s controller which updates the *Main Window*. This update may involve disabling the *Solve* button if the current game gets into invalid state and enabling it back when it gets back to valid.

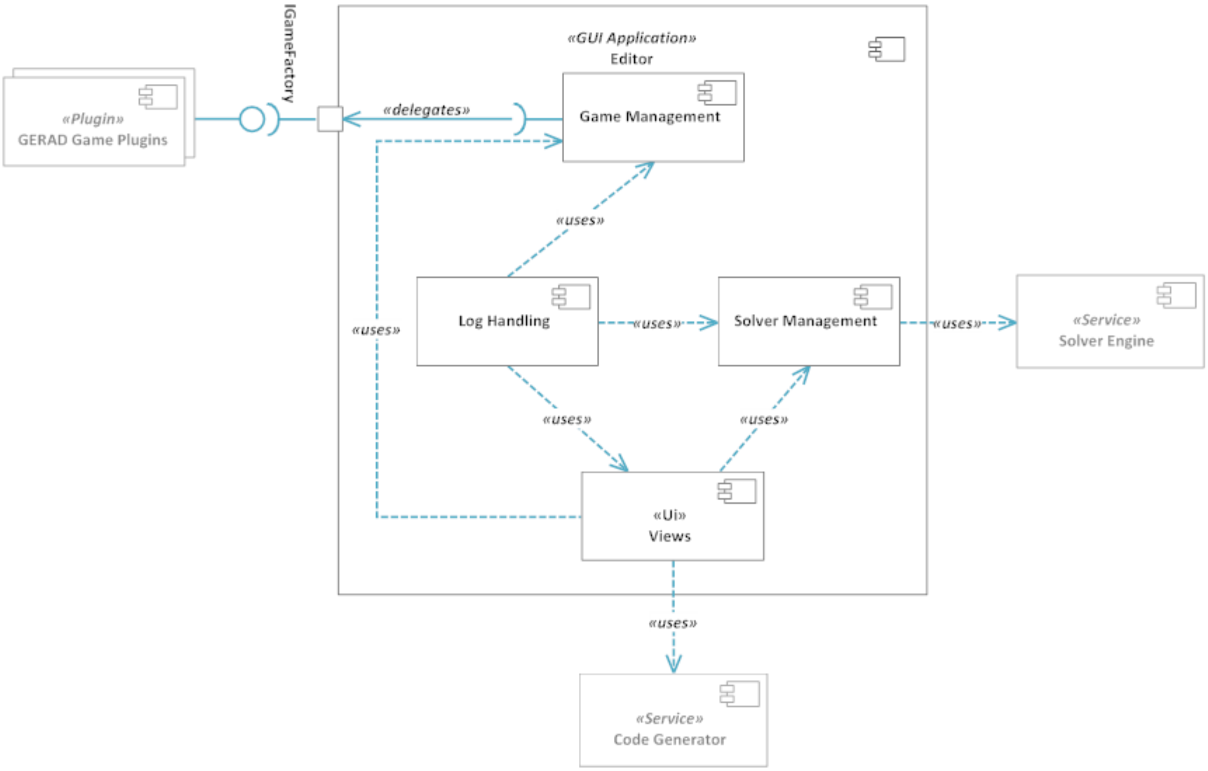

Figure 5: Editor Component*Solver Management Component*

The *Solver Management* component is used as an interface between the *Main Window*'s controller and the *Solver Engine*. Indeed, the *Main Window*'s controller does not invoke the *Solver Engine* directly when user run a solver or a post-processing on a game. It invokes the *Solver Manager* which posts the execution request to the *Solver Engine*. It allows the application to get some information about the *Solver Engine* such as the list of all available solvers and post-processings.

4.1.2. *Log Handling* component

As the name suggests, the *Log Handling* component handles standard outputs redirected from the running processes. Starting a solver or post-processing involves starting a process with the corresponding executable. Since the started process (executable) does not share the same stream buffers as the application, Stream Redirection is then simplest way to retrieve those outputs in order to be displayed through the application. The Streaming Handler is thus in charge of retrieving these buffered outputs and sending them to *Main Window* for displaying in the *Output View*.

4.1.3. *Views* component

The *Views* component represents the main application and its child views. The central child view is the game view.

4.2. *Command Prompt* component

The *Command Prompt* component represents the *XGame Solver* application in *Console* mode. It contains several sub-components which are the *Console Manager*, the *Streaming Handler*, the *Solver Manager* and the *Console Controller* components as illustrated in **figure 6**.

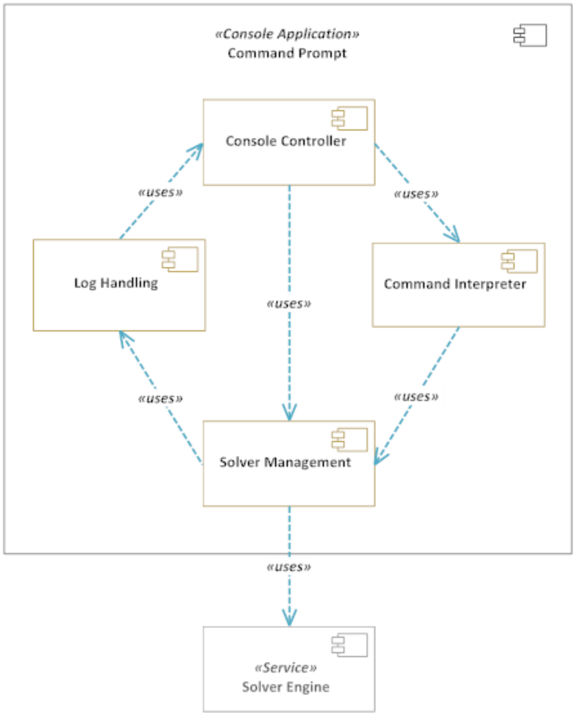

Figure 6: Command Prompt component

4.2.1. *Console Controller* component

This component handles commands entered by user and then forwards them to the *Command Interpreter* component for parsing these commands. It also prints redirected process outputs on screen.

4.2.2. *Solver Management* component

The *Solver Management* component is used as interface between the *Command Interpreter* and the *Solver Engine*. Indeed, the *Command Interpreter* does not invoke the *Solver Engine* directly to execute a command. It invokes the *Solver Management* component which posts the execution request to the *Solver Engine*. It allows the application to get some information about the *Solver Engine* such as the list of all available solvers and post-processings.

4.2.3. *Log Handling* component

This component works the same as the GUI application. The only difference is that after retrieving the redirected outputs, the *Log Handling* component sends them to the *Console Controller* to be printed on screen.

4.2.4. *Command Interpreter* component

The *Command Interpreter* component parses command lines entered by user and invokes the Solver Manager to execute them if they are valid. These command lines are string lists containing the actual command to run and some arguments. The command could be a solver or post-processing to run, a *help* command to get some helps, or an *info request* command to get some information about the *Solver Engine* such as the list of all available solvers.

4.3. *Solver Engine* component

The *Solver Engine* component represents the solver engine of the application. It is used to solve games specifying what solver to use with some extra arguments. To achieve that efficiently, it holds a set of solvers mapped using their unique key names to distinguish a solver from another. Running a solver on a given game involves invoking the solver engine with the solver's unique key name, the algorithm's unique key name to use, the game's path name and some algorithm-specific arguments. Solver executions are tasks run by threads supplied and managed by the solver engine's thread pool.

4.4. *Code Generator* component

The *Code Generator* component is used to generate C++/Qt skeleton code for Game and Solver plugin projects.

4.5. *GERAD Game plugins*

The *GERAD Game Plugins* (**figure 7**) components are the default game plugins provided alongside the application. They make it possible to create or edit Bimatrix, Sequential and Polymatrix games.

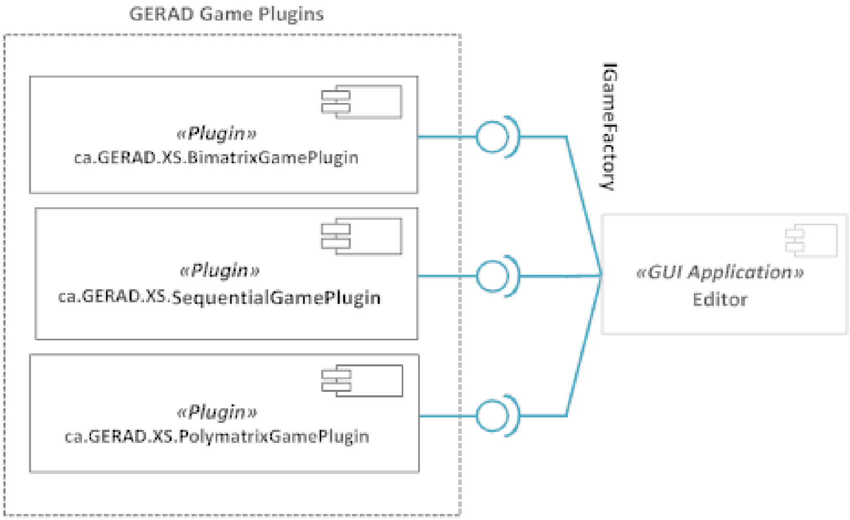

Figure 7: GERAD Game plugins

4.5.1. *Ca.Gerad.XS.BimatrixGamePlugin* component

This plugin provides the default capability to create, load, edit and save Bimatrix games.

4.5.2. *Ca.Gerad.XS.SequentialGamePlugin* component

This plugin provides the default capability to create, load, edit and save Sequential games.

4.5.3. *Ca.Gerad.XS.PolymatrixGamePlugin* component

This plugin provides the default capability to create, load, edit and save Polymatrix games.

4.6. *GERAD Solver plugin*

The *GERAD Solver Plugin* (**figure 8**)component is the default solver plugin provided alongside the application. It provides solvers to solve Bimatrix, Sequential and Polymatrix games.

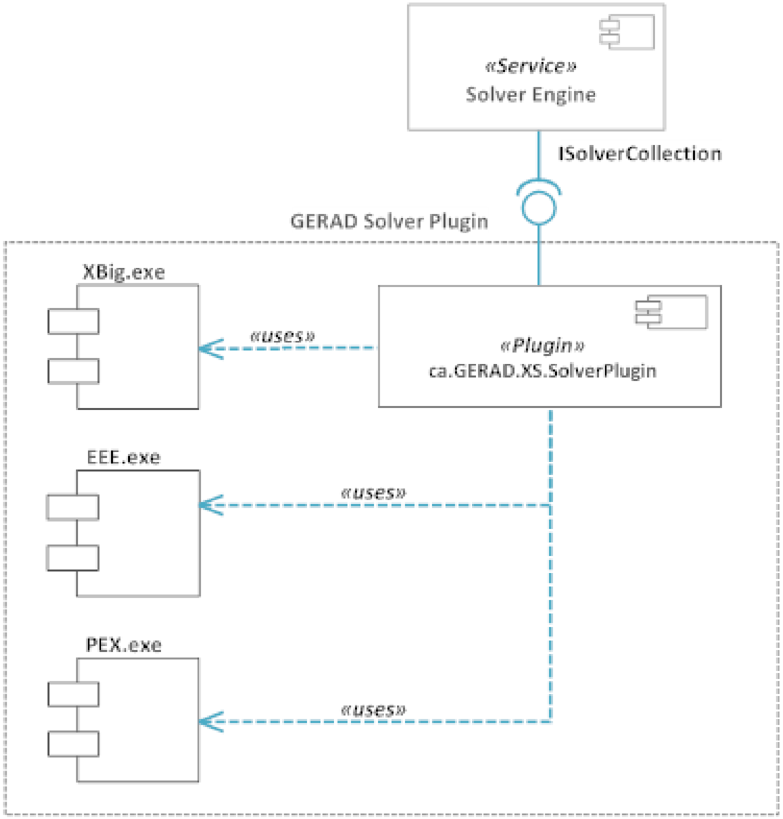

Figure 8: GERAD Solver plugin

4.6.1. *XBig.exe* component

This component is an executable used to solve Bimatrix and Sequential games using the ExMIP algorithms developed by Slim Belhaiza.

4.6.2. *EEE.exe component*

This component is an executable used to solve Bimatrix and Sequential games using EEE algorithm developed by Charles Audet.

4.6.3. *PEX.exe component*

This component is an executable used to solve Polymatrix games using ExMIP algorithm developed by Slim Belhaiza.

5. Classes

This section presents the class diagrams of the system grouped by modules.

5.1. *XSEditor* module

As mentioned above, the *Editor* represents the GUI application. **Figure 9** shows its complete class diagram (class members are hidden for better illustration).

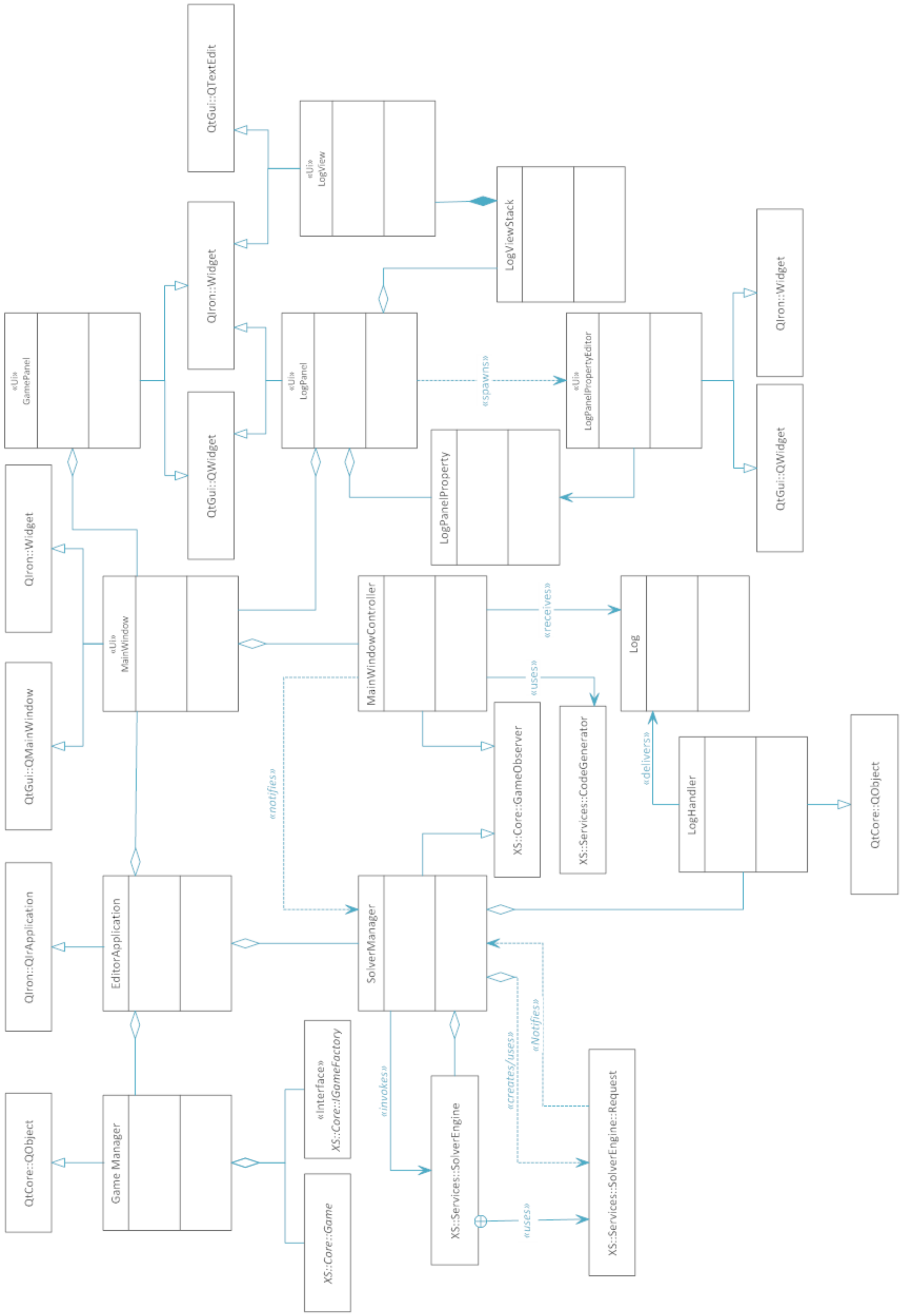

Figure 9: Editor Class diagram

5.1.1. *Editor Application* class

This class represents *XGame Solver* application in GUI mode.

5.1.2. *MainWindow* class

This class manages the main window of the application. It contains the Game panel which displays loaded games and the Log panel which displays logs streamed by the running processings (solvers or post-processings).

5.1.3. *MainWindowController* class

This class represents the controller of the main window. It handles user actions and sends the corresponding commands to the appropriate receivers. Use of commands is a convenient way to undo and redo changes applied to editing objects. For instance, use of commands allows user to undo changes applied to games such as modifying payoffs or strategies of players in a game.

5.1.4. *GamePanel* class

The *GamePanel* class provides a stack of opened game (views) and displays them under tabs.

5.1.5. *LogPanel* class

The *LogPanel* class provides a stack of log views. Each log view is associated with one of the opened games.

5.1.6. LogViewStack class

This class is a stack of log views held by the *LogPanel* Class.

5.1.7. *LogView* class

This class provides a text area displaying logs streamed by the processing running on its associated game.

5.1.8. *LogPanelProperty*class

This class allows user to personalize the appearance of the log views. It makes it possible to change the background color, the text color and font of all stacked log views.

5.1.9. *LogPanelPropertyEditor* class

This class provides a dialog to edit the appearance of the log views.

5.1.10. *LogHandler* class

The *LogHandler* class handles logs streaming by all the running processings and forwards them to the controller of the main window and then to the appropriate log views.

5.1.11. *Log* class

This class represents a log streamed by a running processing. A log is actually nothing but a redirected standard output (using std::cout, printf or std::cerr in C++) from the process associated with a running processing.

5.1.12. *GameManager* class

This class manages all loaded games and game factories. It gets notified of any changes of game states and then dispatches these events to all game observers to get them up to date.

5.1.13. *SolverManager* class

The *SolverManager* class provides an interface between the rest of the application and the actual solver engine. Indeed, the main window does not invoke the solver engine directly to run solvers since it has no visibility on it. It makes it through the solver manager which posts processing requests to the engine. It also allows the application to get some information about the engine such as the list of all available solvers and post-processings.

5.1.14. *ProcessingRequest* class

This class represents a request made by user to run a processing (solver or post processing). Processing request is sent by the solver manager to the solver engine and contains all information about a processing to run and a target game file. It is also used by the engine to send logs to the application. By analogy with client-server model, a processing request may be considered as a socket the solver manager (as client) and the engine (as server) communicate through. The request closes when the associated processing finishes (completion or killed at user's request).

5.2. *XSCodeGenerator* module

As mentioned above, the *Code Generator* is used to generate C++/Qt skeleton code for Game and Solver plugin projects. **Figure 10** shows its complete class diagram (class members are hidden for better illustration).

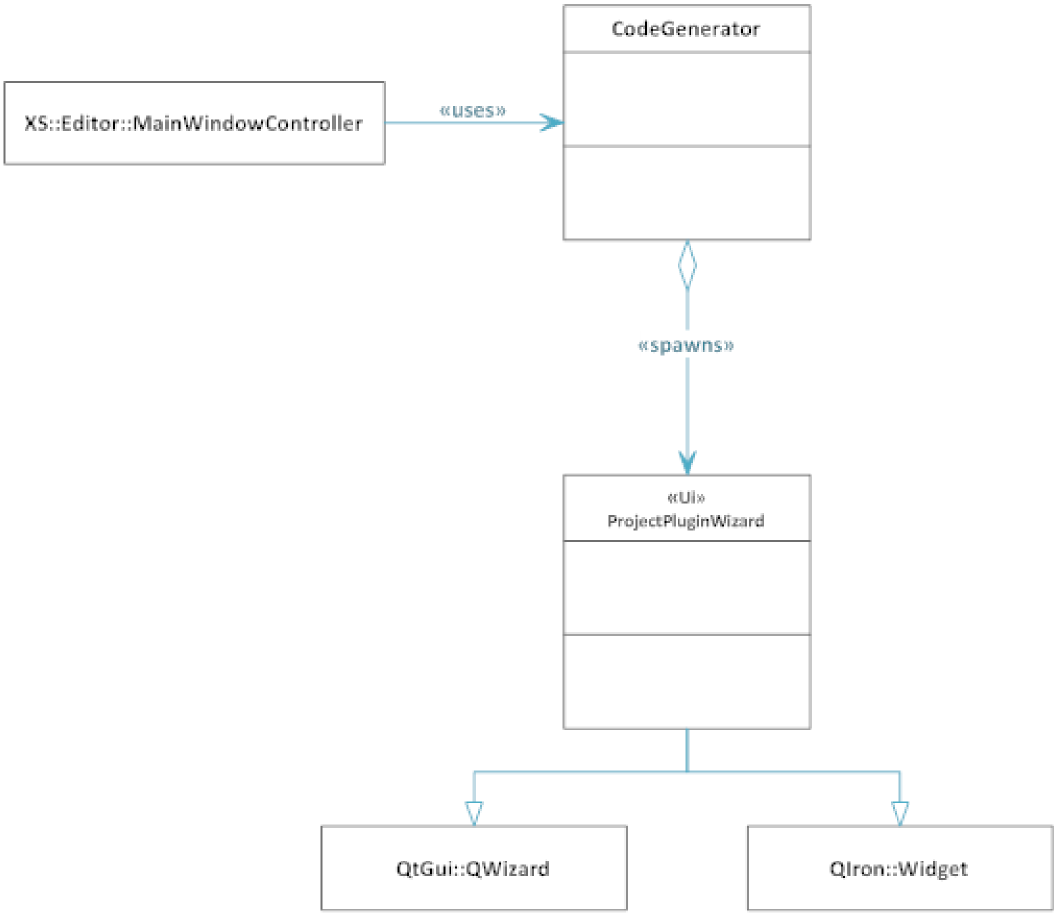

Figure 10: CodeGeneratorClass diagram

5.2.1. *CodeGenerator* class

This class provides functionalities to generate codes. It is used to generate Qt/C++ project in order for users to implement their own plugins for the application.

5.2.2. *ProjectPluginWizard* component

This class provides a wizard spawned by the code generator to guide the user through creation of game or solver plugin projects.

5.3. *XSCommandPrompt* module

As mentioned above, the *CommandPrompt* module represents the Console application. Figure 11 shows its complete class diagram (class members are hidden for better illustration).

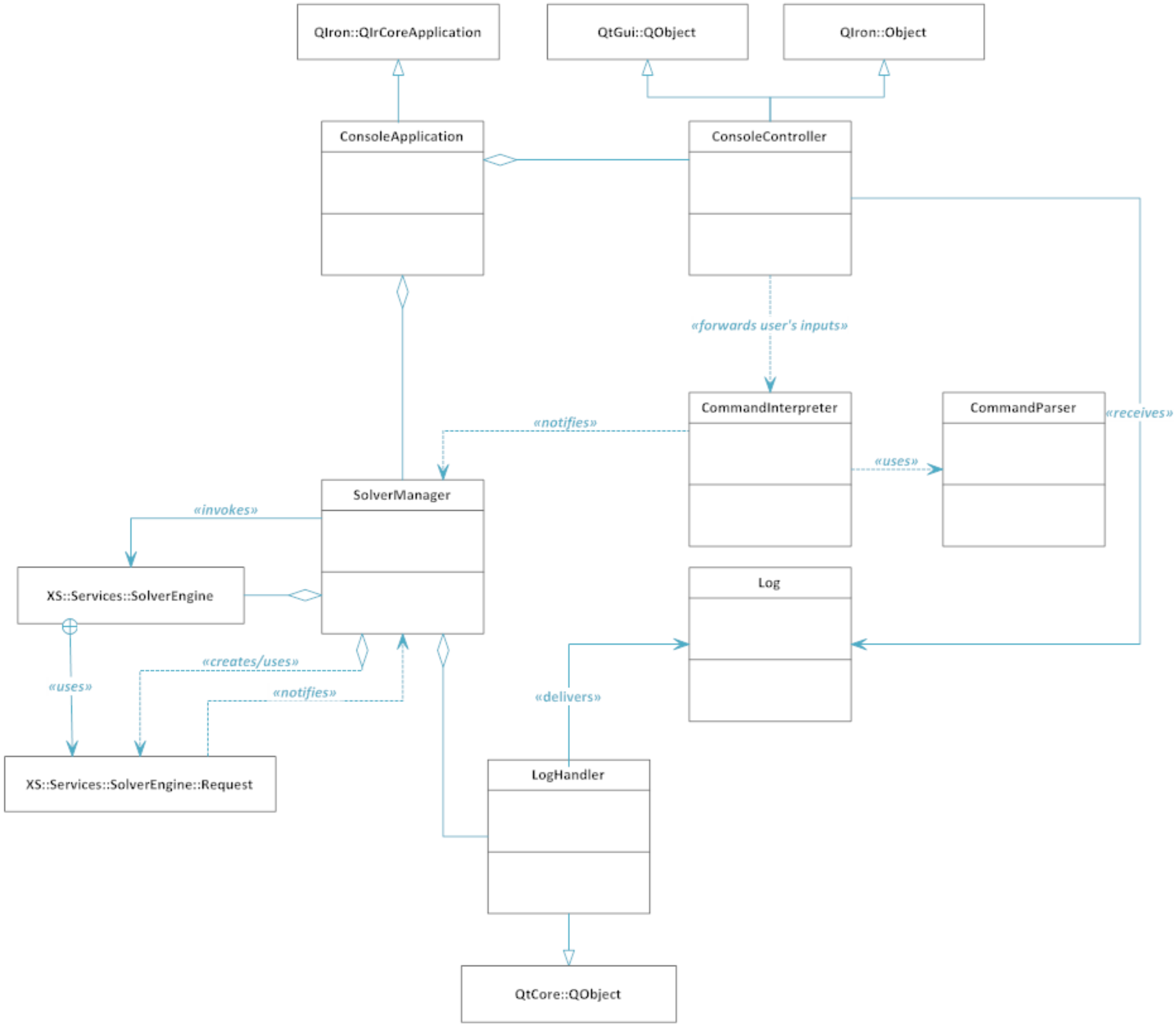

Figure 11: Command Prompt Class diagram

5.3.1. *ConsoleApplication* class

This class represents *XGame Solver* application in Console mode.

5.3.2. *ConsoleController* class

This class handles user inputs and forwards them to the interpreter. It locks the console when a processing is running, which prevents user from entering multiple commands. It releases the console when the processing stops.

5.3.3. *CommandInterpreter* class

The *CommandInterprete r*class interprets user input. It extracts the processing unique name from the command and requests a task to the solver manager.

5.3.4. *CommandParser* class

The *CommandParser* class parses user entry splitting it into processing unique name, target game file path and extra arguments.

5.3.5. *LogHandler* class

The *LogHandler* class handles logs streaming by all the running processing and forwards them to the console controller which prints them on console.

5.3.6. *Log* class

This class represents a log streamed by a running processing. A log is actually nothing but a redirected standard output (using std::cout, printf or std::cerr in C++) from the process associated with a running processing.

5.3.7. *SolverManager* class

The *SolverManager* class provides an interface between the rest of the application and the actual solver engine. Indeed, the console controller does not invoke the solver engine directly to run solvers since it has no visibility on it. It makes it through the solver manager which posts processing requests to the engine. It also allows the application to get some information about the engine such as the list of all available solvers and post-processings.

5.4. *XSSolverEngine* module

As mentioned above, the *SolverEngine* module represents the solver engine used by the applications to solve games. **Figure 12** shows its complete class diagram (class members are hidden for better illustration).

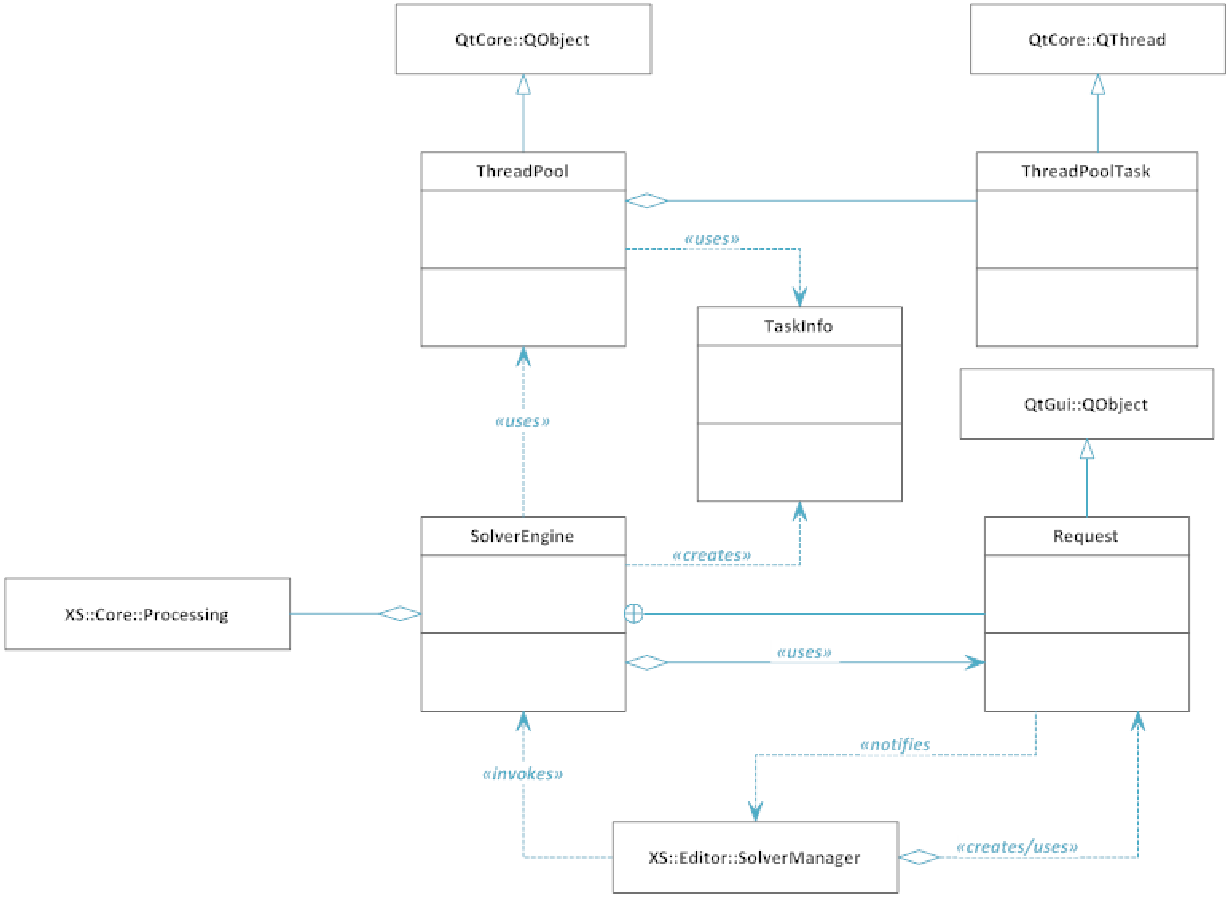

Figure 12: Solver Engine Class diagram

5.4.1. *SolverEngine* class

This class represents solver engine used by the application to solve games. It manages a collection of all available processings to be executed at user's request. To serve user's processing request, it makes use of a thread pool which manages a collection of thread pool tasks (threads). Indeed, when the solver engine receives a processing request from the application, it creates a *TaskInfo* object from details pulled from the request and then invokes the tread pool. The thread pool starts the first available threads picked up from its non-running thread queue using the *TaskInfo* object and returns a unique token (task id) to the solver engine. Any communication between the solver engine and thread pool is made using these tokens to specify what running processing or thread execution is concerned by a given message. For instance, a returned token may be used to kill a specific task or to dispatch processing logs to the proper execution flow.

5.4.2. *ThreadPool* class

This class manages a collection of threads (thread pool tasks) to run solvers.

5.4.3. *SolverEngine::Request* class

This class represents a request made by user to run a processing (solver or post processing). Processing request is sent by the solver manager to the solver engine and contains all information about a processing to run, the target game file and some extra arguments. It is also used by the engine to send logs to the application. By analogy with client-server model, a processing request may be considered as a socket the solver manager (as client) and the engine (as server) communicate through. The request closes when the associated processing finishes (completion or killed at user's request).

5.4.4. *TaskInfo* class

This class contains all information about a processing to run, the target game file and some extra arguments. It is used by the thread pool to just set up a thread pool task before running.

5.4.5. *ThreadPoolTask* class

The *ThreadPoolTask*class represents a thread managed by the thread pool and used to run a specific processing.

5.5. *XSCoreLib* module

The *XSCore* module provides reusable core classes to XGame Solver. Basically, it includes abstraction of games, solvers, matrix transformations. It also provides plugins support for user extensions.

5.5.1. Game Framework

The Game Framework provides base classes to build up games and to develop user game extensions. It is an implementation of the MVC, Abstract Factory and Observer patterns. **Figure 13** shows its complete class diagram (class members are hidden for better illustration).

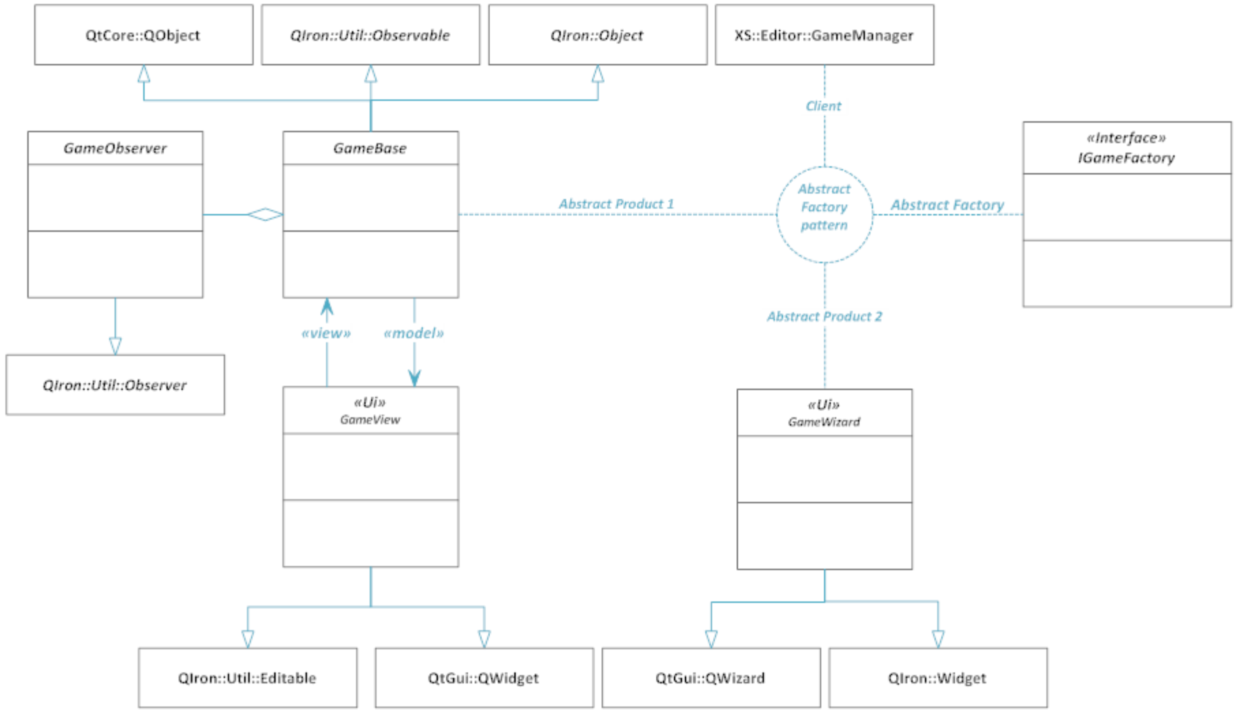

Figure 13: Game framework Class diagram

5.5.1.1. Game class

The *Game* class is the base class of all games. In the *MVC* pattern, this class represents the *Model* whereas the *GameView* class represents the *View*. In the Observer pattern, it represents the *Subject* observed by *GameObserver* objects.

5.5.1.2. GameView class

The *GameView* class is the base class of view of games. Basically, it represents the graphical representation of a game. Any game view should inherit from this class to be used by the *Editor* application.

5.5.1.3. GameObserver class

The *GameObserver* class provides a way to get informed of any changes in games.

5.5.1.4. GameWizard class

The *GameWizard* class represents a wizard used to guide users through creation of new games. Ideally, there should be a wizard created for each type of games. That means there should be a different wizard for *Bimatrix* games, *Sequential* games and

5.5.2. Matrix Framework

The Matrix Framework provides classes to manipulate and visualize matrices. It mainly uses to build up games as they expose their data as matrices. **Figure 14** shows its complete class diagram (class members are hidden for better illustration).

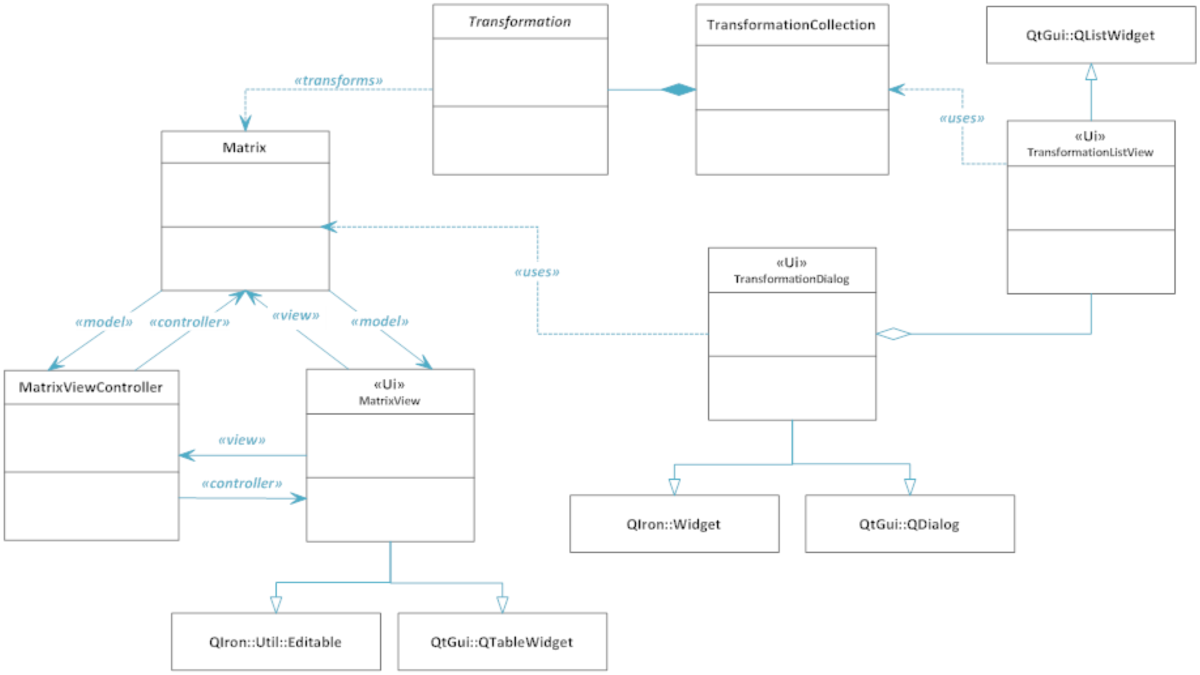

Figure 14: Matrix Framework Class diagram

5.5.2.1. Matrix class

This class represents a*n-m* matrix of integers.

5.5.2.2. MatrixView class

The *MatrixView* class represents a view of matrix.

5.5.2.3. MatrixViewController class

The *MatrixViewController* class represents the controller of a matrix view.

5.5.2.4. Transformation class

This class represents a matrix transformation.

5.5.2.5. TransformationCollectionclass

This class represents a collection of all available matrix transformations.

5.5.2.6. TransformationListView class

This class provides a- view of the matrix transformation collection from which user can select a matrix transformation to apply.

5.5.2.7. TransformationDialog class

This class represents a dialog which displays a matrix transformation list view.

5.5.3. Solver Framework

The Solver Framework provides base classes to create solver and post-processing. It is also used to enable support for user solver extensions. It provides *NashSubset*, *QuasiStrong* and *Perfect* post-processings. **Figure 15** shows its complete class diagram (class members are hidden for better illustration).

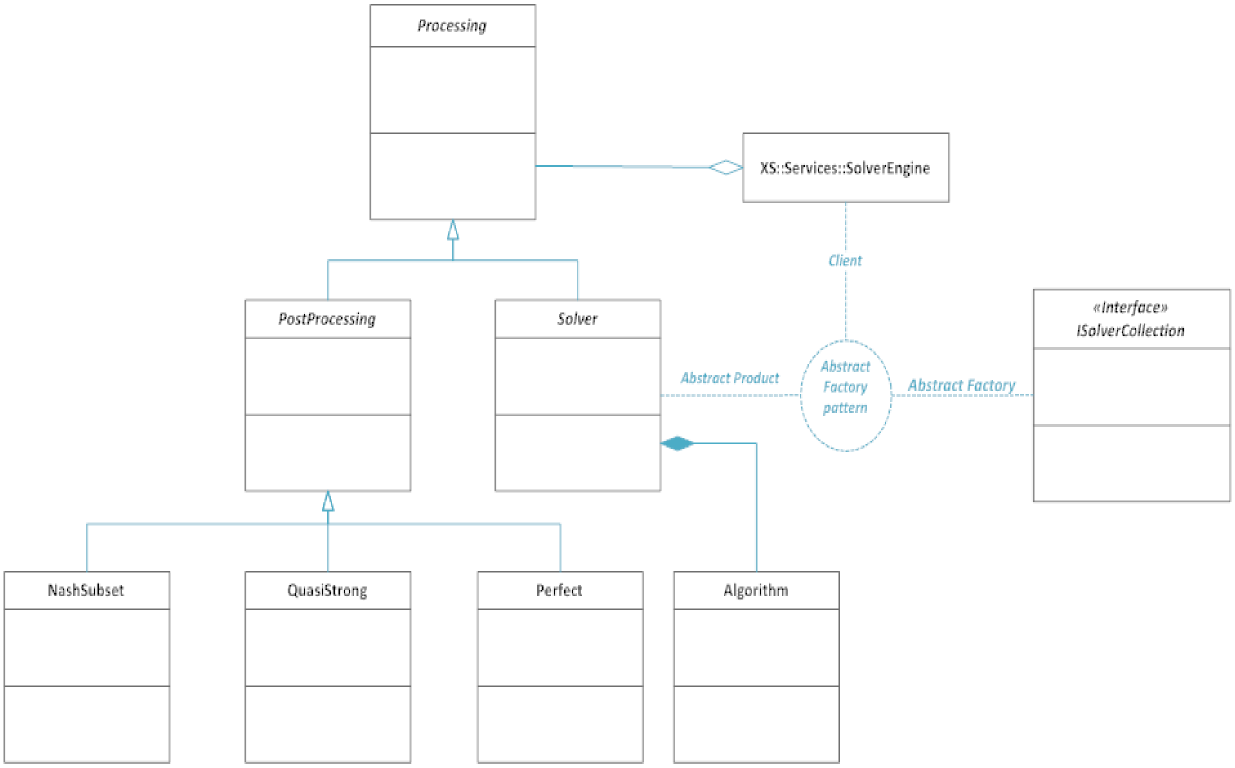

Figure 15: Solver Framework Class diagram

5.5.3.1. Processing class

The *Processing* class is the base class of all processing runnable by the solver engine.

5.5.3.2. Solver class

The *Solver* class is the base class of all solvers runnable by the solver engine. Each solver contains a collection of algorithms, which allow it to solve various types of games.

5.5.3.3. Algorithm class

The *Algorithm* class defines what algorithm a solver must use to solve a game of a given type. The specified algorithm must be contained in the collection of algorithms of the specified solver.

5.5.3.4. Post-processing class

This class represents the base class of all post-processings.

5.5.3.5. NashSubset class

This class represents the *Nash-Subset* post-processing.

5.5.3.6. QuasiStrong class

This class represents the *Quasi-Strong* post-processing.

5.5.3.7. Perfect class

This class represents the *Perfect* post-processing.

5.6. *GERAD Game plugins*

The *GERAD Game Plugins* components are the default game plugins provided alongside the application. They make it possible to create or edit Bimatrix, Sequential and Polymatrix games.

5.6.1. Bimatrix Game plugin

The Bimatrix game plugins allows user to create Bimatrix games. **Figure 16** shows its complete class diagram (class members are hidden for better illustration).

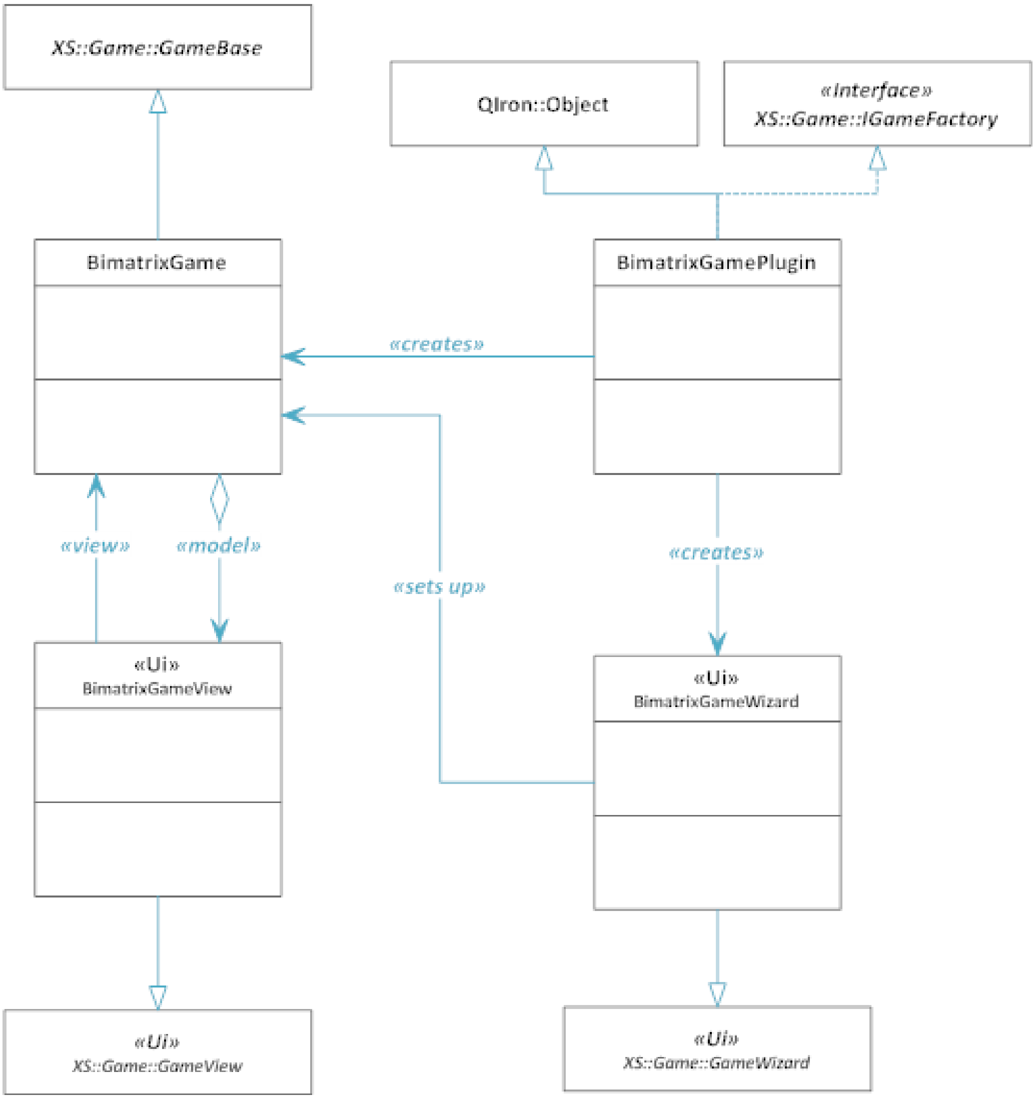

Figure 16: Bimatrix Game plugin Class diagram

5.6.1.1. BimatrixGamePlugin class

The BimatrixGamePlugin class represent the plugin loaded by the application on startup which allows user to create Bimatrix games.

5.6.1.2. BimatrixGame Class

The *BimatrixGame* class represents a Bimatrix game.

5.6.1.3. BimatrixGameView class

The *BimatrixGameView* class represents a graphical representation of a *Bimatrix* game.

5.6.1.4.BimatrixGameWizard class

The *BimatrixGameWizard* class provides a wizard to guide user at the creation of a new Bimatrix game.

5.6.2. Sequential Game plugin

The Sequential game plugin allows user to create Sequential games. The picture below shows its complete class diagram (class members are hidden for better illustration).

5.6.2.1.SequentialGamePlugin class

The SequentialGamePlugin class represent the plugin loaded by the application on startup which allows user to create Sequential games.

5.6.2.2.SequentialGame Class

The *SequentialGame* class represents a Sequential game.

5.6.2.3.SequentialGameView class

The *SequentialGameView* class represents a graphical representation of a *Sequential* game.

5.6.2.4.SequentialGameWizard class

The *SequentialGameWizard* class provides a wizard to guide user at the creation of a new Sequential game.

5.6.3. Polymatrix Game plugin

The Polymatrix game plugin allows user to create Polymatrix games. The picture below shows its complete class diagram (class members are hidden for better illustration).

5.6.3.1.PolymatrixGamePlugin class

The PolymatrixGamePlugin class represent the plugin loaded by the application on startup which allows user to create Polymatrix games.

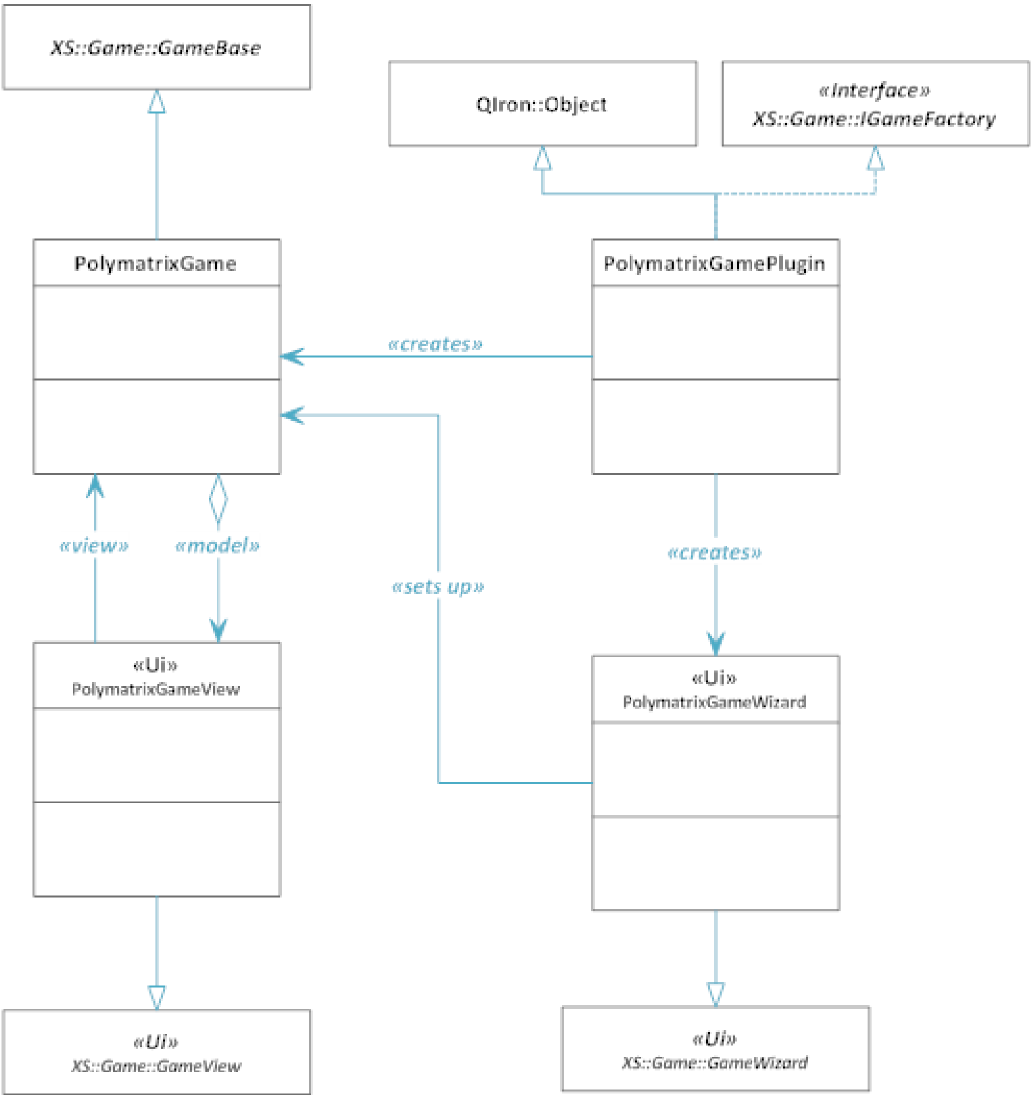

Figure 17: Polymatrix Game plugin Class diagram

5.6.3.2. *PolymatrixGame* Class

The *PolymatrixGame* class represents a Polymatrix game.

5.6.3.3. *PolymatrixGameView class*

The *PolymatrixGameView* class represents a graphical representation of a *Polymatrix* game.

5.6.3.4. PolymatrixGameWizard class

The *PolymatrixGameWizard* class provides a wizard to guide user at the creation of a new Polymatrix game.

6. Conclusion

I this document we presented the different aspects of the XGame Solver software new architecture. We hope that the scientific benefit from our work as it is the case of our previous versions. This new version of the *XGame Solver Software* will be soon available at **http://www.Xgame-Solver.net** for free download by the scientific community.

Acknowledgments

We here thank the ***Deanship of Scientific Research*** at KFUPM for their financial support.

References

- *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked objects*, Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, September 2000.
- *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*, Graig Larman, July 13, 2001.
- *The Unified Modeling Language User Guide*. Addison-Wesley, Ooch, G., Rumbaugh, J. and Jacobson, I. 1999.

KFUPM RESEARCH COMMITTEE

SCHOLARLY OUTCOMES OF A COMPLETED PROJECT

This report must be submitted to the Deanship of Scientific Research before the release of final payments

The information should be brief ***and*** concise. ***It should*** concentrate on ***the*** specific points ***related to the scholarly outcomes of the completed project including*** journal publications, conference publications, students' training, patents, seminars, invited speeches ***and*** other academic-related achievements.

Please provide a concise list of all such achievements.

You are greatly encouraged to directly use (fill in) the formatted Sections below

---

A. PROJECT GENERAL INFORMATION

Project Type:		
Project Title:	AUTOMATIC REFINEMENT OF EQUILIBRIA IN GAME THEORY	
Project Number:	JF100002	
Name of **Principal Investigator**	Dr. Slim Belhaiza	
Department	Mathematics and statistics	
Name(s) of **Co-Investigator**(s) [or Project Consultant(s)]	**1.**	
	2.	
	3.	
Details **Start Date:**	01-12-2009	
Completion Date:	30-11-2010	

	Approved Budget:	58,220 SR

B. DETAILS OF THE SCHOLARLY OUTCOMES

Status: A = Accept/ Published **S** = Submitted **UP** = Under Preparation

● In this Section, list the details as mentioned above (Kindly make sure to include the full details of each item including the dates and the status).			**Status**
I	**Journal Publications**	*XGame Solver Software for Enumeration and Refinement of Equilibria in Game Theory. To be submitted to ACM Transactions on Mathematical Softwares.*	UP
II	**Conference Publications /Presentations**		

B. DETAILS OF THE SCHOLARLY OUTCOMES (continued)

Status: A = Accept/ Published **S** = Submitted **UP** = Under Preparation

● In this Section, list the details as mentioned above (Kindly make sure to include the full details of each item including the dates and the status).	**Status**

III	**Book/Book Chapters**		
IV	**Patents**		
V	**Students' Training**		
VI	**Invited Speeches delivered by the Investigators**		
VII	**Seminars/Talk delivered within the University**		

VIII	**Seminars/Talk delivered outside the University**		
IX	**Others** , Specify	XGame Software Installation Package Online at http://www.Xgame-Solver.com	

Principal Investigator:	Dr. Slim Belhaiza		
Signature:	Slim Belhaiza	**Date:**	03-January-2011