Designing a 3D visualization framework for distributed, synchronized execution

Documentation of hands-on experiences gained from redesigning a single-user physics visualization framework written in Java to an efficient multi-user architecture

Master's Thesis - DRAFT
at
Graz University of Technology

submitted by

Christian Schratter

Supervisor: Dipl.-Ing. Dr. techn. Christian Gütl

Institute for Information Systems and Computer Media (IICM)
Graz University of Technology
A-8010 Graz, Austria

Co-Supervisor: Associate Director V. Judson Harward

Center for Educational Computing Initiatives (CECI)
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

April 1th, 2012

Design eines 3D Visualisierungs-Frameworks für verteilte, synchronisierte Umgebungen

Dokumentation der Überarbeitung eines Java Physik Visualierungs-Frameworks für Desktops zu einer effizienten Client-Server Architektur

Diplomarbeit - ENTWURF
an der
Technischen Universität Graz

Vorgelegt von

Christian Schratter

Betreuer: Univ.-Doz. Dipl.-Ing. Dr. techn. Christian Gütl

Institut für Informationssysteme und Computer Medien (IICM)
Technische Universität Graz
A-8010 Graz, Österreich

Co-Betreuer: Associate Director V. Judson Harward

Center for Educational Computing Initiatives (CECI)
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

01. April 2012

Affirmations

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

………………………………..   ………………………………..

(Place, Date)   (Signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

………………………………..   ………………………………..

(Ort, Datum)   (Unterschrift)

Acknowledgments

First of all I would like to thank my supervisor at the Graz University of Technology Christian Gütl for supporting me with his expertise, inputs and ideas on how to approach and carry out this thesis to come up with interesting results. His continued believe in me and my skills paved the way for an enduring collaboration over the course of the last couple of years, eventually convincing him to recommend me to his scientific acquaintances at the MIT, resulting in this university-spanning project.

Consequentially I have to thank all the people working together with me at the CECI department at the MIT: Judson Harward for co-supervising me, John Belcher and Phil Bailey for all their explanations, inputs and constructive talks on how to evolve TEALsim for a better as well as Meg, Kirky, Maria, Jim and Mark for everything they did to make my stay in the United States as enjoyable and productive as possible.

Furthermore I would like to thank the following institutions for funding this thesis as such and/or its associated stay in the USA:

the Marshall Plan Foundation[1] for granting a 'Marshall Plan Scholarship', the Industriellenvereinigung Kärnten[2] for granting an 'Exzellenzstipendium', the Verband selbstständig Wirtschaftstreibender Kärntens for granting an 'Auslandsstipendium', Dr. Josef Martin for granting a scholarship for studies in foreign countries as well as the Faculty of Informatics[3] at the Graz University of Technology for granting a 'Förderungsstipendium'.

Last but not least I would like to take the opportunity to thank my family for supporting my studies at every point of time in every possible way, enabling me to gather an incredible amount of different experiences and having a great time which I would not want to trade for anything else.

Christian Schratter

Graz, Austria, March 2012

[1] Marshall Plan Foundation - http://www.marshallplan.at
[2] IV Kärnten – http://www.iv-kaernten.at
[3] Faculty of Informatics - http://www.dinf.tugraz.at/

Contents

Part I
General background

Chapter 1

State of the TEALsim project

Currently TEALsim (Massachusetts Institute of Technology, 2012) is not in a releasable state.

This situation held true at the beginning of the work for this thesis and did not change up until now. While there was certainly a point of time when the framework appeared to people – at least those who were not intensely engaged in the project – to work pretty well (see the binaries compiled with older code[4]), several factors negatively impacted the project and pushed the trunk of the source code back into a state resembling a demonstration version instead of something which could be used in a real teaching environment. Amongst others the following reasons can be identified which lead to the current situation:

a) Student projects – over the course of the last couple of years most of the work to enhance TEALsim has been done in the form of student projects. Taking into consideration output limiting constraints like an approximate length of 6 months for a student project, the considerable size of the project paired with the lack of high-quality in-depth (design-) documentation, the general best practice to spend 50% of development time to test the software (50PctTest) in contrast to the desire of students to come up with a maximum of interesting results to support an extensive thesis like this, one can easily spot the area of conflict between these aspects. In the case of TEALsim usually the pursuit of ideas for new features (and thus documentable results) was preferred at the expense of software quality.

b) Lack of (paid,) continued work on the framework – this item has been already partially touched in the former paragraph. The lack of paid and therefore continuing, dedicated work to funnel deviating feature branches, created by temporary collaborators, back into the trunk to match a unified style and design paradigm exacerbated the problem of diminishing software quality. While the absence of detailed documentation or system specification itself indicates that the framework's design was mostly defined on-the-fly

[4] TEALsim binaries based on older code - http://web.mit.edu/viz/EM/simulations/

(and therefore got inconsistent over the years – see the Lava Flow AntiPattern (Brown, Malveau, McCormick III, & Mowbray, 1998)), the missing authority to evolve and watch over the code ultimately lead to an even more lava-flow like situation, which will be discussed in the succeeding paragraphs.

c) Changing client hardware/software – naturally software and hardware is evolving rapidly over the years. Given the fact that the TEAL project was started in the year 2000 (Scheucher, 2010) where Java version 1.3 was the most recent library available, TEALsim was faced with more or less comprehensive new client environments at least a couple of times throughout its lifetime. This circumstance implies the requirement for code maintenance activities of various scales, which in turn becomes an issue in absentia of dedicated workforce (see the former paragraph).

As mentioned TEALsim appears as a perfect example for a project which evolved over time matching the explanations for the Lava Flow AntiPattern given in (Brown, Malveau, McCormick III, & Mowbray, 1998).

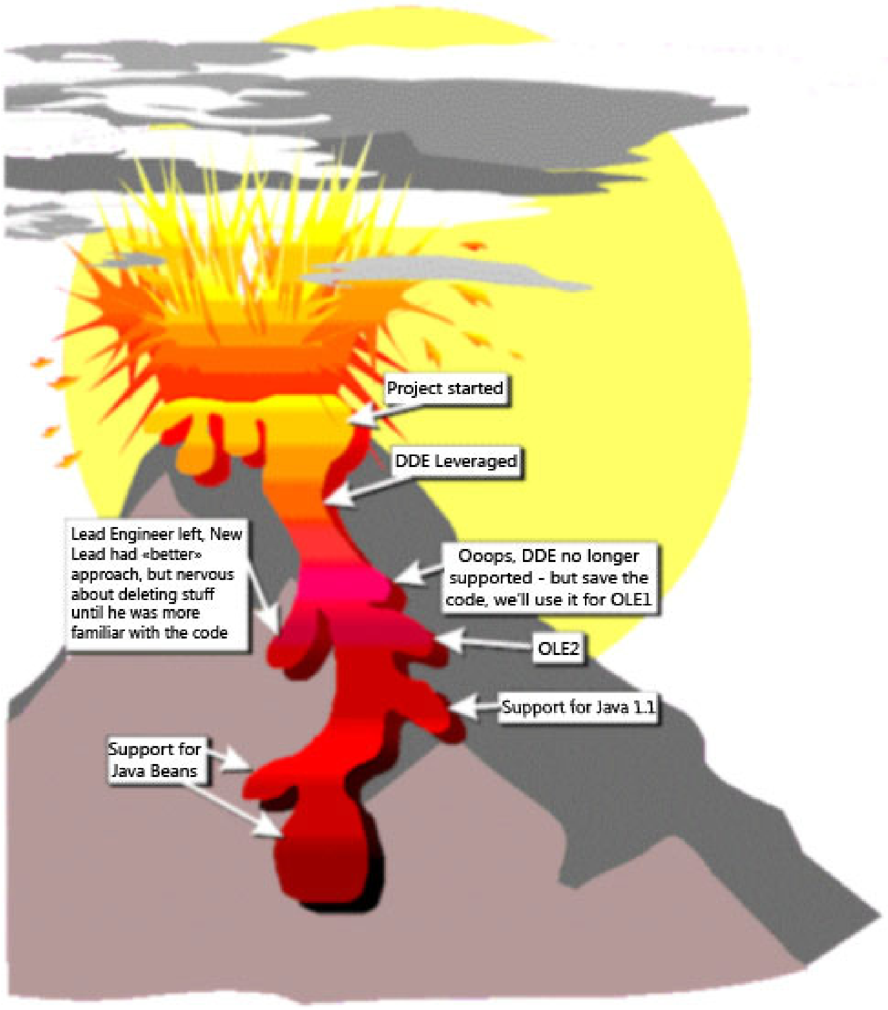

Project started
DDE Leveraged
Lead Engineer left, New Lead had «better» approach, but nervous about deleting stuff until he was more familiar with the code
Ooops, DDE no longer supported - but save the code, we'll use it for OLE1
OLE2
Support for Java 1.1
Support for Java Beans

Figure 1.1: Illustration of the Lava Flow AntiPattern – Source: (Brown, Malveau, McCormick III, & Mowbray, 1998)

To further stress the significance of this AntiPattern, following is a quote of the most evident consequences (that is drawbacks in the case of an 'anti' pattern) with a subsequent check whether or not the particular assertion also held true for the TEALsim framework:

"*It is poor design, for several key reasons:*

a) Lava Flows are expensive to analyze, verify, and test. All such effort is expended entirely in vain and is an absolute waste. In practice, verification and test are rarely possible.

b) Lava Flow code can be expensive to load into memory, wasting important resources and impacting performance.

c) *As with many AntiPatterns, you lose many of the inherent advantages of an object–oriented design. In this case, you lose the ability to leverage modularization and reuse without further proliferating the Lava Flow globules.*" (Brown, Malveau, McCormick III, & Mowbray, 1998)

Brief check of existence of Lava Flow AntiPattern issues in TEALsim source code:

assertion	applicable to TEALsim?	description
a)	✓	Dozens, if not hundreds, of hours were spent analyzing program flow and object creation to trace bugs or find out a way to extend the software. As mentioned in chapter 1.4 prior to this thesis hardly any more comprehensive and centralized (J)unit tests were available.
b)	✓	A good example for this behavior was the rendering system which was fed a list of objects to render which contained every item twice.
c)	✓	Cascades of interfaces assigning roles to classes which made no sense and were not used, e.g. the SimPlayer ultimately was also a TElementManager thus forced to implement all its methods while they were in fact never used (or if so, it was an erroneous usage!)

Table 1.1: Lava Flow AntiPattern issues found in TEALsim source code

Consequentially one part of the practical work for this thesis was to clean up as many dead 'features' and design inconsistencies as possible. To illustrate the results of this effort the next chapter 1.1. summarizes and comments the changes to the code base of TEALsim. Although this task arguably changed the code base to the better, the description for the Lava Flow AntiPattern in (Brown, Malveau, McCormick III, & Mowbray, 1998) also gives a hint at why the framework seemingly worked worse after the end of this thesis than before:

"*As suspected dead code is eliminated, bugs are introduced. When this happens, resist the urge to immediately fix the symptoms without fully understanding the cause of the error.*"

In simple terms, the time spent coding was too short to reach the point where the runtime behavior of the framework inarguably exposed apparent improvements over its former state.

The following sub chapters will cover very general aspects of the TEALsim project, which generally apply to alien projects as well. More technical details like the design of the deterministic simulation engine or the integrated network layer are covered in later chapters.

1.1 Issues with redundant and/or obscure code

As mentioned one of the big challenges encountered during the course of this project was to understand the 'design' of the TEALsim framework and track down its execution flow. Code artifacts of abandoned features and design decisions bloated the amount of code to analyze. The following table 1.1 illustrates with approximate figures the work performed to clean up the frameworks codebase.

	files	blank lines	lines of comment	lines of code
feature branch of TEALsim with SVN revision 150	602	17629	31286	74852
network, runtime argument and test packages added after original check-in to SVN repository	46	772	1930	2654
resulting leftover from original check-in	556	16857	29356	72198
original check-in to SVN repository (revision 2)	582	17902	29534	76205
variance of leftover to original content of check-in	-26	-1045	-178	-4007

Table 1.2: Overview over evolution of lines of code resp. amount of files of TEALsim project[5]

Most mentionable for table 1.1 is the reduction in lines of code by approximately 4 kLOC which would equal to ~5% of the whole project. Usually such code consisted of classes forced to implement methods due to them implementing inappropriate interfaces, or derived classes repeatedly re-implementing the same methods instead of calling into their base class. As reference for 'new code submits' the packages containing TEALsim's new network capabilities, the runtime argument parser and the JUnit test suite were chosen, since these were definitely non-existent previous to this thesis. While a magnitude of changes to files in other packages were applied as well, taking them into exact consideration is arguably quite challenging, for which reason it was assumed that the measurable effects of these changes would cancel each other out.

[5] Figures calculated with the tool CLOC - http://cloc.sourceforge.net/

	files	blank lines	lines of comment	lines of code	lines of comments per line of code
network and runtime argument packages added after original check-in	42	645	1813	2102	0,862511893
resulting leftover from original check-in	556	16857	29356	72198	0,406604061

Table 1.3: Overview over amount of in-line documentation of TEALsim project[6]

To emphasize the improving quality of the framework's code emanating from this thesis' practical work, table 1.3 summarizes the ratio of lines of comments to lines of code for old code versus new code submits. Apparently a degree of documentation more than twice as comprehensive as prior outlines the improving software quality.

Despite the fact that at this thesis' start it was of importance to provide means to effectively test the code base, eventually the attention shifted over to eliminate design flaws and implement new features (e.g. for network capabilities) which mainly left the testing suite behind as a solid starting point for future work in this area. As such, code comprising the JUnit tests are rather unsophisticated in their current state, generally consisting of copied and pasted boilerplate code, which is the reason why the test package was excluded in table 1.3.

	files	blank lines	lines of comment	lines of code	lines of comment per line of code
feature branch of OWL TEALsim module with SVN revision 150	21	437	547	1475	0,370847458
original check-in of OWL TEALsim module	23	783	810	2731	0,296594654
change of project contents from check-in to rev. 150	-2	-346	-263	-1256	0,074252804

Table 1.4: Overview over evolution of OpenWonderland TEALsim module project[7]

For the sake of completeness table 1.4 confirms that changes to the TEALsim framework did not come at the cost of increased complexity in related projects like the Open Wonderland module. Correlative to the developments in the TEALsim code base, the TEALsim OWL module code base was reduced in complexity while getting slightly more thoroughly documented alike. Effectively this turned it into a real player-like wrapper, responsible to embed TEALsim into OWL without redefining aspects or the design of the simulation framework.

[6] Figures calculated with the tool CLOC - http://cloc.sourceforge.net/

[7] Figures calculated with the tool CLOC - http://cloc.sourceforge.net/

1.2 Move from CVS to SVN

One of the first activities was the migration from the formerly used CVS revision control system to the more recent SVN[8] system. Performing this step before anything else was a logic consequence, since experience has shown that the move from one support system to another one often implies the loss of various information while additionally the necessary amount of time to do the migration increases with the amount of information contained in the former system. Motivations for this task were amongst others

- the extended functionality of SVN, which allowed for example to easily move folders (e.g. via drag and drop within Windows Explorer using TortoiseSVN[9])
- a unified workflow while developing TEALsim in parallel to OWL and the corresponding module (since OWL used SVN as well),
- and in general the broader experience with SVN compared to CVS of people involved in this project.

The SVN host of choice was Sourceforge, and the all of the source code is currently publically available[10].

1.3 Integration into Netbeans, rework of the build script

Next on list was the migration of the TEALsim project from Eclipse to Netbeans. Motivation for this work was again the desire to create a more streamlined development workflow, since Open Wonderland itself (and all of its related projects created/maintained by the Open Wonderland Foundation), the TEALsim OWL module as well as one of its most important frameworks – the MTGame Graphics Engine[11] – are also both Netbeans projects. Unifying the amount of utilized integrated development engines to a minimum not only decreases the amount of time required to setup a working station and get involved in the project, but also leverages productivity with the concrete IDE by allowing usage of advanced features like step-by-step debugging with stepping into code of other projects controlled by the same IDE, etc..

With Java and Netbeans getting everything right to allow usage of all of Netbeans' features usually becomes a serious challenge without profound knowledge about Ant[12] build scripts in general and Netbeans way to deal with things in particular. Adding into this consideration the complex structure of the former build script (see figure 1.2), which existed without any documentation or any formal system specification for the deliverables, a serious amount of time was required to achieve a tight and properly working integration with Netbeans.

[8] Subversion revision control system - http://subversion.apache.org/
[9] Tortoise SVN client - http://tortoisesvn.tigris.org/
[10] TEALsim project hosted on Sourceforge - http://sourceforge.net/projects/tealsim/
[11] MTGame Graphics Engine - http://code.google.com/p/openwonderland-mtgame/
[12] Ant software build tool - http://ant.apache.org/

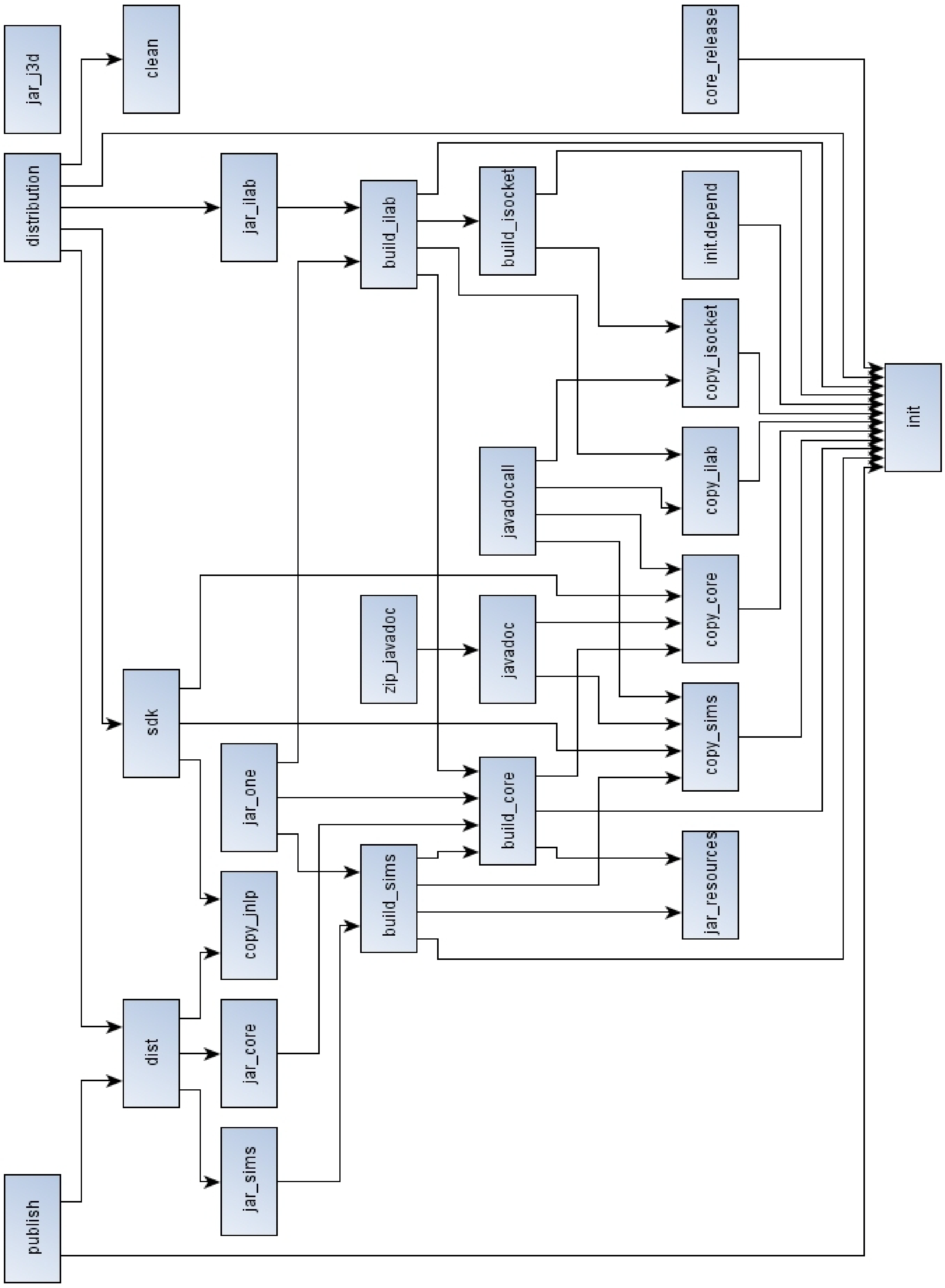


Figure 1.2: Dependency graph of the former TEALsim build script

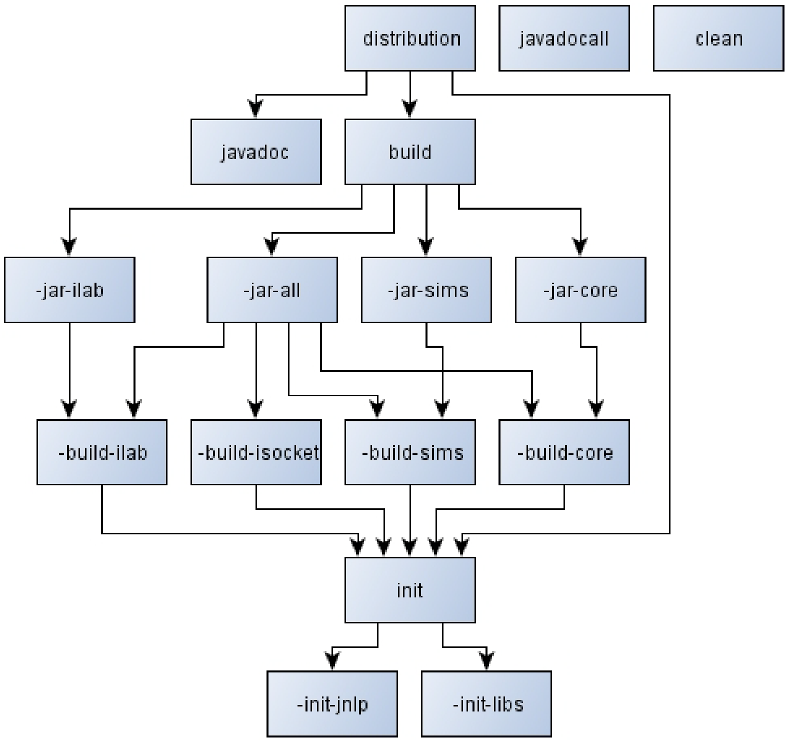


Figure 1.3: Dependency graph of the streamlined TEALsim build file

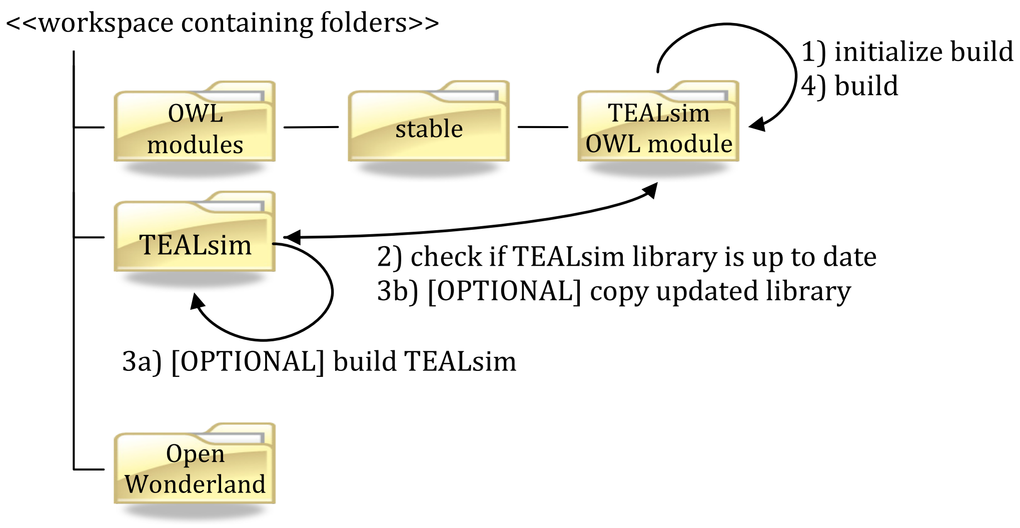

<<workspace containing folders>>
1) initialize build
4) build
OWL modules
stable
TEALsim OWL module
TEALsim
2) check if TEALsim library is up to date
3b) [OPTIONAL] copy updated library
3a) [OPTIONAL] build TEALsim
Open Wonderland

Figure 1.4: Illustration of TEALsim' OWL module build sequence

Figure 1.3 shows the final *build.xml* file used by Netbeans to build, debug and profile the TEALsim project. Besides the TEALsim project also the TEALsim OWL module project was slightly adapted. Most mentionable the module's build script became considerable smarter by usage of a custom Ant-Contrib Tasks[13] build, which allows to check if the module uses the most recent TEALsim library file (that is it checks if compiling TEALsim would yield a more recent binary, and if so triggers the build and fetches the new library – see figure 1.4 for an illustration of this process).

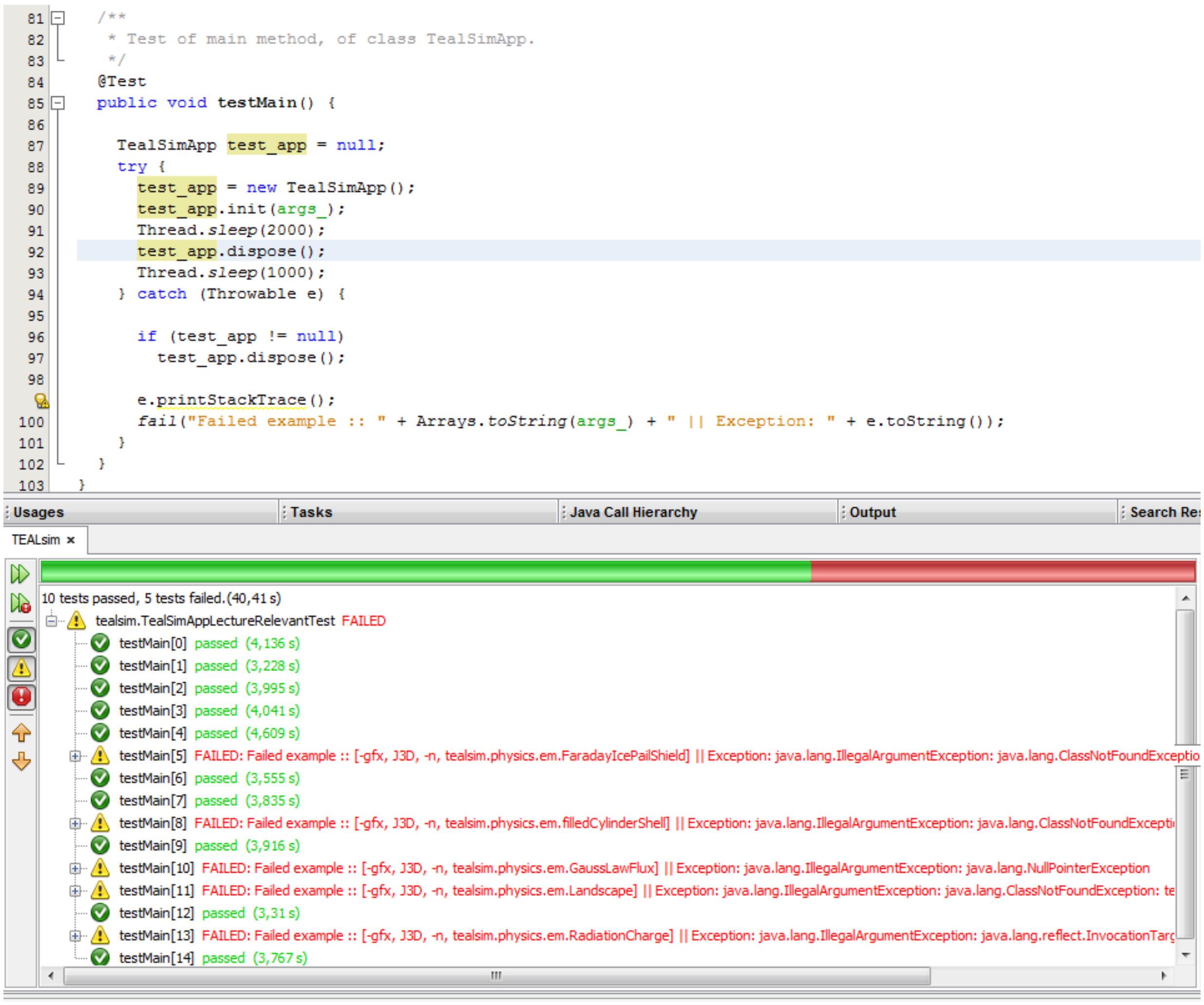
1.4 Addition of (J)Unit Tests

Integral part of any project but maybe very experimental R&D prototypes is the testing suite. Since TEALsim surprisingly did previously not include any extensive package for this task at all, an inevitable duty was to introduce a suitable framework for this purpose. The library of choice was JUnit[14], not least because it is recognized as being a standard tool in Java for test-driven software development paradigms like Extreme Programming (ExtrProgJunit). Another advantage of JUnit is its tight integration into the Netbeans IDE (see figure 1.5) which allows for a smooth workflow while working on the TEALsim code. Due to the amount of time required for other areas of this thesis, unfortunately the testing aspect ultimately ended up being treated rather poorly again. One major issue currently limiting the usefulness of tests was an error with the JME[15] graphics library which refused to close its canvas between single tests, making it impossible to test TEALsim with this particular graphics setting. Presumably the root for this error is way down in the source code of the JME framework. Eventually search for this bug was postponed because of its complexity.

One of the currently available tests resembles a black-box style test (BlckBoxTest) only checking the proper, exception-free startup of TEALsim based on 2 lists of simulations: one comprehensive list, covering almost all available simulations and one subset of the former list consisting of a selection of important simulations used by John Belcher in his classes at MIT.

A few other tests exist as well, checking the correct execution of runtime arguments or the determinism of the simulation engine, but definitely more work in this area would be required.

[13] Ant-Contrib Tasks - http://ant-contrib.sourceforge.net/
[14] JUnit project page - http://junit.sourceforge.net/
[15] JME graphics framework (see the Wonderland branch) - http://code.google.com/p/jmonkeyengine/



```
/**
 * Test of main method, of class TealSimApp.
 */
@Test
public void testMain() {

  TealSimApp test_app = null;
  try {
    test_app = new TealSimApp();
    test_app.init(args_);
    Thread.sleep(2000);
    test_app.dispose();
    Thread.sleep(1000);
  } catch (Throwable e) {

    if (test_app != null)
      test_app.dispose();

    e.printStackTrace();
    fail("Failed example :: " + Arrays.toString(args_) + " || Exception: " + e.toString());
  }
}
}
```

```
10 tests passed, 5 tests failed.(40,41 s)
tealsim.TealSimAppLectureRelevantTest FAILED
  testMain[0] passed (4,136 s)
  testMain[1] passed (3,228 s)
  testMain[2] passed (3,995 s)
  testMain[3] passed (4,041 s)
  testMain[4] passed (4,609 s)
  testMain[5] FAILED: Failed example :: [-gfx, J3D, -n, tealsim.physics.em.FaradayIcePailShield] || Exception: java.lang.IllegalArgumentException: java.lang.ClassNotFoundExceptio
  testMain[6] passed (3,555 s)
  testMain[7] passed (3,835 s)
  testMain[8] FAILED: Failed example :: [-gfx, J3D, -n, tealsim.physics.em.filledCylinderShell] || Exception: java.lang.IllegalArgumentException: java.lang.ClassNotFoundExcepti
  testMain[9] passed (3,916 s)
  testMain[10] FAILED: Failed example :: [-gfx, J3D, -n, tealsim.physics.em.GaussLawFlux] || Exception: java.lang.IllegalArgumentException: java.lang.NullPointerException
  testMain[11] FAILED: Failed example :: [-gfx, J3D, -n, tealsim.physics.em.Landscape] || Exception: java.lang.IllegalArgumentException: java.lang.ClassNotFoundException: te
  testMain[12] passed (3,31 s)
  testMain[13] FAILED: Failed example :: [-gfx, J3D, -n, tealsim.physics.em.RadiationCharge] || Exception: java.lang.IllegalArgumentException: java.lang.reflect.InvocationTarg
  testMain[14] passed (3,767 s)
```

Figure 1.5: Screenshot taken in Netbeans IDE showing the result of a JUnit test run

1.4.1 Addition of a mocking framework

To enable very generic testing of certain components an additional framework providing mocking features was also added to TEALsim. The library of choice was EasyMock[16]. An example of its usefulness is the test which checks the soundness of the algorithm responsible to execute the RuntimeArguments according to the declared specifications (e.g. fail if a developer accidentally declared arguments depending on each other, creating a circular reference). In the concrete test an instance of each available type of RuntimeArguments gets generically created, and this set of arguments is fed to the execution logic. Since the algorithm will eventually execute every single argument - which usually interacts with an environment which is not available for this test - the called method on the RuntimeArgument has to be mocked to prevent exceptions

[16] EasyMock project - http://www.easymock.org/

breaking the test. A mocking library like EasyMock takes care of this task, providing e.g. functions to record calls to mocks. In the test mentioned above every argument has to be executed exactly once to pass the test. Adding mocking capabilities to the test suite enables very elegant, precise tests without the need to create a huge amount of hacks to emulate a required support environment.

1.5 Addition of a logging framework, allowing dynamic adjustment of log settings via e.g. helper windows

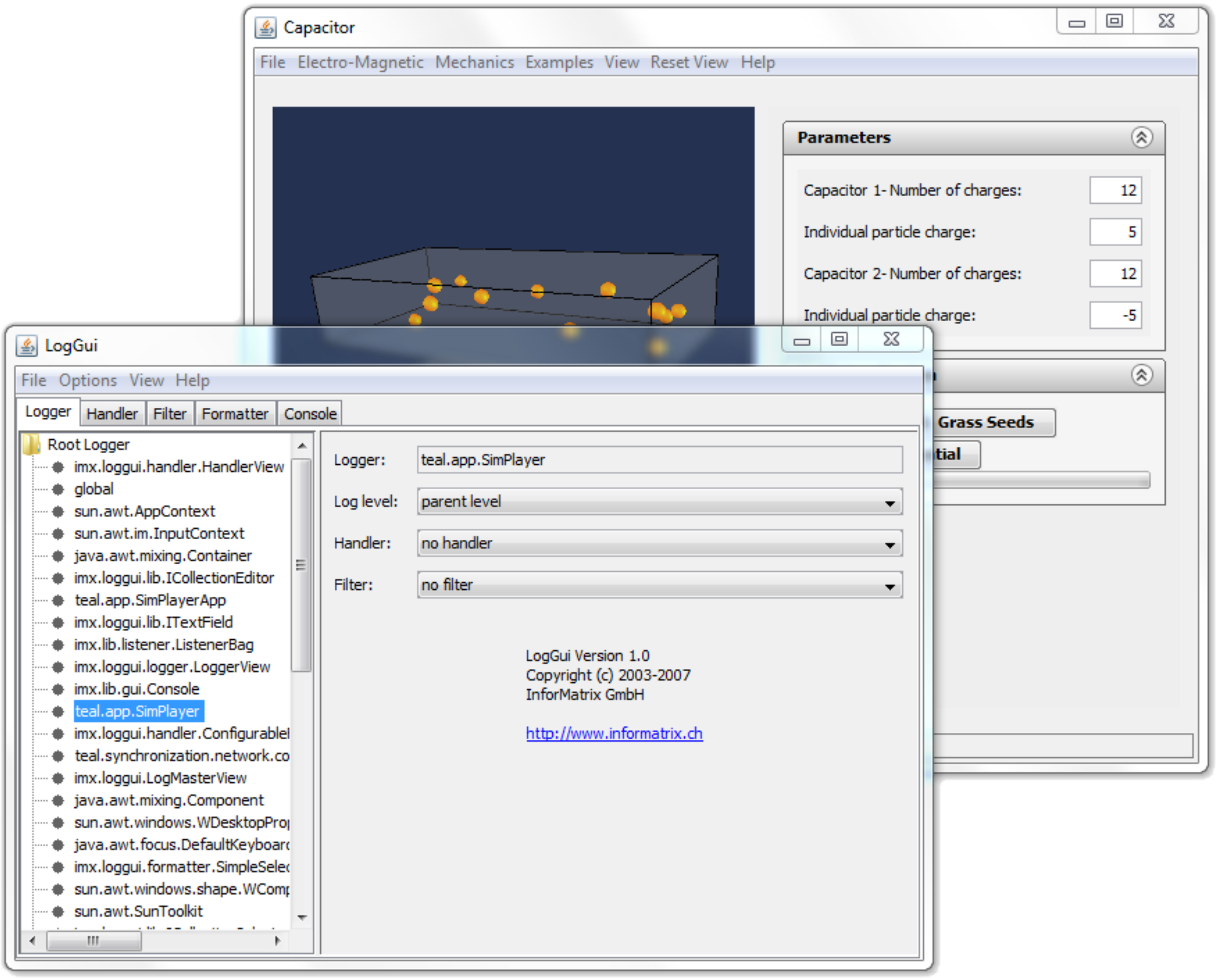
Another missing component indispensable for any larger project was a capable logging facility. While there existed a rudimentary class providing static methods which essentially allowed printing various types of objects to System.out, it definitely suffered from several drawbacks:

- equivalent to the rest of the project hardly any documentation
- resembling old-style C code by being parameterized via plain integers instead of e.g. enums
- being real source code requiring recompilation for changes

To solve these issues any new code written consistently utilized the standardized logging facilities included in the Java framework since JDK 1.4[17], while many of the former debugging messages were changed to do so as well. In the face of the amount of code constituting the TEALsim framework unfortunately not all of the debugging messages utilizing the former debug system could be changed to use the JDK version – this should be done incrementally as further work is done to enhance the framework.

Apparently the JDK logging utilities tackle all of the mentioned problems of the self-made debugging system, adding on top the benefits of outsourcing the task to maintain or even extend this system as well as being a widely recognized standard which allows to use other 3rd party applications to plug into the logging system for even more sophisticated log analyses. As an example for this serves the currently included LogGui (InforMatrix GmbH, 2007) library, which opens up a separate logging window (see figure 1.6)to allow adjustments to the logging output at application runtime!

[17] JDK logging facilities - http://docs.oracle.com/javase/6/docs/api/java/util/logging/package-summary.html


Capacitor
File Electro-Magnetic Mechanics Examples View Reset View Help
Parameters
Grass Seeds
LogGui
File Options View Help
Logger Handler Filter Formatter Console

Capacitor 1- Number of charges: 12
Individual particle charge: 5
Capacitor 2- Number of charges: 12
Individual particle charge: -5

Root Logger
imx.loggui.handler.HandlerView
global
sun.awt.AppContext
sun.awt.im.InputContext
java.awt.mixing.Container
imx.loggui.lib.ICollectionEditor
teal.app.SimPlayerApp
imx.loggui.lib.ITextField
imx.lib.listener.ListenerBag
imx.loggui.logger.LoggerView
imx.lib.gui.Console
teal.app.SimPlayer
imx.loggui.handler.Configurablel
teal.synchronization.network.co
imx.loggui.LogMasterView
java.awt.mixing.Component
sun.awt.windows.WDesktopPro
java.awt.focus.DefaultKeyboar
imx.loggui.formatter.SimpleSele
sun.awt.windows.shape.WCom
sun.awt.SunToolkit

Logger: teal.app.SimPlayer
Log level: parent level
Handler: no handler
Filter: no filter

LogGui Version 1.0
Copyright (c) 2003-2007
InforMatrix GmbH

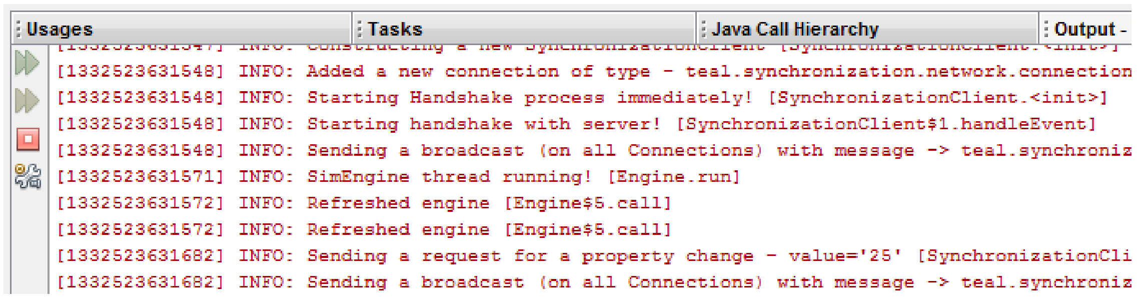
http://www.informatrix.ch

Figure 1.6: Screenshot of TEALsim being started up with an exchangeable tool for adjustments of debugging output at runtime – tool used:(InforMatrix GmbH, 2007)

Besides the mentioned possibility to plug 3rd party applications into the logging system, another of its key features is the dynamic configurability without recompilation. In TEALsim's case this is currently achieved via a configuration file in the project root folder called *logging.properties*. An example for a couple of instructions can be seen in listing 1.1, with a sample of the corresponding logging output in Netbeans shown in figure 1.7.

Listing 1.1: Example for instructions found in *logging.properties* file

```
# Set the standard Loglevels
.level= INFO
# this tells the handler to use our custom formatter
java.util.logging.ConsoleHandler.formatter = teal.util.CustomLoggingFormatter
# you don't need to specify the next line, which demonstrates a very terse log format
teal.util.CustomLoggingFormatter.format =[%t] %L: %m [%C.%M]
```



```
[1332523631547] INFO: Constructing a new SynchronizationClient [SynchronizationClient.<init>]
[1332523631548] INFO: Added a new connection of type - teal.synchronization.network.connection
[1332523631548] INFO: Starting Handshake process immediately! [SynchronizationClient.<init>]
[1332523631548] INFO: Starting handshake with server! [SynchronizationClient$1.handleEvent]
[1332523631548] INFO: Sending a broadcast (on all Connections) with message -> teal.synchroniz
[1332523631571] INFO: SimEngine thread running! [Engine.run]
[1332523631572] INFO: Refreshed engine [Engine$5.call]
[1332523631572] INFO: Refreshed engine [Engine$5.call]
[1332523631682] INFO: Sending a request for a property change - value='25' [SynchronizationCli
[1332523631682] INFO: Sending a broadcast (on all Connections) with message -> teal.synchroniz
```

Figure 1.7: Log messages from customized formatter

1.6 Streamlining the user interface

Another part of the work comprised a slight redesign of the user interface to match widely spread tools which appear to users to have an equivalent behavior like TEALsim. For a screenshot of TEALsim running with its original UI see figure 1.8. The screenshot shows the Capacitor simulation running on a Microsoft Windows machine. This simulation – like almost all of the other available simulations – acts comparable to a movie, that is basically it can be controlled via a set of playback controls familiar from media players (at this place neglecting special controls provided by e.g. ControlGroups, since they were not touched during the course of this project). Figure 1.9 shows a close-up of the previously used playback control (which was called ‘EngineControl’) annotated with the purpose of the particular button.

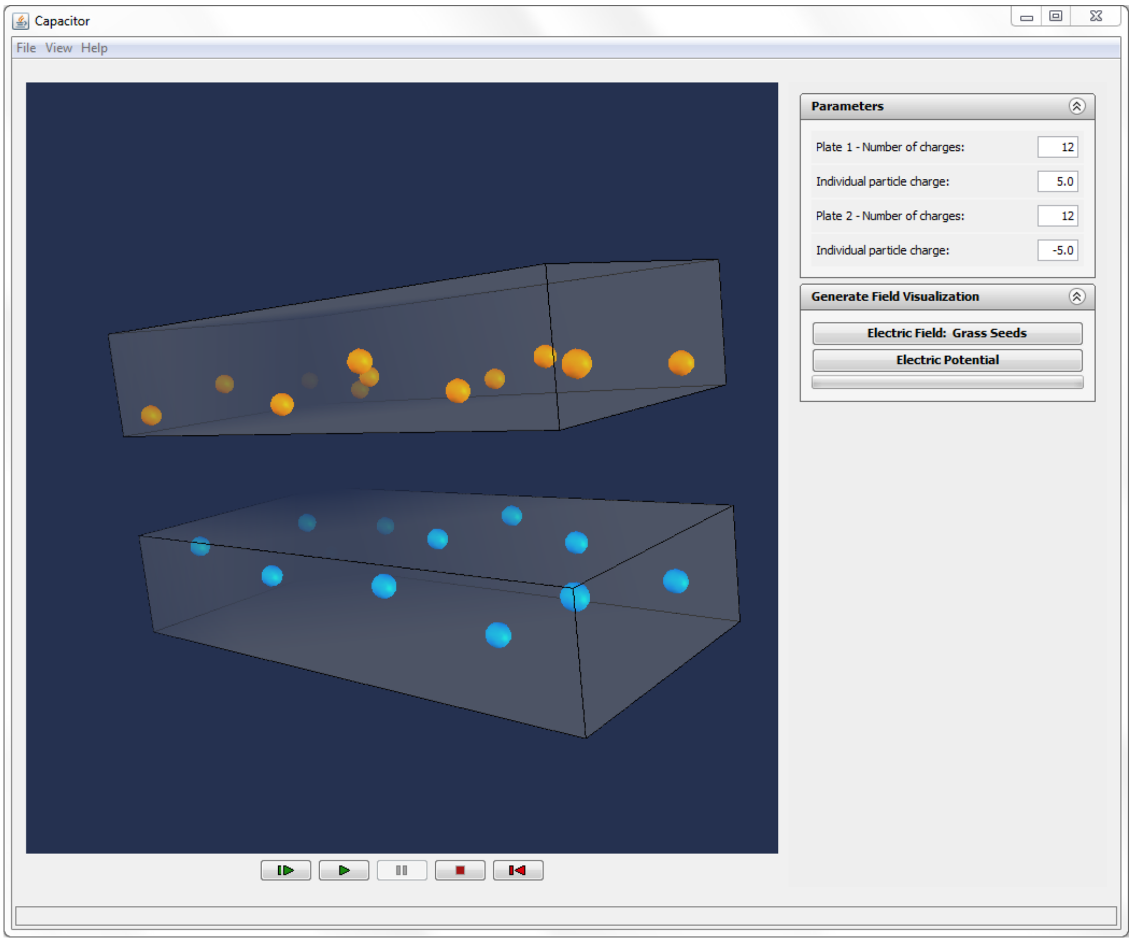


Plate 1 - Number of charges:	12
Individual particle charge:	5.0
Plate 2 - Number of charges:	12
Individual particle charge:	-5.0

Figure 1.8: Screenshot of an older version of TEALsim showing the former user interface

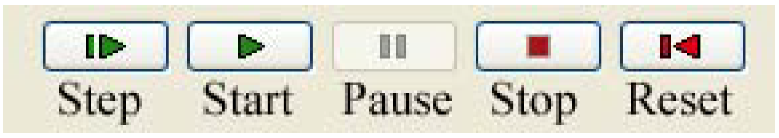


Figure 1.9: The former simulation controls and their associated properties – Source: (Belcher, McKinney, Bailey, & Danziger, 2007)

For a sketch outlining the proposed look of the new UI see figure 1.10. Consecutively is a brief break down of the ideas behind the new buttons:

- Most evidently the new UI emulates modern media player's style to merge the play and pause button to one toggle button, since both states are mutually exclusive.
- The underlying behavior of 'Reset' should change from its former concept of setting everything back to a random value to setting everything back to a defined state (thus repeated resets would always return to the same state). This would be more consistent with the common understanding of what a reset in everyday life usually does. Inevitably this may create the need to provide the means for a real randomized restart

for certain simulations. For this purpose a dedicated button could be placed on demand next to reset, clearly describing its effect.

- The former 'Stop' button was removed since its main purpose was to pause the simulation and prevent it from being started again. For most simulations – e.g. in the case of the capacitor – this did not make much sense after all, while those few simulations which really need this kind of behavior could simply trigger pause and disable the controls themselves.
- Since a button to forward the simulation by a predefined time value ΔT existed ever since, a logic consequence was to provide a button with an inverse behavior as well.
- Last but not least a slider was added to give users control over the pace of the simulation, essentially multiplying the predefined time value ΔT.

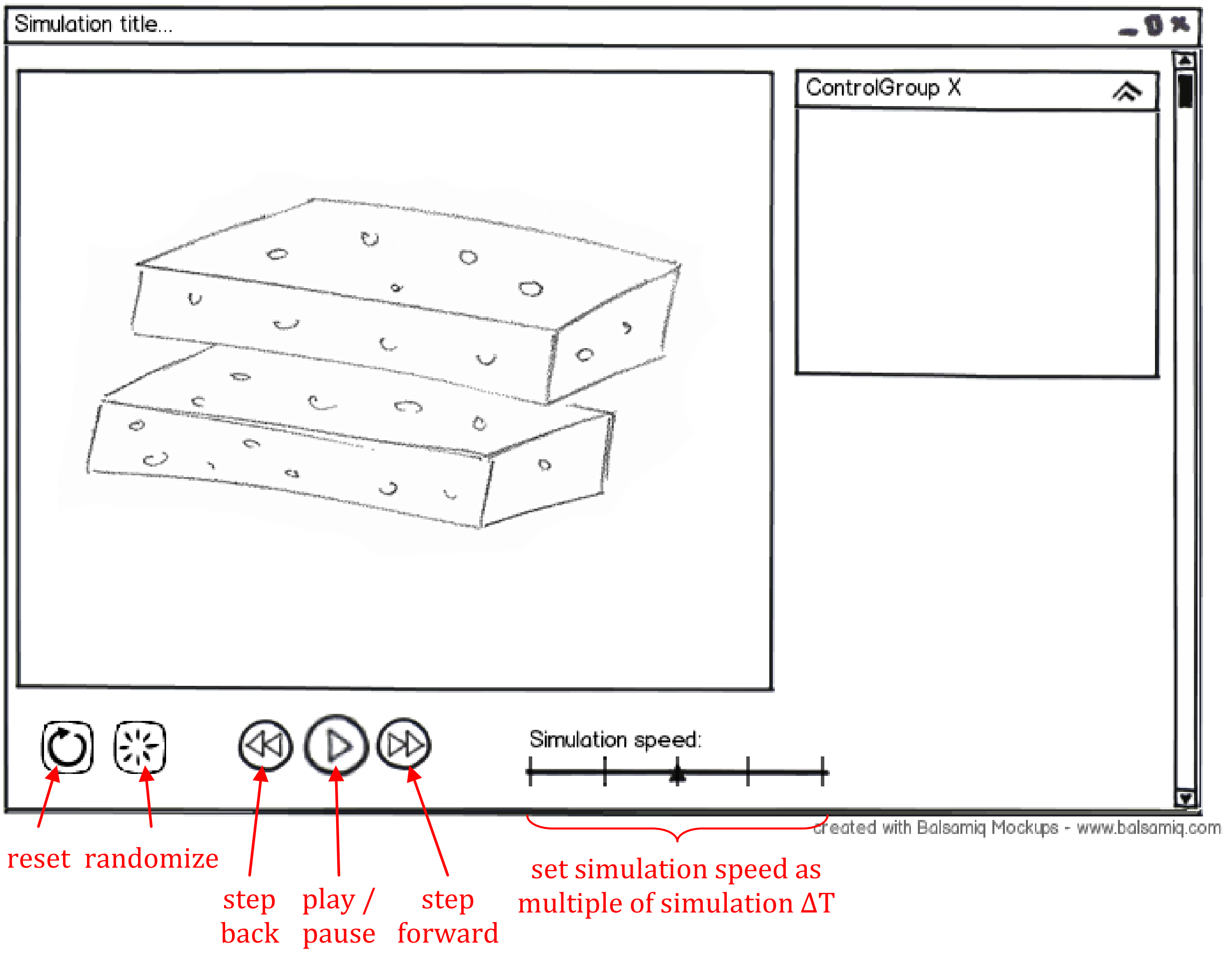

Simulation title...
ControlGroup X
Simulation speed:
created with Balsamiq Mockups - www.balsamiq.com
reset
randomize
step back
play / pause
step forward
set simulation speed as multiple of simulation ΔT

Figure 1.10: Sketch of the proposed refurbished user interface created with the free online version of Balsamiq Mockup tool[18]

[18] free online version of Balsamiq Mockup tool - http://builds.balsamiq.com/b/mockups-web-demo/

Chapter 2

Synchronizing distributed applications

This chapter outlines theoretical aspects related to the synchronization of an application (that is a TEALsim simulation in this case) across multiple clients via a network (and in particular with OWL as host framework). Part of this discourse is a summary of the status quo which was available at the start of this thesis, a suggestion for an alternative approach finishing with a comparison explaining why the new design was eventually introduced to the TEALsim framework.

Subsequently follows a confrontation of the principles and consequences of those two synchronization paradigms which were identified as the fundamental ways how to synchronize user interactions.

2.1 Initial idea

The essential intention behind the former concept was to keep the main project of TEALsim as unchanged as possible, essentially leaving it as a real desktop application while packaging everything related to a client-server architecture into the separate TEALsim OWL module. Inherent to such an approach is the circumstance that for the development of the module in-depth knowledge about TEALsim's architecture is required because core components have to be rebuilt to enable the framework's executability in the distributed environment. Another side aspect is the relative complexity of the module originating from the fact that superimposing a client-server architecture on an underlying desktop system usually causes design inconsistencies and requires considerable hacks to make both worlds work together.

Principles behind the architecture outsourced in a second project

For TEALsim's OWL module the simulation engine was outsourced from the client to the server. To understand the implications of this decision, one has to be aware that – while there are in general many threads running, like in any other modern-day application with a GUI – the simulation-part of the framework adheres to a single-threaded design. In simple terms there is only one thread which is responsible to calculate the state of all of the simulation's elements for the next frame. Upon completion of the calculation the simulation thread informs the rendering package (which can be considered as black-box consisting of an arbitrary amount of threads) to render the view based on the updated states of the elements. Since everything – simulation thread, rendering package, etc. – operates on the same set of states, the simulation thread has to wait for the rendering process to finish before resuming its duty and repeating the same procedure again.

Due to the fact that some possible (usually tricky) use cases were not specified (let alone being implemented – e.g. satisfying solution to slider synchronization) inevitably no final, thorough explanation outlining every detail of the proposed synchronization mechanism can be given. Nevertheless, synchronization of the simulation elements amongst all clients was achieved by putting that part of the simulation logic onto the server-side which is responsible for calculating the dependent values of these objects. In further instance these values were broadcasted to all clients who finished any outstanding auxiliary calculations based on them before triggering the render process. As a result of this design the simulation had to be kept – more or less completely – in memory on both sides - the server as well as on the clients.

In the end this lead to a situation where clients were sort of streaming the simulation states like a video stream.

Performance characteristics of the client-server architecture

To illustrate the effects of this design on bandwidth requirements table 2.1 summarizes the occurring congestion for 3 basic simulations. Measurement of these figures was done by logging the size of EngineMessages being sent from the server to the client (that is after the serialization of a message and before it being handed over to the OWL/RedDwarf Server infrastructure for transmission). As such the values may only be considered as a rough estimation since any further processing of the message on the layers below – possibly increasing or decreasing the total amount of bytes to transmit, e.g. by compressing the messages or adding meta information, etc. – is not reflected in the table. Besides, an important detail is the fact that the table relates to a single client receiving the stream of dependent values. Because OWL currently does not utilize any techniques like multicast (Kaplan, 2012), any successive client joining the simulation increases network congestion by the denoted figures. Additionally the EngineMessages are only dispatched when the simulation is in a running state.

Simulation name	# point charges (PC)	Bytes per frame*	resulting Bytes per PC**	kb/s with 20fps	kb/s with 30fps	kb/s per PC with 30fps
Capacitor	12	1312	109	26	38	3
Capacitor	18	1888	105	37	55	3
Charge by Induction	10	1328	133	26	39	4
Charge by Induction	20	2288	114	45	67	3
Falling Coil	–	224	–	4	7	

* approximate values based on the size of a message dispatched from server to client

** disregarding any overhead potentially caused by e.g. message header, etc.

Table 2.1: Bandwidth usage of OWL module for different simulations

One interesting aspect of table 2.1 is the last column. Considering that the simulation engine is currently capable of calculating at least 200 point charges simultaneously on computer dating from September 2009 (Intel Core i5 2.66 GHz Quad-Core CPU, 4 GB RAM, Nvidia GeForce 260 GTX, Windows 7 64 bit), which would result in approximately 600 kb/s bandwidth utilization for each client streaming such a simulation, it becomes evident there is a conceptual bottleneck (unless network bandwidth is considered to be unlimited).

Two more pieces of information can be extracted from table 2.1 relating to the visual appearance of TEALsim.

First of all the framework used to run with a frame rate of 20 frames per second (fps). Since the human eye requires a frame rate of at least 24 fps to get the impression of smooth transitions (24FPS), the prevailing situation could be considered suboptimal. Therefore the column with the figures for bandwidth requirements with 20 fps is more of a historical record serving as reference point for further thoughts (that is as a basis for the extrapolation for the columns with 30 fps).

Secondly the last two columns show figures with extrapolations of the 20 fps measurements for a frame rate of 30 fps. This value seems to be a fair tradeoff between increased computational load and improved visual presentation, and it is also the (minimum) rate of choice for most video consoles nowadays (30FPS).

Last but not least table 2.1 also outlines that not every simulation takes advantage of the available server-side processing power (and it is open if this could be changed in a way which makes sense). The Falling Coil simulation for example, which is in its current, low-optimized state a very resource hungry application, causes very little network traffic. For simulations like this, the server takes on more of a role matching the scenario describe as alternative approach in the succeeding chapter 2.2 than its intended role as a remote workhorse relieving the clients of computational demanding tasks.

Limitations introduced by 3rd party frameworks

Besides the already mentioned, expectable issues caused by superimposing a client-server architecture on a desktop system, the underlying RedDwarf Server[19] used by OWL introduced another set of limitations, further intensifying the difficulties. Following is a consolidated list of constraints which are of potential relevance for the TEALsim OWL module, taken from (Berger, 2012):

"Code run on PD has to follow several guidelines (RedDwarf Server Application Tutorial, 2010):

a) All objects must implement the serializable interface. Without that the mentioned atomicity of a task can not be provided. PD throws an exception if an object not being serializable.

b) A single managed object must not contain too much data. Otherwise the de-serialization and re-serialization process would take too much time and the task will be thrown away very often.

c) All inner classes should be static since the time taken for the serialization increases significantly if they are not.

d) Synchronization blocks must not be used among managed objects and their members. Since PD uses it's own locks those can conflict with the ones the user defined code uses. This can easily lead to a deadlock.

e) Static fields which are not constant vanish on re-serialization. Although this problem can be solved with Java semantics another problem with this fields appear. Such fields are specific to a single Java virtual machine. This behavior undergoes the feature of PD to run on more than one virtual machine.

f) Java's exception base class java.lang.Exception should never be caught. This is because PD uses its own exceptions which would in this case be caught by the user code. This is especially important for debugging and testing new functionality since the exception base class is often used together with such approaches.

g) No objects except managed objects themselves should be referenced by more than one managed object. After the first serialization process they will not be identical any more since a new object is created on re-serialization."

Recapitulation

Evidently the design explained in this chapter comes with certain limitations which are difficult to overcome based on mere tweaks and minor reiterations. Additionally, complexity of the code turned out to be rather high, for which reason a different approach to handle synchronization will be discussed in the succeeding chapter 2.2.

[19] RedDwarf Server homepage - http://www.reddwarfserver.org/

2.2 Alternative approach

Learn from the best

Starting point for a reconsideration of the existing design is an analysis of what has to be synchronized. In its former state the TEALsim OWL module had to synchronize two things between server and client(s):

a) user inputs (e.g. button clicks, or slider dragging,…)

b) simulation states (of the simulation elements – via EngineMessages)

Item a) – user input – is always going to be unpredictable and will therefore require a mechanism for synchronization hence only item b) – simulation states – remains as a candidate for potential optimization.

Techniques used to build multiplayer video games could help to find the right approach for this issue – after all simulations built upon the TEALsim framework can be regarded as a kind of game as well (particularly since ideas exist to evolve TEALsim into a direction where it allows for even more game-like simulations to be created). For a very popular title in this area, that is the first-person shooter Half-Life, its developer Valve Corporation[20] published a series of articles on its Developer Community web page[21], describing various aspects of their technology. Most interestingly their engine and network logic does not synchronize each frame, but rather expects a certain kind of determinism in the game flow (which equals to the 'simulation flow' with regards to TEALsim). This determinism usually only gets violated by user input. Due to this reason a lot of brainpower was invested to optimize methods for game flow prediction, e.g. by extrapolating and interpolating client states, with one overall goal being to decrease perceivable network latency (subsumed under the term 'lag compensation' – see (Bernier, 2001)). Since a competitive multiplayer game is always prone to hacking attacks by individuals trying to gain an unfair advantage, their system comprises a supervising server instance computing the game flow in parallel to the clients, verifying that user events stay within the boundaries of possible. Still, communication between client(s) and server basically consists of user inputs influencing the game in a way which alter the predictable future, or in other words: only new impulses, changing the 'direction' the game converges to, are transmitted. This is possible because all of the involved parties (server and clients) share the same set of algorithms which calculate the same output based on the same input.

[20] Valve Corporation - http://www.valvesoftware.com/

[21] Valve Developer Community page - https://developer.valvesoftware.com

Apply best practices

Applying this paradigm to TEALsim creates the requirement for the simulation engine to become deterministic. Formally the engine's responsibility is to calculate the next, future state for all simulation elements (= output) based on their former states and a specified amount of time to forward the scenery (= input).

Naturally computational intensive tasks from other, in parallel running processes on the same machine are able to delay the computation of new frames for TEALsim (by racing for computational resources). Beforehand there existed no specified logic to ensure that the engine made up for the time it was lagging behind after such a delay occurred. Put simply, the most important thing that has to be changed to make the engine predictable is its behavior from calculating new outputs on a best effort basis to a stricter specification forcing it to produce defined outputs in certain intervals with the option to fill in supplementary outputs depending on available processing power (and furthermore skip in-between steps/frames as long as it is lagging behind). More details on how the engine is supposed to work will be given in chapter 3.

Assuming that this requirement for determinism can be met, in general no additional synchronization would be required on that score. Just like in the case of Half-Life synchronization could be restricted to mere user inputs, drastically reducing bandwidth utilization. Above all no need for server-side calculations exists for the TEALsim framework because it is currently not intended for use in a real competitive environment hence the issue how to ensure that integrity is maintained is of no concern. For this reason the sever could resemble a lightweight broker service responsible to interconnect its clients, resolve race conditions occurring between clients trying to change the same simulation element within a short time interval as well as store an up-to-date version of the current simulation state (which could be queried from one of the connected clients) for persistence purposes .

Impacts on TEALsim's design

As mentioned adoptions to the simulation engine are necessary to ensure its determinism. Beyond that, the integral part of the proposed design is the shift of TEALsim's design from desktop to client-server architecture, regardless of the actual environment the simulation is intended to run in. The background for this paradigm change is the notion that it is easier – and from a design point of view more consistent – to emulate a distributed environment for a desktop application than it is to superimpose a client-server architecture on a desktop application.

For technical details how this aspect was realized in TEALsim see chapter 6. In general the framework references a minimalistic connection interface for which an appropriate implementation gets instantiated depending on the prevailing host environment.

The client-server architecture itself is of an authoritative nature, which means that the client (the TEALsim simulation) indispensably requires a server to authorize user inputs. Though, due to the abstraction of the connection the client does not care whether the server runs in parallel on the local machine (either as independent process or in the same Java virtual machine) or remotely – it is the concrete connection's responsibility to close the gap between client and server.

Besides the need for a package containing the network layer and changes to the simulation engine, the third mentionable effect of the new design on TEALsim is its requirement to slightly adapt all existing simulations. Similar to the concept of multi-threaded programming, where the Java language provides standardized constructs to synchronize concurrent threads on various levels of scope (e.g. via synchronized methods and blocks, reentrant locks, etc.), equivalent decisions have to be made when designing applications for distributed execution. To reduce the amount of work required to adjust the whole project to adhere to the new paradigm two major types of objects will be part of the network package:

1) A generic property which meets the requirement to set its value asynchronously only after server authorization. In further instance it will also be used to provide synchronization on block level.
2) A package of properties referencing Swing UI elements, which can be synchronized with all other clients and furthermore linked to generic properties

Technical information regarding these properties can be taken from chapter 8.2.4.

Recapitulation

The proposed alternative design is based on paradigms successfully used for proprietary multiplayer games. It will inevitably increase the amount of TEALsim's code base (and consequently also the complexity by a certain degree), since the network capabilities become integral part of the framework instead of being outsourced to secondary projects. In return any project aiming for distributed execution will become more comprehensible and more consistent in design by a magnitude. The optimization of the system for client-server architecture will provide considerably improved performance in this area while suffering – if at all – from negligible performance hits in the desktop scenario. Once adapted to the new architecture, simulations built on the TEALsim framework can be used with little to no adjustments in frameworks like OWL. Beyond that no noticeable additional work to create new simulations should derive from the new architecture due to the availability of simple to use properties which encapsulate the synchronization logic.

2.3 Comparison of former design versus alternative design

In this chapter the term '*former design*' relates to the concept explained in chapter 2.1 Initial idea, whereas the term '*alternative design*' references the ideas depicted in chapter 2.2 Alternative approach.

The following table summarizes various relevant aspects which will be briefly explained in the listed chapters thereafter. The assessment of the superiority of one design over another shall serve as a simple and quick reference to the conclusions which can be drawn from the explanations in the corresponding chapters. In this case a check means that one design could be considered superior to the other design for this particular aspect.

aspect	superiority	
	former design	alternative design
2.3.1 Complexity of TEALsim framework	✓	
2.3.2 Complexity of OWL module		✓
2.3.3 Complexity to create new content (simulations)	✓	✓
2.3.4 Network congestion / scalability	(✓)	✓
2.3.5 Code extensibility / autonomy		✓
2.3.6 Versatility		✓
2.3.7 Server hardware requirements		✓
2.3.8 Client hardware requirements	(✓)	✓

Table 2.2: Summary of pros and cons for the former compared to the alternative synchronization design

2.3.1 Complexity of TEALsim framework

Considering TEALsim's code base isolated from all other related projects, the client-server architecture of the alternative design doubtlessly adds in a certain amount of complexity compared to the former design. In this context it is important not to mix code and complexity reducing effects of the cleanup duties described in chapter 1.1 with the consequences to the framework's code caused by the new design. Instead the former activity has to be seen independently and it would have benefited both designs alike.

2.3.2 Complexity of OWL module

Unlike the reduction of code in TEALsim's main project (as discussed in the former chapter 2.3.1) is the decreased size and complexity of TEALsim's OWL module (see chapter 1.1) directly related to the introduction of the new client-server architecture. This is due to the downgrade of OWL's role to something resembling a simple media player.

In fact everything that was needed to make the TEALsim simulations run in OWL was to create a specific implementation of the connection system as well as provide adapted classes to allow embedding the viewer into the OWL world (which already existed for the module based on the former design).

Currently there are still some classes with redundant/obsolete code left in the module; further work in this area with the goal to completely take away the requirement to know any TEALsim framework internal details, would make it even easier to embed TEALsim in 3rd party applications.

2.3.3 Complexity to create new content (simulations)

Complexity to create new simulations should stay equal regardless of the underlying design of the TEALsim framework. The documentation of the former design owes detailed answers to certain use cases which require synchronization. For this reason it is sensible to assume that eventually similar mechanisms like those described in chapter 8.2.4 would have to be implemented as well to cover the fundamental types of synchronization levels (see chapter 2.5). With such tools as explained in chapter 8.2.4 content developers have a fine-grained control over what they want to be synchronized on an easy to use basis.

2.3.4 Network congestion / scalability

Network congestion caused by synchronization of user inputs should be almost identical for both designs.

Beyond that the flexible operation mode of the alternative design requires a distinction of the expectable benefits depending on the situation:

- Compared to the former design, network traffic is reduced by a magnitude in the alternative design's standard mode, since synchronization amongst all clients is generally done implicitly by a deterministic simulation flow. For one (capable) client little bandwidth usage will arise from the server's subscription to receive regular updates of the simulation state (to share with connecting clients).
- In this context the worst case scenario occurs with computational weak clients (like smartphones or tablets) who could switch into an operating mode resembling the former design (where they will receive a stream of simulation states to avoid having to compute the simulation flow themselves).

2.3.5 Code extensibility / autonomy

With the former design TEALsim's network capabilities originated solely from the (tight integration into the) OWL framework. This situation could be described as a vendor lock-in because no easy integration of TEALsim in any other 3rd party multi-user framework was conceivable.

Furthermore extension of the main project itself was challenging, not least because of the split engine design serving as basis for the OWL module. When trying to change the TEALsim framework this setup made it difficult to entirely comprehend and anticipate the consequences on depended projects like the OWL module. In absence of extensive tests to verify the code's functionality this issue becomes especially severe.

The alternative design packages everything into one coherent project. As a result verification of the correct operation of all features becomes more centralized and consequently easier and more reliable. In further instance the development of new features for TEALsim should become more rapid due to isolating changes to one (small) project instead of having to work with multiple interconnected projects, which all come with their own set of limitations and rules regarding e.g. debugging, compilation etc.

2.3.6 Versatility

Not least due to its flexibility to switch between operating modes, the alternative design opens up additional possibilities for future developments and fields of operation of TEALsim.

On the one hand focus could be put on the simulation core to cover more physics experiments or even other fields of science again (e.g. biochemistry). In line with such developments would be the tasks to try to add more sparkle and/or improve performance on the visualization side. This direction would especially benefit computational powerful computers like laptops or workstations.

On the other hand attention could be directed to work on use cases relating to the execution of TEALsim on slower devices. Such devices often come with a unique set of constraints to mind, like reduced screen sizes which would need special adjustments to the user interface to allow for a good user experience.

While the former design operated similar to the streaming mode of the alternative design and thus shares the same set of possible future fields of operation, its tolerance for more graphical brilliance is potentially limited by the available network capacity.

2.3.7 Server hardware requirements

Hardware requirements for the server are on its bare minimum for the alternative design. In its current conception this design does not offload any computational intensive tasks from the client-side on to the server-side. Instead the server merely has to decide if incoming requests are admissible and forward them. Consequently a server running the alternative design's service could doubtlessly handle a vast amount of clients and/or simulations simultaneously.

With the former design the threshold for concurrent clients and simulations was much lower. For the maximum amount of clients the available network bandwidth was decisive (see chapter 2.1 and in particular table 2.1). Finding a feasible set of simulations the server could handle was even more complex given the awareness that besides significant computational requirements for the simulations themselves also auxiliary tasks of the server, which were running in parallel and consuming performance, had to be considered.

2.3.8 Client hardware requirements

In theory the former design was based on the concept that computational intensive tasks – like calculation of the simulation flow – should be delegated from the client to the server, thereby relieving the client. In practice not all simulations were equally suitable for such a divided computation model (see table 2.1 and the related discussion), hence slow clients could only run a subset of the available simulations whereas fast clients were not always able to take full advantage of their computing power but were often forced to idle waiting for the server to provide missing parts of the calculation.

For slow clients running TEALsim in the streaming mode of the alternative design equal constraints to the former design hold true.

The true benefit of the alternative design lies in its better utilization of otherwise untapped resources. In this regard the client hardware requirements could be considered to be higher, which becomes irrelevant though due to the possibility to switch over to the streaming mode.

2.4 Issues of implicit synchronization of simulation calculation

2.4.1 Double precision divergence across multiple clients

One serious challenge impeding the task to create a flexible but still deterministic simulation engine emanates from the definition of floating point numbers. Since computers on their lowest level can only handle two different types of values – zero and one – ultimately everything has to be mapped to a binary presentation. The same holds true for floating point numbers. Mathematical operations which appear easy to solve to a human might effectively return unexpected results once executed on a computer. An example for such an issue can be taken from listing 2.1 (Retain precision with Doubles in java, 2008).

Listing 2.1: Code sample to demonstrate precision issue with floating point numbers

```java
public class doublePrecision {
  public static void main(String[] args) {
     double total = 0;
     total += 5.6;
     total += 5.8;
     System.out.println(total); //prints 11.399999999999
  }
}
```

Based on the mentioned considerations the following, initial approach to log the elapsing SimulationTime could not be used, even though it might seem perfect written on a piece of paper.

Idea:

$$\begin{array}{ll} \Delta \mathrm{T} & 1000\ [\mathrm{ms}] \\ \Delta \mathrm{step} & x \\ \hline \multicolumn{2}{l}{x = \Delta \mathrm{step} * 1000 / \Delta \mathrm{T}} \end{array}$$

Explanation:

In the example above we increase the simulation exactly by a value of $\Delta \mathrm{T}$ over the course of 1 second. By computing the next engine state (that is simulation state) based on a $\Delta \mathrm{step}$ means to forward the virtual SimulationTime by the factor *x*. Since computation of *x* requires a multiplication and a division the result of the operation is almost certainly prone to minor rounding. Considering the rounding to occur for each frame the engine calculates it becomes obvious that various clients with an alternate frame rate will quickly deviate from each other.

2.4.2 Divergence across multiple clients when computing certain algorithms with different amount of steps

Besides rounding occurring due to floating point numbers exceeding the amount of information storable in their associated objects in computer memory, also the underlying algorithms describing physical effects may produce slightly varying results depending on the amount of sub-steps used to compute an aggregated end result.

2.5 Synchronization at property level compared to synchronization on event level

Listing 2.2: Pseudo code declaring a basic UI and the relation of its elements

```
Button btn = new Button("Increment");
TextField field1 = new TextField(0);
TextField field2 = new TextField(0);

btn.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    field1.setValue(field1.getValue() + 1);
    field2.setValue(field2.getValue() + 1);
  }
});
```

2.5.1 Concept of synchronization on event level

Synchronization on event level could be considered as the counterpart of Java's multithreading synchronization on block respectively method level. With the example in listing 2.2 in mind this would result in a single message to the server trying to synchronize the ActionEvent which occurred after a user clicked the Increment button.

2.5.2 Concept of synchronization on property level

For the example given in listing 2.2 synchronization on property level would mean to synchronize each call to a textfield's setValue method independently. Depending on the context this might cause problems when successive, unsynchronized code references e.g. the textfields which were not yet synchronized (that is the response from the server is still missing).

Chapter 3

Designing a deterministic simulation engine

3.1 Issues of former design

Put simply, the initial simulation engine algorithm was just not predictable respectively configurable enough to use it in an environment with the requirement to have an algorithm which reaches an exactly defined state at an exactly defined point of time to execute arbitrary tasks without impacting its overall determinism. One of the reasons why a complete redesign of the simulation algorithm was preferred over an - as minimalistic as possible – adaption of the existing one was definitely the lack of a thorough design-document specifying the in place procedure. As a result, the only way to get to understand the engine's design was to read the (sparsely, if at all documented) source code. Part of this challenge was the complexity of the engines execution flow based on the amount of states it was able to assume (see Figure 8.3)

3.2 Ideas behind a 'deterministic' algorithm

The new engine algorithm is implemented in a flexible way which allows it to compute more or less sub-steps (so called in-between frames) depending on the underlying hardware capabilities. For each calculation round the engine advances the simulation by a fraction of the totalΔT called Δstep. ΔT is the amount of SimulationTime to forward the simulation from one key frame to the next key frame. Consequently Δstep is the amount of SimulationTime from one in-between frame to the next in-between frame.

For a given simulation ΔT is either set to a specific value by the simulation creator, or otherwise a standard value from the AbstractEngine is used. During runtime this value can be adjusted via the available slider (see figure 1.10).

3.3 Concept of the configurable simulation engine

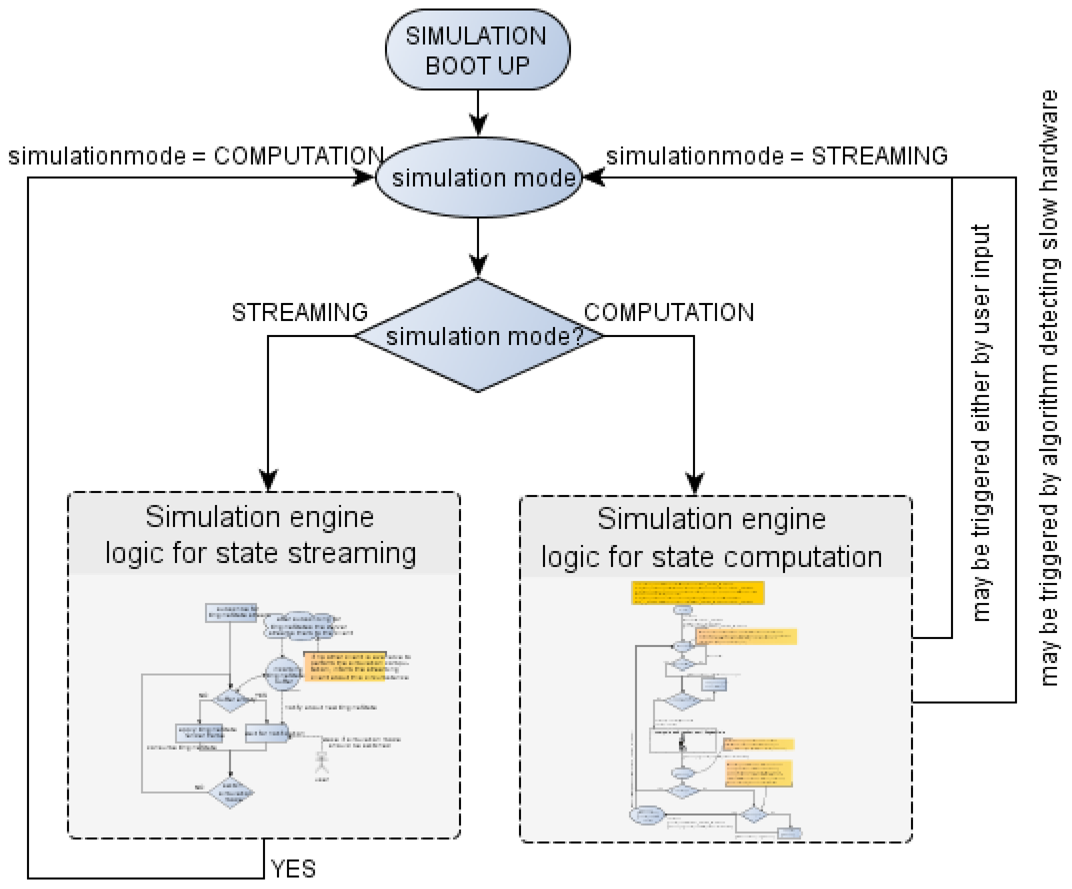
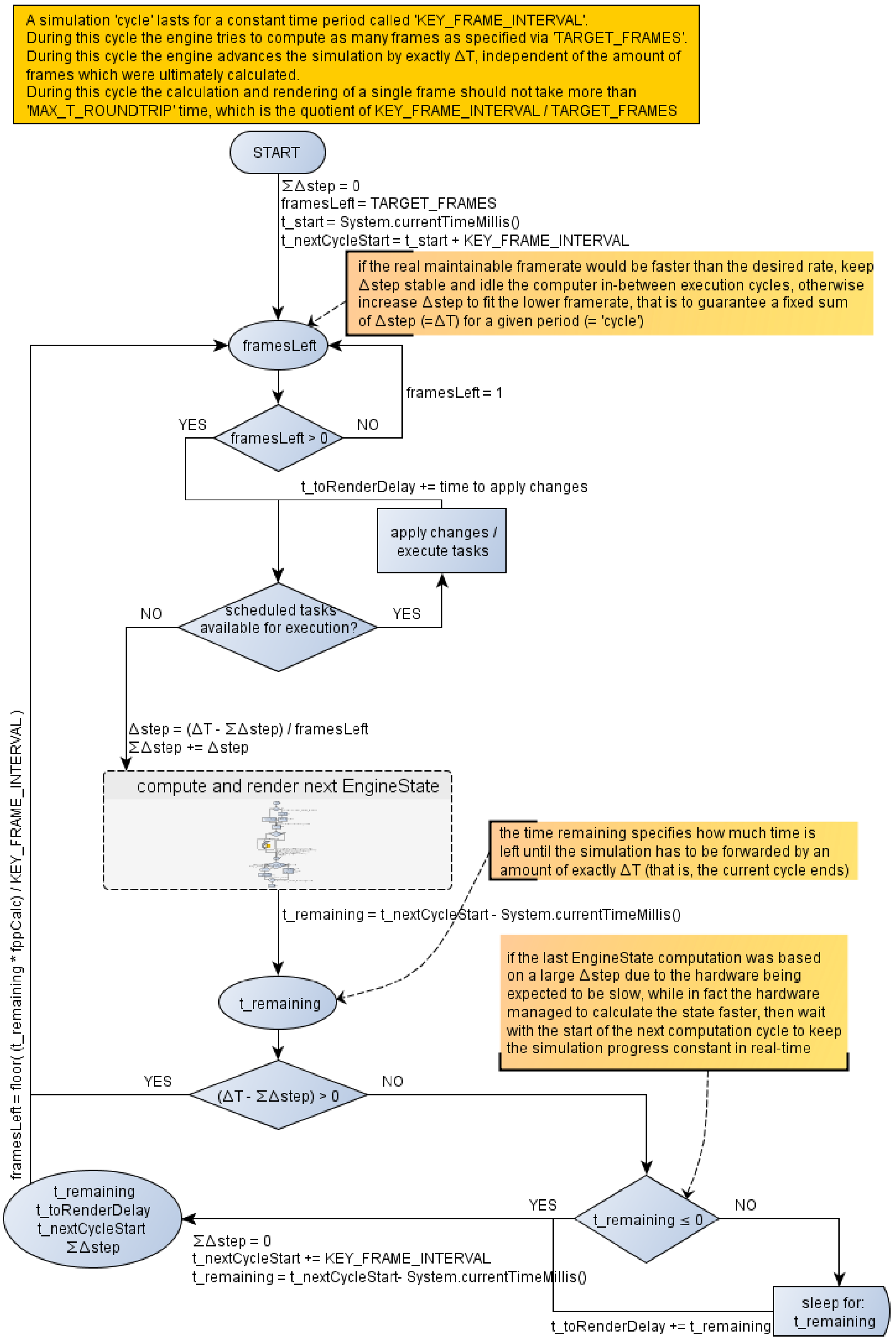

SIMULATION BOOT UP
simulationmode = COMPUTATION
simulation mode
simulationmode = STREAMING
STREAMING
simulation mode?
COMPUTATION
Simulation engine logic for state streaming
Simulation engine logic for state computation
may be triggered either by user input
may be triggered by algorithm detecting slow hardware
YES

Figure 3.1: Diagram of the concept of a switchable simulation engine

3.4 Design of the algorithm for local simulation calculation


A simulation 'cycle' lasts for a constant time period called 'KEY_FRAME_INTERVAL'.
During this cycle the engine tries to compute as many frames as specified via 'TARGET_FRAMES'.
During this cycle the engine advances the simulation by exactly ΔT, independent of the amount of frames which were ultimately calculated.
During this cycle the calculation and rendering of a single frame should not take more than 'MAX_T_ROUNDTRIP' time, which is the quotient of KEY_FRAME_INTERVAL / TARGET_FRAMES

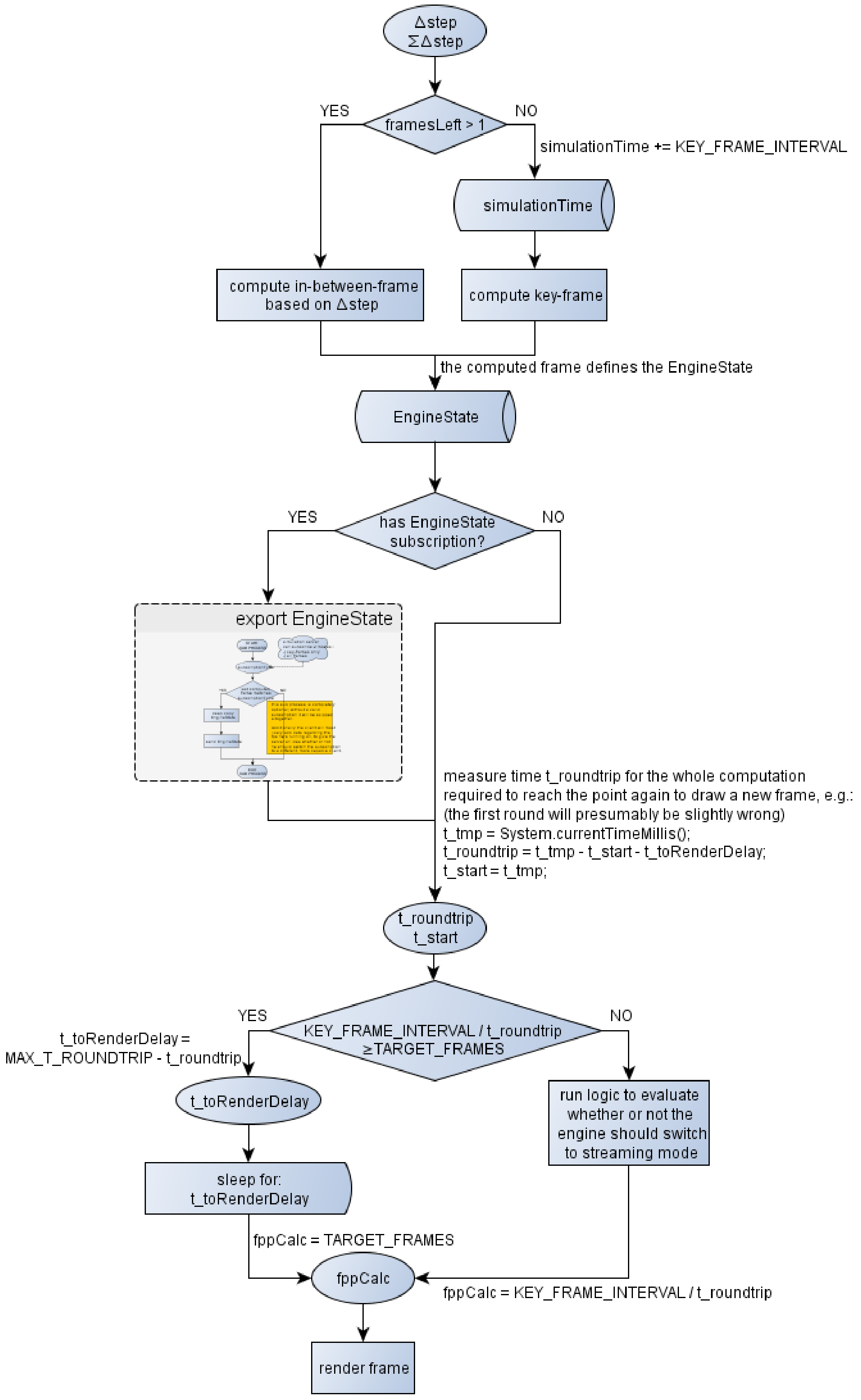
if the real maintainable framerate would be faster than the desired rate, keep Δstep stable and idle the computer in-between execution cycles, otherwise increase Δstep to fit the lower framerate, that is to guarantee a fixed sum of Δstep (=ΔT) for a given period (= 'cycle')

the time remaining specifies how much time is left until the simulation has to be forwarded by an amount of exactly ΔT (that is, the current cycle ends)

if the last EngineState computation was based on a large Δstep due to the hardware being expected to be slow, while in fact the hardware managed to calculate the state faster, then wait with the start of the next computation cycle to keep the simulation progress constant in real-time

START
ΣΔstep = 0
framesLeft = TARGET_FRAMES
t_start = System.currentTimeMillis()
t_nextCycleStart = t_start + KEY_FRAME_INTERVAL
framesLeft
framesLeft = 1
YES
framesLeft > 0
NO
t_toRenderDelay += time to apply changes
apply changes / execute tasks
NO
scheduled tasks available for execution?
YES
Δstep = (ΔT - ΣΔstep) / framesLeft
ΣΔstep += Δstep
compute and render next EngineState
t_remaining = t_nextCycleStart - System.currentTimeMillis()
t_remaining
YES
(ΔT - ΣΔstep) > 0
NO
framesLeft = floor((t_remaining * fpsCalc) / KEY_FRAME_INTERVAL)
t_remaining
t_toRenderDelay
t_nextCycleStart
ΣΔstep
YES
t_remaining ≤ 0
NO
ΣΔstep = 0
t_nextCycleStart += KEY_FRAME_INTERVAL
t_remaining = t_nextCycleStart- System.currentTimeMillis()
sleep for: t_remaining
t_toRenderDelay += t_remaining

Figure 3.2: Overall design of the deterministic algorithm for local simulation calculation


ΔstepΣΔstep
framesLeft > 1
YES
NO
simulationTime += KEY_FRAME_INTERVAL
simulationTime
compute in-between-frame based on Δstep
compute key-frame
the computed frame defines the EngineState
EngineState
has EngineState subscription?
YES
NO
export EngineState
t_roundtrip t_start
KEY_FRAME_INTERVAL / t_roundtrip ≥TARGET_FRAMES
YES
NO
t_toRenderDelay = MAX_T_ROUNDTRIP - t_roundtrip
t_toRenderDelay
sleep for: t_toRenderDelay
run logic to evaluate whether or not the engine should switch to streaming mode
fppCalc = TARGET_FRAMES
fppCalc
fppCalc = KEY_FRAME_INTERVAL / t_roundtrip
render frame

```
measure time t_roundtrip for the whole computation
required to reach the point again to draw a new frame, e.g.:
(the first round will presumably be slightly wrong)
t_tmp = System.currentTimeMillis();
t_roundtrip = t_tmp - t_start - t_toRenderDelay;
t_start = t_tmp;
```

Figure 3.3: Computational part of the algorithm doing simulation calculation locally

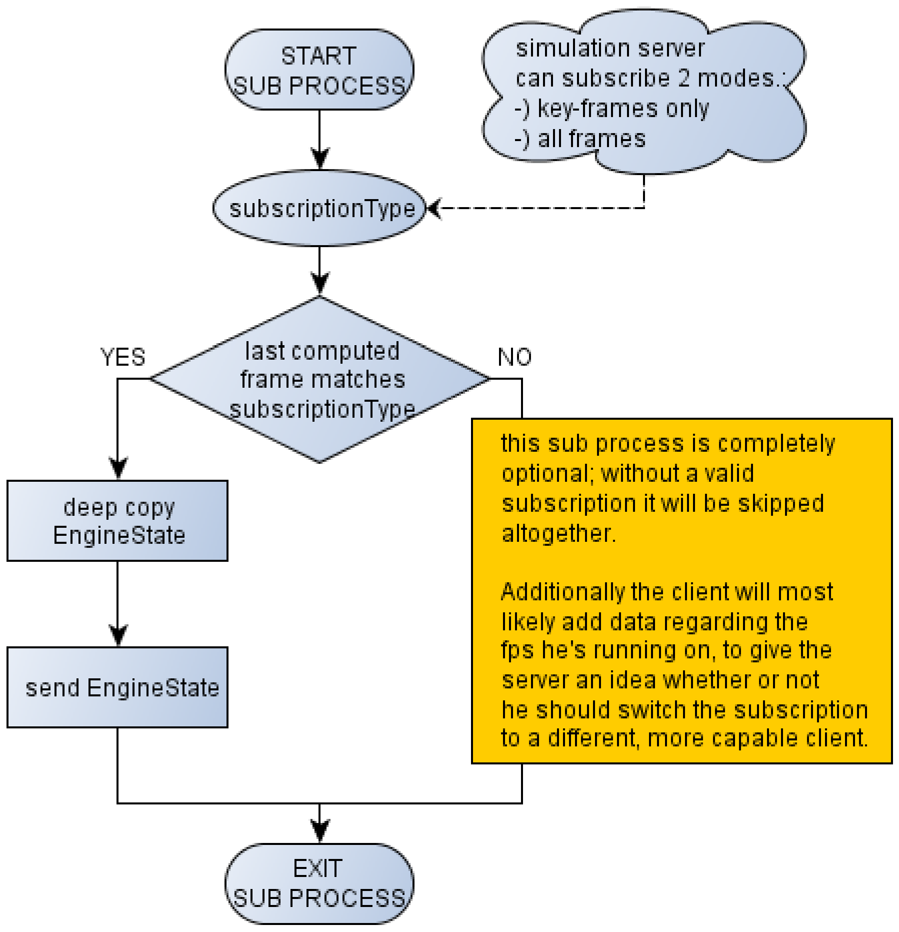

START SUB PROCESS
simulation server can subscribe 2 modes.:
-) key-frames only
-) all frames
subscriptionType
last computed frame matches subscriptionType
YES
NO
deep copy EngineState
send EngineState
this sub process is completely optional; without a valid subscription it will be skipped altogether.
Additionally the client will most likely add data regarding the fps he's running on, to give the server an idea whether or not he should switch the subscription to a different, more capable client.
EXIT SUB PROCESS

Figure 3.4: Logic to export engine states to the remote computers (most notably the server)

3.5 Details of the simulation engine running in state streaming mode

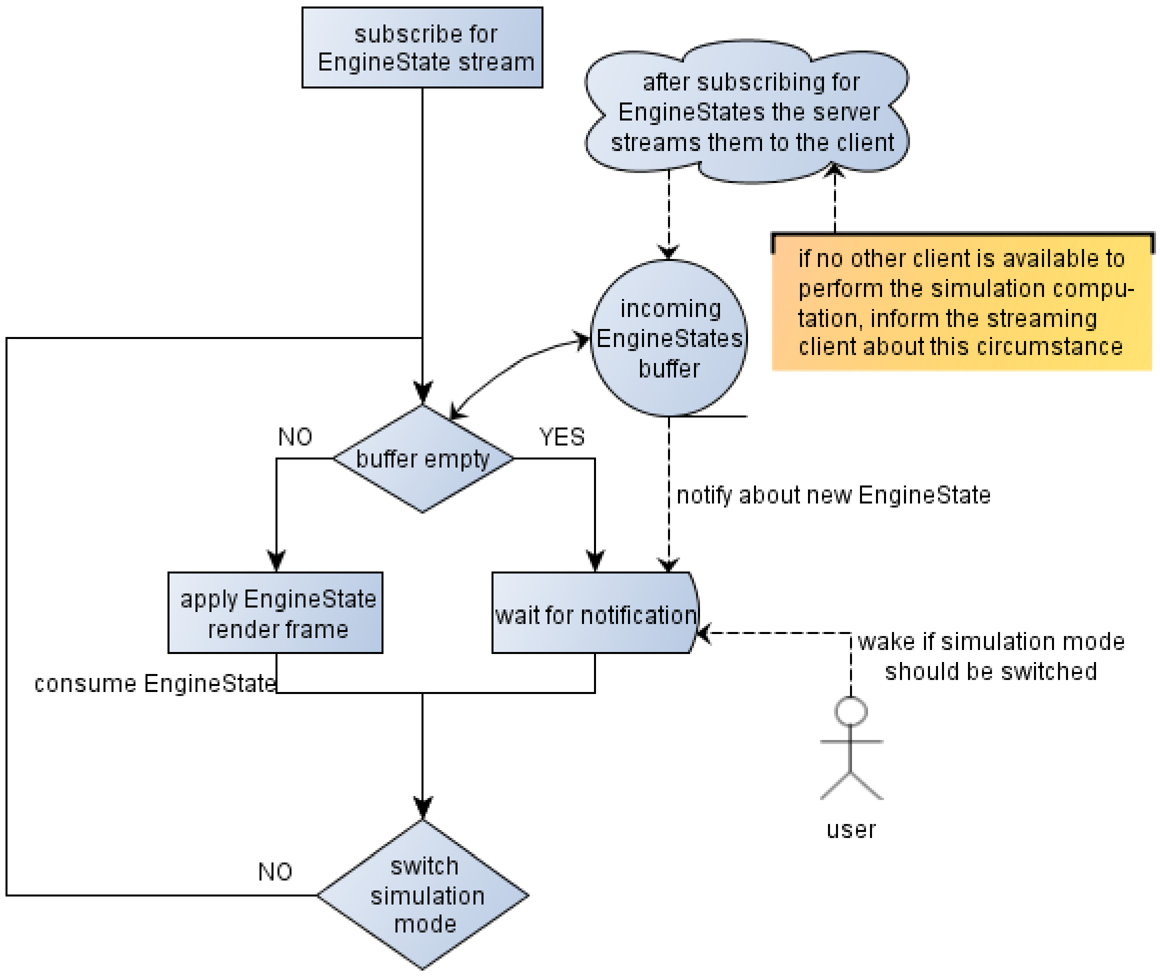

subscribe for EngineState stream
after subscribing for EngineStates the server streams them to the client
if no other client is available to perform the simulation computation, inform the streaming client about this circumstance
incoming EngineStates buffer
NO
buffer empty
YES
notify about new EngineState
apply EngineState render frame
wait for notification
wake if simulation mode should be switched
consume EngineState
user
NO
switch simulation mode

Figure 3.5: Execution flow for simulation engine when streaming simulation states

Chapter 4

Defining the expected system behavior

In the successive chapter – unless explicitly specified in another way – the expression ‘*client*’ refers to the TEALsim client side implementation whereas the expression ‘*server*’ refers to the TEALsim server side implementation.

The expression ‘*ControlState*’ serves as a placeholder term for something which may be changed by the user but which has to be synchronized between all clients (and therefore has to be previously authorized by the server). An example for this would be the simulation state (running or paused). Furthermore by using the term ControlState to describe something which has to be synchronized implies that changes to this element (after authorization) also changes all of its associated elements (e.g. if ‘ControlState’ relates to the value of a text field, which is used to specify the charge value of an arbitrary amount of point charges, changing the ControlState will subsequently also change the point charges).

‘SimulationTime’ is defined as the sum of |ΔT| the simulation engine progressed the simulation (state), hence a paused simulation does not change its SimulationTime.

4.1 Use cases for important design aspects

4.1.1 Simulation and graphics engine logic

Use case 1: Multiple clients running with out-of-sync SimulationTime	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is paused
Story	Client A requests to start the simulation. Due to the authoritative principle of the underlying design specifying how to handle user input, the request to start the simulation is forwarded to the server before anything may happen. In further instance the server verifies whether or not the requested ControlState change may be authorized (see the use cases in the succeeding chapter 4.1.2

	Synchronization of ControlState). Assuming that the change is authorized, all of the clients connected to the server are notified about the new state (that is to start the simulation). Due to differing latencies between the server and its connected clients, all of the clients will ultimately start running their simulation at a different point of (real world) time.
Post-condition	All clients are running but ultimately they might be out of sync assuming an omniscient supervisor capable tracking such marginal differences.
Remark	Assuming latencies to be in the area below 1000 [ms] for modern networks, having 2 clients next to each other connected e.g. via LAN to a local server, the overall difference of alternating SimulationTimes between both clients should not be visible with the human eye. While it probably may never be possible to achieve a 100% synchronization, certain techniques could be implemented to decrease the effective difference in SimulationTime amongst all clients – for example by accounting for diverging client latencies when starting a simulation, and therefore slightly fast forwarding a simulation on demand

4.1.2 Synchronization of ControlState

Use case 2: Change of ControlState <u>without</u> user input collisions and the simulation is running	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is running
Story	Client A causes a change of the ControlState (e.g. by user input). Client A logs the SimulationTime and sends a ControlState change request to the server. The server receives the ControlState change and verifies in its internal lookup table that the changed element(s) was(/were) not changed by any other client at a later SimulationTime. Upon successful validation of this constraint the server broadcasts the ControlState change to all connected clients and updates its internal ControlState (that is state and timestamp of last change). The broadcasted ControlState change message contains a field telling the clients at which SimulationTime to execute the state change. For details regarding the calculation of this SimulationTime value, see chapter 0 Details regarding SimulationTime calculation All of the connected clients receive the broadcast message and have one of two options: a) For example if they have a fast connection to the server (that is low latency) their internal SimulationTime might be before the timestamp in the ControlState change message specifying the point of time to update the simulation. In this case the particular client caches the received order and executes the change as scheduled. b) Contrary if they would have a slow connection to the server their simulation will be beyond the point of time specified in the ControlState change message. In

	this case they have to record their current SimulationTime and roll back the simulation by negating ΔT until the point of time the ControlState change has to be executed. Then the change gets applied and the simulation is fast forwarded to the original SimulationTime.
Post-condition	All clients and the server share the same ControlState. At a global point of time X (that is real world time) it is not guaranteed that clients display exactly the same rendered view because it is not guaranteed that their internal SimulationTime is exactly the same relative to point X. It is guaranteed though that at a given SimulationTime clients will eventually show exactly the same rendered view.
Remark	

Use case 3: Change of ControlState <u>without</u> user input collisions and the simulation is not running	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is not running
Story	Client A causes a change of the ControlState and sends a request to the server. The server receives the ControlState change, runs the verification process (which should always succeed) and sends a broadcast message to all clients with SimulationTime set to: *(SimulationTime from change request)* All of the connected clients receive the broadcast message and apply the contained change.
Post-condition	All clients and the server share the same ControlState.
Remark	This use case is related to use case 2 – due to this reason some technical details have been omitted.

Use case 4: Change of ControlState <u>with</u> user input collisions – case 1 – lagging behind client B	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is running
Story	Client A causes a change of element X. Client A logs the SimulationTime Y and sends a ControlState change request to the server. The server receives the ControlState change and verifies and broadcasts it to all connected clients. Client B causes a change of element X before receiving respectively applying the broadcasted change from the server. Client B logs the SimulationTime (Y - Z) [where

	Z > 0] and sends a ControlState change request to the server. The server receives the ControlState change but the verification process fails causing the change not to be authorized (that is not sent back). Eventually all (including client B) connected clients will receive the broadcasted change originating from client A and update their simulation accordingly. Additionally client B might receive a notification that he's lagging behind compared to other clients which caused one change to be dropped.
Post-condition	All clients and the server share the same ControlState based on the change caused by client A.
Remark	Constraint: in general, clients' SimulationTime is assumed to be not too far off amongst each other (see use case 1). This use case is related to use case 2 – due to this reason some technical details have been omitted.

Use case 5: Change of ControlState <u>with</u> user input collisions – case 2 – advanced client B	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is running
Story	Client A causes a change of element X. Client A logs the SimulationTime Y and sends a ControlState change request to the server. The server receives the ControlState change and verifies and broadcasts it to all connected clients. Client B causes a change of element X before receiving respectively applying the broadcasted change from the server. Client B logs the SimulationTime (Y + Z) [where Z > 0] and sends a ControlState change request to the server. The server receives the ControlState change and verifies it. In this case the server's verification process will succeed, causing it to save the new state and issue a new broadcast to all clients with the state of client B. Eventually all clients will receive the broadcasted change originating from client A and update their simulation accordingly. In further instance all clients will receive the change originating from client B. Regular rules apply with regards to the way clients deal with simulation changes, that is they will – if needed – roll back their simulation state to the timestamp value contained in the change message, apply the change and fast forward the simulation to the former point of time.
Post-condition	All clients and the server share the same ControlState based on the change caused by client B.
Remark	Constraint: messages are expected to arrive in fixed order. This use case is related to use case 2 – due to this reason some technical details have been omitted.

Use case 6: Change of ControlState <u>with</u> user input collisions – case 3 - equal SimulationTime	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is running
Story	Client A causes a change of element X. Client A logs the SimulationTime Y and sends a ControlState change request to the server. The server receives the ControlState change, verifies and broadcasts it to all connected clients. Client B causes a change of element X before receiving respectively applying the broadcasted change from the server. Client B logs the SimulationTime Y and sends a ControlState change request to the server. The server receives the ControlState change, runs the verification process and reacts on it depending on which particular of the following two possible scenarios applies: 1. The simulation is in the state ‘running’ with both clients causing a contradicting change at exactly the same point of SimulationTime, but for example client B’s latency is greater than client A’s. This issue is resolved by applying a first-come, first-serve paradigm on the server resulting in the rejection of client B’s change and preferring the change arriving sooner. 2. The simulation is currently paused (therefore the SimulationTime is temporarily constant). Since successive changes will always wear the same timestamp none of them may be dropped. Therefore any change will overwrite prior ones (resulting in new change broadcasts), in effect favoring changes caused by high latency clients.
Post-condition	1. If the simulation is in running state all clients and the server will share the same ControlState based on the change caused by client A. 2. If the simulation is in paused state all clients and the server will share the same ControlState based on which change was received last by the server.
Remark	This use case is related to use case 2 – due to this reason some technical details have been omitted.

Use case 7: Change of ControlState <u>with</u> user input collisions – v4	
Pre-condition	2 clients (A and B) opened the same simulation, connected (and therefore synchronized) with the server, and the simulation is running
Story	Client A causes a change Y of element X. Client A logs the SimulationTime and sends a ControlState change request to the server. The server receives the ControlState change, verifies and broadcasts it to all connected clients. Client B causes a change Y of element X before receiving the broadcasted change from the server. Client B logs the SimulationTime and sends a ControlState change request to the server. The server receives the ControlState change and runs the verification process.

	Since the state of element X on the server is equal to the requested change by client B, the request by client B gets dropped silently. Eventually all (including client B) connected clients will receive the broadcasted change originating from client A and update their simulation accordingly.
Post-condition	All clients and the server share the same ControlState based on the change caused by client A which is equal to the change requested by client B.
Remark	This use case is related to use case 2 – due to this reason some technical details have been omitted.

4.2 Details regarding SimulationTime calculation

The SimulationTime defining the point of time when changes should be applied to the simulation is calculated as follows:

$$\begin{array}{rl} & \text{(SimulationTime from change request)} \\ + & \text{(originating clients 1-way latency)} \\ + & \underline{\text{(median 1-way latency of all connected clients)}} \\ = & \text{SimulationTime to apply changes} \end{array}$$

The formula above seems to be a robust implementation to determine the point of time when to apply changes to a simulation. While this design naturally causes some delay for user input, it also reduces the amount of simulation roll back computation required on other clients (and therefore graphic ‘stuttering’). Additionally this design pays tribute to the idea to reward “high efforts” in a sense that clients can directly influence the perceived feeling of input lag by improving their own network connection and therefore reduce one of the delay factors.

The second factor influencing the delay of user interactions impacting the whole system can be regarded as some kind of social parameter to improve the experience for slower clients. While theoretically this value could be abused for fraud (in a sense of disturbing the experience for others), its practical security relevance is supposedly minor since it requires approximately as much evil clients as those who are present with good ambitions.

Part II
Implementation details

Chapter 5

General aspects of a client-server architecture for TEALsim

5.1 Security

The system was designed to be used in a closed environment which is assumed to be free of aggressors. Due to this reason security concerns did not influence the design of the implementation in general. Evident security risks were tried to be addressed as good as possible unless it involved a serious effort of redesign or increase of complexity. Throughout this thesis known issues are mentioned purely for the sake of documentation without even trying to compose a comprehensive list covering a fair amount of aspects. For this purpose an independent, in-depth analysis would be required which presumably would require a huge amount of work.

Chapter 6

The underlying network layer

6.1 General design of the network layer

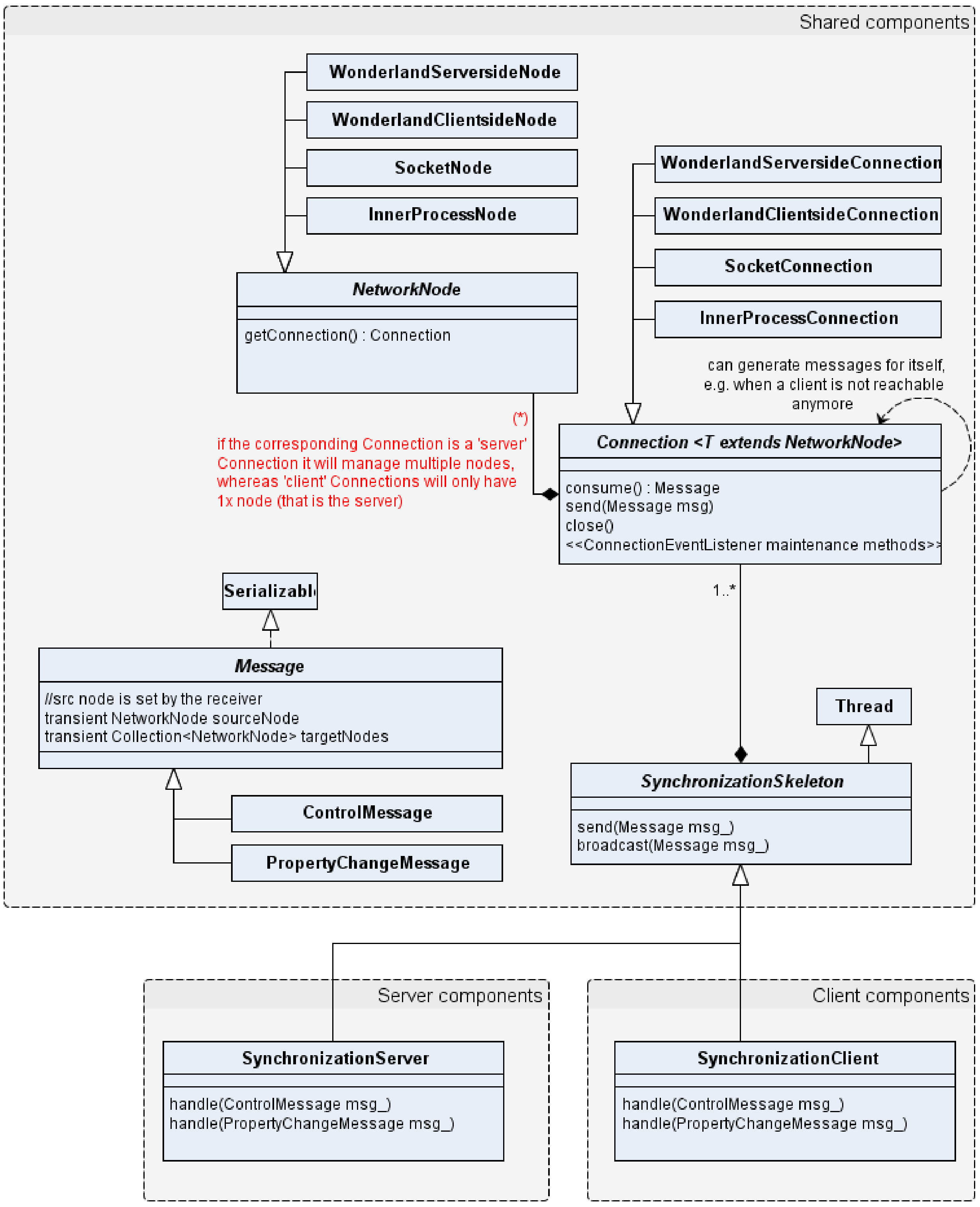

Shared components
WonderlandServersideNode
WonderlandClientsideNode
SocketNode
InnerProcessNode
NetworkNode
getConnection() : Connection
WonderlandServersideConnection
WonderlandClientsideConnection
SocketConnection
InnerProcessConnection
can generate messages for itself, e.g. when a client is not reachable anymore
Connection <T extends NetworkNode>
consume() : Message
send(Message msg)
close()
<<ConnectionEventListener maintenance methods>>
(*)
if the corresponding Connection is a 'server' Connection it will manage multiple nodes, whereas 'client' Connections will only have 1x node (that is the server)
1..*
Serializabl
Message
//src node is set by the receiver
transient NetworkNode sourceNode
transient Collection<NetworkNode> targetNodes
ControlMessage
PropertyChangeMessage
Thread
SynchronizationSkeleton
send(Message msg_)
broadcast(Message msg_)
Server components
SynchronizationServer
handle(ControlMessage msg_)
handle(PropertyChangeMessage msg_)
Client components
SynchronizationClient
handle(ControlMessage msg_)
handle(PropertyChangeMessage msg_)

Figure 6.1: Class diagram of TEALsim's network layer

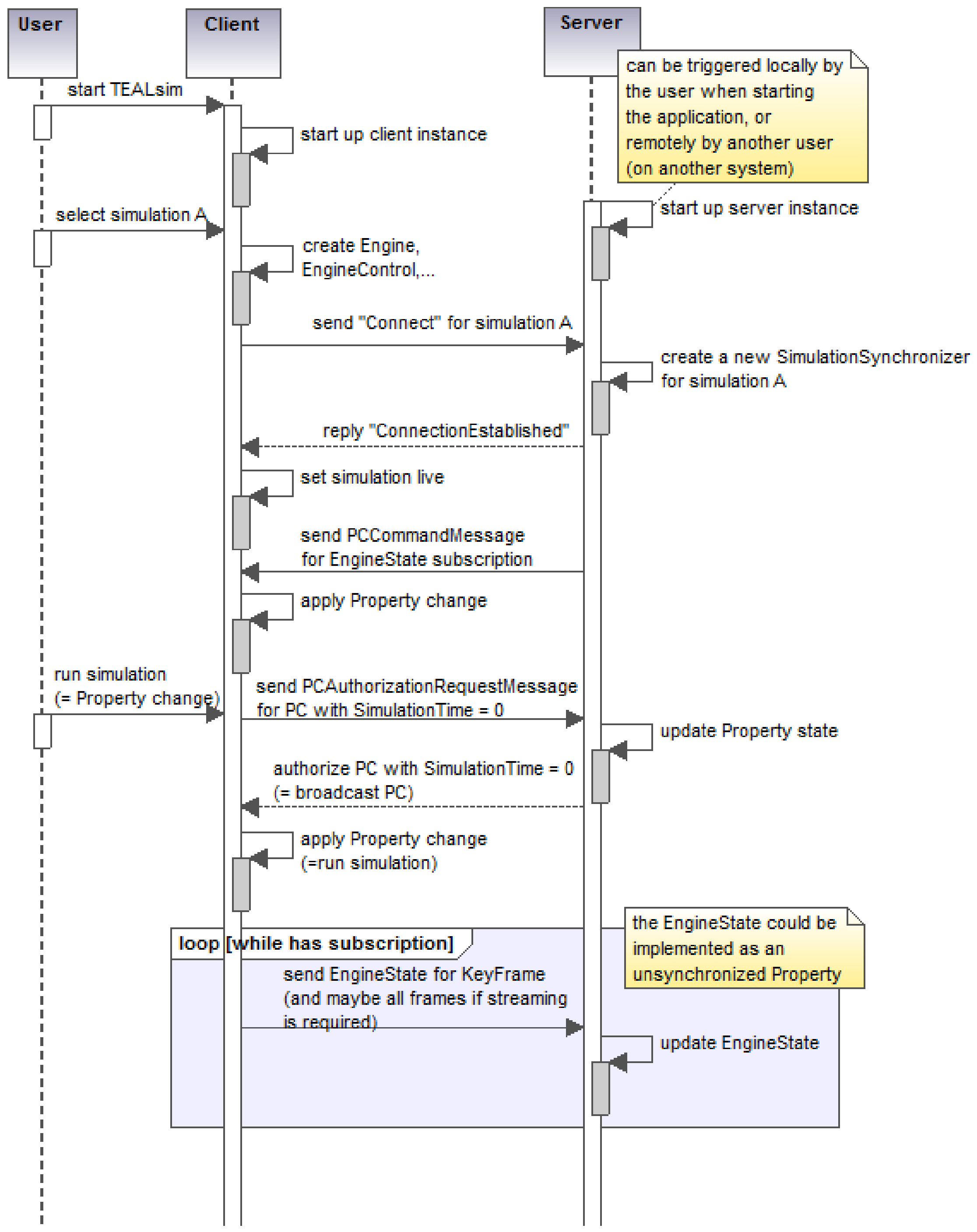

User
Client
Server
start TEALsim
start up client instance
can be triggered locally by the user when starting the application, or remotely by another user (on another system)
start up server instance
select simulation A
create Engine, EngineControl,...
send "Connect" for simulation A
create a new SimulationSynchronizer for simulation A
reply "ConnectionEstablished"
set simulation live
send PCCommandMessage for EngineState subscription
apply Property change
run simulation (= Property change)
send PCAuthorizationRequestMessage for PC with SimulationTime = 0
update Property state
authorize PC with SimulationTime = 0 (= broadcast PC)
apply Property change (=run simulation)
loop [while has subscription]
the EngineState could be implemented as an unsynchronized Property
send EngineState for KeyFrame (and maybe all frames if streaming is required)
update EngineState

Figure 6.2: Standard start up sequence of TEALsim's client-server architecture

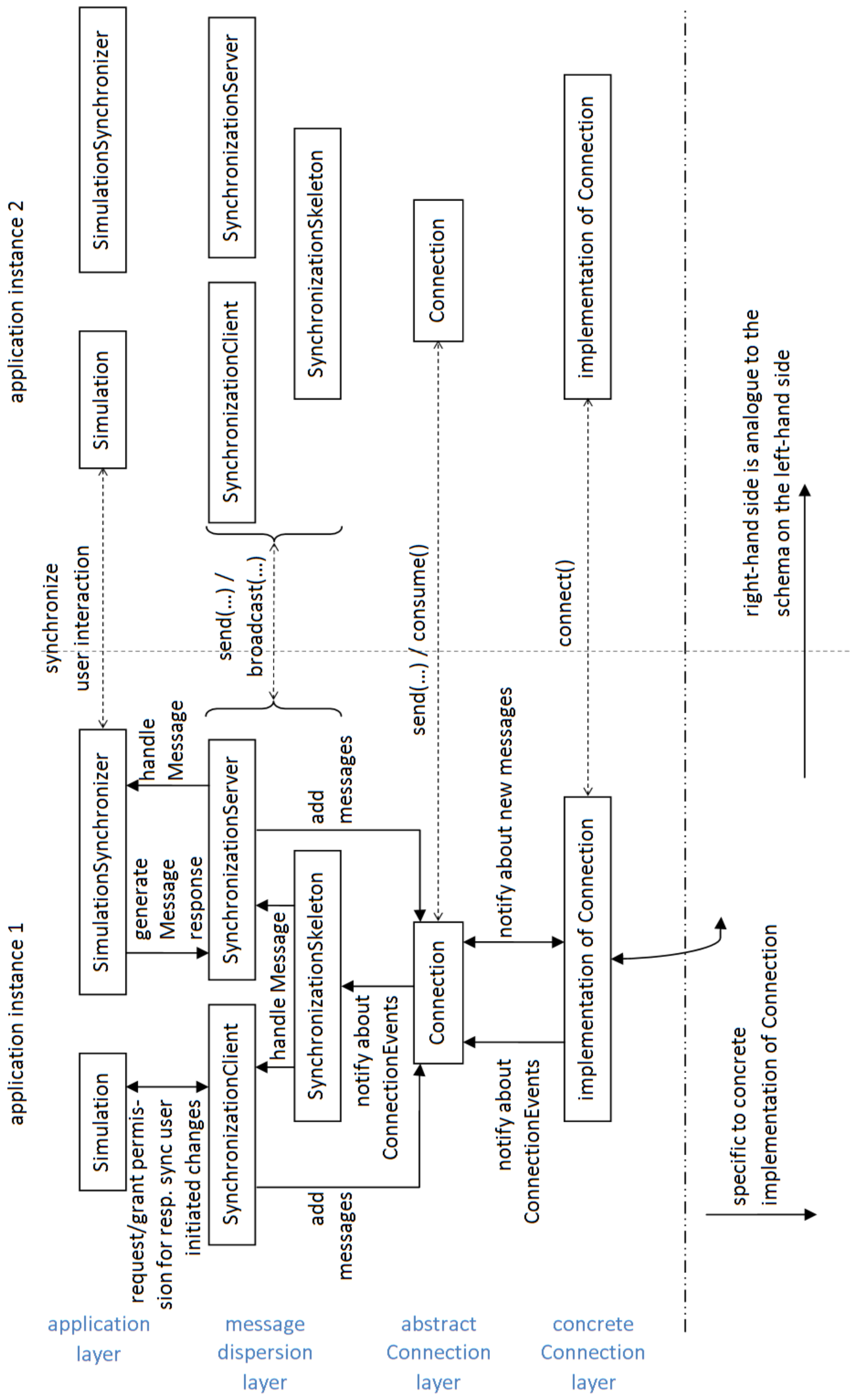

application instance 1
application instance 2
Simulation
SimulationSynchronizer
SynchronizationClient
SynchronizationServer
SynchronizationSkeleton
Connection
implementation of Connection
request/grant permis-sion for resp. sync user initiated changes
generate Message response
handle Message
handle Message
add messages
add messages
notify about ConnectionEvents
notify about ConnectionEvents
notify about new messages
synchronize user interaction
send(...) / broadcast(...)
send(...) / consume()
connect()
specific to concrete implementation of Connection
right-hand side is analogue to the schema on the left-hand side
application layer
message dispersion layer
abstract Connection layer
concrete Connection layer

Figure 6.3: layers of TEALsim's network architecture

6.2 Socket based implementation of Connection

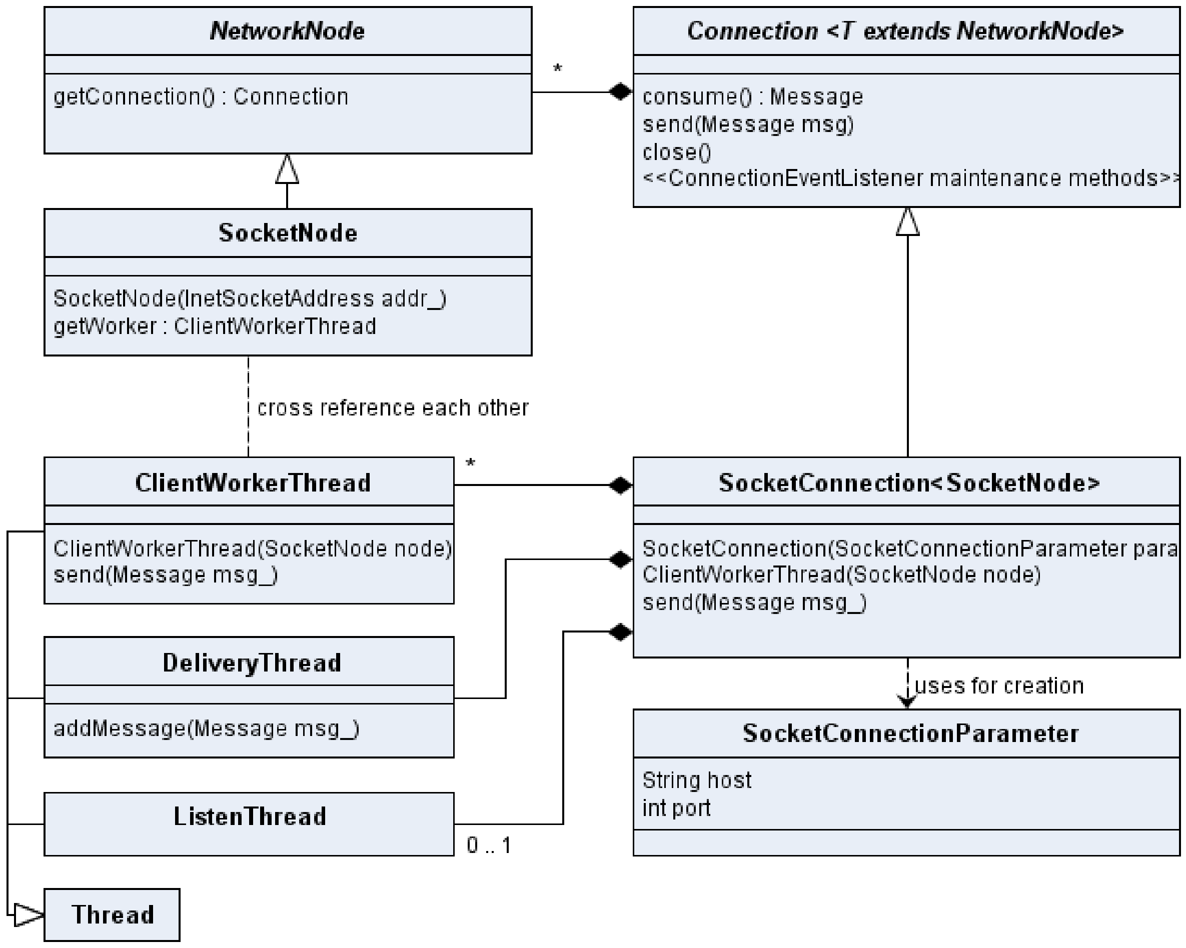

NetworkNode
getConnection() : Connection
Connection <T extends NetworkNode>
consume() : Message
send(Message msg)
close()
<<ConnectionEventListener maintenance methods>>
SocketNode
SocketNode(InetSocketAddress addr_)
getWorker : ClientWorkerThread
cross reference each other
ClientWorkerThread
ClientWorkerThread(SocketNode node)
send(Message msg_)
SocketConnection<SocketNode>
SocketConnection(SocketConnectionParameter para
ClientWorkerThread(SocketNode node)
send(Message msg_)
DeliveryThread
addMessage(Message msg_)
ListenThread
0 .. 1
uses for creation
SocketConnectionParameter
String host
int port
Thread
*

Figure 6.4: Class diagram of the socket based implementation of Connection

6.3 Implementation of a Connection for communication within a single JVM

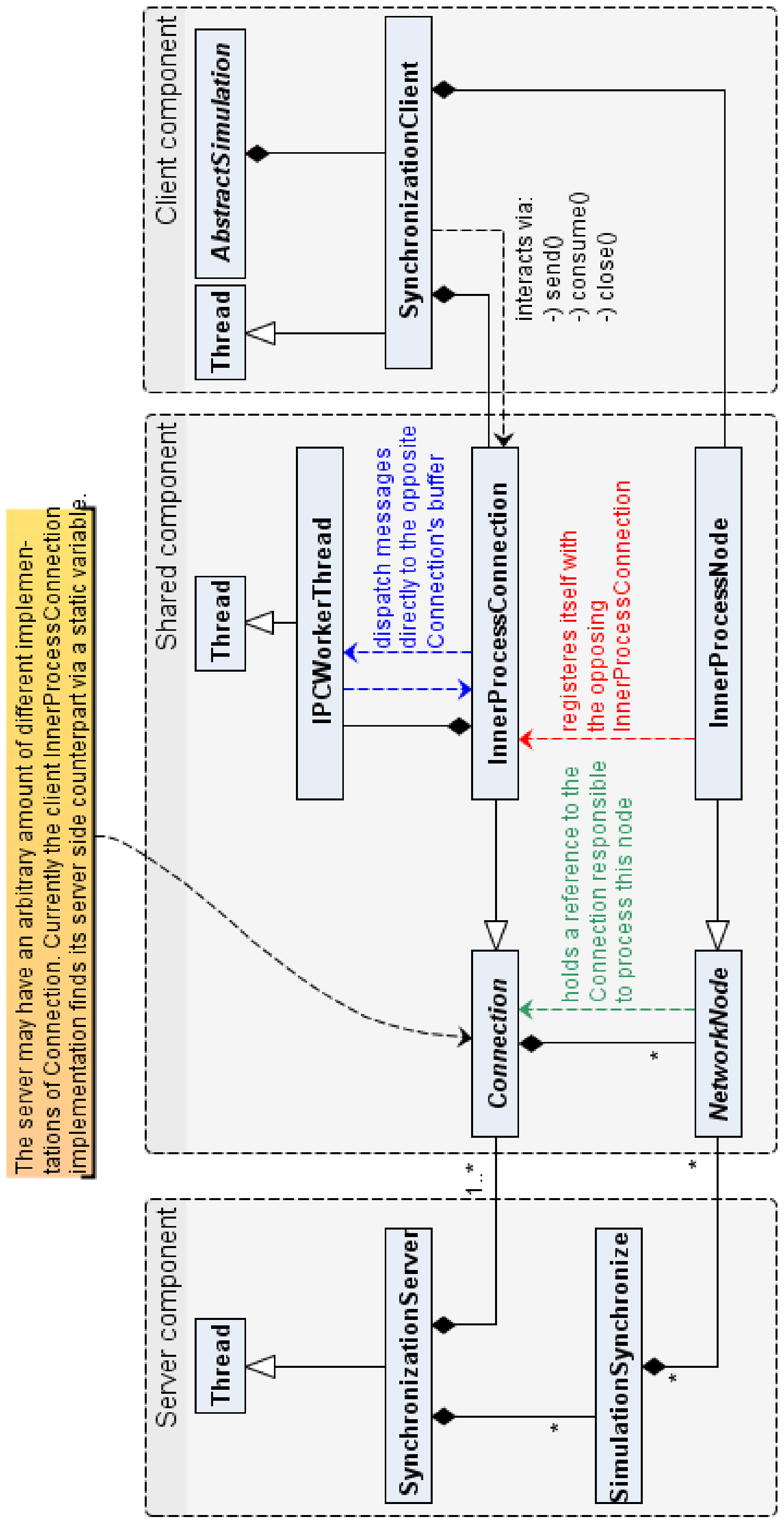

Client component
Thread
AbstractSimulation
SynchronizationClient
interacts via:
-) send()
-) consume()
-) close()
Shared component
Thread
IPCWorkerThread
dispatch messages directly to the opposite Connection's buffer
InnerProcessConnection
registeres itself with the opposing InnerProcessConnection
InnerProcessNode
holds a reference to the Connection responsible to process this node
Connection
NetworkNode
The server may have an arbitrary amount of different implementations of Connection. Currently the client InnerProcessConnection implementation finds its server side counterpart via a static variable.
1..*
Server component
Thread
SynchronizationServer
SimulationSynchronize
*

Figure 6.5: Class diagram of the implementation of Connection used to run client and server within one JVM

6.4 Open Wonderland specific implementation of Connection

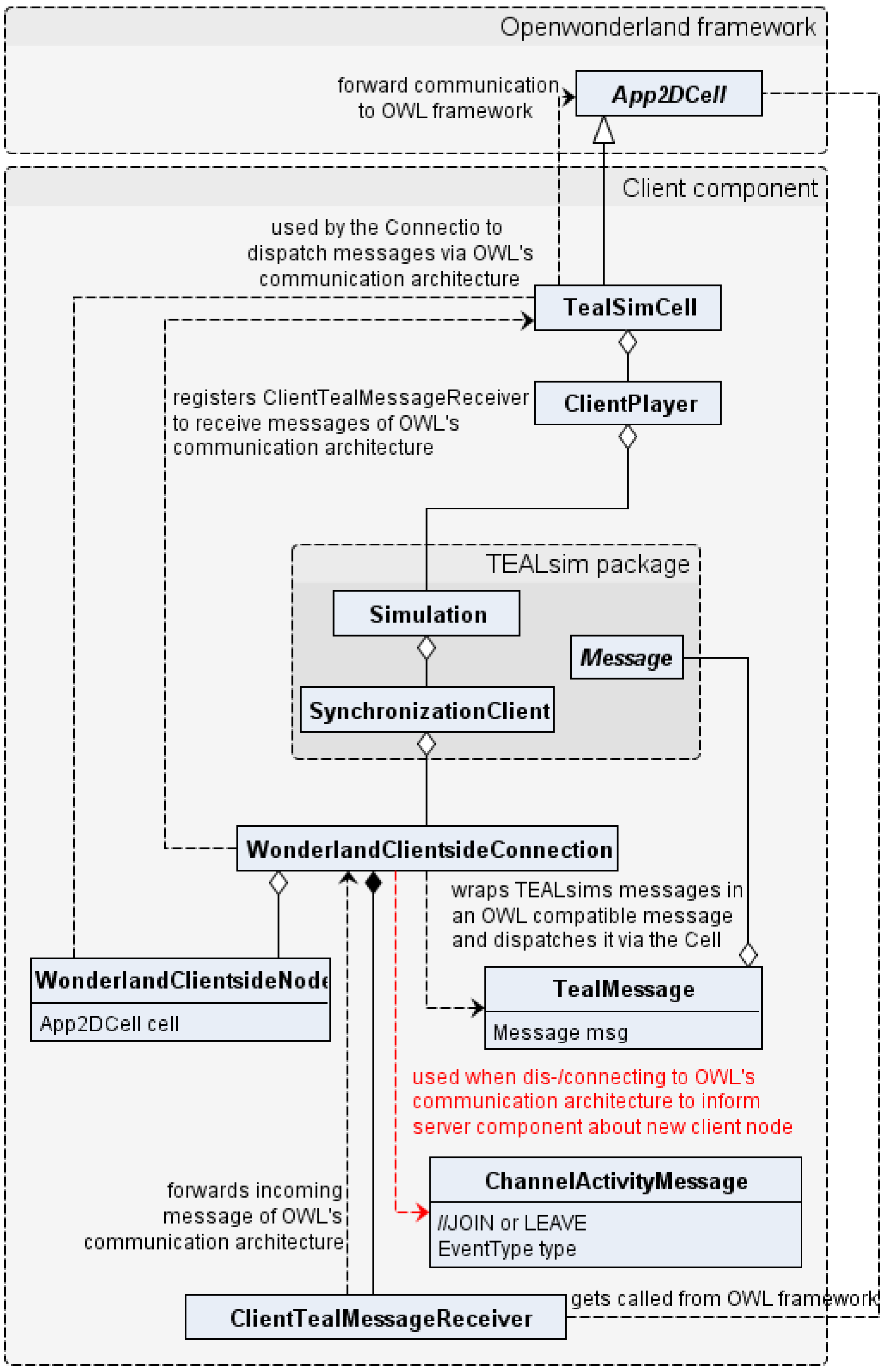

Openwonderland framework
forward communication to OWL framework
App2DCell
Client component
used by the Connectio to dispatch messages via OWL's communication architecture
TealSimCell
registers ClientTealMessageReceiver to receive messages of OWL's communication architecture
ClientPlayer
TEALsim package
Simulation
Message
SynchronizationClient
WonderlandClientsideConnection
wraps TEALsims messages in an OWL compatible message and dispatches it via the Cell
WonderlandClientsideNode
App2DCell cell
TealMessage
Message msg
used when dis-/connecting to OWL's communication architecture to inform server component about new client node
ChannelActivityMessage
//JOIN or LEAVE
EventType type
forwards incoming message of OWL's communication architecture
gets called from OWL framework
ClientTealMessageReceiver

Figure 6.6: Class diagram of the client-side implementation of Connection for OWL

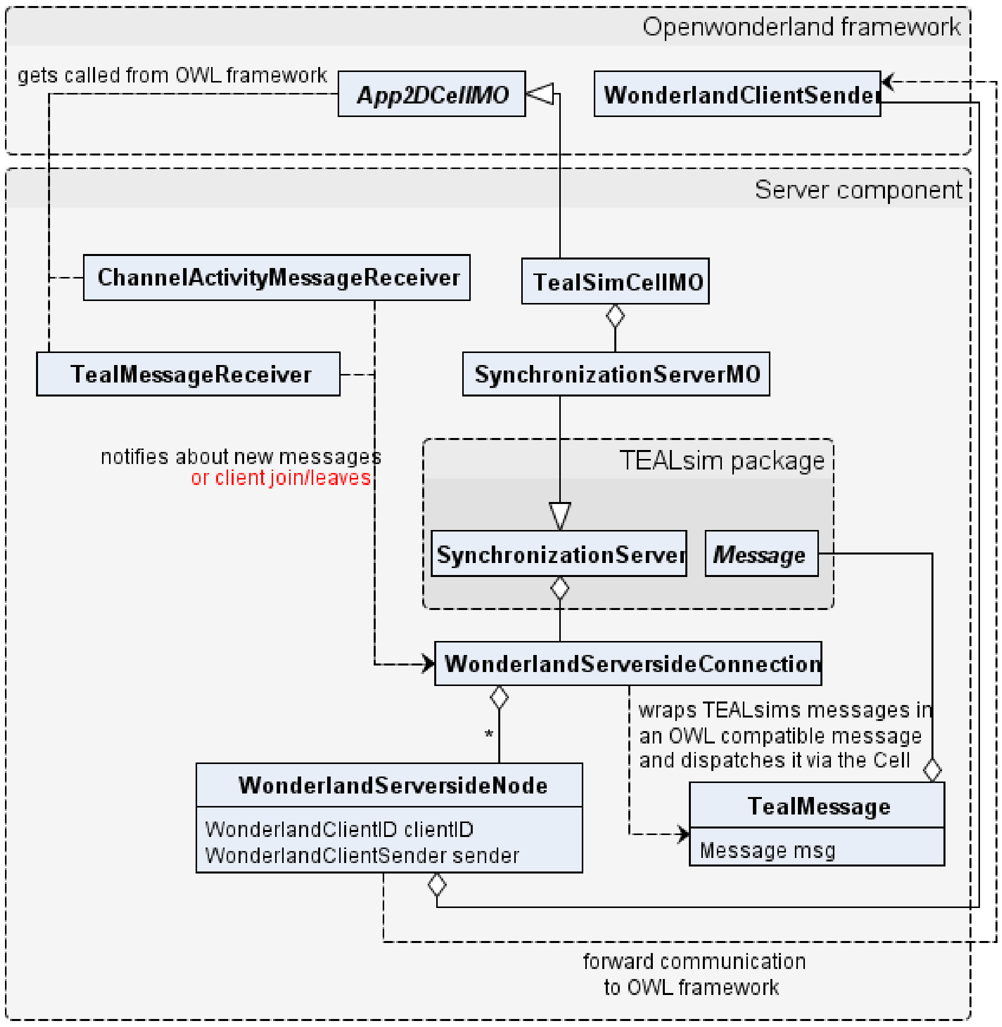

Openwonderland framework
gets called from OWL framework
App2DCellMO
WonderlandClientSender
Server component
ChannelActivityMessageReceiver
TealSimCellMO
TealMessageReceiver
SynchronizationServerMO
notifies about new messages
or client join/leaves
TEALsim package
SynchronizationServer
Message
WonderlandServersideConnection
wraps TEALsims messages in an OWL compatible message and dispatches it via the Cell
*
WonderlandServersideNode
WonderlandClientID clientID
WonderlandClientSender sender
TealMessage
Message msg
forward communication to OWL framework

Figure 6.7: Class diagram of the server-side implementation of Connection for OWL

Chapter 7

Starting / using the TEALsim framework

7.1 Options to start the desktop version of TEALsim

7.1.1 Configuring the client and server component

By utilizing TEALsim's runtime argument feature its desktop version may be started in different ways. Especially the behavior when no server instance is required may seem odd at first glance but it is owed to the way the RuntimeArgument mechanism is implemented. Basically the possible ways to start TEALsims desktop version are …

- **with no particular runtime arguments at all**
  this will cause TEALsim to start with a local server instance running with exactly one associated Connection of the sub-type InnerProcessConnection. Instances of simulations will create SynchronizationClient instances utilizing themselves a Connection of sub-type InnerProcessConnection. This behavior ensures a minimum amount of conflicts which may be caused by firewalls, etc. when the socket based implementation of Connection would be used for desktop use of TEALsim. Furthermore the InnerProcessConnection performs much better compared to the socket based Connection type due to the fact that messages are exchanged by passing around pointers.
- **with arguments for the server part of TEALsim but no special arguments for the client part**
  this will cause TEALsim to start with a local server instance running with all of the specified Connections plus one additional of the sub-type InnerProcessConnection. Instances of simulations will create SynchronizationClient instances utilizing themselves a Connection of sub-type InnerProcessConnection. Remote clients may connect to the local server by means of any of the instantiated Connection types.
- **with a 'no server' argument for the server part of TEALsim and a specific server for the client part**
  this will cause TEALsim to start with a local server instance running with an instance of

the Connection sub-type InnerProcessConnection. After this instance is created the server related runtime argument will cause the server to shut down. Instances of simulations will create SynchronizationClient instances utilizing a sub-type of Connection matching the given runtime argument.

In a nutshell:

- If not explicitly specified in a different way the **client** side of TEALsim (that is the SynchronizationClient) will use instances of Connections of the sub-type InnerProcessConnection.
- Unless specified via the 'no server' argument a local **server** instance will be created with at least one Connection of sub-type InnerProcessConnection.

7.2 Using TEALsim framework in a 3rd party application like OpenWonderland

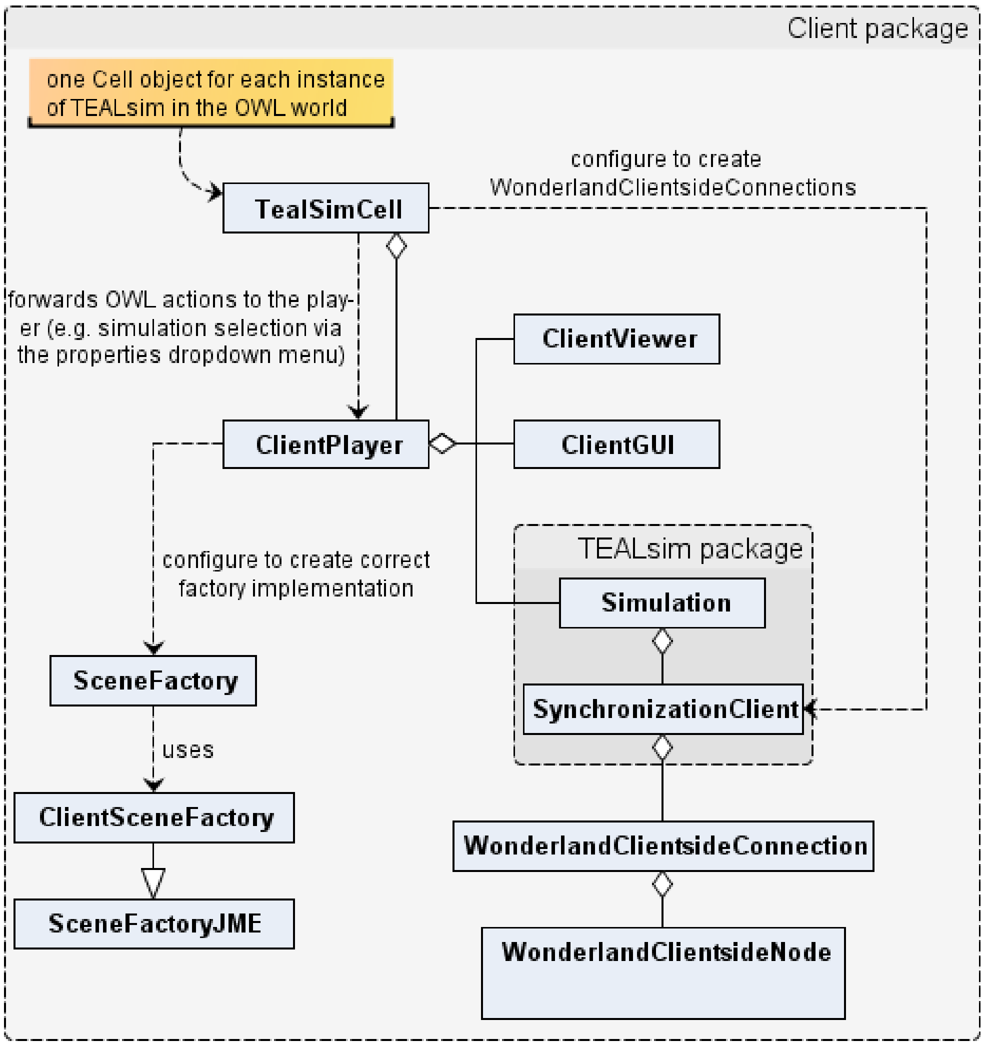

Client package
one Cell object for each instance of TEALsim in the OWL world
TealSimCell
configure to create WonderlandClientsideConnections
forwards OWL actions to the player (e.g. simulation selection via the properties dropdown menu)
ClientViewer
ClientPlayer
ClientGUI
TEALsim package
configure to create correct factory implementation
Simulation
SceneFactory
SynchronizationClient
uses
ClientSceneFactory
WonderlandClientsideConnection
SceneFactoryJME
WonderlandClientsideNode

Figure 7.1: Schema of required client-side components to integrate TEALsim into OWL

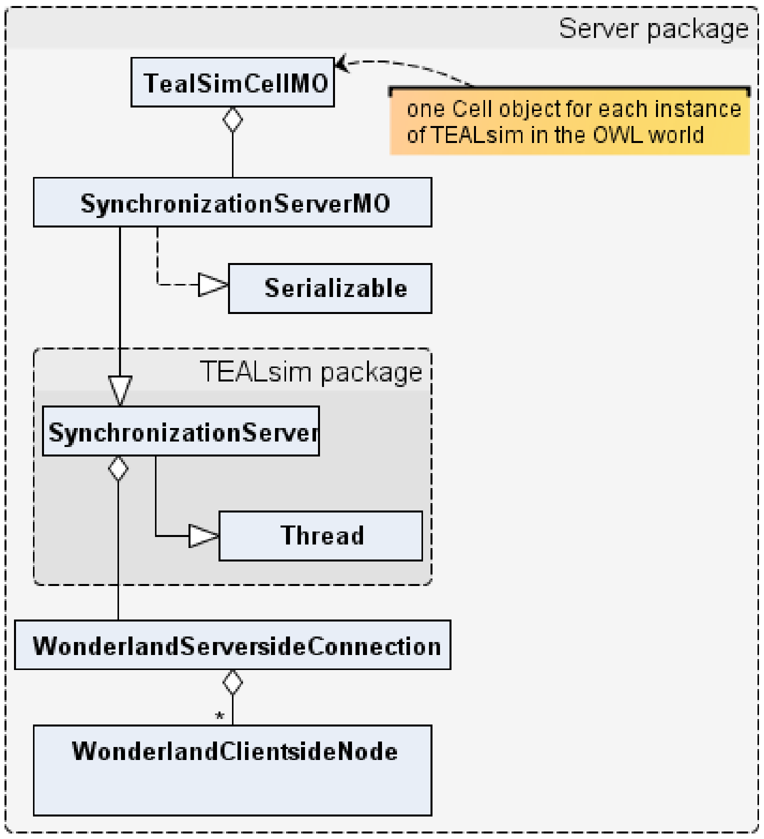


Figure 7.2: Schema of required server-side components to integrate TEALsim into OWL

Chapter 8

Analysis of the design of various TEALsim components

8.1 Introducing clear object ownership

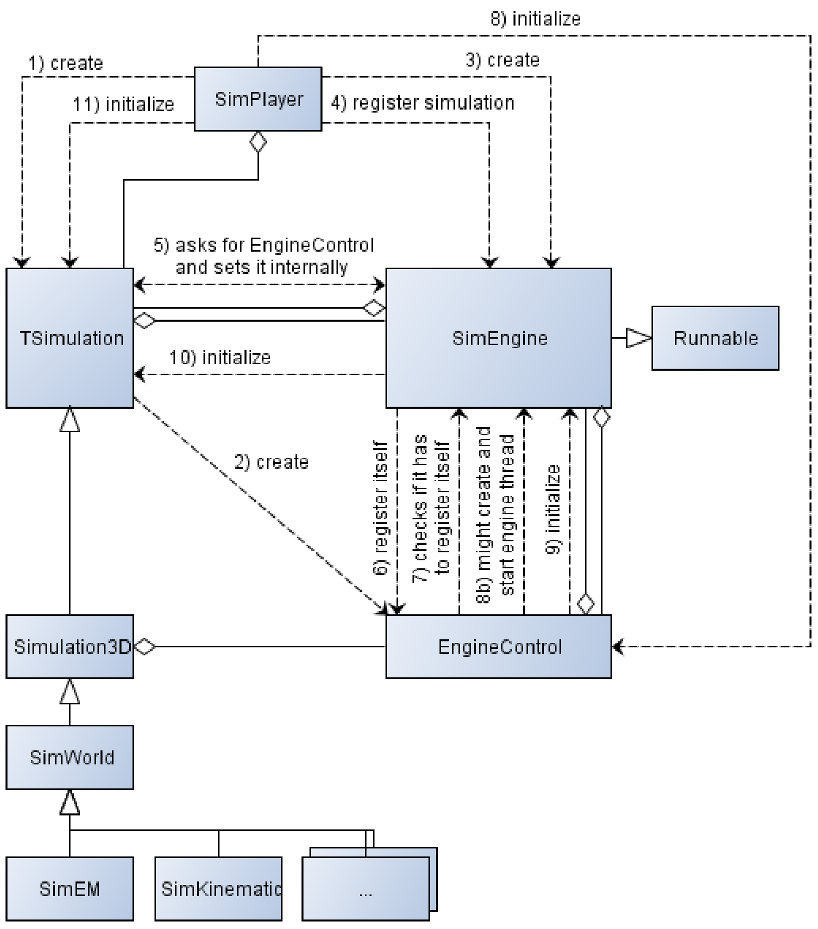


Figure 8.1: Snapshot of execution flow and object ownership of former simulation construction

8.1.1 Resulting design for TEALsim's simulations and their more general components

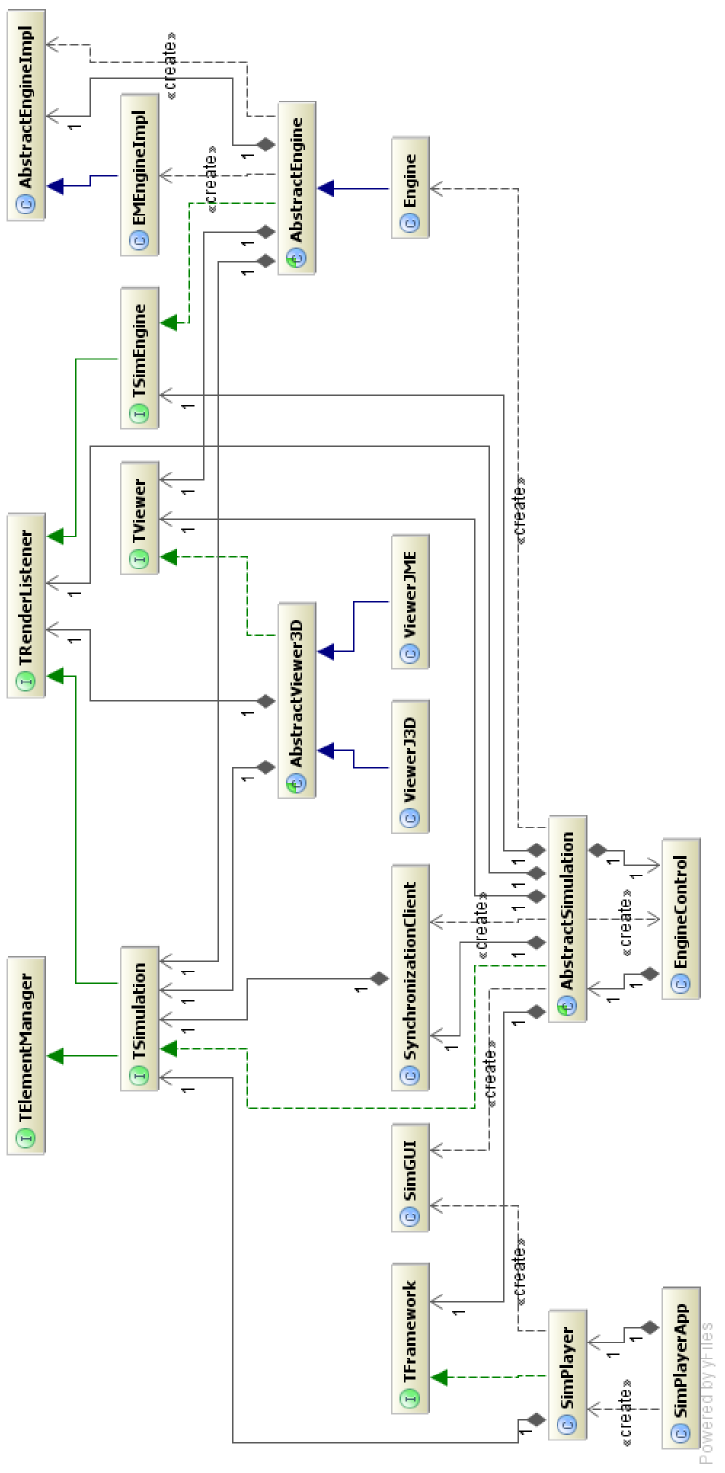


Figure 8.2: Design of TEALsim (based on client-server architecture)

8.2 Design of the components involved in computing and rendering the simulation

8.2.1 State of the former simulation engine

Due to the vast amount of objects allowed to change the simulation's state to any state an accurate state diagram outlining the allowed transitions is difficult to extract from the code.

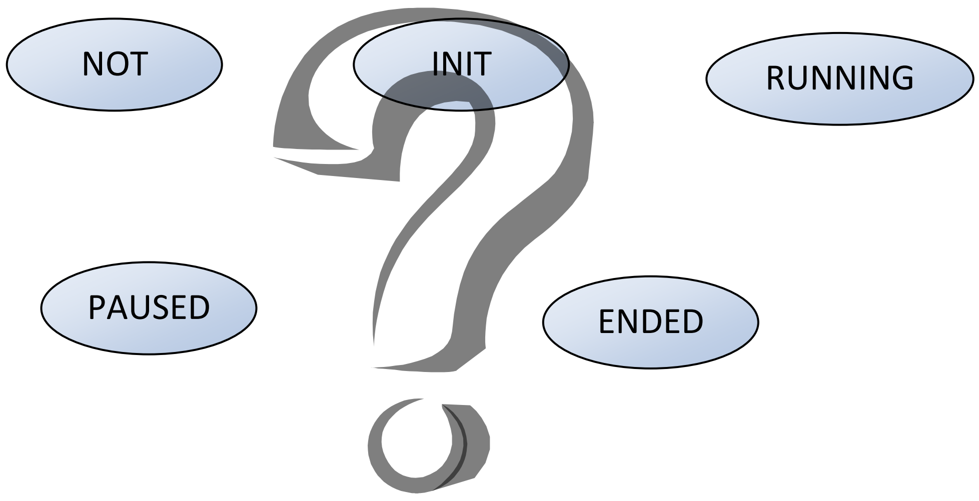


Figure 8.3: State diagram of the former simulation engine

8.2.2 Design of the revamped simulation engine

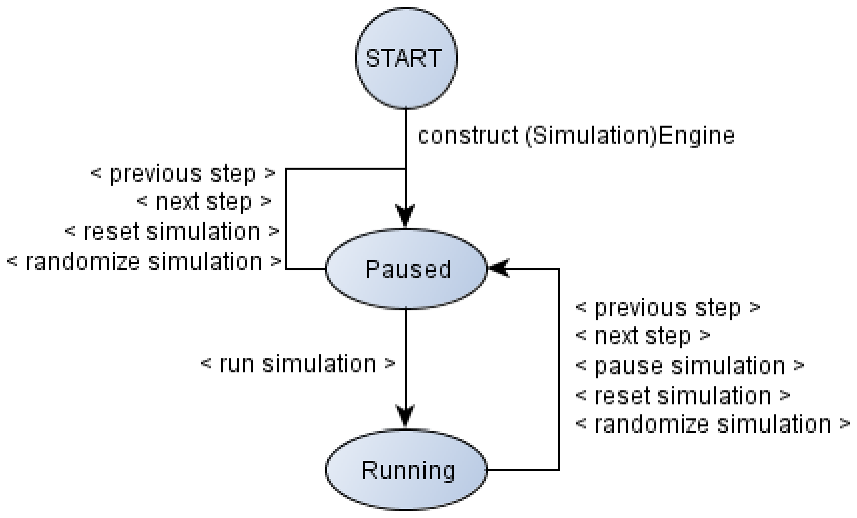


Figure 8.4: State diagram of the deterministic simulation engine

8.2.3 Application flow of the rendering process

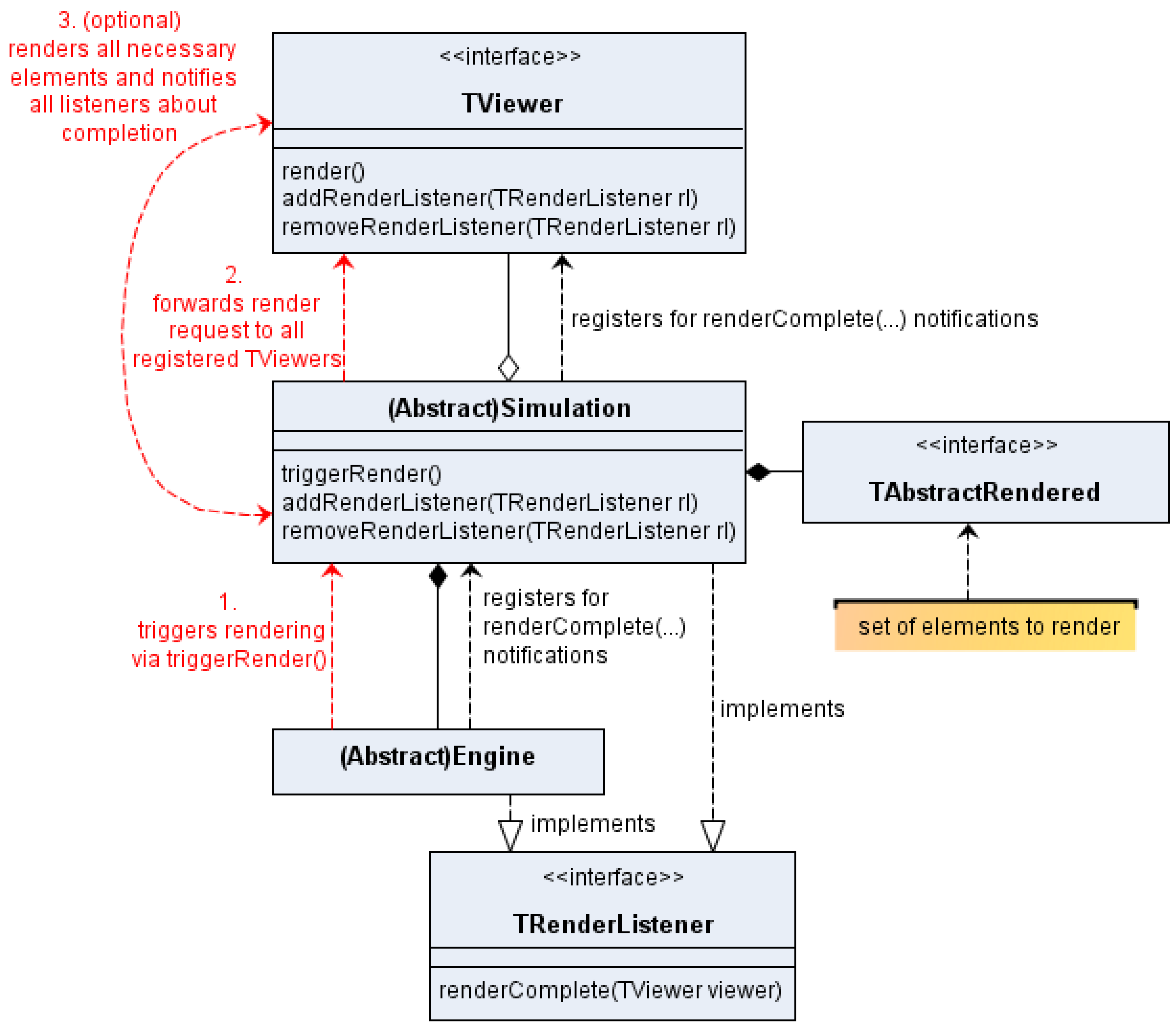

3. (optional) renders all necessary elements and notifies all listeners about completion
<<interface>>
TViewer
render()
addRenderListener(TRenderListener rl)
removeRenderListener(TRenderListener rl)
2. forwards render request to all registered TViewers
registers for renderComplete(...) notifications
(Abstract)Simulation
triggerRender()
addRenderListener(TRenderListener rl)
removeRenderListener(TRenderListener rl)
<<interface>>
TAbstractRendered
set of elements to render
1. triggers rendering via triggerRender()
registers for renderComplete(...) notifications
implements
(Abstract)Engine
implements
<<interface>>
TRenderListener
renderComplete(TViewer viewer)

Figure 8.5: Schema of the rendering process in TEALsim

8.2.4 Design of SynchronizableGenerics

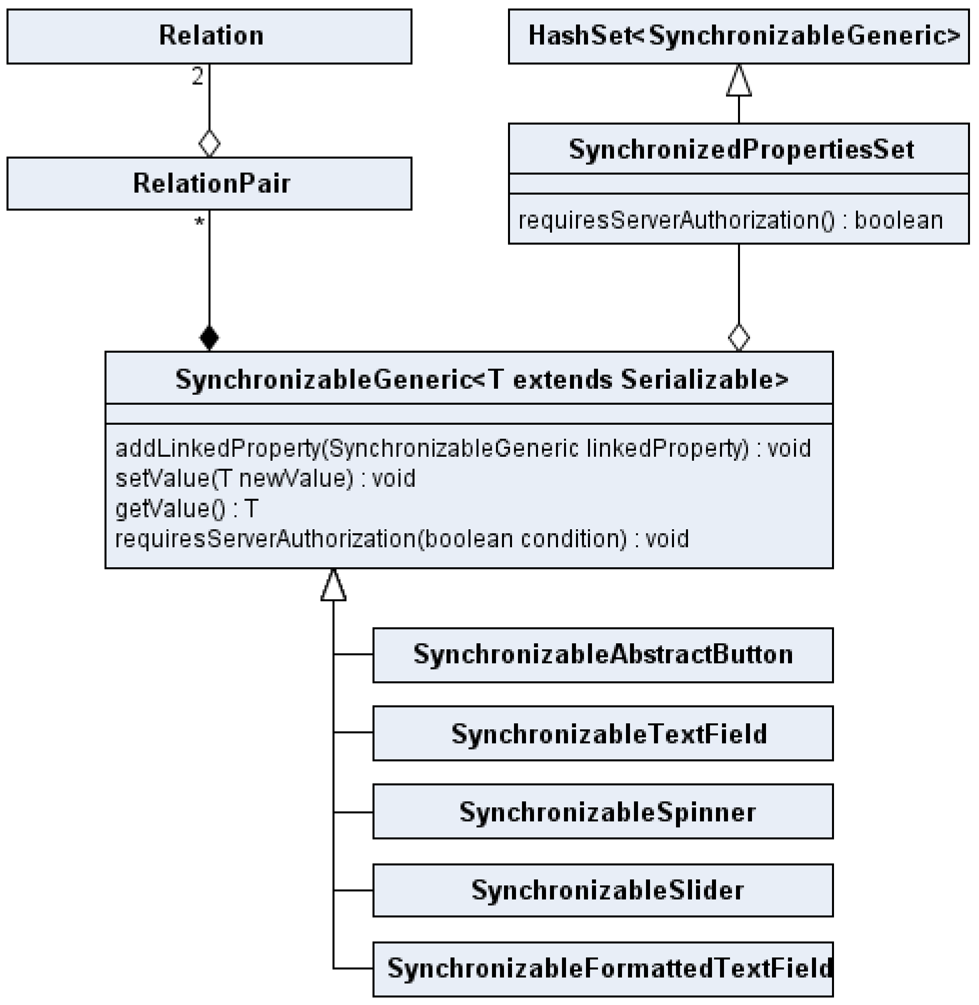

Relation
2
RelationPair
*
HashSet<SynchronizableGeneric>
SynchronizedPropertiesSet
requiresServerAuthorization() : boolean
SynchronizableGeneric<T extends Serializable>
addLinkedProperty(SynchronizableGeneric linkedProperty) : void
setValue(T newValue) : void
getValue() : T
requiresServerAuthorization(boolean condition) : void
SynchronizableAbstractButton
SynchronizableTextField
SynchronizableSpinner
SynchronizableSlider
SynchronizableFormattedTextField

Figure 8.6: Class diagram of SynchronizableGenerics and the associated UI wrapper objects

Part III

Outcome

Chapter 9

Ideas for future work respectively open issues

9.1 Important issues to fix

- An important issue to fix is the synchronization of calls to the randomization oracle (and subsequently enabling a reset of the oracle)
- Properly rolling back the simulation engine state to a former point of time – currently it is not exactly clear which variables have to be set back
- While stepping forward in simulations works, the inverse action of stepping backward does not, thus the UI button currently has no function
- A bug in the JME library seems to prevent the canvas from being closed. This is particularly bad because it prevents effective unit testing of the TEALsim desktop application

9.2 Design issues in need of a redesign

- SynchronizableProperties – they capture changes triggered by the user at a certain point of time. Based on the current design connecting clients start their execution based on the last computed engine state + the set of all changed SynchronizableProperties (that is their last state). What happens if changes besides the last change to a SynchronizableProperty alter the simulation in a way which is not currently reflected by the engine state (e.g. elements are added to the simulation by repeatedly clicking a UI button etc.)?
- Improve the SimulationEngine algorithm to become more stable in situations where the required computing power exceeds the available hardware capacities. The current logic exposes the drawback that in situations where the hardware is unable to calculate at least the key-frame within the given KEY_FRAME_INTERVAL period, delays are accumulated causing the algorithm to stay unresponsive for an increased amount of

time, even if for example the simulation was changed by the user to a state which could be calculated faster (for example less elements).

- Another issue of current design is its single threaded execution flow which does computation of the next frame on demand (that is when it is required for rendering). A better solution would be a consumer/producer scenario where engine states get calculated (that is ’produced’) in advance and eventually ‘consumed’ by the rendering engine. This would supposedly also help to change the engines design to be capable of multi threaded execution, which would help to increase performance on modern and future hardware.
- ID generation of SynchronizableProperties currently has to be done manually by content developers. The question is if there is a generic (automatic) way to do this, or is it always up to the user to set unique IDs?
- Currently TEALsim still comprises a serious amount of inconsistent design which creates a redundant, bloated codebase. For example the use of factory methods which could be configured to produce the right type of objects is not utilized throughout the TEALsim (and its correlated OpenWonderland module) project. In this context the ClientPlayer class in the OWL module might be cleaned up considerably – potentially up to the point where it becomes a featureless shell.

9.3 List of nice-to-have features

- Picking, selecting dragging via the mouse
- Verification of simulations soundness via Unit-Tests
- Formal specification of Ant build scripts deliverables
- Change all of TEALsim to use java.util.logging
- Fix the broken test which searches for circular dependencies in the set of existing derivations of RuntimeArgument. Since those derived classes are no longer inner static classes of RuntimeArgument, the technique to query for all declared classes in RuntimeArgument obviously does not yield any results anymore

Part IV
Appendix

Chapter 10

Remarks

10.1 Diagram legends

The diagrams, schemas, charts, etc. in this thesis do not strictly stick to any specific standard (like the UML standard[22]). Basically there are 2 reasons for this decision:

1) the chapters of this thesis related to TEALsim and the TEALsim Open Wonderland module do not intend to serve as a full technical documentation in the narrow sense, but shall rather give an overview over the performed programming work and the ideas and concepts behind the implemented features
2) almost all diagrams were created manually (instead of automatic generation from source code), not least because no suitable, affordable tools were available during the course of the project

As such the diagrams sometimes mix different sets of graphical notations, even though effort was taken to adhere to familiar and wide spread norms. After all, the careful selection of content by a human being - to keep the drawings as minimalistic as possible while maximizing its pictured information - should outweigh the loss of standardization by far.

Following there is a compilation with descriptions of the most commonly used notations.

[22] Specification of the UML standard - http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

10.1.1 UML class diagram like content

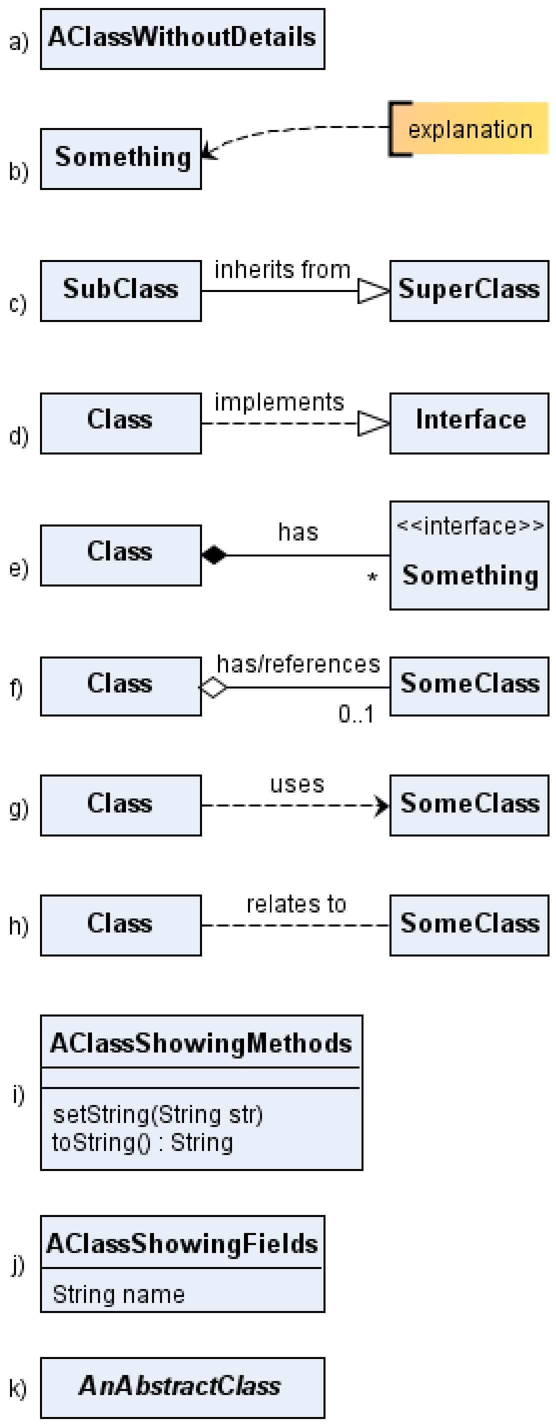


Figure 10.1: Drawings with UML class diagram like content

a) usually class/interface/etc. representations (during the course of this chapter referenced as ‘elements’) are kept as minimalistic as possible, hiding most or all methods/fields/etc. (during the course of this chapter referenced as ‘properties’).

Therefore a simple box without any other information but the name of it does not imply that there is no functionality contained in this element, but it shall rather be seen as a black box fulfilling all of the tasks which can be logically derived from its name. If one particular property may be of special interest or may further help to clarify the purpose of the element, it may be included in the drawing.

Usually access levels of properties are omitted, but generally one can assume that all properties included in drawings are publically accessible, either directly or via access methods

b) if something may not be immediately evident out of the drawing but might be necessary/useful for its understanding (respectively to understand the context of the pictured elements), a box with an orange to yellow gradient was used to add more details in a textual form

c) inheritance of one element from another is indicated with a solid line and an arrow, usually without the textual hint attached to the line

d) a class implementing an interface is indicated with a dashed line and an arrow, usually without the textual hint attached to the line. In some places a stereotype notation is used to indicate that a box constitutes an interface (for example if a class relates to other classes/objects via an aggregation but only demands the related classes to implement a certain interface – for instance the next item d))

e) since composition and aggregation symbols are very often controversially defined in literature, little effort was taken to align all diagrams with one final, strict rule. Usually a composition in this thesis indicates that the part-element may (or at least should) not exist without the owner-element. Due to the blurry difference between composition and aggregation, without doubt one notation may be substituted by the other occasionally.

   Additionally in this thesis a specialty of Java – that is Inner Classes – are usually connected to its parent class with a composition. While there exist various recommendations for non-standardized symbols like (Holub, 2011), for the sake of simplicity this approach was chosen.

   If omitted, then multiplicity is assumed to be 1

f) as indicated in item e) aggregations usually put elements in a relation which may exist independently. Again, if omitted, then multiplicity is assumed to be 1

g) shows a class which uses another class in some way. Usually additional textual information is available indicating the purpose of this interaction

h) shows a class which relates to another class in some way. Usually additional textual information is available indicating the purpose respectively nature of this relation

i) shows an element where at least two methods seem to be of particular interest for its understanding. While the 'setString(…)' method has a void return type, the 'toString()' method returns an object of type 'String'

j) shows a class where at least the 'name' field seems to be of particular interest for its understanding

k) classes showing an italicized name are definitely abstract

Additionally group nodes were used in various places to give a hint about how classes belong together in a broader sense, but this notation should be self-explanatory.

Chapter 11

References

11.1 References

Belcher, J., McKinney, A., Bailey, P., & Danziger, M. (2007, December 16). TEALsim: A Guide to the Java 3D Software (Version 1.1). Cambridge, Massachusetts, USA.

Berger, S. (2012). *Virtual 3D World for Physics Experiments in Higher Education*(Master's Thesis). Graz, Austria: Graz University of Technology.

Bernier, Y. W. (2001). *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. Retrieved from Valve Developer Community: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization

Brown, W. J., Malveau, R. C., McCormick III, H. W., & Mowbray, T. J. (1998). *Anti Patterns.* New York: John Wiley & Sons, Inc.

Holub, A. I. (2011, September 26). *Allen Holub's UML Quick Reference*. Retrieved from Hollub: http://www.holub.com/goodies/uml/index.html

InforMatrix GmbH. (2007). *InforMatrix LogGui.* Retrieved from http://www.informatrix.ch/loggui/index.html

Kaplan, J. (2012, March 28). *Open Wonderland Forum: Utilization of Multicast in OWL*. Retrieved from Google Groups: http://groups.google.com/group/openwonderland/browse_thread/thread/5a6cdec9ca72fd75

Massachusetts Institute of Technology. (2012, March 1). *TEALsim Project at MIT*. Retrieved from http://web.mit.edu/viz/soft/visualizations/tealsim/index.html

RedDwarf Server Application Tutorial. (2010, March). Retrieved from RedDwarf Server Project: http://sourceforge.net/apps/trac/reddwarf/attachment/wiki/Documentation/RedDwarf %20ServerAppTutorial.odt

Retain precision with Doubles in java. (2008, November 27). Retrieved from Stack Overflow: http://stackoverflow.com/questions/322749/retain-precision-with-doubles-in-java

Scheucher, B. (2010, March). *Remote Physics Experiments in 3D*(Master's thesis). Graz, Austria: Graz University of Technology. Retrieved from http://www.iicm.tugraz.at/thesis/MA_%20Bettina_Scheucher.pdf

11.2 Table of the most important software (tools) used

TEAL Simulation Framework
http://web.mit.edu/viz/soft/visualizations/tealsim/index.html
http://sourceforge.net/projects/tealsim/

Open Wonderland
http://www.openwonderland.org

Netbeans IDE
http://www.netbeans.org/

IntelliJ Idea Ultimate (evaluation version)
http://www.jetbrains.com/idea/

yEd Graph Editor
http://www.yworks.com

Balsamiq Mockups
http://www.balsamiq.com/

CLOC
http://cloc.sourceforge.net/

EasyMock
http://www.easymock.org/

Ant-Contrib Tasks
http://ant-contrib.sourceforge.net/

InforMatrix LogGui
http://www.informatrix.ch/loggui/index.html

TopThreads JConsole plug-in
http://lsd.luminis.nl/top-threads-plugin-for-jconsole/

Chapter 12

Listings

12.1 Table of figures

12.2 Table of tables

12.3 Table of listings

12.4 Table of use cases

Chapter 13

Index

13.1 Glossary

CECI	–	Center for Educational Computing Initiatives
CVS	–	Concurrent Versions System or Concurrent Versioning System
fps	–	frames per second
kLOC	–	lines of code x 1000
MIT	–	Massachusetts Institute of Technology
PD	–	Project Darkstar
TEAL	–	Technology Enabled Active Learning
TEALsim	–	TEAL Simulation Framework
UI	–	user interface

13.2 Index

No index entries found.

13.3 Digital assets