**Complexity Research Initiative for Systemic InstabilitieS**

**FP7-ICT-2011-7-288501-CRISIS**

# Deliverable D5.2
# Integration Plan

| | |
|---|---|
| Author(s) | László Gulyás (AITIA), Tamás Máhr (AITIA), Péter Rieger (AITIA), Richárd Legéndi (AITIA) |
| | |
| Abstract | Our goal in this document was to describe how the simulation models are integrated into the on-line game architecture. To this end, we discussed both the integrated architecture and the integration process. First, we discussed some architectural requirements and presented a brief architectural overview. We then treated functional requirements in the form of use cases, and showed how the required functions are implemented by the different components of the game. We discussed usage cases of players, game owners, and game masters, and showed how the three components, the web portal, the game engine, and the game client realize these functions. Following the discussion of the game components, messages and data, we describe the integration process we apply in developing the game. |

| | | | | | |
|---|---|---|---|---|---|
| **Distribution level** | Public | **Status** | Final | **Version** | 01 |
| **Contractual delivery date** | 31/07/2012 | **Actual delivery date** | | 31/07/2012 | |

The CRISIS project is funded by the European Commission under the Seventh Framework Programme (ICT)

# Table of Contents

# Introduction

One of the goals of the project is to produce a unified simulation model of the macro economy and the financial system. To create this model, we extend the Mason[1] simulation platform into a library with predefined but configurable elements modeling different entities in the economy. Such a library can be used by the researchers to build economic models and experiment with different algorithms. This core library is designed to contain both macroeconomic and financial model elements to allow a more complete simulation of the economy than it was possible before. On this core simulation library and the unified model, two other products of the project relies: the on-line game (developed in WP5), and the economic simulator (developed in WP6). Since the unified model forms the basis of both software, it is critical to deliver and integrate it in time.

A key problem in the development of the integrated simulation library and the unified model is that numerous project partners contribute to it. 3 partners develop the macro-economy models, 5 the financial models, and 2 more is concerned with putting it all together. It is mainly the role of WP8 to ensure the integration of the different contributions into a unified code base, but it is done in close cooperation with WP5 and WP6, as these work-packages heavily depend on the integrated library.

In this document, we describe how the different pieces fit together. We start by outlining the relation of the integrated simulation, the game, and the economic simulator in the next section. Then, this document concentrates on the game, and describes the software architecture in detail, elaborating topics like messages sent between the game clients and the simulation, and the data infrastructure needed for the game. In the end, an overview of the integration process is given, explaining which software tools are used, and how the developers interact with the tools.

## Integrated simulation model, game, and economic simulator

The integrated simulation model, as depicted in Figure 1, is composed of a macro-economic (MABM) and a financial model (FABM). The macro-economic part describes households, and firms, while the financial part describes banks, central banks, and other financial institutions. These entities interact through different markets (deposit, loan, labour, etc.) that can be implemented by different mechanisms like limit-order-book or over-the-counter market. The macro-economic model, the financial model and the market implementations are melted into a common CRISIS base layer, the integrated simulation library that can be used to implement various different economic models. This library is extensible, new model elements can be added in the future to support other complex economic simulations.

[1] http://cs.gmu.edu/~eclab/projects/mason/

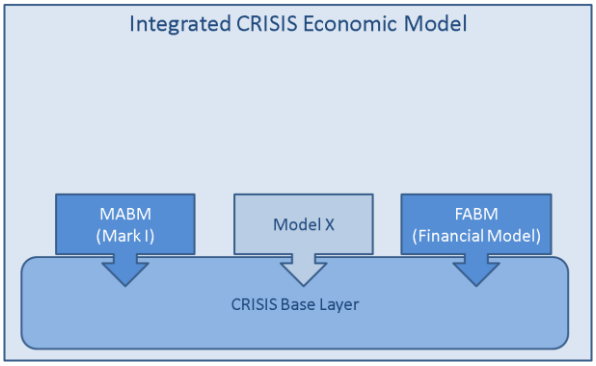


**Figure 1 - The integrated model**

The integrated simulation model is built upon the integrated simulation library. On the integrated model, two other artifacts of the project depends: the game and the economic simulator. How the game is built around the integrated simulation is depicted in Figure 2. In the center of the architecture is a web portal that allows players to find and connect to games, and launches pre-configured simulations on request. The portal is also used by administrators (see the section on Game Master) to upload new versions of the game simulation engine, and to create new games by configuring the simulations. When a game is started, the corresponding game simulation is started either at the same server, or on a remote server. The game clients connect directly to the newly started simulation, and interact with it during the game. Consequently, for each game one simulation is started, and the different games can be based on different game simulation engines with different parameter settings.

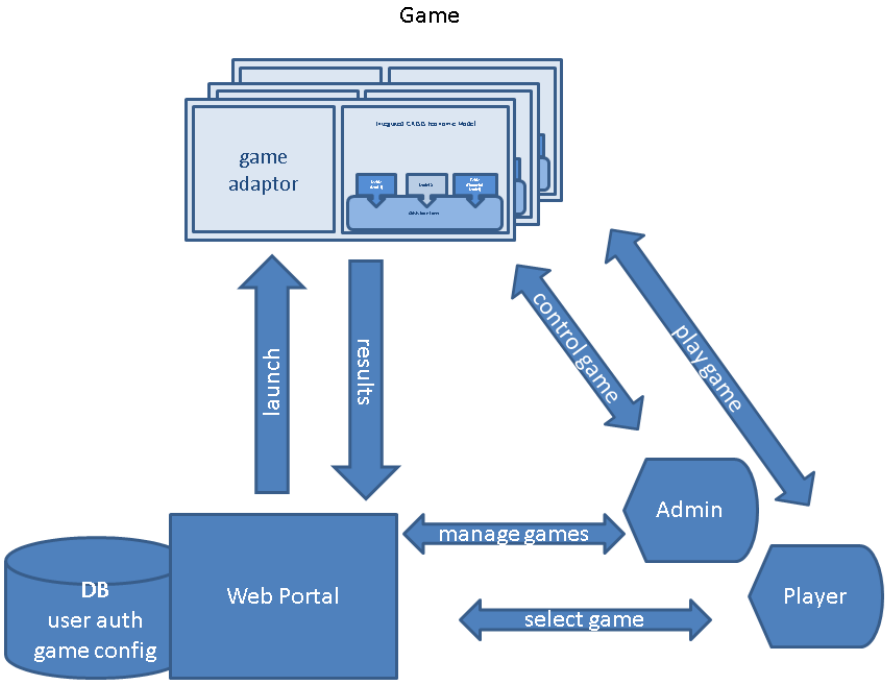


**Figure 2 – Game Overview**

In contrast, the economics simulator is used to explore the behavior of simulation models, therefore (see Figure 3) it starts multiple instances of the same simulation model with different parameter settings. From a simulation model built upon the integrated simulation library, the user can generate a WUI that allows himself or others to set up a parameter-sweep experiment. The parameter-sweep experiment is executed either at the server hosting the WUI, or remotely on some cloud infrastructure. Depending on the available resources, the simulations of a user (the same model with different parameter settings) can run parallel on several different machines. The WUI helps the user to follow the completion of the simulations, and to access the results.

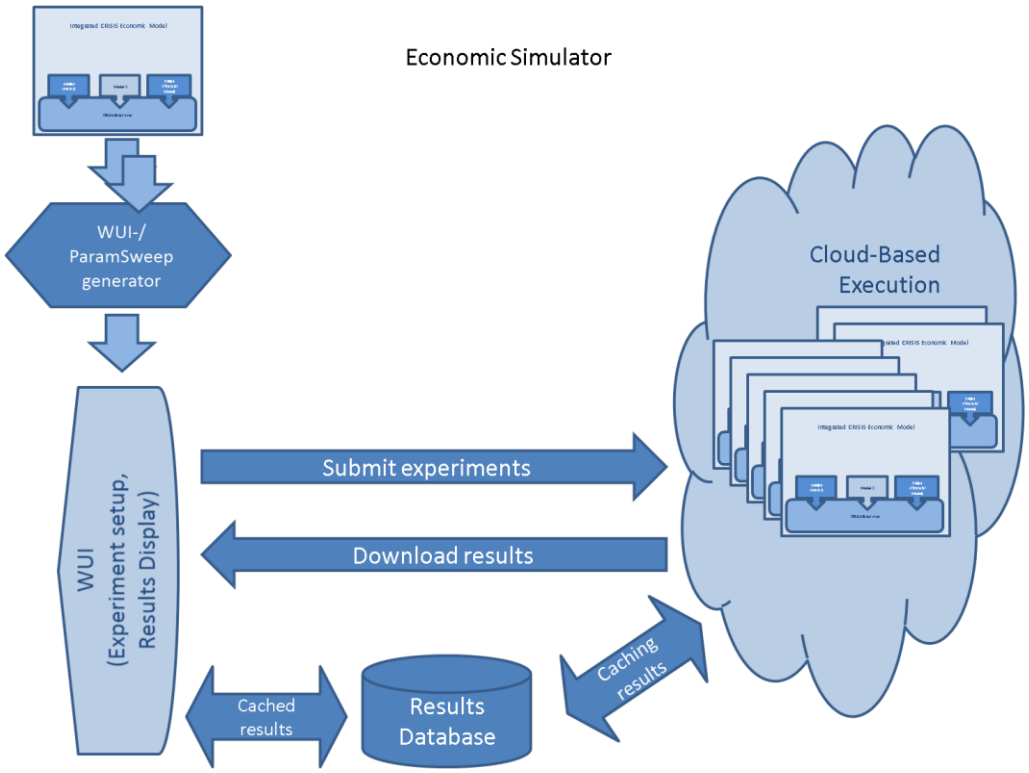


**Figure 3 – Economic simulator overview**

To summarize, both the game and the economic simulator uses the integrated simulation model, or more precisely a simulation model built with the integrated simulation library. The main difference is that while in a sole use-case of the game, a single simulation is started with a given parameter setting and the users/players interact with the simulation while running, in an economic simulator use-case, multiple simulations are started with different parameter settings (a.k.a parameter sweep) in a non-interactive mode.

Following this comparison of the integrated simulation, the on-line game, and the economic simulator, this document elaborates on the integration of the unified simulation model and the on-line game. In the next section, we describe how the simulation integrates into the game, while in later sections we describe the integration process and the tools that are used to ensure a coordinated process.

# Game Integration

There are a few requirements we have to consider during the design of the game. Since in the CRISIS project several different partners develop the integrated simulation model and the game, we need an architecture which allows independent development of these components. Additionally, we may need to handle multiple versions of the simulations; we need to be able to organise/group the simulation models. This can be achieved by a pluggable design, where simulation models can be developed, deployed, and removed as plugins, independently from the rest of the game architecture.

In order to create games, simulation models have to be configured. Similarly to the models themselves, it should be possible to create configurations in multiple versions. Users should be able to create multiple different configurations to the same simulation model, and they should be able to configure any model. This leads to a directory of configurations, each of which represents a game that can be started.

The above discussed requirements relate to administrative or '*game master*' usages of the game. In addition to this, the main usage is of course playing the game. These two roles, the *game master* and the *player*, should be clearly separated. Such a role-based separation of functions helps the design and implementation of the game.

Finally, the last requirement is derived from one of the goals of the game in the project, the behavioral data collection. The simulation models created in the project are configured as a game in order to facilitate data collection on decision making in a simulated financial environment. To live up to this requirement, the game architecture should facilitate the data collection from the simulation, and the storage of the data. The game, however, does not need to provide tools to process and analyse this data.

In order to highlight how the CRISIS game satisfies the above requirement, we introduce the design of the on-line game in the following sections. First, we provide a brief architectural overview to clarify the main components. This is followed by the description of the use cases, which stands for a more elaborate requirement definition. After the use cases, we describe the components of the game in detail, and list the messages exchanged between the components. The section ends with the definition of the data structures required by the game in order to store game configurations, results, user scores, etc.

## Architectural Overview

As already outlined in the previous section, the on-line game consists of three main components: the browser-based client, the web portal with a database, and the game simulation engine (see Figure 2). The *game client* is an application running in a browser. It is a *rich internet application* that facilitates playing the game. It can be run in two modes: owner and player mode. The owner user, who created the game, can start, pause, or stop (control) the game in addition to playing it as normal players.

Next to playing the game, the browser is used to access the *web portal* to list, search, or manage games, as well as to review past performance of the players. If the user logs in as an administrator, he/she can upload new simulation code, configure simulations to create games, publish games to make it available for the players to start game instances, or alternatively un-publish and remove games. If a user is not logged in, or logged in as a player, he/she can browse or search games, join and play games, and review his/her scores in past games. While browsing the portal, the game client is launched in the browser when the user starts or joins a running game instance.

When a user starts a new game instance, a game engine is launched. The launched game engine is run independently from the web portal under a different URL, and the browser clients are redirected to the new URL. The game engine is composed of two parts: the integrated simulator and the game adaptor. The integrated simulator is the unified macro-economic and financial simulation model developed in WP8. The game adaptor is a custom made library that controls the simulation and handles the user interactions mediated by the game clients. When a game engine is started, it receives all configuration data in files. When the game is over, the game engine sends the results, logs, saved states back to the web portal. If the players are logged in (i.e. registered), their results are accounted for in their records.

The above outlined architecture is based on the idea, that the web portal should be able to handle numerous simultaneous visitors, therefore time and resource consuming operations, i.e. the game-engine executions have the option to run on a separate infrastructure, independently from the portal. This ensures that users can browse the portal without it being slow to respond, and at the same time, games can be played simultaneously on shared or individual hardware depending on the resource requirements of the games.

## Use cases

Following the general overview presented above, in this section we describe the main use cases of the on-line game. To discuss the use cases, we use so called *use-case diagrams*. The definition of a use-case diagram on the Wikipedia[2] is:

*,,A use case diagram at its simplest is a graphical representation of a user's interaction with the system and depicting the specifications of a use case. A use case diagram can portray the different types of users of a system and the various ways that they interact with the system. This type of diagram is typically used in conjunction with the textual use case and will often be accompanied by other types of diagrams as well."*

In the following sections, we discuss the various use cases of the game grouped by the different roles. The three main roles, discussed in this order, are the player, the game owner, and the game master.

### Player

Players are the central actors in the on-line game. The use cases related to players are separated into two groups in Figure 4. On the left, usages of the web portal, while on the right, usages of the game engine are highlighted. As discussed earlier, on the web portal, players should be able to browse, search and *find* published games and running game instances. If they find a running game instance they like, they should be able to *join* it, otherwise they should be able to *create* a new instance and invite people to play it, or publish it as open access. In addition to playing games, the players should have a forum page where they can discuss various aspects of the game, or ask for help. Finally, the website should support logged in players in reviewing their scores and the history of previous games.
In the game, players have four main use cases. Since players control banks, their activities are bound to the four options banks have in the game. They can give loans to firms (*commercial loans*) or banks (*interbank loans*), they can attract new deposits by *adjusting the interest of deposits*, and they can *trade shares* of firms. As we have seen in previous sections, the game engine utilizes the integrated simulation model to provide the environment for these four activities, and it uses the game adaptor to facilitate the interaction of the players and the simulation.

[2] http://en.wikipedia.org/wiki/Use_Case_Diagram

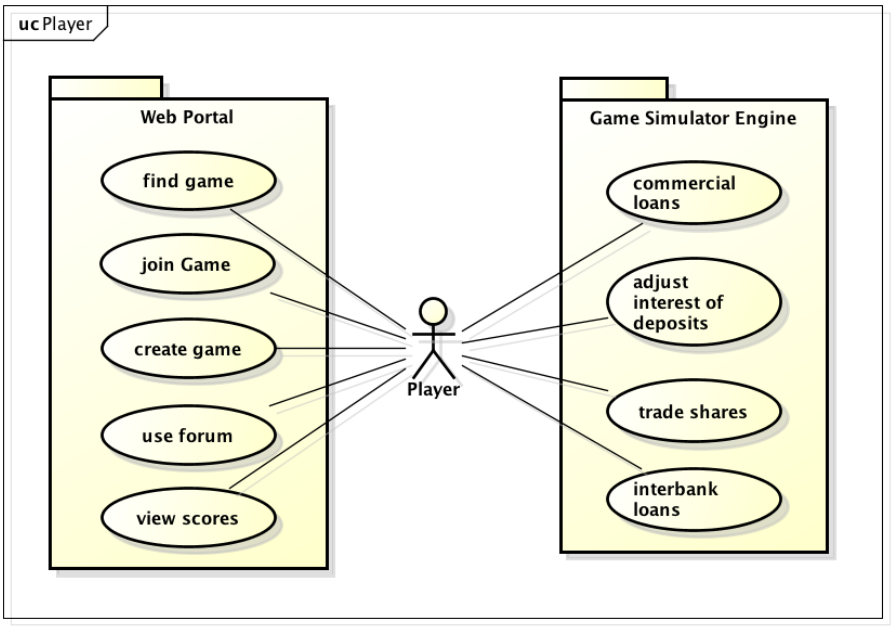


**Figure 4 – Player use cases**

## Game Owner

When players browse the portal looking for a running game instance to join, they might not find any suitable running game, or they may want to explicitly invite fellow players for a game. In such cases, players have the option to create a new game instances. They do this by selecting the appropriate pre-configured game type, and launching a new game engine. The player who creates a new game instance in this way becomes the *game owner*. The game owner participates in the game in a similar manner as other players, but he/she has a few extra options. These use cases are listed in Figure 5.

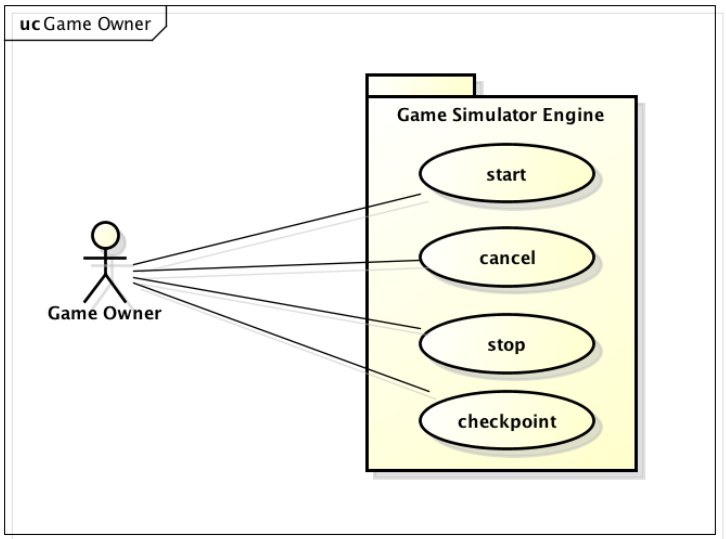


**Figure 5 – Game owner use cases**

After starting a new game instance, the game owner is redirected to the URL of the game, and the system is stalled to wait for the other players to join. In this state, the game owner has two extra options of *starting* or *cancelling* the game. If he/she acts to start, the game is started with the players already joined. If he/she cancels the game, all already joined player is directed back to the game portal, the game engine is stopped and the game URL becomes invalid.
During the game, the game owner has two additional options to *stop* or to *checkpoint* the game. If he/she stops the game, then the simulation is stopped, the final state is saved, the game engine is removed, and the players are redirected back to the portal. If he/she checkpoints the simulation, then the current state is saved to allow a rewind from a later stage. This feature enables start further experiments from an intresting simulation state, or to simply pause (interrupt) the experiment and continue it later.
Players and game owners manipulate and play games that are based on preconfigured simulation models. Game masters are responsible for creating such preconfigured game versions.

### Game Master

Game masters are in control of the game types that are available on the web portal. First of all, game masters should be able to handle a repository of *uploaded simulation models*. This includes uploading, organising, and removing them. The simulation models can then be used to create games by *configuring the simulations*. By providing configurations to the models, game masters can define the initial state of a simulation as well as the behaviors of the models. Such preconfigured simulations can be seen as game definitions that can be instantiated to run a game. Before the game definitions can be accessed on the portal, they have to be *published* by the game master who provided the configuration. Game masters can choose to publish the games as public, i.e. available to non-logged in users, or private which can be accessed only by logged in users. Finally, game masters should be able to remove published games from the portal to disallow the creation of further game instances. These use cases are listed in Figure 6.

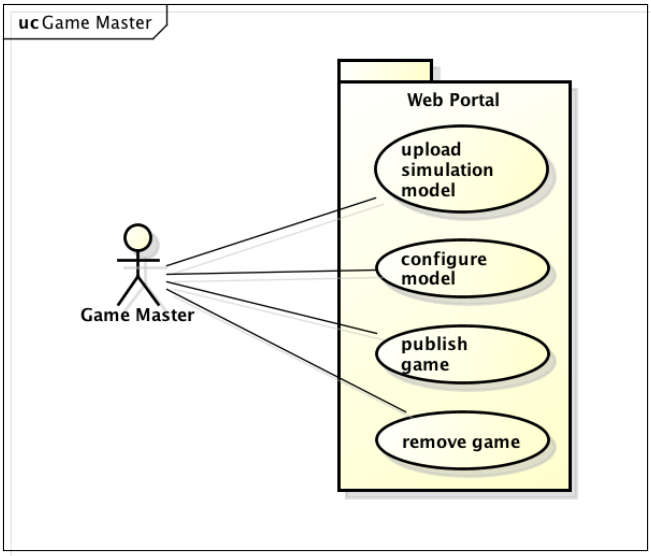


**Figure 6 – Game master use cases**

## Components

After discussing the different use cases of the game, i.e. how players will interact with the game, in this section we describe the main components of the system. For each component, we define the functions it has to implement, and we discuss the technologies we use to realize the components. This section should shed some light on how exactly the agent-based simulation is integrated into the game, and made available for the players to interact with.

The use cases have already revealed two of the components, the web portal and the game engine. These two components provide the background for the game. The third component, the game client is the front-end. It is the game client the players directly interact with during the game, and it is its role to provide a graphical experience to the player. While in the following discussions we concentrate on how the different components facilitate the interaction between the simulation and the players, we also describe the interaction between the two back-end components, the web portal and the game engine,

### Web Portal

The web portal serves as a meeting point for the player community, it provides the interface where games can be created and accessed, and it is also a communication medium between the game maintainers and the community. The portal is based on a standard Liferay[3] portal which provides much of the community-portal functionalities. This is extended into a game portal by several custom designed modules implemented specifically for the on-line game within WP5. The Liferay portal is a standard (JSR-286[4]) portlet container meaning that it can host several independent or with each-other collaborating applications, which are called portlets. In the following paragraphs, we discuss the components that are used to implement the on-line game. First, we go through the modules provided by the standard Liferay distribution, then we introduce those implemented specifically for the game.

[3] http://www.liferay.com

[4] See JSR 286: Portlet Specification 2.0, http://jcp.org/en/jsr/detail?id=286

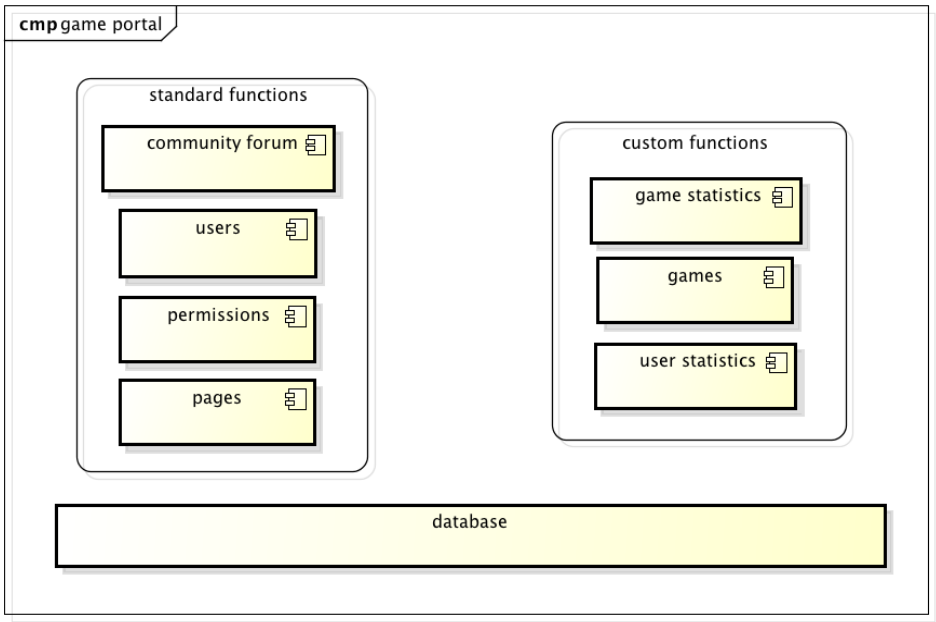


**Figure 7 – Game portal functionalities**

As depicted on Figure 7, much of the functionalities in the portal is built upon a database. This database stores all data required by the portal, such as the users' e-mail address and password, or the pages of the portal. In addition to the standard Liferay data-structures, the database is extended to support the custom components.

As Figure 7 highlights, the functions of the game portal are partitioned into two groups. On the left, the functionalities that are part of the standard Liferay distribution and are used in the game portal are listed. Of primary importance here is the *user handling*. The portal allows visitors to read public pages without registration. Considerable amount of content and even games can be provided by the portal as public, allowing people browsing or searching the web to find the site and the games. It is also possible, however, to register on the portal. Registered users can access private pages that contain, for example, private games, their scores, or other non-public content. At registration, various data might be requested from the user which are stored in the database and can be used later to create statistics about the players and the games. Finally, the user-handling module facilitates authenticating the users. To log in, it is possible to use the registered e-mail address and password, or use an OpenID provided by some OpenID provider, such as Google, Yahoo! or Aol.

Another basic functionality of the Liferay portal is to serve *web pages*. Web pages can contain static or dynamic data in the form of portlets (web applications embedded in a portal). The configuration of each web page is stored in the database, and can be customized on a user or group level. Similarly, the *permissions* module allows us to determine user and group permissions for pages. Using the above two modules, we define public pages that are viewable to anybody browsing the site (containing general descriptions, public games, forums in read-only mode etc.), pages that are displayed to users logged in (with private games, game statistics, shoutbox, forums in read-write mode), and pages that are displayed to users logged in and are different for each such user (displaying user data, user statistics, games by invitation, etc.). The user handling, permissions, and web page handling components offer several extra

functionalities (e.g. public personal pages, user customizable pages, user-group pages) which will not be utilized in the beginning, but can be exploited later on.

In addition to the above-discussed basic portal functionalities, we use the standard Liferay forum (message boards) portlet to facilitate community discussions. Forum messages can be displayed read-only to any user, but only logged in users can post new messages or replies. Forum discussions are organized into categories. We create a category for each game published on the site: the “Never-ending” game with subcategories for each world, and the challenge games (e.g. “1973 Oil Crisis”, “2008 Financial Crisis”, etc.). Additionally, we create a “General Discussion” category with subcategories such as “Bank Strategies Discussion” or “FAQ”. Finally, we create a “Technical Support” category to help players with technical problems. The general principle is that only site maintainers can create discussion categories, but any logged in user can initiate new discussions within the categories.

On the right-hand side of Figure 7, a list of custom functionalities is presented. These functionalities turn the standard Liferay portal into a game portal for the CRISIS game. Similar to the standard functions, these rely on the database. In the center of the new functions is the *game handling*. The game handling module enables site maintainers to upload new simulation models and to create new games by configuring these models on line. The games can be published as a public game, which allows players to play anonymously, or as a private game, which can only be played by logged in users. Once a game configuration is published, it is displayed on the site where users can browse and search the games and create game instances. A game instance is a running version of a game and it is owned by the user who created it. Any user can instantiate a game. When a game is instantiated, the game engine is started and made available under a unique URL. This URL can be published by the owner on the site to allow anybody to join the game, or he/she can invite players by sending the URL to the right persons. The game handling module provides portlets that lists game URLs on the site for general access, and portlets that show the invitation-only games to the invited users on their private pages. In addition to these portlets that facilitate the joining of users to games, a ‘My Games’ portlet is also being developed that lists past and currently running games of a user (only if logged in) to allow them to re-connect to a game if the connection was somehow broken.

*Statistics Module*

Related to the game-handling module, two other modules are developed for the game portal. These are the *user statistics* and the *game statistics* modules. These display statistics after a game is finished. The game statistics present the end result of a given game, as well as the all time scores, e.g.:

- Most Experienced Bankers (ordered by total score)
- Most "Hazardous" Bankers (ordered by average leverage ratio)
- Most Successful Investors (ordered by profit)

At the same time, user statistics present the history of the player and his/her overall score and ranking.

- Games played: Number of games the user played so far
- Highest score: Best game score the player achieved over all games
- Worst score: Worst game score the player achieved over all games
- Total score: Total score of the player over all games (experience points)
- Best rank: Best game rank the player achieved over all games
- Average rank: Average game rank the player achieved over all games

The above discussed custom modules offer a complete set of functions that turn the standard Liferay portal into a game portal. They extend the standard functionalities with the handling of games and game instances, user scores and game histories. An important property of this solution is that it is game agnostic, the same portlets can be used to implement a game portal for any kind of game.
In this section, we discussed the web portal as one of the three components that forms the on-line game. Another component is the simulation back-end that is executed during a game to provide the world in which the game is played. This component is described in the next section.

### Simulation back-end

The simulation back-end, a.k.a. the game engine, is started by the web portal when a user initiates a new game. The game engine is deployed as a web application in a servlet container (e.g. Apache Tomcat), which can be the same container than the one the web portal is using, or, if the resources are running short, another one running on a different machine. The lifecycle of the game engine is handled by the servlet container: it starts after it is deployed in the servlet container, and it stops when it is undeployed.
The game engine is primarily composed of two modules (see Figure 8). The *integrated simulation* (on the left) is based on the simulation model produced in WP8. For the most part, it is exactly the same as the integrated MABM+FABM model, with a few things adapted to the game (e.g. synchronous vs asynchronous event scheduling). The *game adaptor* component (on the right) is responsible for integrating the simulation into the game. It handles the initialization and the controlling (starting up, stopping, etc.) of the simulation, and all the in-game communication between the simulation and the players.

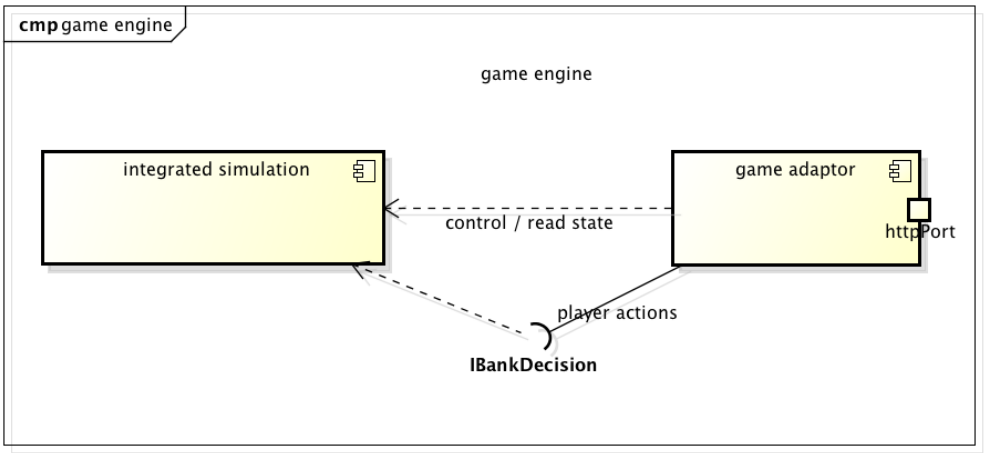


**Figure 8 – Game engine component diagram**

To be able to communicate with the web-based clients, the game adaptor opens an http port and waits for incoming requests (as shown in Figure 9). Most requests are simple queries or actions to be performed on behalf of the player. These requests are answered instantly. Some are, however, so called *long poll* requests that are queued and not answered immediately. These requests are sent by the game clients to enable the game engine to report changes in the game whenever they happen. For optimal resource usage, all incoming http request are handled by the game adaptor using an asynchronous (non-blocking) API[5]

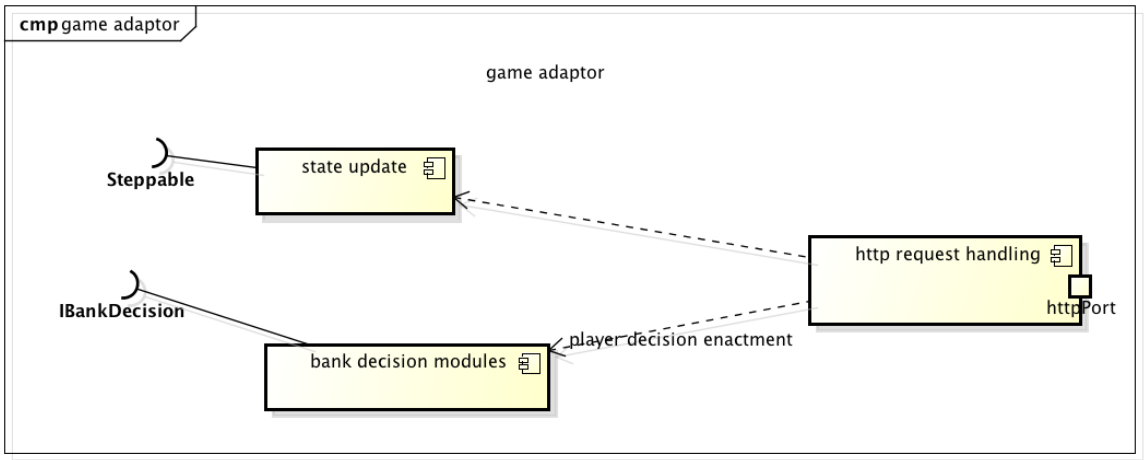


**Figure 9 – Game adaptor component diagram**

To be able to relay the players' decisions to the simulation, the game adapter implements the *IBankDecision* interface declared by the simulation. The simulation model employs this interface to allow for pluggable decision modules to be developed for banks. That is, whenever the bank has to perform an action (make an offer on some market), the decision module that is actually plugged in the bank agent is consulted. The game adaptor implements such a decision module by simply forwarding the players' decisions when it is requested. To be able to provide decisions

[5] http://tomcat.apache.org/tomcat-7.0-doc/aio.html

on behalf of each player, the game adaptor has a list of such *IBankDecision* implementations, one of each bank-player pair.
In addition to enacting the players' decision, the game adaptor has to keep the game clients up-to-date regarding the simulation state. This is achieved by the *state update* module that hooks into the scheduler of the simulation and receives calls after each turn. At the end of each turn, the new state of the simulation can be observed and forwarded to the game clients using the *long poll* requests queued by the http-request handling module.
In the previous sections, we discussed two components of the on-line game: the web portal and the game engine. In the next section, the last piece, the game client is described to complement the discussion of the whole system.

### Game Client

The game client is the interface through which a player accesses a game. It runs as a browser application on the computer of the player, and communicates with the game engine using a standard http-based protocol. In order to decide on the technology used to implement the client, we considered a few different factors. The game client should

- provide easy access to the game for the players,
- not perform heavy computations,
- not display complicated graphics.

Given the above requirements, a browser-based client seems to fit our needs. It trivially integrates with the web portal (players will easily assess the games), the users do not need to install anything, and modern browsers can handle light computational tasks and simple graphics. Even after selecting the browser as platform, there are multiple options for implementing the game client. In Table 1 we compare the properties of Flash[6] and pure HTML5 applications.

| ***Technology*** | ***Flash*** | ***HTML5*** |
|---|---|---|
| Asynchronous HTTPRequest | + | + |
| Animations | + | + |
| Mobile Compatibility | - | + |
| Source code protection | + | - |

**Table 1 – Technology comparison**

Both technologies can handle asynchronous HTTP requests and animations. In terms of mobile compatibility HTML5 is better positioned, since Flash is not supported on all mobile platforms. In contrast, from a security point of view HTML5 has the disadvantage that the code (being simple HTML and javascript code) is easily readable and manipulable by the users. Although this comparison more or less yields a draw, we prefer HTML5 over Flash because it is a new and

[6] http://www.adobe.com/products/flashplayer.html

upcoming standard spreading rapidly on the net, while Flash is an old technology which seems to be close to the end of its lifecycle even according to Adobe.
Another factor in deciding over the technology to use was the ease of developing applications using that technology. Table 2 summarizes the important development environment (IDE) features we checked for the Adobe Flash Builder, and an HTML5 development tool, JetBrains Webstorm.

| ***IDE Features*** | ***Adobe Flash Builder (Flash)*** | ***JetBrains WebStorm (HTML5)*** |
|---|---|---|
| Code completion | + | + |
| Debugging | + | + |
| Refactoring | + | + |
| Syntax Highlighting | + | + |
| Unit Testing | + | + |

**Table 2 – IDE Comparison**

This comparison (on a very high level) showed no difference between the development tools of Flash or HTML5 applications. In accordance with the previous discussions, we chose to develop the game client as a browser-based application in HTML5.
As a consequence, the game client is developed as a DHTML application which uses HTML5 to display graphics and controls to the user, and manipulates the DOM in the browser to reflect changes in the game. The application has three main screens it presents to the players in different situations (see Figure 10).

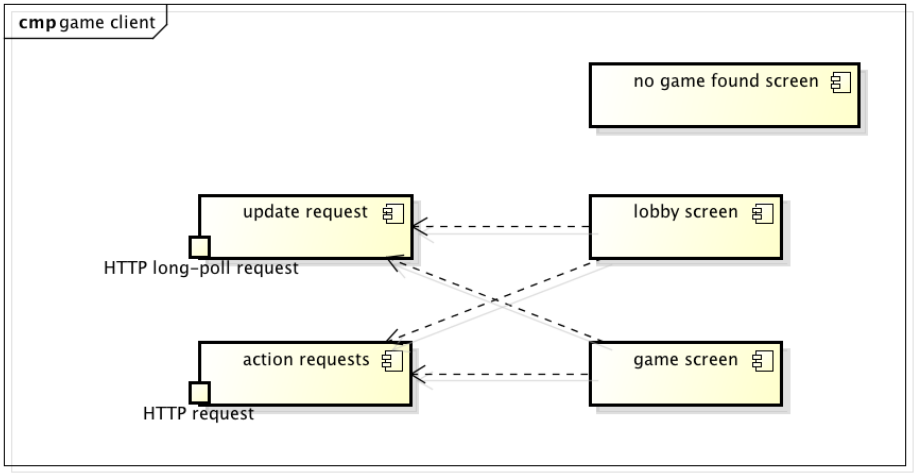


**Figure 10 – Game client components**

When a player connects to a game by opening the URL of the game, the *lobby screen* is presented first. This screen lists the users already connected to the game, shows the number of further users who should join, optionally provides an interface for chatting, and it allows the players to set certain parameters should the game allow for player-specific parameters. The owner of the game has two buttons on this screen, one to start and another one to cancel the game. These buttons activate the *action-requests* component which is responsible for sending the appropriate http requests to the game engine. Parallel to displaying the lobby screen, the *update-request* component sends a 'long poll' request to the game engine to allow it to send data any time. When the owner starts the game, and the appropriate action request is sent to the game engine, the game engine can notify the clients to switch to the *game screen* using these 'long poll' requests from the clients.

The game screen displays the game to the user, and it operates in a similar manner to the lobby screen. Whenever the user makes an action that has to be communicated to the game engine, the action-request component is used to send the request. Parallel to this, the update request screen sends a 'long poll' request to the game engine, which is used to communicate changing game states back to the client. When such a state update is received, the game screen is changed by manipulating the HTML DOM, and a new 'long poll' request is sent immediately to the game engine. In this way, there is always a 'long poll' request sent to the game engine per client, which allows the game engine to keep the clients up-to-date.

The last screen, the *no game found screen* is displayed to the player if he/she followed an URL that has no game associated to it. It might happen that the URL was accidentally modified while being sent to the players, or the game has already been canceled by the time a player tries to join. This screen provides no possible actions to the user, his/her only option is to close the window and/or go back to the portal.

In this section, we discussed the different components that play a role in the general architecture of the on-line game. We presented the functionalities of the game clients, the web portal, and the game engine. In the following section, we list all the messages that are possibly exchanged between the game clients, the web portal and the game engine during the set-up phase and also during the running of the game.

## Messages

As described in the previous sections, the browser-based game client communicates with the web portal and the game engine using http requests. In the following sections, we first describe the general structure of the API, which is followed by the concrete requests. Among the concrete requests, the first group of messages discussed, the control messages, are sent to the web portal to manage game definitions, and to the game to control it. The rest of the sections describe messages sent from the game client directly to the game engine during the game. These messages are grouped by the user screens they are activated on.

### General Message structure

The REST Endpoint URL sample is:

```
http://api.crisis-ecomonics.eu/game/{instance_id}/
```

Request Format:

```
EndpointURL/?method={method_name}[&{arg1}={val1}[&{arg2}={val2}[...]]]
```

Status Codes:

| | |
|---|---|
| 0 | Success Code |
| 0: OK | Successful |
| 1-49 | System Specific Codes |
| 5: Service currently unavailable | The requested service is temporarily unavailable. |
| 12: Method "xxx" not found | The requested method was not found. |
| 50-99 | Message Specific Codes |
| see more at messages | - |
| 100-599 | HTML Status Codes (See more at RFC2616 sec. 10[7]) |
| 100-199: | Informational |
| 200-299: | Successful |
| 300-399: | Redirection |
| 400-499: | Client Error |
| 500-599: | Server Error |
| 666 | Unknown |
| 666: Unknown Error | Something bad happened and the reason is unknown. |

[7] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

### Test

The following methods are used only for testing purposes.

*crisis.test.null (GET)*

Caller may verify if it is able contact the server. There is no response to this call (except the success status code).

| Method name | Description |
|---|---|
| crisis.test.null | Null test |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| `This method has no specific response - It returns an empty success response if it completes without error.` |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.test.hello (GET)*

Caller may verify if the server is able to return proper responses.

| Method name | Description |
|---|---|
| crisis.test.hello | A testing method which sends back the name parameter in the response. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| name | The name to say hello to. | String | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| name | The response value is identical to the input value. | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <test>
    <hello name="world"/>
  </test>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

### Control

The following control interface is provided over the simulation.

*crisis.game.init (POST)*

This method starts a pre-configured game instance. The *gameID* refers to the configured game that consists of a base model, the initial parameters, and the player events if the ID is a checkpoint. During the initialization process, the required simulation and configuration is downloaded to an execution server, and it is started up. After this bootstrapping, the simulation (a.k.a. game engine) waits for client connections. The sender of this message becomes the *game owner*.

| Method name | Description |
|---|---|
| crisis.game.init | Initializes the game instance. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| gameID | Unique Game ID | int | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid gameID | Game not found with this ID |

*crisis.game.start (POST)*

This method, posted by the game owner, starts the game instance. This action triggered after all the players joined to the game, or when the game is in a paused state.

| Method name | Description |
|---|---|
| crisis.game.start | Starts / Resumes the game instance. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.stop (POST)*

This method, sent by the game owner, stops the game instance. After this call, the game sends every information regarding the final game state to the web portal, as well as the players' actions log to facilitate the replay of the game. After this message is sent, the clients are redirected to the web portal, and the simulation is terminated.

| Method name | Description |
|---|---|
| crisis.game.stop | This method stops the game and tries to close everything in a correct state. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.kill (POST)*

The game owner can kill the game instance using this method (i.e., interrupt it immediately). After this call, the game instance disappears. The downloaded files are deleted, and the virtual machine closes.

**WARNING:** Every unsaved information will be lost.

| Method name | Description |
|---|---|
| crisis.game.kill | This method clears everything up. This should be the last command to the game instance. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.pause (POST)*

This method, posted by the game owner, pauses the game instance. The game instance goes to a paused state. In this state, all events are dropped from the users, the in-game time is stopped, and all agent states / properties remain unchanged. This method is useful for *Game Masters*, when they want to point out something in the game.

| Method name | Description |
|---|---|
| crisis.game.pause | This method pauses the game. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.checkpoint (POST)*

This method creates a checkpoint for the game instance. The entire simulation state is saved and sent to the web portal. These checkpoints can be used for future games as initial states. The checkpoints can be selected just like the base models to start a game. The visibility (private / public) will be the same as the current game itself.

| Method name | Description |
|---|---|
| crisis.game.checkpoint | Creates a checkpoint. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.rewind (POST)*

This method rewinds the game to the given check point, and pauses the game instance.
**WARNING:** After the game will start again, every information about the current game state in memory, following the checkpoint, will be lost.

| Method name | Description |
|---|---|
| crisis.game.rewind | Reverts the game to the state of the given checkpoint. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| checkPointID | The ID of the checkpoint to which we want to revert the experiment. | Date | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| - |

| Message Specific Status Codes | Description |
|---|---|
| 50: Incorrect checkPointID | The requested checkpoint is not available. |

### UI Actions - Player Info

*crisis.player.getPlayerInfo (GET)*

This method is called by every client, and returns basic information about the player to display on the graphical interface.

| Method name | Description |
|---|---|
| crisis.player.getPlayerInfo | This method sends back the balance, profitability, return on equity and the risk of the player. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| playerID | Unique Player ID | int | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| balance | The balance of the requested player | double | yes |
| profitability | The profitability of the requested player | double | yes |
| roe | The Return on Equity of the requested player | double | yes |
| risk | The risk of the requested player | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <player>
    <getPlayerInfo balance="18735262.75" profitability="246401.72" roe="14.35"
    risk="12.34"/>
  </player>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: Incorrect playerID | The requested playerID is not available. |

*crisis.game.getDate (GET)*

On every screen there is a calendar to show the in-game date. This virtuale time goes faster in the game than in the real world, that's why we need a method to know the correct time.

| Method name | Description |
|---|---|
| crisis.game.getDate | This method sends back the in-game Date. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| date | The in-game "now" date in the following format: YYYY-MM-DD | Date | yes |

Example Response

```
<rsp code="0" msg="OK">
  <game>
    <getDate date="2012-06-22"/>
  </game>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

### UI Actions - Deposit Screen

On the deposit screen, the player can set the interest rate to pay for clients on their deposits. To facilitate this decision, the game client collects and displays information about the economy (unemployment, GDP, etc.) and the deposit history of the player's bank. A sketch of the deposit screen is depicted in Figure 11. The screen displays various information to the player, such as the bank's balance, its current deposit cost, the current date, some charts on the general state of the economy, and the bank's deposit history. At the bottom of the page, the user can change the offered interest rate for deposits by the slider, or reset the interest rate to the current value, or submit the the value set on the slider. Some widgets on the sketch are numbered, and in the top-right corner a legend shows which messages are invoked to retrieve the corresponding information, or to send the corresponding message. For example, when the *Submit* button is pressed, the *game.setDepositInterestRate* message is sent to the game engine to communicate the newly selected interest rate for deposits in the player's bank.

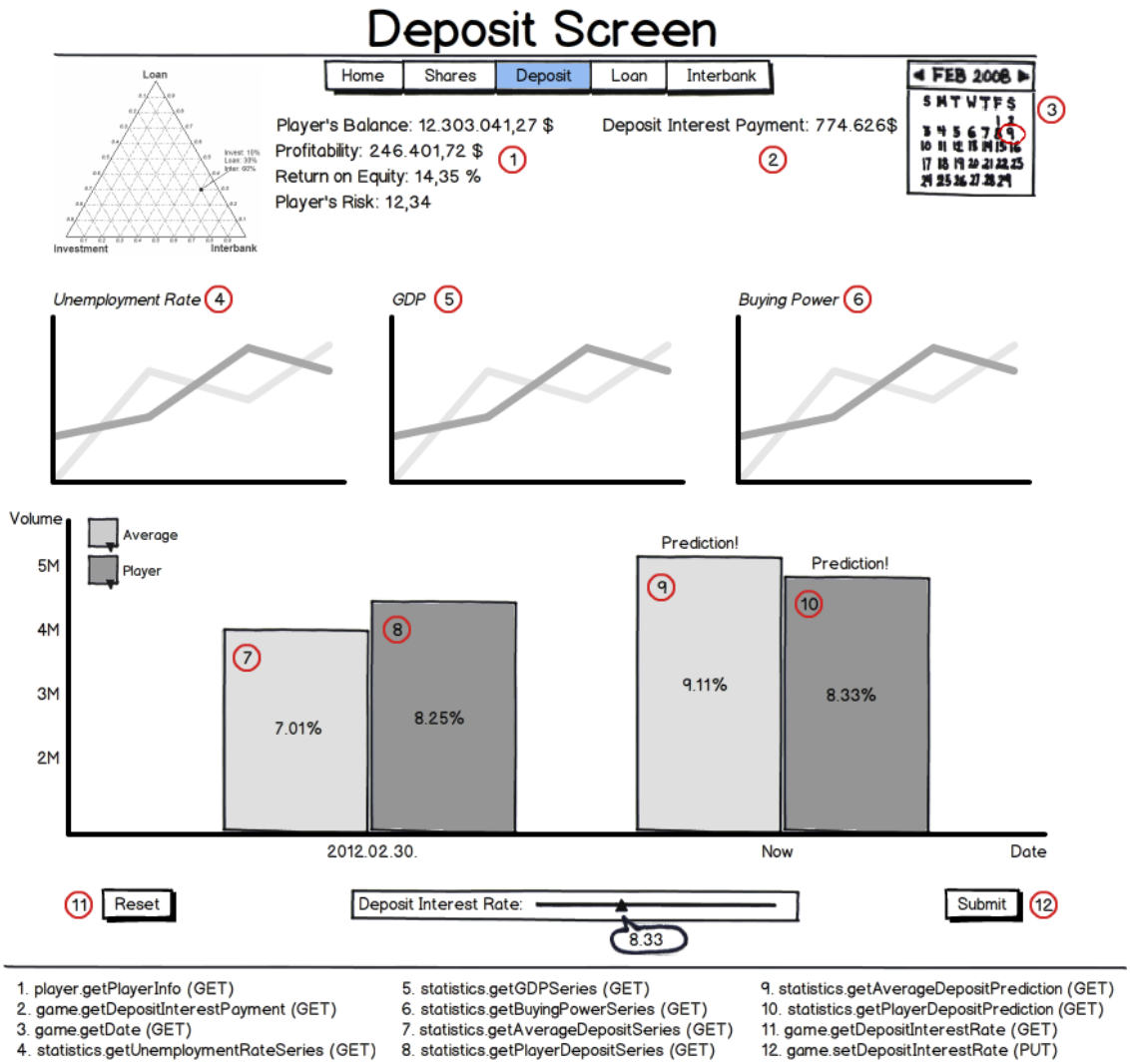


**Figure 11 – Visuals and controls on the deposit screen**

*crisis.game.getDepositInterestPayment (GET)*

This method calculates the interest of deposits the current player needs to pay in the current turn.

| Method name | Description |
|---|---|
| crisis.game.getDepositInterestPayment | This method gives back how much interest you have to pay for the deposits. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| payment | The value of the obligation what the player has to pay for the depositors in the end of current turn. | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <game>
    <getDepositInterestPayment payment="842381"/>
  </game>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.game.setDepositInterestRate (PUT)*

This message communicates the new deposit interest value to the game engine. When the message is received, a new order is submitted to the deposit market, or the currently submitted order is modified. The price of this deposit order is set to the *interestRate* parameter of the message, and the size of the order is set to infinity. If the order is successfully submitted a '0', otherwise an error code is returned.

| Method name | Description |
|---|---|
| crisis.game.setDepositInterestRate | This method updates the deposit interest rate value of the player. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| interestRate | The desired deposit rate adjustment | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| <rsp code="0" msg="OK"><br></rsp> |

| Message Specific Status Codes | Description |
|---|---|
| 50: Invalid argument value | The value of the interest rate must be within 0 and 30. |

*crisis.statistics.getUnemploymentRateSeries (GET)*

The historical data of the unemployment rate indicates the health of the economy. This factor may help players to make decisions.

| Method name | Description |
|---|---|
| crisis.statistics.getUnemploymentRateSeries | This method sends back a time series of Unemployment Rate. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles, represents the unemployment rate in the given period | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getUnemploymentRateSeries>
      <series>
        <value>8.453</value>
        <value>8.512</value>
        <value>8.389</value>
        <value>8.234</value>
        <value>8.109</value>
        <value>7.982</value>
        <value>8.123</value>
      </series>
      <freq>month</freq>
    </getUnemploymentRateSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |

*crisis.statistics.getGDPSeries (GET)*

The historical data of GDP signals the growth and recession of the economy and tends to translate to an increase or decrease in productivity.

| Method name | Description |
|---|---|
| crisis.statistics.getGDPSeries | This method sends back a time series of Gross Domestic Product. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles, represents the GDP on the given period. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getGDPSeries>
      <series>
        <value>28596</value>
        <value>28650</value>
        <value>28902</value>
        <value>29123</value>
        <value>28744</value>
        <value>28744</value>
        <value>28936</value>
      </series>
      <freq>month</freq>
    </getGDPSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |

*crisis.statistics.getBuyingPowerSeries (GET)*

The historical data of buying power (sometimes referred to as purchasing power) shows how the cash reserves of households changed.

| Method name | Description |
|---|---|
| crisis.statistics.getBuyingPowerSeries | This method sends back a time series of the households buying power. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles, represents the buying power on the given period. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getBuyingPowerSeries>
      <series>
        <value>8.453</value>
        <value>8.512</value>
        <value>8.389</value>
        <value>8.234</value>
        <value>8.109</value>
        <value>7.982</value>
        <value>8.123</value>
      </series>
      <freq>month</freq>
    </getBuyingPowerSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |

*crisis.statistics.getPlayerDepositSeries (GET)*

The series of deposit volumes representing the trend of households' willingness to place their savings in the player's bank.

| Method name | Description |
|---|---|
| crisis.statistics.getPlayerDepositSeries | This method sends back the time series of the player's deposit volume and interest rate at the given period. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles, represents the deposit volumes of the player on the given period. | double[] | yes |
| interests | Array of doubles, represents the deposit interest rate of the player on the given period. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getPlayerDepositSeries>
      <series>
        <value>198657</value>
        <value>234845</value>
        <value>256879</value>
        <value>215487</value>
        <value>172839</value>
      </series>
      <interests>
        <value>7.25</value>
        <value>8.13</value>
        <value>8.52</value>
        <value>8.52</value>
        <value>8.1</value>
```

```
        </interests>
        <freq>month</freq>
      </getPlayerDepositSeries>
   </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |

*crisis.statistics.getPlayerDepositPrediction (GET)*

To help the players set the optimal deposit interest rate, we calculate the expected deposit volume with the given *interestRate*. This value is estimated and does not represent the exact volume. The estimation is based on the data collected in the past.

| Method name | Description |
|---|---|
| crisis.statistics.getPlayerDepositPrediction | This method sends back the predicted deposit volume for the player at the given conditions. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| interestRate | The deposit interest rate of the next month | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| volume | The predicted deposit volume for the player | int | yes |

| Example Response |
|---|

```
<rsp code="0" msg="OK">
  <statistics>
    <getPlayerDepositPrediction volume="156487"/>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: Invalid argument value | The value of interest rate must be within 0 and 30. |

*crisis.statistics.getAverageDepositSeries (GET)*

The deposit volume and interest rate information is collected and averaged in each simulation turn. This information can help the player to adjust the deposit interest rate.

| Method name | Description |
|---|---|
| crisis.statistics.getAverageDepositSeries | This method sends back the average volume and interests of the deposits in all banks at the given period. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles, represents the averaged deposit volumes of banks. | double[] | yes |
| interests | Array of doubles, represents the averaged interest rate of banks. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getAverageDepositSeries>
      <series>
        <value>198657</value>
        <value>234845</value>
        <value>256879</value>
        <value>215487</value>
        <value>172839</value>
      </series>
      <interests>
        <value>7.25</value>
        <value>8.13</value>
        <value>8.52</value>
        <value>8.52</value>
        <value>8.1</value>
      </interests>
      <freq>month</freq>
```

```
    </getAverageDepositSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |

*crisis.statistics.getAverageDepositPrediction (GET)*

To help the players set the optimal deposit interest rate, we calculate the expected average deposit volume and interest rate of all banks. This value is estimated and does not represent the exact volume. The estimation is based on the data collected in the past.

| Method name | Description |
|---|---|
| crisis.statistics.getAverageDepositPrediction | This method sends back the expected average deposit volume and interest rate of all banks. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| volume | The expected average deposit volume for the next turn | int | yes |
| interest | The expected average interest rate for the next turn | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getAverageDepositPrediction volume="156487" interest="8.14"/>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: Invalid argument value | The value of interest rate must be within 0 and 30. |

*crisis.game.getDepositInterestRate (GET)*

This method is used to initialize the deposit screen or to reset the slider. The current deposit interest rate is set based on the information from the *Game Adaptor*.

| Method name | Description |
|---|---|
| crisis.game.getDepositInterestRate | This method sends back the current deposit interest rate of the player. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| value | The current deposit interest rate of the player | double | yes |

| Example Response |
|---|

```
<rsp code="0" msg="OK">
  <game>
    <getDepositInterestRate volume="6.7"/>
  </game>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

### UI Actions - Loan Screen

On the loan screen, players can decide about the interest rates of loans provided to firms. To facilitate this decision, the game client collects and displays information about all the firms (balance, dividends and production in previous turns). A sketch of the deposit screen is depicted in Figure 12. The screen displays various information to the player regarding the firms, such as the history of their balance, dividends, production on separate charts, and a table containing the current classification of firms into *Risky* and *Safe* categories (firms declared *Risky* shown with red background, while firms declared *Safe* shown with green background). The player may directly manipulate these categories by adjusting a slider. The player can also set different interest rates for the different categories.

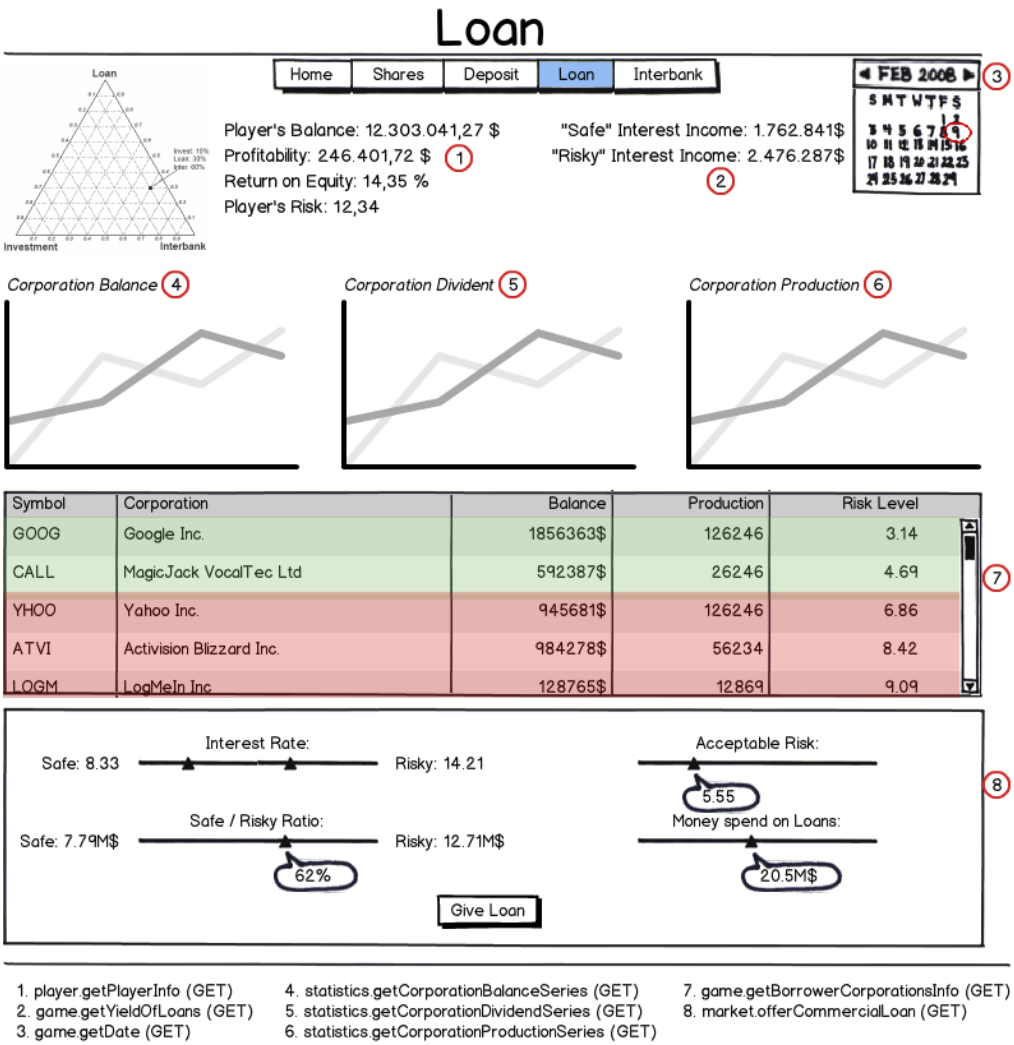


**Figure 12 – Commercial loans**

*crisis.market.offerCommercialLoan (POST)*

The game client sends this message to make commercial loan offers. As a result in the game engine, two networked limit-orders (defined in the CRISIS layer) are submitted to the corresponding market. One for the “safe” investments with a network constraint applicable to firms with risk below the *acceptableRisk* parameter, and one for the “risky” investments with a network constraint applicable to firms with risk above the *acceptableRisk* parameter. The price of the orders are given by the safeInterestRate and *riskyInterestRate* parameters, and the size of loan offers are determined by the *amountForSafe* and *amountForRisky* parameters. If the orders are submitted successfully, a ‘0’, otherwise an error code is returned.

| Method name | Description |
|---|---|
| crisis.market.offerCommercialLoan | This method creates commercial loan offers according to the given arguments. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| amountForSafe | The amount of money that the player wants to spend on safe investments | float | yes |
| amountForRisky | The amount of money that the player wants to spend on risky investments | float | yes |
| acceptableRisk | This parameter specifies the boundary between safe and risk investments; if the investment’s leverage rate is below this value, the investment is called safe, above that it is called risky | float | yes |
| safeInterestRate | The interest rate of the safe investments | float | yes |
| riskyInterestRate | The interest rate of the risky investments | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| <rsp code="0" msg="OK"><br></rsp> |

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid argument value (“xxx”) | The value of "xxx" must be within 0 and 30. |
| 51: negative argument value ("xxx") | The value of argument "xxx" must be greater or equal to zero. |

*crisis.statistics.getCorporationDividendSeries (GET)*

The historical data of corporation dividends help the player to decide what corporation to lend to.

| Method name | Description |
|---|---|
| crisis.statistics.getCorporationDividendSeries | This method sends back a series of dividend of the given corporation. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| corpID | The unique corporation ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the corporation dividend. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getCorporationDividendSeries>
      <series>
        <value>8453</value>
        <value>8512</value>
        <value>8389</value>
        <value>8234</value>
        <value>8109</value>
        <value>7982</value>
        <value>8123</value>
      </series>
      <freq>month</freq>
    </getCorporationDividendSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid corpID | Corporation not found with this ID |

*crisis.statistics.getCorporationBalanceSeries (GET)*

The historical data of corporation balance is a good indicator of the corporation's success and its power on the market. Therefore it is also an indicator if it is worth to lend to.

| Method name | Description |
|---|---|
| crisis.statistics.getCorporationBalanceSeries | This method sends back the series of the given corporation's balance. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| corpID | The unique corporation ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the corporation balance. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getCorporationBalanceSeries>
      <series>
        <value>328453</value>
        <value>348512</value>
        <value>338389</value>
        <value>378234</value>
        <value>368109</value>
        <value>357982</value>
        <value>408123</value>
      </series>
      <freq>month</freq>
    </getCorporationBalanceSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid corpID | Corporation not found with this ID |

*crisis.statistics.getCorporationProductionSeries (GET)*

The historical data of corporation production shows the amount of products that were produced at a given time.

| Method name | Description |
|---|---|
| crisis.statistics.getCorporationProductionSeries | This method sends back the series of the given corporation's production volume. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| corpID | The unique corporation ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the corporation production. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getCorporationProductionSeries>
      <series>
        <value>328453</value>
        <value>348512</value>
        <value>338389</value>
        <value>378234</value>
        <value>368109</value>
        <value>357982</value>
        <value>408123</value>
      </series>
      <freq>month</freq>
    </getCorporationBalanceSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid corpID | Corporation not found with this ID |

*crisis.statistics.getYieldOfLoans (GET)*

This method calculates the yield of safe and risky loans of the player at a given time. These values can help players find their optimal risk rate.

| Method name | Description |
|---|---|
| crisis.statistics.getYieldOfLoans | This method sends back the safe and risky loan's profits in the given period. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| acceptableRisk | The risk limit of the safe and risky loans | float | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| safe | The yield of safe loans | double | yes |
| risky | The yield of risky loans | double | yes |

| Example Response |
|---|

```
<rsp code="0" msg="OK">
  <statistics>
    <getYieldOfLoans safe="408123" risky="566891"/>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the "from" parameter must represent an earlier date, than the "to" parameter. |

## UI Actions - Trading Shares Screen

On the trading shares screen (see the sketch as Figure 13), players can review their shares, their currently active buy and sell orders, and the list of firms they can buy shares of. The players can click on a firm in the *Investments* list, and submit a limit order to buy a certain amount of shares of the selected firm for a given price. Similarly, the players can click on a firm listed in their portfolio to place a limit order to sell a certain amount of the firm's share for a given price limit. Additionally, players can cancel currently active buy and sell orders by selecting them from the pending orders list, and clicking on the *Cancel Order* button.

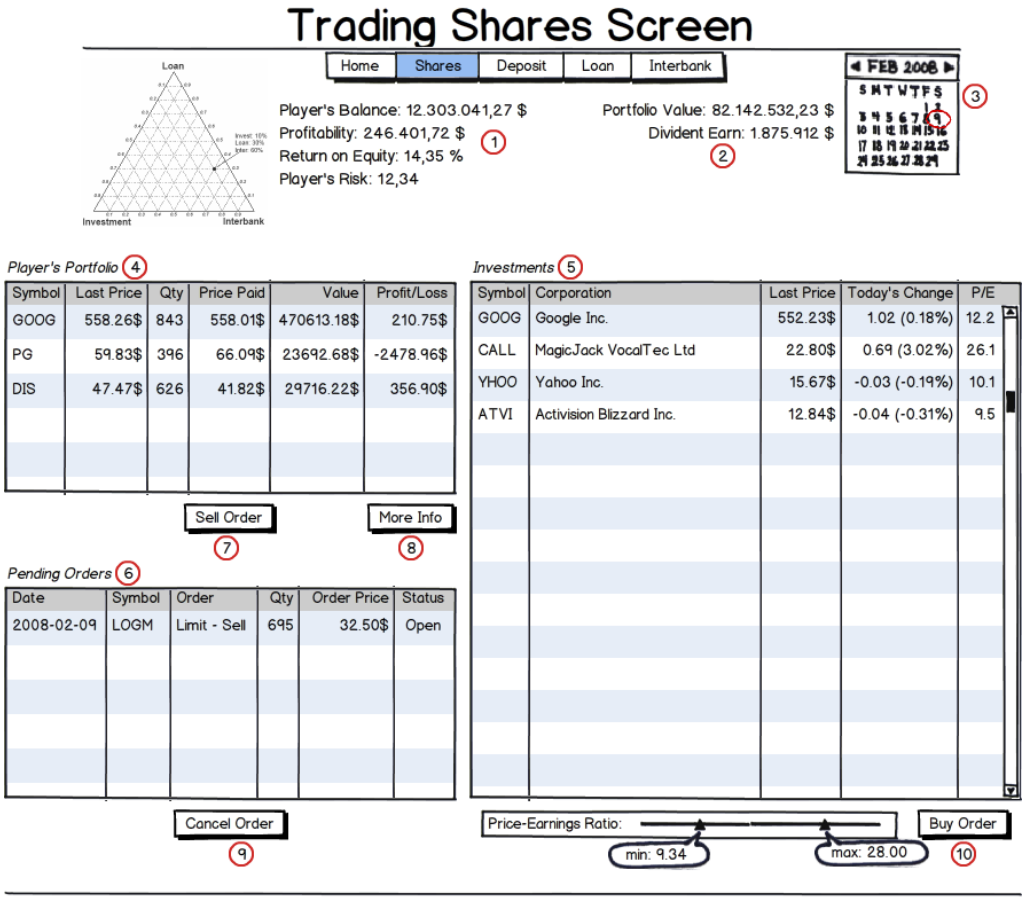


**Figure 13 – Trading shares**

*crisis.market.placeBuyOrder (POST)*

This method places a new buy order on the market of corporation shares. The price of the order is given by the *priceLimit* parameter, and the size is given by the *quantity* parameter.

| Method name | Description |
|---|---|
| crisis.market.placeBuyOrder | This method creates a buy order with the given parameters. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| corpID | A unique corporation ID | int | yes |
| quantity | The number of share you want to buy | int | yes |
| priceLimit | The maximum price of the shares | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| id | The ID of the order | int | yes |

Example Response

```
<rsp code="0" msg="OK">
  <market>
    <placeBuyOrder id="15467"/>
  </market>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid corpID | Corporation not found with this ID |

*crisis.market.placeSellOrder (POST)*

This method places a new sell order on the market of corporation shares. The price of the order is given by the *priceLimit* parameter, and the size is given by the *quantity* parameter.

| Method name | Description |
|---|---|
| crisis.market.placeSellOrder | This method creates a sell order with the given parameters. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| corpID | A unique corporation ID | int | yes |
| quantity | The number of share you want to sell | int | yes |
| priceLimit | The minimum price of the shares | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| id | The ID of the order | int | yes |

Example Response

```
<rsp code="0" msg="OK">
  <market>
    <placeSellOrder id="15467"/>
  </market>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid corpID | Corporation not found with this ID |
| 51: quantity too great | You have not so much share to sell |

*crisis.market.cancelOrder (DELETE)*

The player has the ability to cancel a pending, non-complete orders by this method.

| Method name | Description |
|---|---|
| crisis.market.cancelOrder | This method cancels an incomplete buy/sell order with the given parameters. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| orderID | A unique order ID | int | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| <rsp code="0" msg="OK"><br></rsp> |

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid orderID | Order not found with this ID |

*crisis.player.getPortfolioValue (GET)*

For the better information, the server calculates the value of the players' portfolio in each turn. This method returns the current portfolio value. However, this value can also be calculated manually by the *portfolioDetails* method.

| Method name | Description |
|---|---|
| crisis.player.getPortfolioValue | This method sends back the value of the player's portfolio. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| value | The value of the player's portfolio | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <player>
    <getPortfolioValue value="1428453"/>
  </player>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.statistics.getPortfolioValueSeries (GET)*

The historical data of the value of a player's portfolio shows the success of the player in the stock market.

| Method name | Description |
|---|---|
| crisis.statistics.getPortfolioValueSeries | This method sends back the series of the value of the player's portfolio. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the value of the player's portfolio. | double[] | yes |
| freq | The frequency of the series elements. Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getPortfolioValueSeries>
      <series>
        <value>1328453</value>
        <value>1348512</value>
        <value>1438389</value>
        <value>1378234</value>
        <value>1468109</value>
        <value>1457982</value>
        <value>1408123</value>
      </series>
      <freq>month</freq>
    </getPortfolioValueSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.player.getPortfolioDetails (GET)*

The portfolio details represent every information about the player's open positions. Symbols, quantity, average price of shares bought, etc.

| Method name | Description |
|---|---|
| crisis.player.getPortfolioDetails | This method sends back the details of the player's portfolio. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| shares | Array of Share represents the information about the player's open positions. | Share[] | yes |
| └ symbol | The symbol of the company | String | yes |
| └ quantity | The quantity of shares | int | yes |
| └ value | The current value of the shares | double | yes |
| └ profit | The profit made on the shares | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <player>
    <getPortfolioDetails>
      <shares>
        <value symbol="GOOG" quantity="61868" value="30934506" profit="124387"/>
        <value symbol="YHOO" quantity="13623" value="3279452" profit="247256"/>
        <value symbol="CALL" quantity="8268" value="734500" profit="34500"/>
      </shares>
    </getPortfolioDetails>
  </player>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.statistics.getPlayerPortfolioDividendSeries (GET)*

The historical data of the dividends of a player's portfolio shows information about the corporations paying dividends. It is possible that the prices of shares are not going up, but the periodical dividend is rewarding.

| Method name | Description |
|---|---|
| crisis.statistics.getPlayerPortfolioDividendSeries | This method sends back the series of the dividends of the player's portfolio. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the dividend of player's portfolio. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getPlayerPortfolioDividendSeries>
      <series>
        <value>328453</value>
        <value>348512</value>
        <value>368109</value>
        <value>357982</value>
        <value>408123</value>
      </series>
      <freq>month</freq>
    </getPlayerPortfolioDividendSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the "from" parameter must represent an earlier date, than the "to" parameter. |

### UI Actions - Interbank Screen

On the interbank screen (see Figure 14), players can borrow and lend money to other banks.
To lend money, the player must set a limit value to classify the banks into “safe” and “risky” interbank investments. The player can set the interest rate and the volume for these groups. After the confirmation of settings two order appears in the market with the proper constraints.
To borrow an interbank loan, the player simply needs to set the maximum interest rate (accepting all offers) and the volume of the loan. (The operation needs to be confirmed by the appropriate button.)

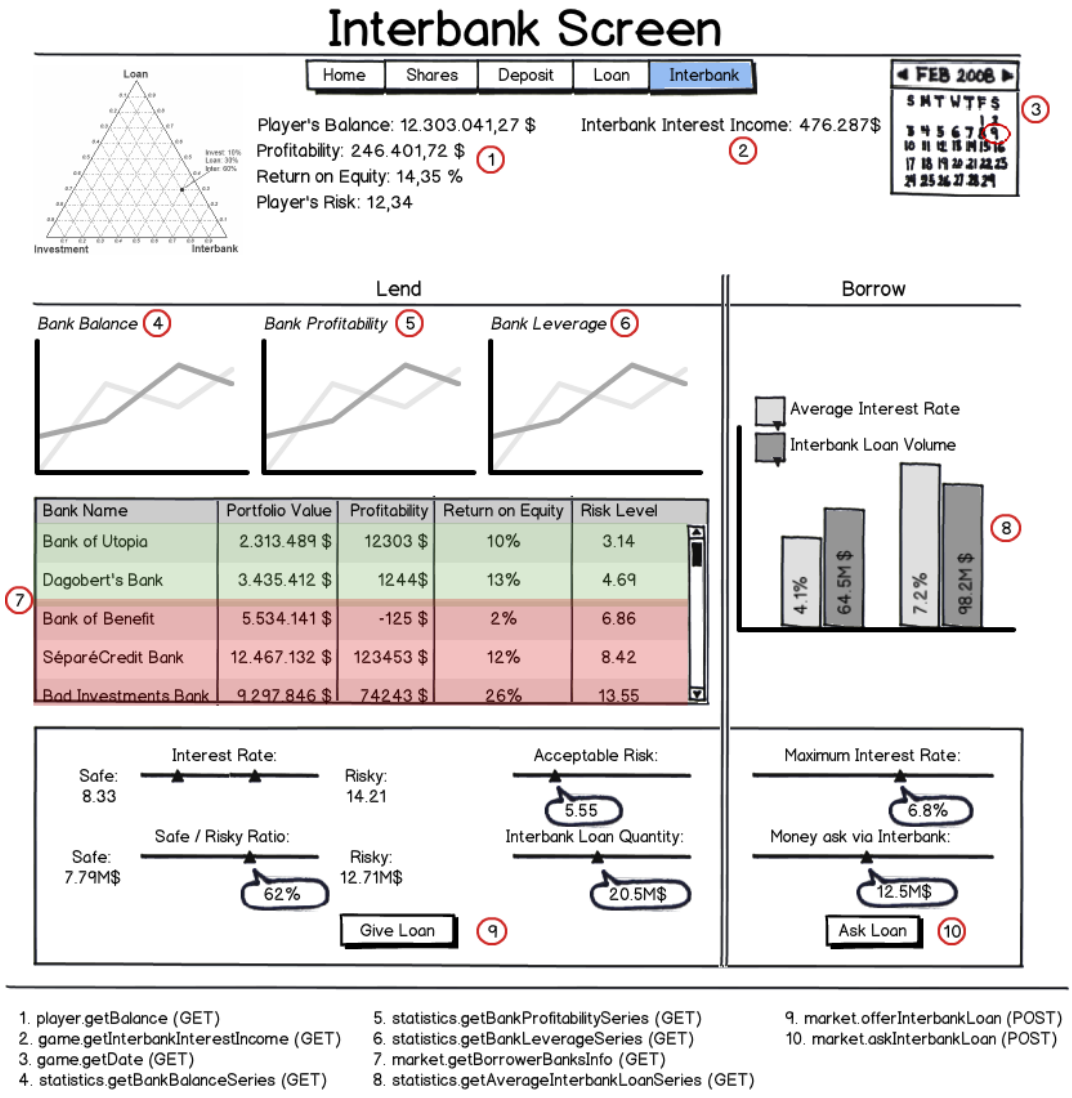


**Figure 14 – Lend/Borrow Interbank Loans**

*crisis.statistics.getInterbankInterestIncome (GET)*

The interbank interest income shows the success of the player in this market.

| Method name | Description |
|---|---|
| crisis.statistics.getInterbankInterestIncome | This method sends back the interbank interest income in the current turn. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| income | The player's income in the current turn. | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getInterbankInterestIncome income="476287"/>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.statistics.getBankBalanceSeries (GET)*

The historical data of bank balance is a good indicator of the banks' success and their power on the market (i.e., is it worth the risk to lend them).

| Method name | Description |
|---|---|
| crisis.statistics.getBankBalanceSeries | This method sends back the series of the given bank's balance. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| bankID | The unique bank ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the series of bank balance. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getBankBalanceSeries>
      <series>
        <value>328453</value>
        <value>348512</value>
        <value>338389</value>
        <value>378234</value>
        <value>368109</value>
        <value>357982</value>
        <value>408123</value>
      </series>
      <freq>month</freq>
    </getBankBalanceSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid bankID | Bank not found with this ID. |

*crisis.statistics.getBankProfitabilitySeries (GET)*

The historical data of a bank's profit can provide useful information about the strategy and success of the requested bank.

| Method name | Description |
|---|---|
| crisis.statistics.getBankProfitabilitySeries | This method sends back the series of the given bank's profit. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| bankID | The unique bank ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the series of bank profitability. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getBankProfitabilitySeries>
      <series>
        <value>328453</value>
        <value>348512</value>
        <value>338389</value>
        <value>378234</value>
        <value>368109</value>
        <value>357982</value>
        <value>408123</value>
      </series>
      <freq>month</freq>
    </getBankProfitabilitySeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid bankID | Bank not found with this ID. |

*crisis.statistics.getBankLeverageSeries (GET)*

The historical data of bank leverage shows the bank's willingness to take risk.

| Method name | Description |
|---|---|
| crisis.statistics.getBankLeverageSeries | This method sends back the series of the given bank's leverage rate. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| bankID | The unique bank ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| series | Array of doubles represents the series of bank leverage. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
     <getBankLeverageSeries>
        <series>
           <value>10.1</value>
           <value>9.2</value>
           <value>11.5</value>
           <value>13.4</value>
           <value>14.8</value>
           <value>9.2</value>
           <value>12.6</value>
        </series>
        <freq>month</freq>
     </getBankLeverageSeries>
  </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid bankID | Bank not found with this ID. |

*crisis.market.getBorrowerBanksInfo (GET)*

The methods returns all information about the banks asking for interbank loans (e.g., name, profitability, return on equity, risk rate and portfolio value).

| Method name | Description |
|---|---|
| crisis.market.getBorrowerBanksInfo | This method sends back the information about the banks asking for loans. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| infos | Array of BankInfo represents the information about the borrower banks. | BankInfo[] | yes |
| └ name | The name of the bank | String | yes |
| └ pvalue | The portfolio value of the bank | double | yes |
| └ roe | The return on equity in percent | double | yes |
| └ risk | The risk factor of the bank | double | yes |

Example Response

```
<rsp code="0" msg="OK">
  <market>
    <getBorrowerBanksInfo>
      <infos>
        <value name="Bank of Utopia" pvalue="2852653" roe="10" risk="3.14"/>
        <value name="Dagobert's Bank" pvalue="5636256" roe="13" risk="4.69"/>
        <value name="Bank of Benefit" pvalue="1513256" roe="2" risk="6.86"/>
      </infos>
    </getBorrowerBanksInfo>
  </market>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| - | - |

*crisis.statistics.getAverageInterbankLoanSeries (GET)*

The historical data of interbank loan volumes and average interest rates can help to find the optimal interest rate settings when placing or asking for interbank loans.

| Method name | Description |
|---|---|
| crisis.statistics.getAverageInterbankLoanSeries | This method sends back the series of interbank lending interests and volumes in the given period. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| bankID | The unique bank ID | int | yes |
| from | The beginning of the time series | Date | yes |
| to | The end of the time series<br>default value: the in-game now date | Date | no |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| volumes | Array of doubles, represents the volumes of completed interbank transactions. | double[] | yes |
| interests | Array of doubles, represents the averaged interest rate of the completed interbank transactions. | double[] | yes |
| freq | The frequency of the series elements.<br>Possible values: [day, week, month] | String | yes |

Example Response

```
<rsp code="0" msg="OK">
  <statistics>
    <getAverageInterbankLoanSeries>
      <volumes>
        <value>198657</value>
        <value>234845</value>
        <value>256879</value>
        <value>215487</value>
        <value>172839</value>
      </volumes>
      <interests>
        <value>7.25</value>
        <value>8.13</value>
        <value>8.52</value>
```

```
          <value>8.52</value>
          <value>8.1</value>
        </interests>
        <freq>month</freq>
      </getAverageInterbankLoanSeries>
   </statistics>
</rsp>
```

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid time period | Invalid time period, the “from” parameter must represent an earlier date, than the “to” parameter. |
| 51: invalid bankID | Bank not found with this ID. |

*crisis.market.offerInterbankLoan (POST)*

The game client sends this message to make interbank loan offers. As a result in the game engine, two networked limit-orders (term defined in the integrated CRISIS layer) are submitted to the corresponding market. One with a network constraint allowing banks with risk below the *acceptableRisk* parameter ("safe"), and one with a network constraint allowing banks with risk above the *acceptableRisk* parameter ("risky"). The price of the orders are given by the *safeInterestRate* and *riskyInterestRate* parameters, and the size of loan offers are determined by the *amountForSafe* and *amountForRisky* parameters.

| Method name | Description |
|---|---|
| crisis.market.offerInterbankLoan | This method submits two networked limit-order to the interbank market as loan offers by the player. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| amountForSafe | The amount of money that the player wants to spend on safe interbank requests | float | yes |
| amountForRisky | The amount of money that the player wants to spend on risky interbank requests | float | yes |
| acceptableRisk | This parameter sets the boundary between safe and risk interbank requests. If the bank's leverage rate is below this value, the request is called safe. Above the limit it is called risky | float | yes |
| safeInterestRate | The interest rate of the safe interbank requests | float | yes |
| riskyInterestRate | The interest rate of the risky interbank requests | float | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| <rsp code="0" msg="OK"><br></rsp> |

| Message Specific Status Codes | Description |
|---|---|
| 50: invalid argument value (“xxx”) | The value of "xxx" must be within 0 and 30. |
| 51: negative argument value ("xxx") | The value of argument "xxx" must be greater or equal to zero. |

*crisis.market.askInterbankLoan (POST)*

This message submits a new order on the interbank market, or overwrites a previously submitted, currently valid order. The price of this order is set to the *interbankInterestRate* parameter of the message, and the size of the request is set by the *interbankVolume*.

| Method name | Description |
|---|---|
| crisis.market.askInterbankLoan | This method submits a new order to the interbank market as a loan request by the player. |

| Argument name | Description | Type | Required |
|---|---|---|---|
| interbankInterestRate | The maximum acceptable interest on the interbank request | int | yes |
| interbankVolume | The maximum amount of money that the player needs from other banks | int | yes |

| Response Attribute Name | Description | Type | Required |
|---|---|---|---|
| - | - | - | - |

| Example Response |
|---|
| <rsp code="0" msg="OK"><br></rsp> |

| Message Specific Status Codes | Description |
|---|---|
| 50: Invalid argument value | The value of interest rate must be within 0 and 30. |

## Data

The following section describes the data structures required for, or recorded by the *Game Engine*. The data tables are enumerated in tables, indicating the type and value of their fields, providing a description and discussing wheter they permit *null* values.

### User Info

Users should be able to login to the portal using an OpenID account. This way the registration process is not necessary to play the game, but as a trade-off most of the general user information is not available. Only the user's e-mail address is stored, other fields are initialized with empty default values (the user however, is able to set all of these fields later on).

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID of the player. | Primary key Auto increment | x |
| Name | Nickname displayed for all other players | String | x |
| EMail | E-mail address of the user (OpenID) | String | x |
| FullName | Full name of the user | NULL: String | |
| TimeZone | Time zone setting of the user | +0:00: Time | |
| RegistrationDate | Date of registration (or date of first usage) | NOW: Date | x |
| Locale | User locale preference | EN: String | |
| Bio | Short description about the user displayed on the portal (HTML formatted text) | NULL: String | |
| AgreedToTermsOfUsage | Indicates if the user accepted the *Terms of Usage* | FALSE: Boolean | x |
| Location | Geolocation of the user | NULL: String | |
| DateOfBirth | Date of birth | NULL: Date | |

### User Role

For the game we need to assign different privilege levels to different users. To have an extensible list of user role definitions, we keep all the necessary information in the database.
By default, the following roles are created for the users:

- *Player*: Allowed to join any previously created game.
- *Administrator*: Creates and manages running games, may download simulation results and data required to replay the experiment.

This table stores only the definitions of the different privileges.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for the role | Primary key Auto increment | x |
| Description | Description of the specific role that is available on the site | String | x |

### User Role Definitions

Since users might own have several privileges at the same time, we have to store the relationship between users and privileges as well. This table maps the users to the roles they currently hold.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for the role definition | Primary key<br>Auto increment | x |
| UserID | User who owns the role | Foreign key<br>[UserInfo] | x |
| RoleID | Role that which the user holds | Foreign key [UserRole] | x |

### User Statistics

Conducting experiments (i.e., running games) yields data that could help us understand the analyzed phenomena with a deeper understanding. For this reason, we store the final results of each player in each separate game. This data is enough to calculate several statistics we described in section Statistics Module.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for the statistic | Primary key<br>Auto increment | x |
| PlayerID | Player who played the game | Foreign key<br>[UserInfo] | x |
| GameID | ID of the game that was played | Foreign key<br>[Game] | x |
| Score | The score the player achieved for the given game | Integer | x |

### Model

Users can upload several different models (simulations) to the portal. A good candidate to handle this is through an existing, standard portal feature is the *Document Library*, where users can upload and organize different kinds of resources (like scientific papers, diagrams, executable models, documentation and so on). This way it will be easy to keep track of different models in an organized way, they would be visible by default, to every registered user, who can also make copies of the uploaded models, etc. Moreover this makes easy to create user friendly URLs for the resources and to provide the users with a familiar interface like folder and file manager applications. In addition, the *Document Library* already offers an application programming interface to access the resources programmatically, and can be customized easily.

The models uploaded this way are stored in the file system, not in the database. However, we also store some additional metadata about the uploaded models in the database through the following table (e.g., the user who uploaded the resource, the date of upload, a brief description of the model, etc.).

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for an uploaded model | Primary key<br>Auto increment | x |
| Name | The name of the model | String | x |
| Description | A brief description of the model<br>(HTML formatted text) | "": String | |
| UploaderID | The user who uploaded this model | Foreign key<br>[UserInfo] | x |
| UploadDate | The date of the upload | NOW: Date | x |
| URL | Location pointing to the uploaded model in the file system | String | x |

### Game Configuration

An uploaded model first have to be configured (i.e., properly parametrized) before it is started. The current architecture design allows the creation of different model families (i.e., different game configurations for the same model). This is an important feature, because users can create different scenarios for the same model (e.g., a version of the model might exclude artificial agent players completely, while another version might have an equal number of human and computer players). These configurations will be displayed directly on the website for the administrators, who can start those pre-configured models with a single click (administrators do not even have to know the model in details, it is enough if the configuration have a descriptive name or a proper description).

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for the game configuration | Primary key Auto increment | X |
| Config | Configuration that the simulation handles through the *LoadConfig* method<br>(JSON format) | String | X |
| UploaderID | User who uploaded this model | Foreign key [UserInfo] | X |
| ModelID | The ID of the model which is configured | Foreign key [Model] | X |
| Name | Name of the configuration to be displayed at the model list on the website | String | X |
| Description | A brief description of the current version of the model that helps to distinguish this configuration from the others even for users who are not familiar with the parametrization of the model in depth.<br>Displayed on the website under details.<br>(HTML formatted text) | String | X |

### Game

Each game ever started is recorded in the database. The stored data include the timestamp and date when the simulation was started, a reference to the used game configuration, the end time (if it is already finished) and the user who started the game.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID for the played game | Primary key<br>Auto increment | x |
| ConfigID | ID of the used game configuration which is used for this game | ForeignKey [GameConfiguration] | x |
| Begin | Start of the game experiment | NOW: Date | X |
| End | End of the game/experiment | NULL: Date | |
| Submitter | User who created this simulation | Foreign key [UserInfo] | x |

### Game Data

During the simulation, it might be necessary to store simulation-specific data for further analysis. A simple way to evaluate these data is to replay the simulation, but this approach might be cumbersome.

For this reason, an API will be supported for the modellers to record any data (like statistics, agent or model states, etc.) at the end of each time step. This data will be automatically stored in the database. The following is the table that stores the data for a given game (for all time steps). The recorded data is saved in standard JSON[8] format.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID of an entry | Primary key<br>Auto increment | x |
| GameID | ID of the Game | Foreign key<br>[Game] | x |
| Tick | Actual tick | Integer | x |
| Data | Recorded data (JSON format) | String | x |

[8] JavaScript Object Notation (JSON) is the *de facto* standard for lightweight data-interchange format. For the specification, see http://www.json.org/.

### Log

A database table that stores the sequence of events that happened during the simulation. It includes *all events* during the simulation, in order to keep the reproducibility of the simulation.
The communication protocol (i.e., the user actions) might also change during the full lifecycle of the project. To keep the reproducibility of previous simulations, we also store the API version that was in use when the game was played.
User actions are stored in the form of the incoming request (i.e., the accepted REST call is stored directly before processing). Additional logging information might be attached to each entry in the form of a description.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID of an entry | Primary key<br>Auto increment | x |
| Time | Timestamp when the action took place | Time | x |
| UserID | ID of the user who performed the action | Foreign key<br>[UserInfo] | x |
| Description | Additional description of the action | String | |
| Action | The full request that came from the client | String | x |
| APIVerID | The API version that was in use when the Action was issued. | Foreign key<br>[UserActionProtocol] | x |

### User Action Protocol

The user action protocol (i.e., the protocol of the REST interface the clients use to perform various actions) should change only in special circumstances and only in a well-documented way. Keeping a list of each previously used versions is essential to maintain reproducibility of the performed experiments.

| Name | Description | Value: Type | Non-null |
|---|---|---|---|
| ID | Unique ID of an entry | Primary key<br>Auto increment | x |
| Description | Name of the protocol version (e.g., “V1.0”) | String | x |

# Integration Process

The development of the economic simulator might be splitted into different milestones. Some of these milestones can be achieved in parallel, while others require several developers to work on the related projects at the same time to finish before the deadline.

During the development we have to pay attention to the management of the source code, to the integration of the work of different developers, and to perform the required (and automatized) tests in a separate sandbox (test) environment before putting the code into actual production.

Having a methodical, designed process to integrate sources from various sources, check their correctness and interoperability, etc. helps us to release new versions of the game (and later, the economic simulation) smoothly and on time, and in a dependable manner.

The overhead of deisgning and setting up this process and the automated tools required for it is minimal. We have installed automatized tools to perform the compilation in the form of a *Continuous Integration environment*[9] which is a continuous process of applying quality control on the software under development. It covers the whole development cycle from the point of checking sources out from the repository, via building the developed software components and performing automated tests on the results (e.g., unit tests and integration tests), to publishing the development and major versions of the software in the form of nightly builds[10], and to automatically generating all of the related development statistics through static program analysis[11], reports and documentations. These latter will include the analysis of the source files through standard software metrics (like ratio of source-level documentation, total and effective lines of code[12], cyclomatic complexity[13], duplicated code), testing and code coverage reports[14], etc.

Having such an environment results in several advantages. First, it becomes possible to build the software anytime during the development regardless of the platform and the developer environment. If the builds are ready, the automated tests help us recognize any introduced errors or malfunctions in the integrated software components. Code coverage is important to know which parts of the software is covered properly with enough test cases and which parts require additional ones. The standard code metrics help identifying parts of the software that do not accomodate certain conventions/rules, or became overcomplicated and time has come to redesign them.

A simple architectural overview is shown on Figure 15, where developers interact through an Source Version Controlling server. The Continuous Integration system also pulls all changes from the related projects and generates all the configured project reports (summary of the unit and integration tests is a bare minimum).

[9] http://en.wikipedia.org/wiki/Continuous_integration
[10] http://en.wikipedia.org/wiki/Nightly_build
[11] http://en.wikipedia.org/wiki/Static_program_analysis
[12] http://en.wikipedia.org/wiki/Source_lines_of_code
[13] http://en.wikipedia.org/wiki/Cyclomatic_complexity
[14] http://en.wikipedia.org/wiki/Code_coverage

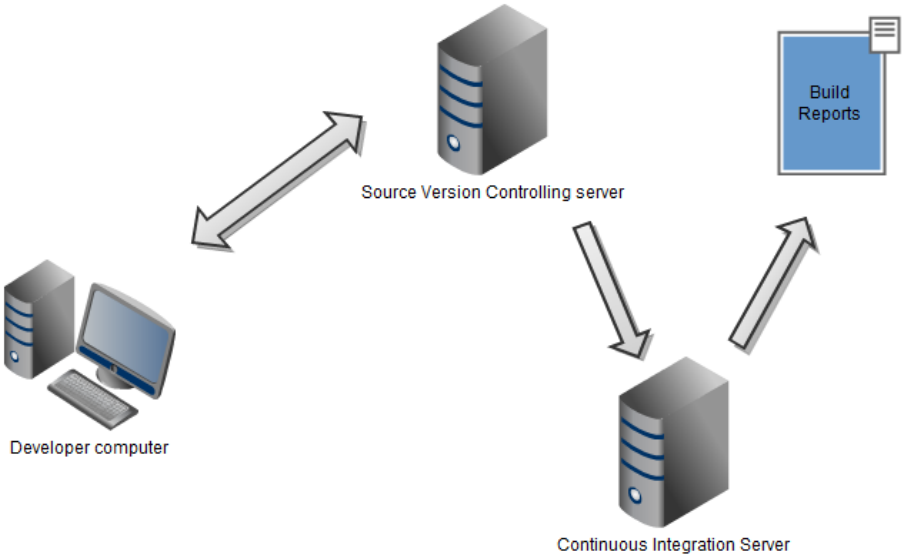


**Figure 15 – Architectural overview of the integration process**

## Source-Code Handling

The project has several contributors who are geographically distributed. Source code management is a crucial point, the design decision have a great impact on the cooperation between the CRISIS developers, the integration architecture and many other important aspects (like backups for instance).

Having a version control system (VCS) based on client-server repository model would have several drawbacks. It requires strict cooperation between members even if the work affects different (even unrelated) components, causing heavy communication overhead - which might be hard to resolve.

The primary advantages of using distributed version controlling is that it offers more freedom in the first place. It narrows the required communication to a few persons in the Integration Group, and the developer teams working at the same place can resolve the in-project source code conflicts relatively quickly without having to interact with external developers.

At the moment, the most popular distributed version controlling systems are Git[15] and Mercurial[16]. In the beginning of the project, we decided to use Mercurial, because i) it is a bit more mature project, ii) its interface is more user friendly, and iii) its tool support is wider.

### General Information about the Used Source Versioning System

- **Web-based source control manager:** https://www.crisis-economics.eu/scm/
- **Root of the repositories:** https://www.crisis-economics.eu/scm/hg/
  All Mercurial repositories are put under this directory. The web-based viewer is enabled for all of the repositories by default, they can accessed by the concrete repository name, e.g., https://www.crisis-economics.eu/scm/hg/financial-core
- **Repository administrators:**
  Personnel owning privileges to create or modify any repository.
  - Tamás Máhr (tmahr@aitia.ai)
  - Richard O. Legendi (rlegendi@aitia.ai)
- **Integration server:** https://www.crisis-economics.eu/jenkins/
  Currently it is running on the same machine, in a separate sandbox environment.

## Integration Strategy

Under integration, we mean the process of taking the code from different repositories and merging them into a dedicated, unified repository while resolving all occurring conflicts in a semi-automatized way. The heart of any integration process is a well defined repository structure.

Our proposal for the development is to create a set of small, modular projects with clear dependencies. It is important to separate components in order to efficiently split work between the partners. A distributed VCS offers great support for this approach. Table 3 describes the repositories we have set up and are using at the moment. There are several additional planned

[15] http://git-scm.com/
[16] http://mercurial.selenic.com/

repositories for the integrated CRISIS economic model, the game engine, and for the online game.

| Name | URL | Description |
|---|---|---|
| financial-core | https://www.crisis-economics.eu/scm/hg/financial-core | A core modelling API for financial simulations (contains structures like balance sheets, assets, liabilities and different contracts, and supports in-memory historical queries). |
| financial-core-example | https://www.crisis-economics.eu/scm/hg/financial-core-example | A demo application that demonstrates the usage of the *financial-core* API with the Mason simulation platform |
| financial-model | https://www.crisis-economics.eu/scm/hg/financial-model | A repository for an integrated financial model (*empty at the moment*) |
| interbank | https://www.crisis-economics.eu/scm/hg/interbank | Integrated interbank model that will use the financial-core API |
| mason-all | https://www.crisis-economics.eu/scm/hg/mason-all | A project for the Mason platform integration |
| mason-demo | https://www.crisis-economics.eu/scm/hg/mason-demo | A simple skeleton project that can be used as a prototype for any Mason-specific agent-based models. Built on top of the *mason-all* project. |
| mason-macroabm | https://www.crisis-economics.eu/scm/hg/mason-macroabm | Implementation of the Mark 1 model. Based on the *mason-demo* project, built upon the mason-all library and will incorporate the *financial-core* API. |

**Table 3 – Summary of the currently used repositories**

Following the distributed version controlling approach, the CRISIS project participants can work with their own derivative (or *branch*) of a given repository, without interfering with any external user. This process will help us, for instance, to create different bank strategies for the Online Game, the Tournament or the Crisis Simulator.

## Automated Tests

During the development, we introduce automated software tests on different levels to ensure software quality and to catch errors. First, each component must be tested properly with unit tests. It is advised to share code in the mainline development branch (*master*) for which all the created unit tests pass without indicating any errors. These tests must be prepared by the developer of the appropriate branch/project.

Even if we had several properly tested and functional components, integrating them could reveal additional problems and misunderstandings. In order to catch these problems early on the development stage, we also prepare *integration tests*.

Since these tests might require considerable computational resources, they are run only on a regular basis (e.g., *nightly builds* if there was any recent change in the mainline code base) and they are run in a separate environment.

We have chosen the Jenkins[17] platform to use in the project, which is open for all project members to use. At the moment, it is also used to generate different project sites, reports and documentation that can be accessed directly from the project website after every automatic build. An example might be the the API documentation of the financial-core programming interface[18] or the description of the Mark 1 replication[19].

In case of any problems, all owners of the project repository are notified about the issue via e-mail. In addition, every developer may subscribe to the build events through various external applications (like RSS feeds).

## Preliminary Release Plan

In WP5, three releases of the on-line game is produced during the project. The releases are timed as follows.

| R1 | R2 | R3 |
|---|---|---|
| M12 | M18 | M36 |

The first release (R1) is planned to come out at month 12. This release will launch the game portal featuring challenge games[20] that are real-time CRISIS games played over the internet. The second release (R2) is the version that can be played on a local network. The main change in this release is going to be in the control of the game, which will support games on local networks. The third release (R3) is scheduled at the end of the project. It will bring a new version of the web-portal-based game environment featuring a never-ending turn-based version of the game.

[17] https://www.crisis-economics.eu/jenkins/
[18] http://www.crisis-economics.eu/jenkins/job/financial-core/javadoc/index.html
[19] https://www.crisis-economics.eu/jenkins/job/mason-macroabm-nightly/site/index.html
[20] See D5.1 Game Architecture

# Summary

Our goal in this document was to describe how the simulation models are integrated into the online game architecture. To this end, we discussed both the integrated architecture and the integration process. First, we discussed some architectural requirements and presented a brief architectural overview. Since several different project partners develop the simulation models, the integrated library, and the game, one of the main requirements was that these components had to be developed independently. The proposed pluggable architecture, that allows simulation models to be uploaded onto the web portal, ensures this separation. In addition to develop the components separately, this architecture has the further advantage that the simulation can be run independently from the portal, possibly on other machines. This allows us to manage computational resources efficiently, and delegate CPU intensive games to separate machines.

After the architectural requirements, we treated functional requirements in the form of use cases, and showed how the required functions are implemented by the different components of the game. We discussed usage cases of players, game owners, and game masters, and showed how the three components, the web portal, the game engine, and the game client realise these functions. From the integration point of view, the game engine is the most interesting component. It encapsulates the simulation model, and completes it with a game adaptor. This game adaptor relays the players' actions to the simulation, and the simulation state to the players. For each simulation model, a separate game adaptor has to be created, one that can answer each request originating from the players. Since computing the required statistics might be different in different simulations, the game adaptor should be different for each simulation as well. Given its central position between the simulation model and the players, the game adaptor is one of the key elements of the integration.

After discussing the game components, we defined the message format and the message contents the browser and especially the game client sends to the portal and the game engine before and during a game. Most of these messages retrieve information from the game engine to the player, but some of them transmit the players' actions. In addition to the message passed between the components, the last subsection in the 'Game Integration' section describes the data structures needed for the game.

Following the discussion of the game components, messages and data, the next section describes the integration process we apply in developing the game. Since different parts are developed by different partners, it is crucial to define the processes and responsibilities related to developing, testing, and integrating the code. In addition to explaining how the code is moved between repositories, this section also discusses the testing procedures, as well as defines a release plan for the three iterations of the game. According to this schedule, we will first release the on-line game at month 12, then the version played on a local network is scheduled at month 18, and finally a last version is due at the end of the project.