

A Generic Framework for Discrete Simulation Based Optimization

1. Abstract

The aim of this work is to provide generic, performant software tools for simulation based optimization. The object oriented solution consists of two main software libraries. The first library is a framework for the implementation of discrete simulation models, including some simple models for demonstrations and comparisons. The second library is a framework for the problem-independent implementation of optimization algorithms with a few ready-to-use algorithms to demonstrate the technique.

About the simulation library

There are a number of potential hazards in simulation based optimization, some of which originate in the simulation engine itself and shall be avoided in this work. One of these hazards lies in the way random numbers are generated for stochastic simulation experiments: Large models ($\gg 1.000$ entities) are often built dynamically (e.g. in [Speckle 2009a]) using databases for the configuration and / or parametrization.

Dynamic model creation is also required in optimization of layout configurations, no matter the size of the model. In this context, reproducibility has to be considered very carefully. If the order in which model components are created is not fixed due to the ongoing optimization, due to unstable sorting algorithms in the model initialization code or due to DBMS / ORM related reasons, the way random numbers or seeds are distributed between simulation entities will also change. The same problem arises when entities with stochastic properties are created dynamically during a simulation run and the order of creation changes due to a changed parameter set, which is not an uncommon scenario in optimization. This may cause only subtle changes in the outcome which will not be noticed and lead to invalid experiment results. The problem is of particular relevance during optimization where single simulation experiments run unattended in the background. Therefore a mechanism has to be implemented so that random numbers or seed values can be linked to identifiers.

A very similar problem may arise in the ordering of events, if they are not prioritized explicitly (the engine allowing). Because of that a simulation engine for use in optimization should differentiate between events, event instances and event handlers (similar to the way this is done in programming languages like C#, [ECMA 2006b]) and allow fine grained prioritization in all three layers.

About the optimization library

Usually, in optimization either the target function's inputs and outputs have to be adapted to match the optimization algorithm's data structures or the optimization algorithm has to be adapted to the target function. This problem shall be alleviated in the proposed framework by making use of the

generics paradigm present in current programming languages and by employing a sophisticated set of interfaces for both the target functions and the solution algorithms. These interfaces should additionally allow for the tuning of optimization algorithms where this is applicable.

The proposed framework will make it possible to design and debug simulation based optimization solutions using Java [Sun 1996] and different .NET languages [ECMA 2006a] with the support of modern development environments. The use of modern, object oriented programming- and software development paradigms and tools like unit testing, contract based programming, generics, lambda expressions, profiling, plugin architectures and others should make the development of simulation models and optimization algorithms less error prone and produce more scalable and highly performant results.

1. Abstract.....	1
2. Introduction.....	6
2.1 Scientific Concepts.....	6
2.1.1 Discrete Event Simulation	6
2.1.2 Metaheuristic Optimization.....	7
2.2 Technologies.....	7
2.2.1 Programming Languages	8
2.2.2 Coding Conventions.....	9
2.2.3 Development Environments.....	10
2.2.4 Code Documentation.....	10
2.2.5 Hosting and Versioning	10
2.2.6 Unit Testing	11
2.3 Development Methodology	11
3. Problem Definition.....	11
3.1 Functionality.....	12
3.2 Reliability	13
3.3 Usability	13
3.4 Efficiency	13
3.5 Maintainability	14
3.6 Portability	14
4. Simulator.....	14
4.1 Engine.....	14
4.1.1 Event Scheduler	15
4.1.2 Model	15
4.1.3 Entities	16
4.2 Event Concept	17
4.2.1 Event	20
4.2.2 Event Instance	21
4.2.3 Event Handler	21
4.2.4 Priorization.....	22
4.3 Stochastics	22
4.3.1 Random Number Generators	23
4.3.2 Distributions.....	25
4.3.3 Seed IDs and Random Seeds	26
4.4 Resource Management	28
4.5 Templating Concept	28
4.5.1 Basic Implementation	28
4.5.2 Simple Entity.....	29

4.5.3	Stochastic Entity	29
4.5.4	Resource Entity	29
4.5.5	State Machine Entity	29
4.5.6	Task Machine Entity	30
4.5.7	Mobile Entity	30
4.6	Extensibility and Interfaces	30
4.6.1	Identification of Entities.....	30
4.6.2	Containers and Attachment	31
4.6.3	Dynamic Creation of Entities.....	31
4.6.4	Reception, Storage and Buffering	31
4.6.5	Further Interfaces	31
4.7	Simple Templates	31
4.7.1	Source.....	32
4.7.2	Buffer	32
4.7.3	Delay	32
4.7.4	Server	32
4.7.5	Sink	32
4.8	Further Utilities	33
4.8.1	Resource Manager.....	33
4.8.2	Path Network.....	33
4.8.3	State Machine.....	33
4.8.4	Task Machine	34
4.8.5	Simulation Logger.....	34
5.	Optimizer	34
5.1	The Framework	34
5.1.1	Problem Interface	35
5.1.2	Solution Interface	35
5.1.3	Strategy Interface	35
5.1.4	Configuration Interface	35
5.1.5	Operator Interface	36
5.1.6	Further Interfaces	36
5.2	Implementations	36
5.2.1	Random Strategy	36
5.2.2	Simulated Annealing	36
5.2.3	Evolutionary Algorithms	36
6.	Discussion	37
6.1	Maturity	37
6.2	Open Questions	38
6.3	Future Tasks.....	38

7.	Acknowledgments.....	39
8.	Literature.....	40
9.	List of Figures.....	45
10.	Appendices.....	45
10.1	Tests & Benchmarks.....	45
10.2	Coding Conventions.....	51
10.3	License.....	57
10.4	UML Class Documentations	60
10.4.1	Interfaces.....	61
10.4.2	Base Entity Framework.....	64
10.4.3	Entity Templates.....	66
10.4.4	Logging Framework.....	71
10.4.5	The Optimization Framework.....	72
10.4.6	Further UML Diagrams.....	77
10.5	Code Excerpts.....	85
10.5.1	The SQSS Model	85
10.5.2	Helper Functions for the Evolutionary Strategy	88
10.5.3	Default Evolutionary Delegate Implementations.....	89
10.6	Logging Framework.....	92

2. Introduction

The aim of this work is to document and discuss the architecture of a discrete simulation library and a heuristic optimization library ("C# Toolbox"), each written in the C# language [ECMA 2006b] and later ported to Java [Sun 1996] ("Java Toolbox"). Altogether the framework is well tested and documented (see appendix 10.4).

The following parts of this chapter provide some fundamentals for the scientific concepts (discrete modelling and simulation as well as heuristic and metaheuristic optimization) and technologies (software engineering, programming languages, object oriented programming and others) on which this work is based.

2.1 Scientific Concepts

This chapter touches on the topics "Discrete Event Simulation" and "(Meta- / Hyper-)Heuristic Optimization" as far as this is required for the understanding of this work and as far as there are new ideas or approaches towards these concepts in the related software libraries. A fundamental knowledge will be assumed though, as this is not a textbook on simulation and optimization.

2.1.1 Discrete Event Simulation

Discrete-Event Simulation (in short "Discrete Simulation") is a technique to represent a physical system as a sequence of chronological events. (see [Banks 2010] or [Law 2000] for comprehensive definitions) These events mark changes of the model state and may trigger some logic, optionally including the subsequent triggering of further events. The simulation then continues until no more events are left or some ending condition is met.

In this work events are defined to be instantaneous. Activities or processes are modeled as sequences of events as opposed to simulation engines which allow interval-based events or which are based on activities instead of events. Furthermore the simulation engine presented in this work allows multiple events to occur at the same point in time. This requires synchronization and prioritization mechanisms which will be discussed in chapter 4.2. Therefore a plain list (as can be used in simpler simulation engines) is not sufficient to hold the references to scheduled events in this case.

Simulation models are usually built up from components which correspond to entities in the real world. Their interactions are then simulated on a discrete time domain where the time in between events can be skipped. Commercial simulation applications like AnyLogic [XJTech 2010], Arena [Kelton 2010], Enterprise Dynamics [SimPlan 2010] or Plant Simulation [Siemens 2010] often include a rich set of predefined entities for different industry domains. (see [ORMS 2010] for a more complete list of available simulation software tools)

The simulation library presented in this work does not contain concrete model objects except for very basic ones but it does provide certain entity interfaces and classes ("templates") which facilitate the implementation of user defined model objects. These are discussed in chapter 4.5.

Usually in such simulation models there is at least one active element, which automatically creates at least one primary event. Alternatively there may be some general startup logic which can be used to trigger further events. Complex dynamics may then emerge due to the fact that the logic behind events can dynamically create further events. In this work both approaches are supported.

Often discrete models are used to analyze statistical properties of the simulated system. Therefore most real world scenarios require the usage of stochastics in the simulated processes for results to be meaningful and significant. These have to be based on statistical distributions with the randomness generated by pseudo random number generators. If such a feature is available (which it is in most modern simulation software) the software can be used to model stochastic processes.

Vice versa stochastic processes can be used to mathematically represent certain simulation models. Another way of mathematical representation with better readability are petri nets [Petri 1966]. This approach was taken in [Löschner 2009], where models are represented as stochastic petri nets.

Simulation software can be divided into out-of-the-box applications and simulation libraries. [Speckle 2009a] and [Speckle 2009b] define some criteria to distinguish between the two kinds of software and discuss advantages and disadvantages of simulation libraries versus out-of-the-box simulation software. The aim of commercial simulation applications is to provide the user with a way to quickly build models without much prior knowledge of discrete simulation. Graphical modelling tools, predefined components and simple scripting facilitate this. The aim of simulation libraries on the contrary is to give the user as much freedom as possible in the creation and usage of his models.

The simulation engine presented in this work is clearly a simulation library. It can however be used as the core of a full simulation application. This possibility was considered in the software design, for example, by allowing dynamic creation of models, entities and all other required elements.

Modern simulation software often combine classic discrete-event simulation with the agent based simulation paradigm [XJTech 2010]. There is no clearly defined border between object oriented discrete simulation and agent based discrete simulation. However, the engine presented in this work supports the agent approach. For very large agent systems though it is recommended to abstain from the use of the templates presented in chapters 4.5.2 to 4.5.7 as they are mainly designed for comfort and not performance.

2.1.2 Metaheuristic Optimization

In many real world problems, direct mathematical optimization is not viable due to the search space being too large or due to the problem being too complex for a manageable mathematical representation. In such cases metaheuristic optimization [Glover 1986] may provide better chances to find the optimum or at least useful local optima.

Simulation problems are often based on permutations of customer- or production orders. Such permutation problems can quickly reach unmanageable dimensions. Depending on the real process it may be necessary to include hundreds or thousands of orders into the optimization process for a meaningful result. Straightforward approaches are often additionally impeded by a complicated set of constraints imposed upon a valid ordering of items. Finally if stochastics have to be considered as well, even the more sophisticated direct methods will quickly reach their limits, especially on conventional desktop computers.

Algorithms like very large-scale neighborhood search (VLNS) [Ahuja 2000], genetic algorithms [Holland 1992], ant colony optimization [Dorigo 1992] and others are specifically designed for such huge and often heavily fragmented search spaces. The choice which kind of algorithm to use on a given problem depends on the nature of the problem and is not a trivial one. But in many cases the tuning of a chosen algorithm is more efficient than comparing multiple algorithms against each other (see also [Luke 2009]). Therefore and because of its simplicity a genetic algorithm has been used in this work to demonstrate the usage of the accompanying optimization library.

The main goal of the provided optimization library is not metaheuristic optimization though. Inspired by the challenges of simulation based optimization I have rather attempted to provide a framework to facilitate the implementation of hyper-heuristic solutions (see [Fisher 1961], [Fisher 1963], [Gratch 1993]) which can be seen as a learning or optimization method to select or compose the optimization method(s) for the actual problem.

2.2 Technologies

This chapter describes the tools and technologies which were used to implement the software library presented in this work.

2.2.1 Programming Languages

As mentioned above the simulation and optimization libraries in this project are being developed in two different languages simultaneously. The base implementation is written in the C# programming language [ECMA 2006b] and a further implementation is provided in Java [Sun 1996]. For both implementations the most recent available language version was used. (Up to the time of completion of this work the most recent version of C# and the .NET Framework was version 4.0 and the most recent Java version was version 1.7.)

Both languages are sometimes said to be "platform-independent" or "cross-platform" [Sun 2010], [Microsoft 2000], [Troelsen 2007]. Java provides its own platform in the form of the Java Virtual Machine (JVM) [Lindholm 1999] while C# is based on the Common Language Infrastructure (CLI) [ECMA 2006a] both of which have implementations for different classic platforms like Windows, Mac-OS-X or Linux. So obviously there are restrictions to independence: for a platform to actually execute code from one of these languages, either the JVM or the CLI has to be implemented for the target machine or operating system. Company policies, personel qualifications and personal preferences of responsible IT managers can further limit the possibilities of actually using software of a certain language. This is the main reason for implementing a software in both mentionned languages.

Fortunately Java and C# are very similar so that significant parts of the code can be ported without change. But there are a number of code constructs in C# which do not exist in Java. Also, there are some concepts which are not interpreted in the same way. Yet, the comparison between the two languages is not in the scope of this work, so the following chapters will cover only the most notable differences and how they were dealt with. More information about the general differences between the two languages can be found in [Obasanjo 2007], [Albahari 2000] and [Hejlsberg 2000].

2.2.1.1 Generics

Both C# and Java have a "Generics" concept in their more recent versions. But while the syntax is quite similar, the semantics and implementation of these concepts differ in many ways. Unfortunately Java generics are a compile time feature based on "Type Erasure" (the compiler removes all information related to type parameters and adds runtime casts) and not a real language feature. This is the cause of some of the shortcomings of java generics as compared to C#. Obvious differences include the following:

- Java does not allow creating arrays of generic parameter types.
- Java does not allow querying the runtime class of a generic type parameter or comparing types using the keyword "instanceof".
- C# generics in contrast to Java generics allow type restrictions like "where T : new()", which means that the type must have an empty constructor. This can be used in C# to create instances of a generic parameter type.
- C# also provides a mechanism to acquire a default instance of an unknown type with the "default" keyword, allowing to set number types to zero. In Java the default value is null for all types.
- Java does not allow references to type parameters of non-static classes from static methods.

Further information about differences between the languages' generics concepts can be found in [Pryor 2007] and [Hejlsberg 2004].

To avoid problems with the portation from C# code to Java the following strategies were used:

- In general, generics were kept as simple as possible and nesting of generic parameters was avoided.

- Patterns requiring the usage of reflection on generic types were avoided.
- Occurances of the new() constraint were replaced with an interface method providing a default instance or factory method in Java.

Furthermore, in some cases a non generic wrapper class was provided for generic classes. This was less to overcome shortcomings of Java generics but more to support developers who are not yet proficient in the use of generics.

2.2.1.2Delegates

The C# language has a type called "delegate" which is similar to a C++ function pointer. The difference is that the delegate is type safe and object oriented. In Java one has to implement a command pattern (as in Java "Listeners") to get a similar behavior. The command pattern is basically an interface defining the required command signature. Delegates allow the usage of any method with the correct signature though. So library methods which accept delegate parameters can consume methods from third party software without a wrapper class in C#.

Additionally the C# language has the predefined Func<...>, Action<...> and Predicate delegates which are generic templates to simplify the use of delegates. For all delegate types the command pattern was used in Java to deal with the difference.

2.2.1.3Lambda Expressions

Since language version 3.0 lambda expressions as in lambda calculus have been available for C#. This concept is well known from functional programming languages like LISP and Haskell. But imperative languages can also provide an equivalent to lambda expressions by using function pointers or delegates. In Java this was resolved again by using the command pattern. Furthermore the use of lambda expressions was limited to very simple, internal cases and not exposed in library methods.

2.2.1.4Expression Trees

An Expression Tree is a representation of code in a tree data structure with each node being an expression. These are available in C# and can be very useful in meta-programming scenarios. Implementing such a feature in Java would be extremely cumbersome, therefore the use of expression trees in the simulation and optimization libraries was avoided completely. However, the C# Toolbox uses a logging framework which the author has written independently and which uses expression trees to allow the user to pass arbitrary data to methods using a very comfortable syntax. In the Java implementation this was completely replaced with a simple Map (Dictionary in C#).

2.2.1.5Extension Methods

The C# language allows the programmer to add methods to existing types without modifying the original type. These extensions are defined as static methods in a static class using a keyword to point to the type which is to be extended. The methods can then be called directly via the type, as if they were ordinary member methods. In the Java implementation these were simply replaced with normal static methods.

2.2.2 Coding Conventions

For this and other software development projects the author has defined a set of coding conventions. These conventions and best practices are intended to help with a number of issues which typically occur when programmers write code without adhering to some common guidelines:

- Code may be difficult to read and hard to understand
- Performance is not optimal and it is hard to identify the bottlenecks

- It is hard to say which parts of the software are responsible for what
- The programmer spends a lot of time searching instead of programming
- The software itself will be unreliable
- The correction of errors is difficult
- Code changes are difficult and result in unexpected behaviour
- Code complexity increases with additional features

Most of these problems are due to excess complexity. Additional difficulties arise from a lack of documentation, common standards and practices.

The software presented in this work is written in strict adherence to the above mentioned conventions with only very few exceptions. One notable exception is that in C# there are separate regions for initialization and reset code as these play an important role in reproducibility issues and must never be overlooked in a discrete simulation context. Another exception is that (simulation) events and handler methods have been grouped into regions on top level due to their special role in discrete simulation. In the very rare case that a class contains native C# events as well as simulation events these have been kept in the same regions because native C# events can be discerned from simulation events by the "event" keyword.

2.2.3 Development Environments

For Java development we used the free Eclipse development environment [Eclipse 2010], version 3.5.2 (Codename "Galileo"). Subversion was integrated using the Subclipse plugin [Subclipse 2010] version 1.6.10. A plugin for unit testing (see chapter 2.2.6) with JUnit versions 3.8.2 or 4.5.0 is integrated per default.

The C# code was mainly developed using Microsoft's free Visual Studio 2010 Express development environment [VSEExpress 2010]. For profiling and unit testing one of the first beta versions (starting with 4.0.0.5835 in May 2010) of SharpDevelop 4 (Codename "Mirador") [IC#Code 2010] was used. All further development starting from June 2010 was done in SharpDevelop as well. The Visual Studio Express editions do not support add-ins which makes it impossible to use common SVN integration and unit testing tools. There is also no integrated profiling tool available as in SharpDevelop.

2.2.4 Code Documentation

Java sourcecode was extensively documented using Javadoc [Sun 2009]. The C# source code was documented using Sandcastle [Sandcastle 2010] and the Sandcastle Help File Builder 1.8.0.3 [Woodruff 2010].

Unified Modeling Language (UML) diagrams (see [OMG 1997], [ISO 19501] and [ISO 19505]) of selected C# classes and interfaces have been created with Visual Studio. Additional UML documentation for both versions of the framework was created using Enterprise Architect [Sparx 2010]. Enterprise Architect supports source code engineering and reverse source code engineering for Java, C# and many other programming languages, meaning that source code can be created from UML and vice-versa. All UML documents were created from the source code using this feature. Therefore no discrepancies are to be expected between the UML documentation and the actual code. The UML Documentation is appended in chapter 10.4.

2.2.5 Hosting and Versioning

Source code versioning was managed using "Subversion Repositories" (SVN) [Pilato 2009]. At the beginning of the project in December 2009 these were hosted on Project Kenai [Kenai 2010]. Other hosting services like SourceForge [Geeknet 2010] and Google Code [Google 2010] were also

evaluated. The main feature that primarily leads to the selection of Project Kenai over others was the fine-grained rights management which the Kenai platform provides. Unfortunately Project Kenai has been migrating to Java.Net since the beginning of 2010 and Oracle mistakenly announced that Project Kenai would be terminated completely. The author therefore temporarily migrated all his software projects to a commercial hosting platform while searching for a better solution. A few days later the announcement was corrected and migration to Java.net was scheduled for the end of March. However, as of now (July 2010) there is still no official announcement nor a schedule for the completion of this migration project. Therefore, the projects were subsequently moved to Origo [Oriact 2010], the free hosting service provided by the ETH Zürich University, in June 2010, and after the closure of this service in 2012 open hosting was discontinued. The software is now being hosted and maintained by the private company AREC GmbH in Austria.

2.2.6 Unit Testing

The software presented in this work contains a fair amount of unit tests written for JUnit (for the Java solution) [JUnit 2010] and NUnit (for the C# solution) [NUnit 2010]. Parts of the software were developed using a Test Driven Development (TDD) [Beck 2002] strategy but most of the code was tested after implementation. Detailed test- and coverage results can be found in the appendices.

2.3 Development Methodology

Common software development models like the V-Model XT [Rausch 2005], Scrum [Schwaber 2004] or Extreme Programming (XP) [Beck 2005] are meant for software development in teams. Here, two relatively small software libraries were developed each by a single person. Therefore using a formal methodology would have been impractical.

Nevertheless some ideas and methods from the above mentioned methodologies – especially from the agile ones – were used, the most important one being pair programming [Beck 2005]. Due to the developers living on opposite ends of Austria, this was achieved using the Virtual Network Computing (VNC) software TightVNC [TightVNC 2010] and the commonly known Voice-over-IP (VoIP) Software Skype [Skype 2010].

Apart from that the following quality criteria for software as defined in the ISO/IEC 9126 standard [ISO 9126] were always observed:

- Functionality (suitability, accuracy, interoperability, security)
- Reliability (maturity, fault tolerance, recoverability)
- Usability (understandability, learnability, operability, attractiveness)
- Efficiency (time behaviour, resource utilization)
- Maintainability (analyzability, changeability, stability, testability)
- Portability (adaptability, installability, co-existence, replaceability)

It should be noted that the ISO 9126 has been superseded by the ISO/IEC 25000 [ISO 25000] in 2005 but this doesn't change the fact that the above quality criteria are still important and valid.

3. Problem Definition

The presented work is aimed at providing the software tools required to develop and maintain discrete simulation models and to be able to use them in heuristic optimization. Further requirements are:

- Stochastics Support – Common distributions should be provided and the modeller should be prevented from breaking reproducibility as well as possible without restricting his modelling freedom.
- Antithetic Experiments – It must be possible to perform antithetic experiments by simply

switching a model to antithetic mode. The antithetic variates method is a technique to reduce variance in stochastic experiments.

- **Dynamic Model Creation** – It should be possible to create and manipulate models dynamically, including database driven generation. Variations in the generation process should not have any impact on reproducibility of experiments.
- **Experiment Automation** – It should be possible to automate the creation and execution of simulation and optimization runs. Automation should not be restricted technologically and should make use of modern interfacing practices like inversion of control.
- **Metaheuristic Support** – It should be possible to use simulation models in metaheuristic and other optimization methods and to tune these algorithms. This means that a huge number of simulation runs will be required to evaluate solution candidates and it imposes a significant performance requirement on the software.
- **Hyperheuristic Support** – It should be possible to use optimization strategies in hyperheuristic scenarios. Therefore it is required to provide generic interfaces for the automation of the strategy configuration.
- **Possibility for Hybrid Models** – It should be possible to extend the software to support hybrid models, combining discrete simulation and continuous simulation methodologies.
- **Possibility for Distributed Models** – It should be possible to extend the software to support distributed models.

The following chapters list some of the problems the author and his colleagues experienced in their work on discrete simulation and simulation based optimization projects which finally led to the author's decision to implement a completely new simulation engine. The problems are categorized according to the quality criteria from [ISO 9126] as listed in chapter 2.3. The solution approaches are briefly explained with each problem.

3.1 Functionality

Many common simulation applications cannot be efficiently used in automation scenarios, making it hard to use them in optimization experiments and unattended background operations in general. Automation is even impeded on purpose in some of the software (notably AnyLogic, [XJTech 2010]), probably as a kind of copy protection mechanism.

This also makes interoperability harder. Even though in the established commercial solutions some interfaces are usually provided and well documented, they are somewhat limited. Some packages only provide native interfaces, others only managed ones and most packages interact with databases only via Microsoft's "Open Database Connectivity" (ODBC) or Oracle's "Java Database Connectivity" (JDBC) if at all. But even if many interfaces are provided, complete freedom in how to interact with other software only exists in a few of the simulation libraries.

The software presented in this work makes no interfacing or automation restrictions whatsoever. All the possibilities provided by the .NET Framework and the Java technology are open to the user. Furthermore the simulation engine is specifically designed to be used in optimization scenarios and an optimization library including a demonstration project is included. Of course this does not mean that optimization can only be performed using the provided library as in some of the commercial products.

Another weakness of the commercial simulation applications is dynamic model generation. These packages are usually intended for the manual graphical design of models. Except for build-order dependent random seed issues this is normally not a problem in simulation libraries.

Finally in most of the simulation software (applications as well as libraries) there are limits in the way the user can prioritize events. This is important to control the ordering of events in simultaneous

occurrences. The simulation library presented in this work uses a three stage prioritization approach in events, event instances and handlers to give the user the optimal freedom (see chapter 4.2.4).

3.2 Reliability

Reliability in the more established simulation software is not a prime issue. Most of the big commercial packages are sufficiently mature and stable. But problems frequently arise in code which was added by the user. In our daily work at Profactor and V-Research we have experienced quite a few crashes and freezes with all of the commercial packages we used (mainly Anylogic [XJTech 2010], FlexSim [FlexSim 2010], Enterprise Dynamics [SimPlan 2010] and Plant Simulation [Siemens 2010]), some of which did not seem to be related to user code.

Of course one cannot prevent the user from writing faulty code. But the use of managed code like Java or .NET as in the presented work does alleviate such problems. Managed code is type safe and protects the programmer from quite a few of the common pitfalls in native programming like memory leaks, pointer errors and buffer overflows.

Basic debugging of user code is usually possible in commercial simulation applications. But as soon as execution control changes to the application the user loses his context and can only guess what is wrong if the actual error occurs outside his own code (even though it may be caused by his own code).

Libraries for which source code is available can be fully debugged of course. Conditional breakpoints, watches and tracing are available as a matter of course. Java and .NET even allow changing code during execution to a certain extent. Finally, versioning, profiling and unit testing are possible. This finally gives the modeller luxuries which in software development have been state of the art for decades.

3.3 Usability

While some of the simulation applications have a good usability this cannot be said about most of the simulation libraries. Of course software libraries are not intended for the typical user. In general, software libraries are intended for software developers. But engineers working in discrete simulation are often mathematicians, physicists, electrical engineers and scientists from other disciplines who are not really specialized in software development. Therefore it is not advisable to expose them to the latest and most complicated patterns and practices in computer programming. To allow even relatively unexperienced programmers to fully utilize the software presented in this work we have kept it as simple as possible from the outside but without impairing the use of advanced programming patterns for the more experienced software developer.

Common discrete simulation engines (in libraries as well as in desktop applications) do not differentiate events, event instances and handler methods, at least not in a way that enables the user to prioritize each separately. Furthermore most simulation software forces the user to write event logic concerning one event all in the same place. These problems have been avoided in the current work by the use of a special event/handler pattern as presented in chapter 4.2.

Finally, as far as applicable the ISO 9241 (see [Weers 2010] for a summary of the contents) and the heuristics from [Shneiderman 1998], [Norman 1990] and [Nielsen 1993] have been applied in the design and development of the presented software. The author believes usability to be of extreme importance in all kinds of software, not only in end-user desktop applications.

3.4 Efficiency

Most common simulation applications use up considerable working memory and processor time. Even some of the mainstream packages have been found not to be linearly scalable. This means that linear incrementation of model complexity results in greater than linear resource consumption (see appendix for detailed performance test results). The performance goal of these applications is limited by the requirements for one single manually designed (and therefore not particularly large) model.

Furthermore graphical animation plays an important role in these applications but it seems that in most of the software it has been implemented in a way so that it impedes performance even when the animation feature is turned off.

Simulation libraries are usually more efficient in resource consumption and runtime behaviour. The software presented in this work has a lightweight engine at its core and was specifically designed for the use in optimization experiments with large numbers of evaluations. As a part of the authors coding conventions profiling of the software was done on a regular basis.

3.5 Maintainability

Usually discrete simulation models have a textual representation, even in graphical modelling environments. Unfortunately in common simulation applications this textual representation cannot be exported easily. This would be very helpful for versioning and improve changeability. Even Anylogic [XJTech 2010], where the models are based on Java code and the modelling environment is based on eclipse, versioning of models is not possible. Instead models are saved to proprietary binary formats to which no interfaces are available.

Another extremely helpful feature is unit testing. It is hard to understand why such features are not available in scientific software where verification and validation (analyzability and testability in usability terms) of results is so important.

Fortunately versioning, unit testing and profiling can be applied to most simulation libraries. Furthermore it is possible to analyze the heart of the simulation engine, the event scheduler, which of course in commercial applications is hidden from the user. For the scientific community it is not viable to work with black box tools, so even if other drawbacks of commercial desktop simulation applications were irrelevant, they cannot be used in scientific projects which must be reproducible.

3.6 Portability

Commercial simulation applications are usually available for one platform only with the commendable exception of AnyLogic [XJTech 2010], which is based on Java and Eclipse and can be installed on Linux, MAC OS and Windows. In contrast, only a few of the available simulation libraries are restricted to certain platforms due to their implementation languages.

The library presented in this work is available in Java and C# and thereby portable to all platforms which either have a JVM implementation or support the CLI. Furthermore with the C# implementation there is a specialized version for Windows platforms which is often preferable in the commercial sector while with the Java implementation there is a specialized version for the scientific and open source community.

On the topic of replaceability it has to be mentioned that there is not yet an established standard of how to represent discrete simulation models so that they can be utilized by different engines. Replaceability is thereby restricted for all simulation software. Commercial simulation environments impose even more restrictions by the use of proprietary, undocumented binary file formats and by not providing a way to export models to a commonly readable file format. The simulation library presented in this work allows to export simulation models in any way the user wished. It is recommended though to use XML serialization and include rich textual documentation as long as no common model representation is established in the scientific community.

4. Simulator

4.1 Engine

Typically, a discrete-event simulation engine consists at least of some kind of clock, a datastructure to hold events and some mechanism to execute logic. Additionally there may be a datastructure to hold references to model components ("entities"). The simulator presented in this work has an

"Engine" namespace ("MatthiasToolbox.DiscreteSimulation.Engine") which basically consists of these elements plus some additional structures. These are discussed in the following chapters.

4.1.1 Event Scheduler

Core of the simulation engine is a mechanism to manage events on a timeline. This mechanism is manifested in the class "EventScheduler". This event scheduler is only an example. Other event schedulers may be implemented without restrictions for different models or event scheduling concepts (e. g. for distributed or hybrid scenarios). The event scheduler not only holds references to the events but is also responsible for the invocation of the logic which may be associated with these events.

One of the central properties (getter in Java) is the EventList, which is a sorted dictionary structure containing another sorted dictionary. The outer key is the timestamp. The inner dictionary contains all events which are to occur at the point in time as specified by the key. The inner key is the priority of the events, which are the value members of the inner dictionary. Priorities are made unique as described in chapter 4.2.4.

A logging flag can be used to generally turn event logging off or to turn it on for events which have their logging flag set to true. Furthermore, as soon as events are available, the scheduler keeps track of the time and instance of the next scheduled event according to simulation time and to the most recently processed event instance if events have already been processed. Internally the scheduler also holds a reference to the model as a private class variable. The event scheduler cannot be instantiated without a model because it is dependent on the model's simulation time. In distributed scenarios a new event scheduler has to be implemented which synchronizes the different model parts. In this case the model reference will have to be the reference to the common model which manages the distributed sub-models.

The event scheduler allows to add events to the schedule using the Add or the AddImmediate method. Add does allow to add events to the current point in simulation time but this is not according to intended use and the behaviour is undefined. Adding events to a past point in time may even result in an infinite loop and subsequently a deadlock of the engine. Adding events through the model class prevents such errors. The remove method allows removal of event instances from the timeline. Events that have already been processed are removed automatically.

The above adding and removing methods are normally used exclusively by the model. They have been kept public nevertheless for allowing remote models to add events in case of distributed models or simulations.

The core of the event scheduler is the "ProcessNextPointInTime" method. It is also triggered by the model normally, and retrieves the events for the current simulation time as a first step. These are then raised in order of their priority. Handlers may add immediate events during this process which are raised subsequently. The adding of immediate events during immediate event processing is not supported. Ideally immediate events should be avoided in general but this is only the author's opinion and still a matter of intense discussion amongst experts (see also chapter 6).

Finally there is a reset method to be able to re-use the same event scheduler for multiple simulation runs. Further methods and properties are described in the code documentation.

4.1.2 Model

The simulation framework defines a common interface "IModel" (see chapter 10.4.6 figure 24) for models and a default implementation (Model). This allows using a different model class (and optionally a different event scheduler) for simulations than the one which is provided.

The model is responsible for the management of most model components and controls the event scheduler. Therefore it contains methods to add and remove events and entities. It also keeps track of

the current simulation time and allows starting, pausing and stopping simulation experiments. The default implementation additionally allows stepping through a simulation in freely definable time steps and simulation based conventional and conditional breakpoints. These help in debugging or analyzing simulations by allowing to pause the model execution at a certain simulation time without entering actual break mode as in normal debugging.

The model interface has been kept as small as possible to minimize the effort required to implement a model class and to maximize the designers freedom in doing so. Further interface members may be implemented as needed but this should be considered carefully in every case. In most cases it will be easier to extend or modify the model class to meet the requirements. Nevertheless further growth of the interface will be unavoidable whenever a part of the engine is extended which is not directly referencing the default implementation.

The default model implementation also acts as the central seed source (see chapter 4.3) for the complete simulation. Further seed sources will be initialized using a seed which is provided by the model's central seed generator. This seed generator allows the retrieval of indexed seeds to make them independent from the order in which they are retrieved, so that the same simulation using the same seed will produce the same results even if the order in which model components are created and added to the model changes.

This architecture also ensures that all random generators (including dynamically created ones) are set to their antithetic mode if the model's antithetic-switch is used. The same is valid for the NonStochasticMode switch (see chapter 4.3). Altogether this allows a very dynamic approach to discrete simulation, allowing the user to even change the model itself as a part of his simulation and still have reproducible results. (see chapter 4.3 for more details on the implementation of stochastics)

Finally the default model implementation contains events which can be used to hook into the starting and finishing processes of the simulation and a default resource manager for the use in standard resource seizing scenarios (see chapter 4.4).

4.1.3 Entities

Theoretically the simulation framework can be used with standard classes and objects from the respective programming language. However, certain features like seed management require the implementation of the interface "IEntity" (see chapter 10.4.6 figure 25) by the entity designer. It is therefore recommended that all simulation objects should implement this interface. There are only two items prescribed: a name and a model property (getter and setter in java) to be able to reference the model in which the entity resides. (Additionally a reset method and a string identifier are prescribed by the inherited interfaces) One of the reasons for this requirement is the possibility of distributed models in which entities can transist from one part-model to another. This may be useful in cluster, grid or cloud computing scenarios for very large models or in other parallelization and distributed computing scenarios.

The main linking and initialization steps for entities are implemented in the engine's abstract "AEntity" class. The patterns used in this class should not be diverted from without detailed understanding of the involved concepts. In distributed simulation scenarios, however, it may be helpful to implement a slightly different approach. Please feel free to contact the author in case of doubt.

Also, note that "AEntity" doesn't contain the mechanisms for an entity to be used as a resource or to make use of random numbers as required by the model. For using these features there are more sophisticated templates in the Entities namespace. In most scenarios the modeller will never have to work directly with "IEntity" or "AEntity" and instead derive his simulation objects from one of the classes in the Entities namespace (see chapter 4.5).

4.2 Event Concept

The event concept presented here is one of the main innovations of this work. In most simulation software and literature an event is seen as something atomic, usually implemented as a method call. [Banks 2010] like many others writes about events "being executed", suggesting that an event is directly connected to execution logic. Many modern programming languages like C# [ECMA 2006b], Delphi [Pacheco 2001] or Visual Basic [Microsoft 2010] have taken a different approach to events though, distinguishing between events and the related execution logic which is coded in handler methods. The approach taken in this work is related closely to the latter: The simulation engine provides an event structure which stands for the semantics (the meaning of the event), an event instance structure which stands for the occurrence of an event on the timeline and handlers, which stand for executable logic and are dynamically attachable to (and detachable from) events.

The atomic view on events forces the modeller to put all the logic (code) related to a given event together in one place. This can lead to complex methods and finally to unmanageability as shown in the following example:

Let us assume that we want to model a production facility consisting of different machines which are constantly being supervised by technical personell. These machines use exchangeable components and different resources to create a product from raw material in multiple subsequent stages. The machines have a color coded light on top and a digital display. The top light turns red on failure conditions and the digital display shows more detailed information on what the machine is actually doing.

Common examples of such scenarios include injection moulding, microprocessor fabrication or testing and different metal working processes. A good approach to model the machines in such a scenario is to use state machines with stochastic distributions for transitions and some customized logic. In some of the more complex real world scenarios it may be necessary to use the actual machine software to emulate the machines' behavior though.

For the current example let us assume that we have modelled the machine as a state machine and that the simulation software has a state transition event which we can extend with our own logic. This could look like the following pseudo-code:

```
method Machine.StateChanged
    switch (CurrentState)
        case ComponentOverheated
            (...)
        end case
        case MaterialExhausted
            (...)
        end case
        (...)
    end switch
end method
```

Lets assume that the red light on top of the machine will be triggered in both of the example cases above. If the component is overheated, it may just need some time to cool down but it may also mean that something is broken and the component has to be exchanged. In the second case where some material is exhausted this material will have to be replaced. In both cases, one of the supervisor personell will have to react. Assuming there is only one supervisor responsible for the machine, the code could be like the following:

```
method Machine.StateChanged
    switch (CurrentState)
        case ComponentOverheated
            Worker.CheckMachine(CurrentMachine)
        end case
        case MaterialExhausted
```

```

        Worker.RefillMachine(CurrentMachine)
    end case
    (...)
end switch
end method

method Worker.CheckMachine
    WalkTo(CurrentMachine)
end method

method Worker.RefillMachine
    TakeUpMaterial
    WalkTo(CurrentMachine)
end method

method Worker.ArrivedAtMachine
    switch (CurrentMachine.CurrentState)
        case ComponentOverheated
            if CurrentMachine.MustReplaceComponent
                ReplaceComponent(CurrentMachine)
            else
                WalkTo(DefaultPlace)
            end if
        end case
        case MaterialExhausted
            RefillMaterial(CurrentMachine)
        end case
        (...)
    end switch
end method

```

Of course there are many other ways how this could be implemented, depending on the actual simulation engine and the design of the model itself. The code in the methods "Worker.CheckMachine" and "Worker.RefillMachine" could also be written directly into the switch block of the machine for example. But assuming the "WalkTo" command is something that takes time and triggers an event, the code in "Worker.ArrivedAtMachine" must be written in the event method.

Here I had to duplicate at least a part of the switch statement from the machine. Code duplication is usually a bad thing, because it means that if something in this execution logic changes, the code will have to be changed in two places and thereby potentially affect two constructs instead of only one. An alternative would be to somehow store what the worker will have to do but this is not really an option in this special case: the machine could already have cooled down again when the worker arrives, meaning that the state "ComponentOverheated" may no longer be active when the worker arrives at the machine.

Let us now assume that in such a case the worker will see the red light disappear while being on his way and will know that he doesn't have to check the machine any more. This means that another "Machine.StateChanged" event will occur in which the machine switches back to some non-failure state. To notify the worker of this event and make him walk back to his default place if he was just on the way to check the machine, we could extend our code like this:

```

method Machine.StateChanged
    switch (CurrentState)
        case ComponentOverheated
            Worker.CheckMachine(CurrentMachine)
        end case
        case MaterialExhausted
            Worker.RefillMachine(CurrentMachine)
        end case
        case Working
            Worker.NotifyChangedState(CurrentState)
        end case
    end switch
end method

```

```

        case Idle
            Worker.NotifyChangedState(CurrentState)
        end case
        (...)
    end switch
end method

```

The worker class could then be extended to cancel his operation if the failure state ceases while he is walking to the machine.

At this point we may think about refactoring the mechanism so that independent of the machine state the machine will call `Worker.NotifyChangedState` and then migrate all the logic concerned with the workers actions into this method. This may seem to be a nice solution for the moment (except for the code duplication – usually the machine itself also has to execute logic on it's state changes). But consider a scenario with multiple workers and multiple machines now: The worker's "NotifyChangedState" method will have to synchronize all workers and machines in this case. So this is not a very tidy solution either.

Fortunately this is exactly the kind of scenario for which most simulation software provides resource management mechanisms. The drawback of these is that usually their logic cannot be changed freely. They may allow prioritization of seizing operations and preempting and some are even able to detect resource deadlocks. But the actual logic to execute when a resource is given to the waiting entity will have to be written in the respective resource receiver event method of this entity. So we end up implementing code for the worker in the machine class again. Extend the scenario with different kinds of forklifts, workers and machines and you will end up with a machine or worker class which contains nearly all the logic of your model, effectively preventing you from writing object oriented code according to modern best practices.

But it could be getting even worse: Who says that the worker is a passive entity, waiting to be summoned by lights on top of the machines? The workers could have their own agenda like routine maintenance and surveillance operations. So they could want to actually do something with the machines while those are idle. In this case you might well end up with a second resource scenario in which the workers have to seize machines. Now you have resources seizing each other and resource deadlocks will be guaranteed if only one resource manager is available (as in most of the simulation software). And even if it is possible to employ multiple resource managers which are able to detect reciprocal resource deadlocks you will have ended up with a lot of messy code in functional programming style which is really hard to debug, change and maintain.

As mentioned above the simulation framework presented in this work alleviates these problems by discerning events, event instances and handlers. Events are not method calls but objects which are semantically linked to the real event. Event instances are occurrences of an event on the timeline. And handlers are signatures for methods to be invoked when an event instance occurs. All three concepts are implemented as interfaces and some exemplary default implementations are provided (see chapters 4.2.1 to 4.2.3). Among other advantages this allows the modeller to group his code in an object oriented way.

In terms of the above example:

The machine may be implemented to have a `StateChanged` event or concrete events based on the meaning of the state transition (like "Overheated" and "MaterialExhausted") or even both. In the latter case one could bind logic to state changes in general or to selected state changes only. The machine and worker classes can each have their own handler methods to execute logic upon each other's events. That way the modeller can organize his code naturally and intuitively. Furthermore prioritization allows the modeller to manipulate his execution logic on an abstract level not only in case of simultaneous occurrence of events but also among handler methods.

The following subchapters give a more detailed view on the different event related interfaces and

datastructures as well as on prioritization.

4.2.1 Event

Event implementations as defined in the presented simulation library have to implement a generic interface `IEvent<THandler>` defined in the `Engine` namespace. The interface inherits `Comparable<IEvent<THandler>>` where the comparer implementation should compare the priorities of the given instances.

4.2.1.1 The Event Interface

The interface proscribes a name. This does not necessarily have to be unique but it is recommended to only use unique names of the form `"EntityClassName.EventName"` (e. g. `"Machine1.StateChanged"`). Furthermore it proscribes a `Priority` (see chapter 4.2.4) and a boolean logging flag `"Log"`. The logging flag allows the modeller to set logging on or off not only on model level but also more fine grained for events. This is helpful in debugging as it is common to have certain frequently recurring events which would be distracting during a search for minor irregularities in other, less frequent events. (see chapter 4.8.5 for more information on logging)

Furthermore a getter for the number of currently attached handlers `"HandlerCount"` is included in the interface. This is mainly used for runtime statistics which are often helpful as a first indication for reproducibility problems.

Finally the interface proscribes methods to attach and detach handlers. `"void AddHandler(THandler handler)"` and `"void AddHandler(THandler handler, Priority priority)"` are used to attach handlers. If no priority is given, it must be set to the according default priority. `"bool RemoveHandler(THandler handler)"` is used to detach handlers by reference. Implementors should return false if the given handler was not attached to the event and true if the handler was successfully detached.

Due to the fact that no unique identifiers are proscribed the implementor has to keep references to handlers which he wishes to detach and optionally re-attach dynamically. It is of course possible to provide implementations including a unique ID and some kind of lookup to retrieve events by their ID. But it should be noted that all event related issues are extremely performance critical. Dynamically detaching event handlers is not a very common scenario for discrete simulation models and the effort of keeping references to those detachable handlers is minimal. Therefore cost and benefit of implementing indexing and lookup for events should be carefully weighed against each other.

4.2.1.2 Abstract Implementation

The abstract and default implementations provided in the `Events` namespace do not provide indices and lookups for the reasons given above. The base for all further implementations is the abstract generic class `"AbstractEvent<THandler>"` which implements all required logic for attaching and detaching handlers, comparing instances and for prioritization. Only the `HandlerCount` getter is abstract and has to be implemented by concrete classes. It could have been implemented within the abstract class as well but for scenarios in which certain handlers or kinds of handlers should not be counted this would be uncomfortable.

Another thing which is not implemented nor proscribed in the abstract class are factory methods for event instances. This is due to the fact that no restrictions were made on the engine level as to what handlers actually are. While in C# it is very comfortable to simply use delegates (`"SomeEvent += HandlerMethodName;"`) this is not possible in Java (no delegates and no operator overloading), where handler interfaces have to be used to wrap the method signatures. This could also have been resolved by differing to a greater extent between the Java and the C# engines. But the way it was done not only keeps the two engines more similar, it also allows developers to use interfaces and classes for handlers in C#. This could be helpful in complex scenarios like mixed platform distributed models where the

exchange of delegates is not possible.

4.2.1.3 Concrete Implementations

Three concrete event implementations are provided which should suffice for all but the most extravagant scenarios: UnaryEvent, BinaryEvent and TernaryEvent.

In programming languages making use of event patterns (like VisualBasic and C#) all events are binary, meaning that the event invoker will pass on two parameters to the handler methods. The first argument is usually the sender / invoker itself. The second parameter is an "event arguments" data structure which contains the arguments specific for the kind of event in question. This convention has been established since the very early times of object oriented programming and has been proven to be practical and efficient. Therefore it is recommended that the modeller using the presented framework should make use only of the BinaryEvent and always provide a reference to the sender as first argument.

The provided implementations provide "GetInstance" factory methods to generate event instances. These can be parametrized with a priority and one (UnaryEvent) to three (TernaryEvent) arguments. If no priority is given, the instance will take over the priority of the event. The arguments can either be provided directly or by using a delegate. This is very useful in cases where one of the arguments is a simulation object which has to be created when the event occurs and not before. It is also possible to remove scheduled event instances from the future timeline. If this happens frequently, it is possible to improve performance by using delegates to retrieve event arguments on demand instead of building them during the creation of an instance.

Finally there is a "Raise" method which calls all handlers in order of their priority. It should be avoided to call this manually (from outside the event scheduler). Doing this may break the ordering of events and invalidate basic statistics. If you need to execute handlers, you can do so directly and if you need to raise the event at the current simulation time, it is recommended to do so using the immediate event mechanism.

4.2.2 Event Instance

Event instances are based on the interface "IEventInstance". Event instances have their own priority so that it is possible to enforce ordering not only of different events on the same point in time but also the ordering of different instances of the same event. Event instances also have their own logging flag and a "Time" property which will be set by the event scheduler. Setting the time of an event instance manually can lead to inconsistencies and errors (setting the time for one of the provided event instance implementation will not update the scheduler) and should be avoided. Finally there is a "Raise" method which will call the "Raise" method of the event to which the instance belongs.

The class "AbstractEventInstance" implements most of the interface in a straight forward fashion except for the "Raise" logic. This cannot be implemented in the abstract class because it is dependent on the concrete handler type which is a generic parameter of "AbstractEventInstance".

Concrete implementations are provided for the three event implementations as described above. These implement raise by retrieving the event arguments and passing them on to the event's raise method to which they keep a reference. The event reference is also exposed as a property "InnerEvent".

If the event arguments are passed directly to the instance, they can be changed via the "EventArgs" property up to the moment when the event is finally raised. After that the setter will throw an exception.

4.2.3 Event Handler

As mentioned above the C# version of the engine uses delegate types as handlers in the pre-defined

event implementations. The handler of a `BinaryEvent<IEntity, SomeEventArgsType>` will therefore be an `Action<IEntity, SomeEventArgsType>` and any void method with exactly two parameters of the given types can be used to attach execution logic to this kind of "BinaryEvent".

Note that there is also a "BinaryEvent" implementation which accepts only one type parameter (assuming the sender is of type "IEntity") and a non-generic "BinaryEvent" which uses "object" as event argument type.

In Java the handler is a simple interface with a proscribed method to emulate the behavior of a delegate.

4.2.4 Priorization

All priorities are of the type "Priority" in the Engine namespace. A priority consists of a value (double precision number), a priority type (enumeration) and an order number (integer). The priority type is used to discern the classes of priorities defined in the "PriorityType" enumeration:

- **LowLevelBeforeOthers** – On rare occasions this priority class is used internally for events, event instances or handlers which must precede all others. Using this outside the engine is not recommended.
- **SimWorldBeforeOthers** – This is intended for events, event instances and handlers which are related to the "simulation world" or "environment" and should be raised / handled prior to the rest.
- **User** – This is the default priority. It is recommended to use this by default and document all other cases clearly.
- **SimWorldAfterOthers** – This priority class is intended for special events, event instances or handlers (related to the "simulation world" as above) which are to be raised / handled after the above.
- **LowLevelAfterOthers** – Like LowLevelBeforeOthers this is used only on a few events, event instances or handlers in the engine code and it should be avoided in user code.

Events, event instances and handlers which are in the same priority class are ordered by their priority value which should be seen as the main priorization system. Note though that the engine cannot handle priority values of "Double.NaN" (for Not a Number) and it is strongly advised not to use Double.Infinity (neither PositiveInfinity nor NegativeInfinity). By default the priority value will be set to zero.

For events, event instances and handlers with the same priority class and the same priority value there is the order number which keeps track when an event instance was added to the timeline or when a handler was attached to an event so that those with equal priorities are processed in a stable ordering as well.

Summarizing this one can say that event priorities are the default priorities for the respective event instances and handlers are ordered and raised primarily by the event instance priority and secondarily by the handler priority.

4.3 Stochastics

The following chapters describe the random number generators and distributions as well as the "seeding" concept used in this work. This is a very sensible issue in discrete simulation.

Stochastic simulation experiments must be reproducible for the experimenter to be able to analyze the stochastic effects on the distribution of results. It is usually very difficult to estimate the number of experiments required for a statistically significant experiment result because of the complex propagation patterns which can emerge. Therefore it is crucial that all parts of the stochastics system work perfectly correct and are transparent.

Usually non-reproducibility can be discovered quickly by just running an experiment with the same seed twice. But there are cases in which it is not so easy. A reproducibility issue might occur only with certain seeds or certain simulation parameters. And in especially unfortunate cases it is not possible to trace such an error back to its actual cause even though one might be able to correct it. So in the worst case just a tiny mistake may invalidate experiment results reaching back for years!

To avoid such catastrophic failures the following measures have been taken in this work:

- **Open Source** – Every single part of the presented software which is concerned with stochastics in any way is available as open source software, allowing the community to inspect it and report issues.
- **Unit Tests** – Massive unit test runs have been performed on multiple machines to ensure the correctness of the concerned algorithms. These tests are repeated after every code change. Furthermore the test coverage of all stochastics related classes is kept at 100%.
- **Manual Tests** – Hundreds of generated distributions have been inspected and analyzed manually.
- **Seed Indices** – A special mechanism has been implemented to protect reproducibility from build order issues (see chapter 4.3.3).
- **Multiple Generators** – Four completely independent pseudo random number generators are provided to allow for cross-checks.
- **Pair Programming** – Stochastics related code was written exclusively in pair programming with one programmer being a mathematics expert and the other a programming expert (both with considerable experience in discrete simulation and stochastics).

4.3.1 Random Number Generators

There is a large number of Pseudo Random Number Generators (PRNGs) available for a variety of applications, many of which are well known and tested. This is partly due to their central roles in cryptography and particle physics. It is very important to choose an appropriate, well studied algorithm which is suited for the kind of experiments one has in mind. The unfortunate example of the "RANDU" generator, which was widely used in the early seventies, has demonstrated this – many results from this time are now dubious because of this poor choice [Press 1992b]. [Marsaglia 1987] gives a number of quality criteria for PRNGs, which the author has found to be helpful in these matters.

Both, the .NET Framework and the Java API provide simple PRNGs, which pass basic tests for equidistribution amongst other typical tests for PRNGs (see also [Marsaglia 1996]). Nevertheless their randomness is severely limited. They were mainly selected for their extremely small memory consumption and the simpleness of the involved calculations. Both generators have a relatively short period which is usually not sufficient for simulation [Deng 2000] and suffer from serial correlation between successive values [Gentle 2003].

The Java API makes use of a linear congruential generator (LCG) from [Knuth 1998], which belongs to a well established and tested class of PRNGs. The formula is given below. It must be initialized with a seed value X_0 and calculates further numbers recursively:

$$X_{n+1} = (aX_n + c) \bmod m$$

with $m > 0$, $0 < a < m$, $0 \leq c < m$ and $0 \leq X_0 < m$ the period will be less than or equal to m . The Java API uses $m=2^{48}$, $a=25214903917$ and $c=11$ and returns only the bits 48 to 17 as resulting pseudo random number due to the lower order bits having a much shorter period than the higher order bits if m is a power of 2. In general, the n th least significant digit in the base b representation of the output sequence, where $bk = m$ for some integer k will repeat with a maximal period of bn [Knuth 1998]. Furthermore, if an LCG is used to generate points in an n -dimensional space, the points will

lie on at most m times $1/n$ hyperplanes (Marsaglia's Theorem, [Marsaglia 1968]) due to serial correlation between successive values as mentioned above.

The .NET Framework uses a subtractive congruential generator (SCG) – also from [Knuth 1998] – as given in [Press 1992] under the name "ran3". This class of generators is less studied and established than the LCG. Furthermore it is slightly more complicated to calculate but still very fast. As pointed out in [Press 1992] the limitations which undoubtedly exist are probably somewhat contrary to the LCG's due to the principal differences in the generator formula.

Most software development APIs and frameworks provide similar algorithms. As long as one is aware of the limitations they can be used for many simple scenarios. [Entacher 1998] shows the linear structures of many well known generators and should be consulted in order to understand the limitations of a given PRNG.

Other RNGs (especially the ones designed for cryptography) are usually better in their randomness and have a huge period but their number generation process takes a lot of time and / or memory. The .NET Framework's cryptography namespace provides such a generator. Finally there are some generators which are very well suited for discrete simulation, some of which were specifically designed for Monte Carlo Simulation (see [James 1990] for a review).

One notable example is the algorithm known as GFSR(250,103) or simply R250. It is based on generalized feedback shift registers (GFSRs, see [Lewis 1973]) and was introduced by [Kirkpatrick 1981]. The generation formula is

$$X_i = X_{i-p} \text{ xor } X_{i-q}$$

and it is easy to see that the generator will need p initial values before it can start. These are usually generated by some other PRNG. GFSRs have a maximal period of $2^p - 1$ which is the case for the GFSR(250, 103) and for every choice of parameters where p is a Mersenne Prime. One of the main advantages over LCGs is the fact that the period is not dependent upon the number of bits used in the calculation.

The R250 generator was later combined with the GFSR(521,168) (note that 521 is the 13th Mersenne Prime) by [Heuer 1997] in a way that attains better statistical properties than each single generator alone (see also [Janke 2002]). The algorithm (commonly referred to as "R250/521") performs significantly better than the default generators from the Java API and the .NET Framework except for cases where only very few random numbers are calculated. In these cases a certain overhead due to the initialization puts the R250/521 at a disadvantage.

Another notable example of PRNGs which are well suited for simulation experiments is the popular "Mersenne Twister" [Matsumoto 1998]. It is a twisted GFSR(624,397) of rational normal form (TGFSR(R), see [Matsumoto 1992]) which was specifically designed with a large number of serial random decisions in mind. It has a huge period of $2^{19937} - 1$ and generates high quality pseudorandom numbers. The generation algorithm is slightly more complicated than for the above generators and the algorithm needs initial values but it is faster and needs less space than R250/521.

The software presented with this work provides implementations for these four (Java LCG, .NET SCG, R250/521 and the Mersenne Twister) algorithms in both the Java as well as the .NET version. The LCG from the Java API was re-implemented in C# and the SCG from the .NET Framework was re-implemented in Java. By default, all distributions will use the Mersenne Twister. The intended purpose of the R250/521 (as well as the SCG and LCG) is to provide the means for comparisons and testing. For all other purposes the native generators should be avoided if possible.

All PRNGs must implement the `IRandomSource` interface in the `Mathematics.Stochastics.Interfaces` namespace. This interface prescribes initialization and reset methods, a seed property, a name, a flag for antithetic mode and a flag indicating if the instance was already initialized. Furthermore it prescribes the methods "NextInteger" and "NextDouble". The "NextInteger" method is meant to generate a number on the interval $[0, \text{int.MaxValue})$ and the "NextDouble" method to generate a

number on the interval $[0, 1)$.

The PRNG to be used in simulation or optimization experiments is not configured centrally through the model class. Instead each distribution can be configured to use a given `IRandomSource` separately. This allows the user to mix PRNGs according to different requirements on performance and randomness within one model.

To provide uniform access to different PRNGs and different distributions (see chapter 4.3.2) a wrapper ("`Random<T>`") has been provided in the `Mathematics` namespace. The type parameter dictates the type of random number (or object) to be generated by the "Next" method and restricts the type of distribution which can be used in the constructor call. Usually `Random<int>` and `Random<double>` will be used but it is possible to implement distributions (and therefore `Random<T>` instances) of any type. In addition to the exposed `IRandomSource` (see above) and `IDistribution` (see below) members like the `Reset` and `Initialize` methods the wrapper has a "NonStochasticMode" property. This can be centrally configured through the `Model` class for all registered distributions. Usually distributions will just return their mean value in non stochastic mode. This will speed up execution and help to analyze stochastic effects.

Usually the modeller will not directly work with the `Random<T>` wrapper from the `Mathematics` library though. The `DiscreteSimulation.Engine` namespace has its own `Random<T>` class which is inherited from the one in the `Mathematics` namespace and implements the `IRandom` interface (see chapter 10.4.6, figure 26). (From here on `Random<T>` will always refer to the latter and not to the wrapper in the `Mathematics` namespace.) Its constructor must be provided a `ISeedSource` instance and the distribution must not be initialized with a seed upon creating the wrapper. This is meant to disable the manual setting of seeds by the user to enforce reproducibility of experiments: The wrapper will retrieve seeds from the seed source to initialize the distribution in a reproducible way.

Any class or simulation entity can be used as seed source if it implements the `ISeedSource` interface. This interface prescribes a seed, a uniform distribution as seed generator and a list to keep track of `IRandom` instances which were initialized by the seed source. Additionally it prescribes a method to add `IRandom` instances and a method to get a random seed for a given seed identifier (see chapter 4.3.3) Upon instantiating a `Random<T>` object with a distribution the wrapper will retrieve a seed from the seed source and register itself with the seed source using the `AddRandomGenerator` so that the `Model` class can reset, initialize or manipulate all distributions, their seeds and their configuration (antithetic and non-stochastic mode). For the model to be able to do this, all stochastic simulation entities (simulation entities which generate their own random numbers and therefore implement `ISeedSource`, see chapter 4.5.3) must be registered with the model. But as mentioned in chapter 4.1.2 the model itself is a seed source and therefore in simple scenarios `Random<T>` objects may be instantiated using the model instance. Additional seed sources are useful when the workload for the model class must be minimized as for example in distributed models. However, it is not discouraged to use entities as individual seed sources to improve readability of the code.

4.3.2 Distributions

Distributions as provided in the `Mathematics.Stochastics.Distributions` namespace are regulated by the `IDistribution` interface (see chapter 10.4.6 figure 27). It prescribes a "Configured" and an "Initialized" flag among other things. The semantic discrimination between a configuration and an initialization is very important for the use in the `DiscreteSimulation` library. For a distribution to be "Configured" means that its specific parameters have been set (like μ and s for a gaussian distribution). To be "Initialized" means that the distribution can be used together with an "IRandom" instance to generate random numbers, which requires a seed value. The `Random<T>` wrapper (see above) from the `DiscreteSimulation` library throws an `ArgumentException` if it is provided with a distribution which is either not configured or which is initialized already. This is done on purpose to prevent the modeller from setting individual seeds instead of configuring a central seed for the model and letting the model provide seeds to individual `Random<T>` instances.

It should also be noted that there is a reason for the seed being stored at the IDistribution instance instead of the IRandom instance: A distribution can theoretically be used by multiple IRandom instances which retrieve random numbers in turns. The random instance is just a mechanism to wrap a distribution object of a certain type and provide the experienced Java or C# programmer with the already familiar semantics (with the only difference being the generic parameter to avoid confusion).

The following list gives an overview over the currently available random distributions available for simulation experiments:

- Bernoulli
- Beta
- Binominal
- Erlang
- Gamma
- Gaussian
- Geometric
- Histogram
- Logistic
- Log Logistic
- Log Normal
- Negative Binominal
- Negative Exponential
- Pearson T5
- Poisson
- Triangular
- Uniform
- Weibull

Additionally, there is a Constant Distribution available to provide one single value repeatedly, for compatibility reasons. [Troschütz 2007] provides further distributions and additional PRNGs for C# 2.0 which could be adapted for use with the presented software.

The histogram distributions are parametrized with minimum and maximum and a list of frequencies. The distribution will return numbers on the given interval with respect to the frequencies for each section of the interval. For the integer histogram distribution the number of frequencies must be equal to Maximum - Minimum. For the double histogram distribution the interval will be divided into sections based on the number of frequencies provided. Per default the double histogram distribution will interpolate values over the sections using a uniform distribution. If interpolation is switched off, only (Maximum - Minimum) / Frequencies.Count distinct values will occur.

4.3.3 Seed IDs and Random Seeds

In most cases simulation models will be built in a stable order. But exceptions are possible, especially when building models dynamically from a database or from other external data sources which are not or not completely under the modellers control. In such cases reproducibility can be broken and the consequences can be fatal in many ways. The simulation library presented in this work contains a mechanism ("Seed Identifiers" or short "SeedIDs") which allows the modeller to make entities identifiable to the seed source so that an object with a given identifier will be provided with the same

seed no matter when the entity is created.

The use of this mechanism is not obligatory though. But it is recommended to use it whenever models are created dynamically or whenever changes in the model building logic or architecture are to be expected even after experiments have started. By using SeedIDs this can be done without invalidating earlier results or making them incomparable to newer results.

This facilitates a two staged approach to very large simulation experiments: It is possible to build a first model version for a single machine and test it in small scenarios and / or short simulation time periods. As soon as the model logic is validated the model may be converted or re-implemented for a distributed simulation. Such a two staged approach is much less error prone than implementing a distributed simulation model from scratch. The big advantage SeedIDs introduce here is that (given the same central seed) the new, distributed model – in spite of being built in a completely different way – will yield exactly the same results as the simple, single machine model. This provides a very comfortable and easy way to validate complex model architectures by using a simpler prototype model.

Another advantage is that experiments on models which were built with an earlier version of the simulation library can be compared with experiments on the same model after being adapted to a future version of the framework even if this somehow changes the order of the model build process.

As discussed above there is one central seed which can be configured via the model class. The model class then acts as a ISeedSource to provide all IRandom instances with seeds upon their registration. The following class, an entity which will repeatedly generate an event after a gaussian distributed time interval, illustrates this process:

```
public class MyEntity : SimpleEntity
{
    private IModel model;
    private Random<double> rnd;
    private BinaryEvent<int> somethingHappened { get; private set; }

    public MyEntity(IModel model) : base(model)
    {
        somethingHappened = new BinaryEvent<int>("MyEntity.SomethingHappened");
        somethingHappened.AddHandler(MyEntity_SomethingHappened);

        rnd = new Random<double>(model,
                                new GaussianDistribution(5, 1),
                                model.Antithetic,
                                model.NonStochasticMode);

        model.AddEvent(rnd.Next(), somethingHappened.GetInstance(this, 0));
    }

    private void MyEntity_SomethingHappened(IEntity sender, int args)
    {
        model.AddEvent(rnd.Next(), somethingHappened.GetInstance(this, args + 1));
    }
}
```

It is obvious that, when multiple such entities retrieve seeds from the model, their random number stream will depend on the order in which this happens. If this is to be avoided, the code should be changed to the following (changes are highlighted in yellow):

```
public class MyEntity : StochasticEntity
{
    private IModel model;
    private MatthiasToolbox.Simulation.Engine.Random<double> rnd;
    private BinaryEvent<int> somethingHappened { get; private set; }
```

```

public MyEntity(IModel model, int id) : base(model, id)
{
    somethingHappened = new BinaryEvent<int>("MyEntity.SomethingHappened");
    somethingHappened.AddHandler(MyEntity_SomethingHappened);

    rnd = new Random<double>(this,
                             new GaussianDistribution(5, 1),
                             model.Antithetic,
                             model.NonStochasticMode);

    model.AddEvent(rnd.Next(), somethingHappened.GetInstance(this, 0));
}

private void MyEntity_SomethingHappened(IEntity sender, int args)
{
    model.AddEvent(rnd.Next(), somethingHappened.GetInstance(this, args + 1));
}
}

```

This code makes use of the seed id mechanism. Provided that the id itself does not change, the entity will now always get the same seed for a given model seed.

4.4 Resource Management

Resource management is regulated by the interfaces `IResource`, `IResourceReservation` and `IResourceManager` (see chapter 10.4.1 figure 6). Each entity, which implements `IResource` can be supplied to any `IResourceManager` instance, so that other entities can book (seize) it. The resources can then be delivered via callback. The `IResource` interface prescribes a reference to the current holder of the resource, a reference to the responsible resource manager object, a flag indicating if the resource is currently free and a method to release the resource after use. Default `IResourceManager` and `IResourceReservation` implementations are available in the Tools namespace and will be described in chapter 4.8.1.

4.5 Templating Concept

The following sections describe the basic templates available in the presented software. These templates are all based on the `IEntity` interface (see figure 25 in chapter 10.4.6) and build upon each other. The exact interdependencies are shown in 10.4.2. The actual templates for simulation models should always be derived from these base classes if possible.

4.5.1 Basic Implementation

A basic `IEntity` implementation can be found in the abstract class "AEntity". This is the recommended base template for all simulation objects. It manages a reference to the corresponding model so that the model cannot be changed accidentally once it has been set. For entities crossing model boundaries in distributed simulation experiments there is an initialization mechanism, which can be used to re-initialize the entity to work with a different model. The removal of the entity from the previous model is not done automatically.

The model in which the entity will be contained must be provided in the constructor call. An identifier and a name are optional. If no identifier is provided, a default identifier including an auto-incrementing number will be created. The entity will add itself to the provided model during instantiation.

The abstract method "Reset" makes sure that every entity or entity template provides a way to set it back to the initial conditions.

4.5.2 Simple Entity

The simple entity class is the most basic implementation of "AEntity". It extends the base class with an optional 2D or 3D position. The current position is accessible using the "Position" property and will be reset to the initial position as provided to the constructor when the model is reset.

This class is meant to provide a very simple, light weight entity template for the quick implementation of basic simulation objects without any functionality or with very limited functionality on their own.

4.5.3 Stochastic Entity

The stochastic entity class is the recommended base entity for all simulation objects in any stochastic simulation. It is also based on "AEntity" and extends the abstract class with positioning as in the simple entity. Additionally it implements the ISeedSource interface to allow instances to be used to initialize a Random<T> object. The stochastic entity and all derived classes provide the seed identifier mechanism as described in chapter 4.3.3. The SeedID property is protected from inadvertent change and has to be provided for the constructor together with the model. There is a parameterless constructor as well, mainly for deserialization purposes. If it is used, the entity has to be initialized manually, in which case the seed id must be set prior to the model, or by using one of the initialization methods.

If the stochastic entity or any derived class is initialized using a SeedID, that seed id is combined with the model seed using exclusive or (XOR) to create a reproducible seed value for the instance.

The MatthiasToolbox.EmergencyDepartment model, which is a solution to the ARGESIM Comparison [Breitenecker 2010] number 6, demonstrates the use of the stochastic entity.

4.5.4 Resource Entity

The resource entity inherits from stochastic entity and implements IResource, additionally. It holds a reference to a resource manager and to its initial and current holders. Furthermore there is a boolean property "Free", which indicates if the resource is currently in use. The "Free" property as well as the "CurrentHolder" property have get and set accessors. The set accessors are meant to be used by the resource manager exclusively. If you manually set the "Free" flag or change the current holder, the simulation may become corrupted. Instead of using the "Free" flag, use the "Release" method to indicate to the resource manager when you do not need the resource object any longer.

4.5.5 State Machine Entity

The state machine entity (figure 29) is based upon the resource entity and adds state machine functionality using a state machine instance from the tools namespace. Besides the StateMachine property, it also provides direct access to the previous and current state using properties. Finally, a simulation event to signal state changes is implemented.

The possible states and transitions can be configured with strings using the constructors or initialization methods, or directly via the state machine property. In theory it is possible to extend or change possible states and transitions dynamically. However, this is not recommended as it may cause a lot of confusion. It is also possible to create an entity which contains multiple state machines. This should also be avoided for the same reason.

If a real world object consists of multiple components which should be represented using state machines, it is better to model the components as distinct entities and combine them within a container entity (if necessary). This facilitates inspection and debugging of the single components.

The MatthiasToolbox.DiningPhilosophers model, which is a solution to the ARGESIM Comparison [Breitenecker 2010] number 10, demonstrates the use of the state machine entity.

4.5.6 Task Machine Entity

The task machine entity is based upon the state machine entity and allows the configuration of a task sequence for the entity. The sequence will be processed using a TaskMachine (see chapter 10.4.6 figure 28) instance from the tools namespace. This can be used to let the simulation object process a sequence of relatively simple steps and is helpful if the same sequence is required to run repeatedly. However, the modeller should be aware of the fact that this may defy the purpose of discrete simulation to some extent if it is overused. The feature is still a subject of discussion among experts.

Notification of finished tasks and sequences are given via simple delegates in C# and via a callback pattern in Java.

4.5.7 Mobile Entity

The mobile entity is the most complex of the base entities. It inherits TaskMachineEntity and provides support for movement including acceleration and deceleration. It can be configured to drive on a network (see chapter 4.8.2 and figure 10) or between absolute positions.

The movement of a mobile entity is not approximated via sampling. Instead the mobile object keeps track of the point in time at which acceleration was started and calculates its current speed and location based on the speed profile only on request. This is a good example of a combination of discrete and continuous concepts. By abstaining from discretizing linear movements or other linear processes a lot of computing resources can be saved, especially if the simulation is executed without animation.

Even finite element models can be incorporated into a discrete model this way. However, for finite element models there will be a bit of an overhead due to the need to re-calculate the whole finite element system whenever an interaction with the discrete part of the model changes circumstances.

The mobile entity has no built-in collision handling. Collisions can either be managed by using paths or connections like resources and implementing a reservation mechanism or by iterating over all mobile units with collision potential whenever a drive task is started or changed. The "ISimulationConnection" interface (see figure 30) is derived from IResource and an implementation can be found in the "Connection" class (see figure 31) for this purpose.

The MatthiasToolbox.NetworkDriver project demonstrates the use of the MobileEntity together with path finding in a simulation network.

4.6 Extensibility and Interfaces

The simulation engine presented in this work is designed for maximum extensibility using interfaces and generics. There is a special collection of interfaces in the "Interfaces" namespace which should be used by template- and simulation object developers, so that their components are mutually compatible and connectable. The interfaces governing resource management have already been mentioned in chapter 4.4. Further interfaces are described in the following sections.

4.6.1 Identification of Entities

The most basic and also most important group of interfaces is shown in chapter 10.4.1 figure 5 and is concerned with the identification of entities. Everything playing a role in a simulation should be identifiable. Throughout the presented simulation engine identifiability was achieved using the non generic "IIdentifiable" interface which is based on string identifiers. String identifiers have the advantage that one is not restricted to numbers and can avoid collisions when using auto incrementing for different types of entities.

However, most classes do not implement IIdentifiable alone but the inherited interface INamedIdentifiable interface instead, so as to be able to additionally provide a human readable name.

4.6.2 Containers and Attachment

The interfaces shown in figure 7 govern the containment of entities within other entities and the attachment of entities to others. Whenever possible it is recommended to use the `IEntityContainer<T>` and the `IAttachableContainer<T>` interfaces above all others. These are the most specific ones and require the generic parameter to implement `IEntity` (for the entity container) or `IAttachable` (for the attachable container). Only if this is not possible should one fall back to the more general interfaces.

The non generic `IEntityContainer` and `IAttachableContainer` interfaces are provided for the convenience of programmers who are not familiar with generics. Finally, the `IItemContainer` interface is provided as a fallback option in case one has to contain an object which does not implement `IEntity` and which is not known at design time.

4.6.3 Dynamic Creation of Entities

Creating entities dynamically is a very common task in discrete simulation and most simulation software therefore provide some kind of source object. The presented framework also has a source entity (see chapter 4.7.1). But additionally it provides interfaces to govern the development of custom source objects (see chapter 10.4.1 figure 8). These interfaces prescribe a method which allows the sources to be dynamically connected to buffer or sink entities as described in the following section.

If possible, preference should be given to the `IEntitySource<T>` and `IEntitySource<TEntity, TData>` interfaces. Similar to the situation in the previous section alternative interfaces have been provided for unknown objects and for the use without generics.

4.6.4 Reception, Storage and Buffering

Reception, storage and buffering of entities are governed by the interface system shown in chapter 10.4.1 figure 9. All sink and buffer interfaces prescribe a put method for other entities to be able to pass on entities. Furthermore a method is prescribed, allowing the sink or buffer to be connected to a source entity as described in the previous section. The elective buffer is an exception to this rule, because it requires an index value to be provided on put and get operations.

In addition to the above mentioned methods, buffer interfaces prescribe a `Preview` method to inspect the next object waiting for retrieval, a `Get` method to allow retrieval of objects, a `Count` property and a `Full` flag.

Similar to the previous two sections there are interfaces for objects unknown at design time and non-generic versions of most of the interfaces. If possible, the interfaces `IEntitySink<T>` and `IEntityBuffer<T>` should be used instead of their alternative versions.

4.6.5 Further Interfaces

There are two additional interfaces in the `Interfaces` namespace: `ILocatable` and `IPriorityContainer`. `ILocatable` only prescribes a `Position` property of the type `Point` as defined in the `Mathematics` library. `IPriorityContainer` prescribes a `Priority` property of the type `Priority` as defined in the `Engine` namespace.

4.7 Simple Templates

Based on the base classes presented in chapter 4.5 and on the interfaces presented in chapter 4.6 the simulation framework provides a number of demonstrative implementations which can be used as templates. These are described in the following sections. However, it is recommended not to derive simulation entities from these templates. It is usually easier and more efficient to implement simulation entities customized for the actual problem, using the provided templates only as a reference. The patterns applied in these template classes are kept as universal and generic as possible, but with the drawback of considerable overhead.

The templates presented in this chapter are used in a classical "Source – Queue – Server – Sink" model (MatthiasToolbox.SQSSModel) which demonstrates some of the features of these entities. The actual simulation class is appended in chapter 10.5.1. The project additionally contains a windows forms GUI so that the log output can be viewed.

4.7.1 Source

The source classes (see chapter 10.4.3 figure 11) are default implementations of the `IItemSource` interfaces `IItemSource<TEntity>` and `IItemSource<TEntity, TData>`. They inherit from state machine entity (chapter 4.5.5) and can thereby be used as state machines. The actual generation of entities can be left to the source itself, which will use default constructors. Alternatively a generator function can be provided to the source for invocation as required.

The source classes are designed to deliver additional data with the created entities via a ternary event "EntityWithDataCreatedEvent". But they can also be used without this feature. In this case the binary event "EntityCreatedEvent" may be used.

4.7.2 Buffer

The buffer template classes (figure 12) are implementations of the interfaces `IItemBuffer<T>`, `IElectiveBuffer<T, int>` and `IElectiveBuffer<T, string>` and inherit from `StateMachineEntity`. They provide a small number of predefined queueing rules for retrieval of objects and can be parametrized with a custom item selector. They also implement `IEnumerable` and can be parametrized for random access.

A buffer instance can be limited to a maximum capacity and provides a `BufferFullEvent`. Finally a buffer can be connected to one or multiple item sources (`IItemSource<T>`) for input.

4.7.3 Delay

The delay template classes (figure 13) are derived from `StateMachineEntity` and implement `IItemSource<T, double>` and `IItemSink<T>`. The delay can be connected to one or multiple item sources for input.

The delay has a distribution for the delay time. Whenever an item is put into the delay it will be held up for a certain time (calculated using the delay distribution). If an `IItemSink<T>` is connected, the item will be passed on to it after the delay time.

4.7.4 Server

The server template (figure 14) is the most complex of the provided templates. It is a `StateMachineEntity` and unites the features of an `IItemSource` and an `IItemSink`. Additionally it unites a pull and a push model. If it is connected to a buffer and the flag `PushAllowed` is set to true, it will retrieve work items automatically.

The server can be configured to fail depending on a mean time to failure (MTTF) distribution and a mean time to recovery (MTTR) distribution. The time it takes to create a product from the given material can also be configured using a random distribution.

Depending on configuration the server may need different kinds and numbers of material to create a product. Check delegates may be provided to externalize the procedure. The actual creation of a product from the given material can also be externalized.

4.7.5 Sink

The sink classes (figure 15) are derived from `SimpleObject` and implement `IItemSink<T>`. They count the number of received items and can be connected to an `IItemSource<T>`. Furthermore they can be configured to log the reception of an item.

4.8 Further Utilities

There are a number of additional utilities which support the development of complex simulation models. These will be described in the following sections. Please note however, that some of these support systems are not as well documented and evaluated as the engine itself. In case of doubt it is recommended to write custom implementations using only the basic ideas from these systems.

4.8.1 Resource Manager

One of the most complicated classes of the simulation framework is the resource manager. It is based on the `IResourceManager` interface (see figure 6) so that custom implementations can be used.

The default `ResourceManager` implementation (see figure 32) in the `Tools` namespace will be used by the model if no other resource manager is provided. The class requires initialization before use via the model's `ResourceManager` property. This initialization has to be done after all initial resources have been added.

To retrieve a resource, there are a number of "Seize" methods which will trigger a resource reservation. A resource reservation is a list of required types of resources with the required number. Additional requirements to the requested resources can be realized by using the "checkMethod" delegates.

Reservations will usually be handled in order of priority or in the order in which the reservations are placed, if no priority is given. Additionally, there is a special property "AllowStealing" which provides a way to override the prioritization of resource reservations. If `AllowStealing` is set to true, the "Steal" method will provide access to resources which are already reserved for some other entity.

Finally, it is important to note that the resource manager must be notified of manual changes to availability of resources or the status of reservations using the "Update" method. However, it is not recommended to manually interfere with the resource management process. If a different functionality is required, it is better to implement a new or derived resource manager.

4.8.2 Path Network

The "Network" namespace also provides a relatively simple network infrastructure (figures 30 and 31). It can be used for networks with directed and undirected connections and supports weighted connections as well as weighted nodes. The basic `INetwork` interface is extended with a dictionary of fixed paths which can be used to avoid expensive path finding calculations during a simulation. Nodes and connections can be added and removed via methods. "INode", "IConnection" and "IPath" are also extended to support the use with "MobileEntity" (chapter 4.5.7).

The "Path" class manages routes via multiple nodes. Path discovery is provided by an implementation of Dijkstra's Algorithm [Dijkstra 1959] with caching switched on by default. Other algorithms can be used via a simple search function or by using the `IPathFinder` interface. For large networks or optimization algorithms it is recommended to calculate all or at least the most frequently used paths prior to simulation for performance reasons.

The `MatthiasToolbox.NetworkDriver` project demonstrates the use of the `MobileEntity` together with path finding in a simulation network.

4.8.3 State Machine

The state machine class (see figure 33) is a specialized implementation of the interfaces in the "Basics" namespace for the use with simulation entities (see also `StateMachineEntity`, chapter 4.5.5). It provides simulation events and methods to schedule transitions or switch states immediately. Furthermore it can be reset by the owner entity.

4.8.4 Task Machine

The Task Machine system (see figure 28) is mainly intended for the use with the MobileEntity (see chapter 4.5.7) and should be considered experimental due to the unclear scientific status of the combination of task management and discrete event simulation. It provides simulation events to allow the handling of finished tasks or task sequences.

4.8.5 Simulation Logger

The simulation logger is an extension for the Logging namespace, providing an easy way to handle the simulation logging separated from any other logging. In contrast to the default logging, the simulation logger will only react to the logger classes `SIM_INFO`, `SIM_WARNING`, `SIM_ERROR`, `SIM_FATAL` and `EVENT` and will provide the simulation time instead of the system time to `ILogger` implementations.

5. Optimizer

A simulation model provides us with the means to perform experiments which we cannot or do not want to perform in the real world. Especially in operations research, these simulations will usually run orders of magnitude faster than their real counterparts. This makes it possible to use (parametric) optimization techniques on these models (see also [Gosavi 2003]).

While most commercial discrete simulation software provides the means to use selected optimization methods on the models and some of them provide interfaces to optimization software like OptQuest [OptTek 2010], complete freedom in the selection of an optimization method is usually denied. On the other hand, discrete simulation libraries like the one presented in this work are usually focused to the scope of simulation and do not provide direct support for optimization. Instead they usually provide common interfaces so that the choice of simulation software is not restricted.

In general, optimization software usually provides techniques like linear, quadratic and mixed integer programming, constraint satisfaction and some others, which require the optimization problem to be representable in a specific way. Well known examples include IBM's ILOG [IBM 2010], MOSEK [Andersen 2009], LINDO [LINDO 2010] and FICO [FICO 2010]. To optimize discrete simulation based problems, (meta-) heuristic methods are more practical due to their independence of the exact problem structure. They only require a fitness function which can be used to evaluate a solution candidate and a set of operators to create and manipulate ("tweak") candidates.

There are only a few optimization software packages which are designed to optimize discrete simulation models (like OptQuest and ProModel's simRunner [ProModel 2010]). Universally usable (meta-) heuristic optimization software is rare (See [Neumeier 2010] for a large list of optimization software). A notable exception is the HeuristicLab software [HeuristicLab 2010], [Affenzeller 2009], which allows the problem independent implementation of metaheuristic algorithms. However, to solve a problem, all parameters have to be transformed into the special wrapper types provided by HeuristicLab, so that the operator implementations can be applied.

The optimization library presented in this work aims to provide an even better separation between optimization strategies and the optimization problems. This is achieved by a number of simple interfaces which loosely couple the problem to the solution strategy. These interfaces will be described in the following section. Chapter 5.2 will then give a short overview of the algorithms which have been implemented for demonstration purposes.

5.1 The Framework

The following sections describe the interfaces (see also figure 17) which have to be implemented to solve an optimization problem using the framework or to extend the framework with a new

optimization strategy.

To make an optimization problem available to a strategy, the problem interface and the solution interface have to be implemented in addition to operators which may be required by the strategy.

To implement a strategy, the strategy interface and the configuration interface have to be implemented. Interfaces for the required operators have to be derived from the operator interface.

5.1.1 Problem Interface

To solve an optimization problem, a class has to be provided to the strategy, which implements "IProblem". This interface prescribes an "Evaluate" method which the strategy will use to determine the fitness of a given solution candidate. The function's return value indicates if the solution candidate is valid.

An additional "IsValid" method is prescribed, so that the strategy can determine if a candidate is valid without the need to evaluate the candidates fitness. In some cases this may be possible and thereby save processing resources.

A property "OptimumFitness" is prescribed, which should return the global optimum, if it is known for the given problem. Depending on the strategy this may be required.

Finally there is a "GenerateCandidates" method, which most optimization strategies will need to create a number of initial candidates. This method takes a seed value as parameter for cases in which initial candidates are generated randomly, so that the optimization experiment is reproducible. The second parameter indicates the number of candidates to generate.

Any problem which allows at least the implementation of the GenerateCandidates method and the Evaluate method is compatible with the framework and can be solved given a viable strategy.

5.1.2 Solution Interface

The solution interface is an extremely simple interface which extends IComparable<ISolution> and ICloneable. This interface has to be implemented by solution candidate objects to be usable with the optimization framework. In addition to the CompareTo and Clone methods, it prescribes a boolean "HasFitness" property and a double precision "Fitness" property. The fitness value will be maximized in this framework. If "HasFitness" is set to false, the fitness value will be assumed missing even if a number is available. This can be used to indicate to the framework that the fitness value of an object may have been changed by an operator and is no longer valid.

5.1.3 Strategy Interface

The strategy interface prescribes a "Name" property, a boolean "IsInitialized" property and a string "ProcessingStatus". Furthermore an event "BestSolutionChanged" is prescribed to notify the controlling application of the current progress.

Methods for initialization and resetting of the strategy instance are also prescribed. If the configuration class for a strategy actually implements the ISolution interface, the "Tune" method can be used to tune the strategy with a selected set of candidates, using any other strategy including the strategy to be tuned itself.

Finally, the "Solve" method is prescribed. It takes a problem instance as parameter and returns a number of solutions when the stopping criteria are met or the "Stop" method is called.

5.1.4 Configuration Interface

The configuration interface extends the ISolution interface with a "Seed" property for reproducibility of optimization experiments and the properties "NumberOfIterations" and "NumberOfEvaluations". These are required for comparability in terms of optimization cycles or target function evaluations

between different optimization strategy algorithms.

The reason for the configuration interface being derived from ISolution is tuning: if a configuration class for a certain strategy implementation actually implements the ISolution interface, a tuning problem can be implemented to be solved with any of the available strategies in the framework.

5.1.5 Operator Interface

The "IOperator" interface is normally used by the configuration classes to provide operators to a strategy, which can be used on the current optimization problem's solution candidates. It prescribes an "Apply" method which takes a number of solution objects as parameters, depending on the cardinality of the operator. The cardinality is prescribed as a property as well as the name of the operator and a seed, for reproducibility of stochastic calculations.

5.1.6 Further Interfaces

Some optimization algorithms do not require a sophisticated set of operators. For these the ITweakable and IParametrizedTweakable interfaces are provided, which allow the implementation of simple manipulation operations directly on the solution candidate class. These interfaces solely prescribe a "Tweak" method.

The evolutionary algorithms and the simulated annealing strategy, which are described in the following chapters, use these interfaces as the means to provide simple operators for the manipulation of parameters.

5.2 Implementations

To demonstrate the flexibility and usability of the framework, a number of different heuristic optimization strategies have been implemented. These are described in the following sections.

5.2.1 Random Strategy

The random strategy (see figure 20) is the simplest of the strategies. It can be used with any IProblem implementation and with any IConfiguration implementation. The strategy simply uses the GenerateSolutionCandidates method to repeatedly create candidates and memorizes the best solution.

This strategy is meant for comparison with other strategies. If a strategy doesn't perform significantly better than the random strategy, it is useless. However, for the random strategy to work in the intended way it is required that the problem's GenerateSolutionCandidates method creates candidates randomly. If this is not the case, an additional problem implementation can be used to provide randomly created candidates to the random strategy.

5.2.2 Simulated Annealing

The optimization framework provides a simulated annealing strategy (see [Kirkpatrick 1983], [Cerny 1985] and [Metropolis 1953]) with a pre-defined operator for simple usage (see figure 21). The configuration requires a brownian operator. If the solution candidates implement ITweakable or IParametrizedTweakable<double>, the SimpleBrownianOperator can be used, otherwise a problem specific brownian operator has to be implemented using the IBrownianOperator interface.

Furthermore a cooling function has to be implemented to govern the optimization process and an initial temperature is required to begin optimization. By default, the cooling function will decrease the temperature by one in each step. The rest of the implementation is rather straight forward and can be seen in the class diagram (figure 21).

5.2.3 Evolutionary Algorithms

The evolutionary algorithms infrastructure (figure 22) and implementation (figure 23) combine a

number of different evolutionary strategies within one framework. Depending on the configuration, the strategy may act as one of the classic evolutionary algorithm (EA) types like a genetic algorithm (GA) [Holland 1992] or an evolution strategy (ES) [Rechenberg 1973] or any other kind of EA. It may be configured to work in (μ, λ) mode or in $(\mu + \lambda)$ mode. Each variant may occur as generational EA or as steady state EA. Support for elitism and tournament selection is provided as well.

The `EvolutionaryAlgorithmConfiguration` class provides constructors for the most common types of EAs with and without tournament selection. Furthermore, each step of the overall algorithm may be controlled and manipulated using inversion of control. This is achieved by a number of exchangeable delegates, which control the central part of the algorithm:

```
public void ProcessGeneration(List<ISolution> generation)
{
    (...)

    CurrentElite = UpdateElite(generation, CurrentElite);

    List<ISolution> parents = SelectParents(generation);
    List<ISolution> children = Breed(parents).ToList();

    (...)

    List<ISolution> mutatedChildren = Mutate(children);
    List<ISolution> oldies = Kill(parents);
    List<ISolution> newGeneration =
        config.SelectNewGeneration.Invoke(mutatedChildren,
                                          oldies,
                                          CurrentElite).ToList();

    CurrentGenerationAverageFitness = Evaluate(newGeneration);

    (...)
}
```

The helper functions `UpdateElite`, `SelectParents`, `Breed`, `Mutate`, `Kill` and `Evaluate` (see chapter 10.5.2) invoke delegates which can be configured freely via the configuration class. When using the above mentioned constructors for creating a configuration, the default implementation (see chapter 10.5.3) of these delegates will be used.

6. Discussion

6.1 Maturity

Even though the C# version of the presented simulation library is already in use in a commercial project at the company AREC Automatisierungstechnik GmbH [AREC 2010], there are still some remaining issues. The test coverage can still be improved, as can the documentation coverage. The performance has also still some room for improvement, even though the presented simulation engine runs orders of magnitude faster and with much less memory consumption than many commercial packages.

Further commercial and scientific projects will certainly uncover remaining errors and additional potential for improvement. The ARGESIM comparisons [Breitenecker 2010] are a good starting point for further testing. However, due to the models and tests which were already implemented, it can be said that the software is ready and fit for commercial and scientific use.

Compared to the simulation framework, the optimization framework is yet less evolved. But commercial use of the optimization library is currently planned (also by AREC GmbH). The upcoming project will certainly help to further develop the optimization package. Due to the optimization framework being much smaller and less complex than the simulation library, it is expected to be ready for commercial and scientific use significantly faster than the latter.

To protect the software from inadvertent introduction of errors, a series of unit tests has been written. At the time of writing this work, these unit tests cover only a fraction of the code. Further tests have to be added, especially prior to non trivial refactoring operations.

6.2 Open Questions

The concepts of "immediate events" (see chapter 4.1.1) and task machines as part of discrete event simulation (see chapter 4.5.6) should be further investigated and discussed by the scientific community. The author's opinion is that both concepts, even though extremely useful in some scenarios, may be misunderstood easily and are therefore a potential source of problems.

Concerning the optimization library, it would be interesting to find out which types of evolutionary algorithms can be synthesized with the central algorithm presented in chapter 5.2.3 and which types cannot.

6.3 Future Tasks

The development of an infrastructure for distributed or parallel discrete event simulation was not in the scope of this work, but the presented engine was developed with such a scenario in mind. A framework for distributed and parallel discrete event simulation based on the presented engine would alleviate stochastic experiments, allow better use of current hardware and make it possible to create very large models.

Another interesting extension point would be the incorporation of continuous simulation concepts like finite element analysis. As demonstrated with the mobile entity (chapter 4.5.7) a combination of discrete and continuous simulation techniques is possible using the presented engine. This would allow the modeling of event based scenarios, where events are dependent on the outcome of a continuously simulated process.

The optimization library presented in this work is actually meant to alleviate hyper-heuristic techniques. However, to actually implement a hyper-heuristics software tool was not within the scope of this work. This should be done so as to test the practicability of the solution. Furthermore, additional strategies (e. g. particle swarm based algorithms) should be implemented and compared in optimization experiments.

7. Acknowledgments

First of all I want to thank my parents and my fiancé Sylvia and my uncle Dr. Bruno J. Gruber for their indefatigable support, their patience and their corrections. Without their help this work would be even harder to read.

I also want to thank Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Felix Breitenecker and Dr. techn. Thomas Löscher for their support and contributions, based on their vast experience in the field of discrete event simulation.

Additional thanks go to my former colleagues and research partners at the PROFACTOR GmbH (<http://www.profactor.at>), the HEAL Group (<http://heal.heuristiclab.com>) and the POM Institute (<http://prolog.univie.ac.at>) who gave me the opportunity to become acquainted with the topics discrete event simulation and (meta-) heuristic optimization.

Finally, I want to thank Andreas Gruber and the AREC GmbH (www.arec.at) for using my framework in a real industrial environment and thereby keeping the software and technology alive.

8. Literature

Affenzeller 2009: Affenzeller, Winkler, Wagner, Beham, Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications, 2009

Ahuja 2000: Ravindra K. Ahuja, James B. Orlin, Dushyant Sharma, Very large-scale neighborhood search, 2000

Albahari 2000: Ben Albahari, A Comparative Overview of C#, 2000,
http://genamics.com/developer/csharp_comparative.htm

Andersen 2009: Erling D. Andersen, Bo Jensen, Jens Jensen, Rune Sandvik, Ulf Worsøe, MOSEK version 6. MOSEK Technical report: TR-2009-3, 2009

AREC 2010: AREC, AREC Automatisierungstechnik GmbH, , <http://www.arec.at/>

Banks 2010: Jerry Banks, John S. Carson, II, Barry L. Nelson, David M. Nicol , Discrete-Event System Simulation, 2010

Beck 2002: Kent Beck, Test Driven Development: By Example, 2002

Beck 2005: Kent Beck, Cynthia Andres, Extreme Programming Explained: Embrace Change (2nd Edition), 2005

Breitenecker 2010: Felix Breiteneker, ARGESIM, ARGESIM Benchmarks, 2010,
<http://www.argesim.org/index.php?id=68>

Cerny 1985: V. Cerny, A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm., 1985

Deng 2000: Deng L.-Y., D. K. J. Lin, Random number generation for the new century., 2000

Dijkstra 1959: Dijkstra, E. W., A note on two problems in connexion with graphs, 1959

Dorigo 1992: Marco Dorigo, Optimization, Learning and Natural Algorithms, 1992

Eclipse 2010: Eclipse Foundation, Eclipse, 2010, <http://www.eclipse.org/downloads/>

ECMA 2006a: TC39 TG3, Standard ECMA-335, Common Language Infrastructure (CLI), 2006

ECMA 2006b: TC39 TG2, C# Language Specification, 2006

Entacher 1998: Karl Entacher, A collection of selected pseudorandom number generators with linear structures, 1998,

FICO 2010: Fair Isaac Corporation, FICO™ Xpress Optimization Suite 7, 2010,
<http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>

Fisher 1961: H. Fisher, G. L. Thompson, Probabilistic learning combinations of local job-shop

scheduling rules, 1961

Fisher 1963: H. Fisher and G. L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules, 1963

FlexSim 2010: Flexsim Software Products, Inc., Flexsim Simulation Software, 2010, <http://www.flexsim.com/>

Geeknet 2010: Geeknet Inc., SourceForge, 2010, <http://sourceforge.net/>

Gentle 2003: James E. Gentle, Random Number Generation and Monte Carlo Methods, 2003

Glover 1986: Fred Glover, Future Paths for Integer Programming and Links to Artificial Intelligence, 1986

Google 2010: Google, Google code, , <http://code.google.com>

Gosavi 2003: Abhijit Gosavi, Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning (Operations Research Computer Science Interfaces Series), 2003

Gratch 1993: J. Gratch, S. Chien, G. DeJong, Learning search control knowledge for deep space network scheduling, 1993

Hejlsberg 2000: John Osborn, Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg, 2000, http://tim.oreilly.com/pub/a/oreilly/windows/news/hejlsberg_0800.html

Hejlsberg 2004: Bill Venners, Generics in C#, Java, and C++, 2004, <http://www.artima.com/intv/genericsP.html>

Heuer 1997: A. Heuer, B. Dünweg, and A. M. Ferrenberg, Considerations on correlations in shift-register pseudorandom number generators and their removal, 1997

HeuristicLab 2010: HEAL, HeuristicLab 3, 2010, <http://heal.heuristiclab.com/>

Holland 1992: John H. Holland, Adaptation in Natural and Artificial Systems, 1992

IBM 2010: International Business Machines Corporation, ILOG Optimization, 2010, <http://www-01.ibm.com/software/websphere/products/optimization/>

IC#Code 2010: Opensource Community, SharpDevelop, 2010, <http://sharpdevelop.net/OpenSource/SD/Default.aspx>

ISO 19501: International Organization for Standardization, ISO/IEC 19501:2005, 2004

ISO 19505: International Organization for Standardization, ISO/IEC FCD 19505-2, 2009

ISO 25000: International Organization for Standardization, ISO/IEC 25000:2005, 2005

ISO 9126: International Organization for Standardization, ISO/IEC 9126-1:2001, 2001

James 1990: F. James, A review of pseudorandom number generators, 1990

Janke 2002: Wolfhard Janke, Pseudo Random Numbers:Generation and Quality Checks, 2002

JUnit 2010: JUnit.org, JUnit 4.8.2, 2010, <http://www.junit.org/>

Kelton 2010: W. David Kelton, Randall P. Sadowski, Nancy B. Swets, Simulation with Arena, 2010

Kenai 2010: Oracle Corporation, Project Kenai, 2010, <http://kenai.com>

Kirkpatrick 1981: S. Kirkpatrick, E. Stoll, A Very Fast Shift-Register Sequence Random Number Generator, 1981

Kirkpatrick 1983: Kirkpatrick, S., C. D. Gelatt, M. P. Vecchi, Optimization by Simulated Annealing, 1983

Knuth 1998: Donald E. Knuth, Art of Computer Programming, 1998

Law 2000: Averill M. Law, W. David Kelton, Simulation Modeling and Analysis, 2000

Lewis 1973: T. G. Lewis, W. H. Payne, Generalized feedback shift register pseudorandom number algorithm, 1973

Lindholm 1999: Tim Lindholm, Frank Yellin, The Java(TM) Virtual Machine Specification, 1999

LINDO 2010: LINDO Systems Inc., LINDO Systems - Optimization Software, 2010, <http://www.lindo.com>

Löscher 2009: Thomas Löscher, Optimisation of Scheduling Problems Based on Timed Petri Nets, 2009

Luke 2009: Sean Luke, Essentials of Metaheuristics, 2009

Marsaglia 1968: George Marsaglia, Random Numbers fall mainly in the planes, 1968

Marsaglia 1987: G. Marsaglia, A. Zaman, W.-W. Tsang, Toward a Universal Random Number Generator, 1987/1990

Marsaglia 1996: George Marsaglia, DIEHARD: A battery of tests of randomness., 1996

Matsumoto 1992: Makoto Matsumoto, Yoshiharu Kurita, Twisted GFSR generators, 1992

Matsumoto 1998: Matsumoto, Makoto, Takuji Nishimura, Mersenne Twister, 1998

Metropolis 1953: N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, Equations of State Calculations by Fast Computing Machines, 1953

Microsoft 2000: Jeffrey Richter, Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web, 2000, <http://msdn.microsoft.com/en-us/magazine/cc748671.aspx>

Microsoft 2010: Microsoft Corporation, Visual Basic Language Concepts: Events in Visual Basic, 2010, [http://msdn.microsoft.com/en-us/library/ms172877\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms172877(VS.80).aspx)

Neumeier 2010: Arnold Neumeier, Global Optimization Software, 2010,
http://www.mat.univie.ac.at/~neum/glopt/software_g.html

Nielsen 1993: Jakob Nielsen, Usability Engineering, 1993

Norman 1990: Donald Norman, The Design of Everyday Things., 1990

NUnit 2010: Charlie Poole, Jamie Cansdale, Gary Feldman, NUnit 2.5.5, 2010,
<http://www.nunit.org/>

Obasanjo 2007: Dare Obasanjo, A comparison of Microsoft's C# programming language to Sun Microsystems' Java programming language, 2007,
<http://www.25hoursaday.com/CsharpVsJava.html>

OMG 1997: Object Management Group, Inc, Unified Modeling Language, 1997,
<http://www.uml.org/>

OptTek 2010: OptTek Systems, Inc., The OptQuest Engine Java and .NET - Developer's Guide, 2010, <http://www.opttek.com>

Oriact 2010: Oriact GmbH, Origo, 2010, <http://www.origo.ethz.ch/>

ORMS 2010: Institute for Operations Research and the Management Sciences, Simulation Software Survey, 2010, <http://www.lionhrtpub.com/orms/surveys/Simulation/Simulation.html>

Pacheco 2001: Xavier Pacheco, Steve Teixeira, Delphi 6 Developer's Guide, 2001

Perl Foundation 2000: The Perl Foundation, Artistic License 2.0, 2000

Petri 1966: Carl Adam Petri, Communication with Automata, 1966

Pilato 2009: C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick, Version Control with Subversion, 2009

Press 1992: William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, Numerical Recipes in C: The Art of Scientific Computing, 1992

Press 1992b: Press, William H. et al., Numerical Recipes in Fortran 77: The Art of Scientific Computing, 1992

ProModel 2010: ProModel Corporation, SimRunner, 2010,
<http://www.promodel.com/products/simrunner/>

Pryor 2007: Jonathan Pryor, Comparing Java and C# Generics, 2007,
<http://www.jpri.com/Blog/archive/development/2007/Aug-31.html>

Rausch 2005: Andreas Rausch, V-Modell XT 1.3, 2005, <http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.3/V-Modell-XT-Gesamt.pdf>

Rechenberg 1973: Ingo Rechenberg, Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, 1973

Sandcastle 2010: Microsoft Corporation, Sandcastle - Documentation Compiler for Managed Class Libraries, 2010, <http://sandcastle.codeplex.com/>

Schwaber 2004: Ken Schwaber, Agile Project Management with Scrum, 2004

Shneiderman 1998: Ben Shneiderman, Designing the User Interface: Strategies for Effective Human-Computer-Interaction, 1998

Siemens 2010: Tecnomatix, Tecnomatix Plant Simulation, 2010, http://www.plm.automation.siemens.com/en_us/products/tecnomatix/plant_design/plant_simulation.shtml

SimPlan 2010: SimPlan AG, Enterprise Dynamics, 2010, <http://www.simplan.de/de/termine-medien/artikel/func-startdown/84/>

Skype 2010: Skype Limited, Skype, 2010, <http://www.skype.com>

Sparx 2010: Sparx Systems Pty Ltd., Enterprise Architect, 2010, <http://www.sparxsystems.com/products/ea/index.html>

Speckle 2009a: Markus Speckle, Evaluierung und Analyse von Simulations Engines für Logistik- und Produktionsprozesse mit der "Anwendungsplattform Simulation", 2009

Speckle 2009b: Markus Speckle, Matthias Gruber, Martin Saler, Evaluierung und Analyse integrationsfähiger Simulations Engines für die Entwicklung komplexer und detaillierter Simulationsmodelle, 2009

Subclipse 2010: CollabNet, Subclipse, 2010, <http://subclipse.tigris.org/>

Sun 1996: James Gosling, Henry McGilton, The Java Language Environment, 1996

Sun 2009: Oracle Corporation, Javadoc, 2009, <http://java.sun.com/j2se/javadoc/>

Sun 2010: Oracle Corporation, The Java Technology, 2010, <http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html>

TightVNC 2010: TightVNC Group, TightVNC, 2010, <http://www.tightvnc.com/>

Troelsen 2007: Andrew Troelsen, Pro C# 2008 and the .NET 3.5 Platform, 2007

Troschütz 2007: Stefan Troschütz, .NET random number generators and distributions, 2007, <http://www.codeproject.com/KB/recipes/Random.aspx>

VSExpress 2010: Microsoft, Visual Studio Express, 2010, <http://www.microsoft.com/express/>

Weers 2010: Michael Weers, Web-Usability, 2010, <http://www-cg-hci.informatik.unioldenburg.de/~airweb/Seminarphase/MichaelWeers/html/>

Woodruff 2010: Eric Woodruff, Sandcastle Help File Builder, 2010, <http://shfb.codeplex.com/>

XJTech 2010: XJ Technologies Company, AnyLogic Documentation, 2010,

9. List of Figures

Figure 1: Profiling Results for the SQSS Model.....	50
Figure 2: Profiling Results for the Dining Philosophers Model	51
Figure 3: Profiling Results for the Emergency Department Model	52
Figure 4: Profiling Results for the Network Driver Model.....	53
Figure 5: Interfaces for identifiable and named entities.	65
Figure 6: Interfaces for resource management.....	66
Figure 7: Interfaces for containers and attachment.	66
Figure 8: Interfaces for item sources.....	67
Figure 9: Interfaces for sinks and buffers.	68
Figure 10: The Base Entity Framework	69
Figure 11: Source implementations.	70
Figure 12: Buffer implementations.	71
Figure 13: The delay entity templates.....	72
Figure 14: The server entity templates.....	73
Figure 15: The sink entity templates.....	74
Figure 16: Architecture of the logging framework.	75
Figure 17: Interfaces for Optimization.....	76
Figure 18: The Simulated Annealing Implementation	76
Figure 19: Evolutionary Algorithm Implementation	77
Figure 20: The Random Strategy Implementation.....	78
Figure 21: The Simulated Annealing Implementation	79
Figure 22: The Evolutionary Algorithms Infrastructure	80
Figure 23: The Evolutionary Algorithms Implementation.....	81
Figure 24: MatthiasToolbox.DiscreteSimulation.Engine.IModel interface	82
Figure 25: MatthiasToolbox.DiscreteSimulation.Engine.IEntity interface	82
Figure 26: MatthiasToolbox.DiscreteSimulation.Engine.IRandom interface.....	83
Figure 27: MatthiasToolbox.Mathematics.Stochastics.Interfaces.IDistribution<T> interface	83
Figure 28: The task machine class.	84
Figure 29: The StateMachineEntity base class and its state machine.....	85
Figure 30: Simulation Network Interfaces	86
Figure 31: Simulation Network Infrastructure	87
Figure 32: Resource Manager and Reservation	88
Figure 33: State Machine Implementation.....	89

10. Appendices

10.1 Tests & Benchmarks

Currently the test coverage of the presented software is relatively low. The best test coverage is achieved in the stochastics part of the mathematics package, followed closely by the simulation package. Further tests will be written until a satisfactory coverage is achieved.

Profiling sessions have shown that logging and visualization have a significant part in the simulation's processing resource usage. These features should therefore be turned off for large experiments and

optimization. The following screenshots show part of the results of profiling sessions performed with the implemented demonstration models.

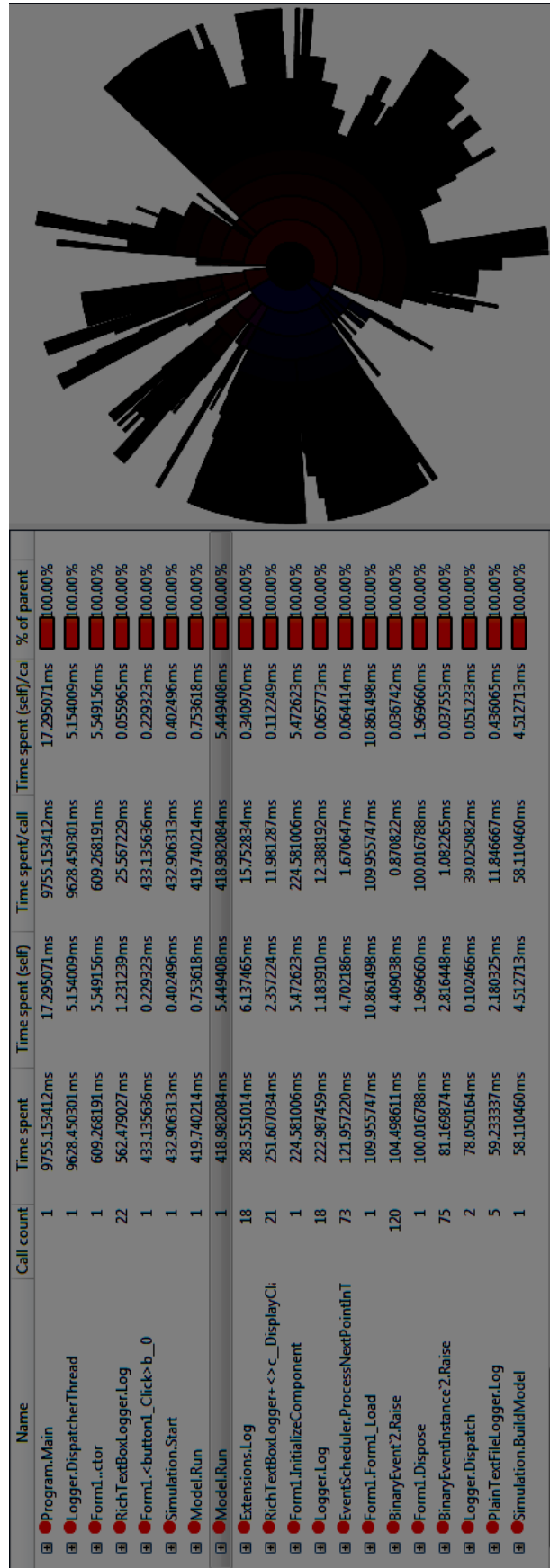


Figure 1: Profiling Results for the SQSS Model

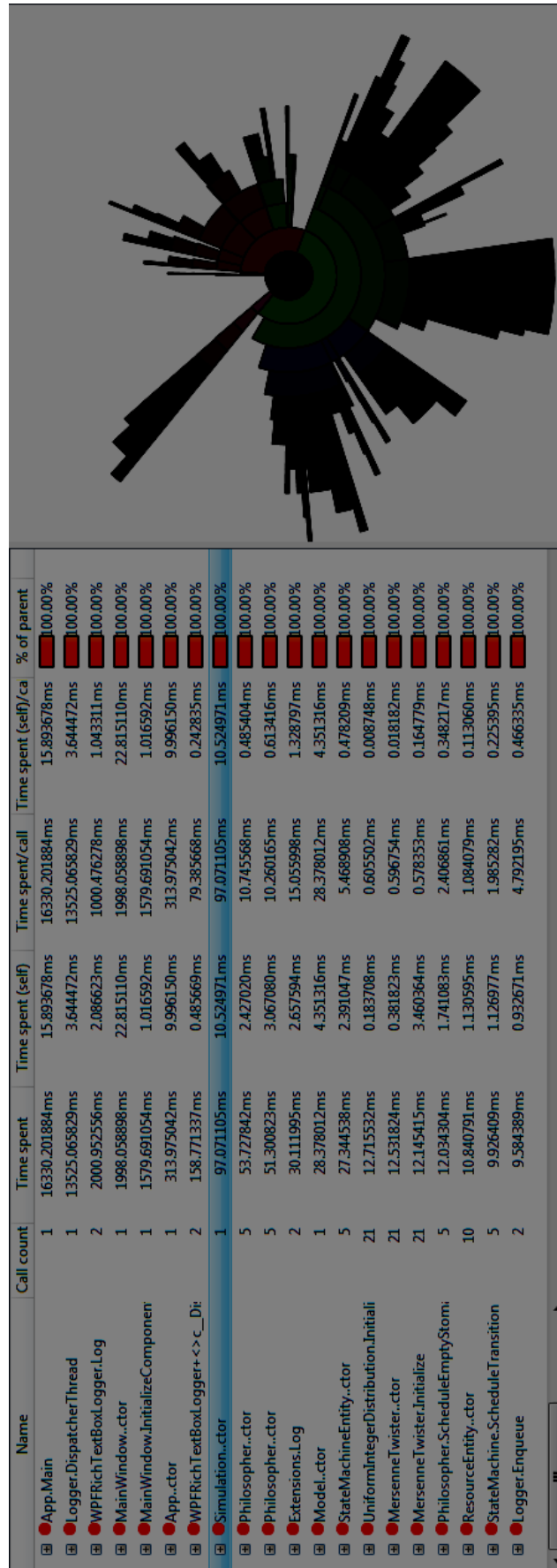


Figure 2: Profiling Results for the Dining Philosophers Model

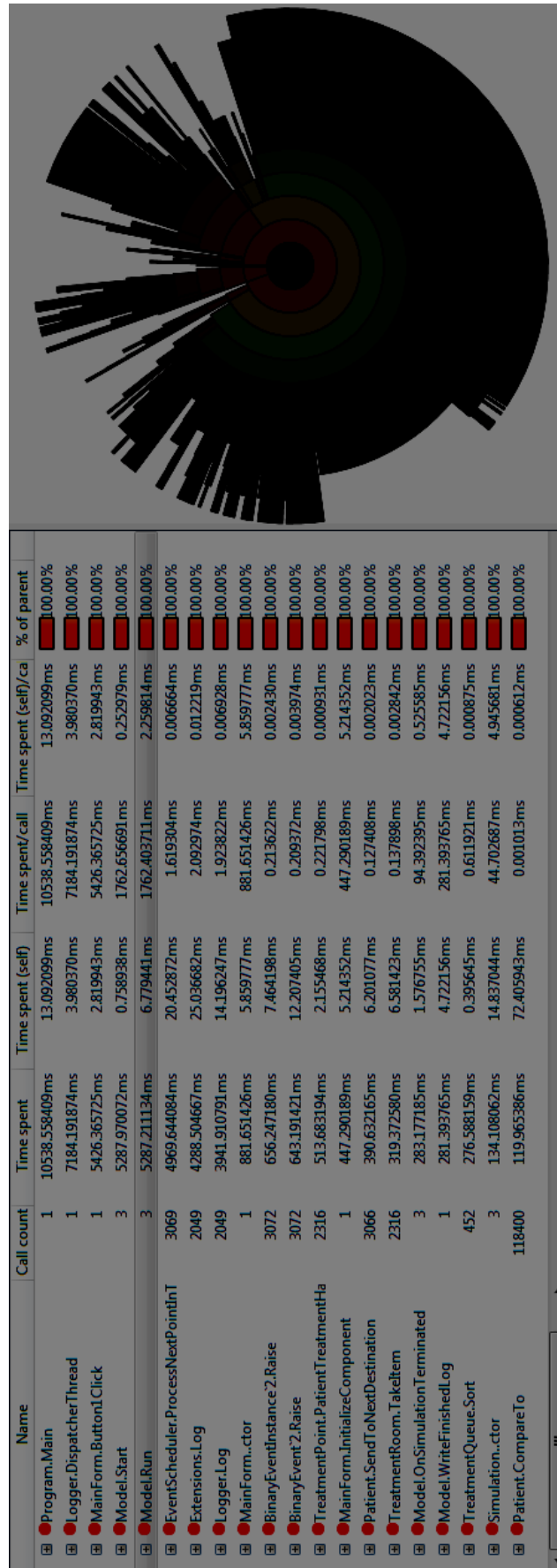


Figure 3: Profiling Results for the Emergency Department Model

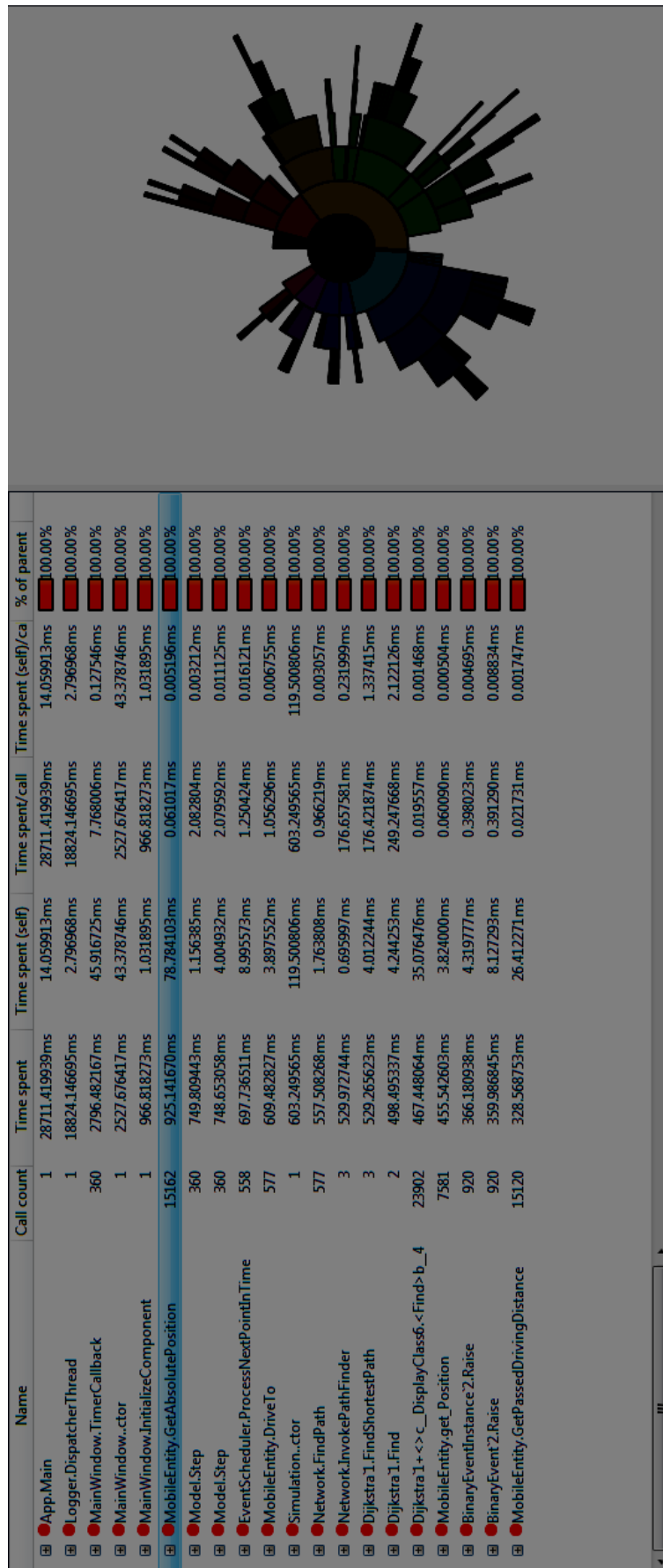


Figure 4: Profiling Results for the Network Driver Model

Direct comparisons with other simulation software is difficult, due to the variation in available templates and other features. Nevertheless, simple source-queue-server-sink models have been built in AnyLogic [XJTech 2010], Arena [Kelton 2010], Enterprise Dynamics [SimPlan 2010] and Plant Simulation [Siemens 2010] to compare speed and memory consumption. Even though the results are only partially comparable, the differences are so great, that some significance can be assumed: On a machine taking a few seconds for the processing of 10.000 items in the above mentioned simulation applications, the presented simulation engine processes 80.000 events per second. And while the above mentioned applications typically consume 50 to 100 megabytes of system memory, the MatthiasToolbox.SQSS application only takes up about 8 megabytes.

However, it is important to note, that the measure "events per time" depends very much on the complexity of the logic associated with the given events, so that different models will produce different event speeds.

10.2 Coding Conventions

General:

Use CamelCase naming for all identifiers if not specified otherwise. Only full words are "camel cased". Abbreviations are written in capital letters. Examples: UserID (not UserId), Database (not DataBase).

Files:

All source code files should be named after the main class / struct / interface or other structure which they contain. Additional extensions like "User.Mapping.cs" and "User.Logic.cs" are used to name files in partial classes. (only C#)

Prefer small source code files with only one main item defined in it. If possible even enumerations are kept in separate files.

Assemblies:

Microsoft assemblies are named after the main namespace (company or solution name) followed by the name of the project to which they belong. This is according to Microsoft .NET Best Practices.

In Java all packages should be URIs starting with a domain name and followed by the project name as above.

All assemblies should be signed with a strong name for commercial releases.

Namespaces:

Namespaces work the same way as above. Always set the root namespace to the company or solution name followed by the name of the project you are working on.

Classes:

Classes, interfaces, structs and other structures should always be named with a meaningful name in order to allow easy understanding of code. Avoid building a collection of "Manager" classes ("UserManager", "RightsManager", "AuthorizationManager") without clearly delimited functionalities (For a method UserManager.CreateAuthorization nobody can decide at first glance what it will really do). This is a relict from functional programming. Instead use classes to model the

real world (like "User" and "Permission") and put methods where they semantically belong. (e. g. for Permission.Grant or User.Create it is very obvious what these methods will do)

In C# use regions to organize your code. On the top level, group code mainly by the type of constructs (e. g. Constructors, Properties, Events, Delegates, Class Variables, Methods) and not by "topics" or access modifiers. In a team agree upon a top level structure and a concept for the subsequent levels (like grouping by access modifiers or topics) and see it through for all files larger than screen size.

If possible, group implementations of specific interfaces or abstract classes into a new subfolder and postfix the name with the name of the interface / abstract class (without the 'I' for interfaces).

Example of an interface "IEvent": All classes that only implement that single interface (so they are primarily there to provide different implementations of a certain pattern) should be put into a subfolder "Events" and be named accordingly (e. g. "UnaryEvent", "BinaryEvent" and so on).

When implementing multiple interfaces, sort the regions by the inheritance level if applicable. Assuming you have an "interface IBasic" and "interface IExtended : IBasic" and a class which implements these: The region "Properties" should contain a region "IBasic" containing implementations of the IBasic properties followed by a region IExtended. The same goes for the "Methods" and "Events" regions.

Always seal classes that are not to be derived from furthermore.

If the class is only a functional module, make it static to disallow instantiation. (not possible in Java)
If there is only a single instance at any time, consider using a Singleton design pattern. But be careful with singleton design patterns: most of them cannot really guarantee that only one instance will be possible (e. g. some do not consider multithreading).

Avoid huge class files (longer than ~1000 lines of code or ~2000 with comments). If grouping into regions becomes difficult consider breaking it up into partial classes.

Interfaces:

Interfaces use the same CamelCase naming convention as all other identifiers prefixed by a capital letter 'I'. If possible, interfaces should be centered on only one concept. (e. g. IComparable, IEquatable and ICloneable, not ICloneableEquatableComparable) If an interface is bigger than one page of text, consider breaking it up in multiple interfaces.

Structs: (only C#)

The same guidelines as for classes apply. Consider implementing a valuetype (struct in C#) whenever a class has a maximum of 32 bytes of data (MSIL code). This can dramatically improve performance, but remember that you are using pass by value with value types and not by references.

Enumerations:

For all enumerated constants define an enum with a meaningful name. Values should also have meaningful names. Never change the order of enumeration members after they have been committed for the first time! If possible avoid enumerations and use static readonly / const instances.

Generics:

Contrary to what Microsoft defines in their best practices documents, use meaningful CamelCase names also for type parameters and not the short "T, K" etc. names that Microsoft recommends, especially if there is more than one type parameter or if it is not completely free and self-explaining.

Avoid deeply nested generics.

Try to stay away from uses of the “Object” base class as much as possible. This is only mandated when dealing with reflection or other low level services. Always try to use Generic classes / methods instead where possible, since they improve type safety thus reducing errors in your programs. Also note that your code will get a lot shorter since you can leave out many casts and checks for the correctness of those casts.

Also, when dealing with value types, performance gains (only C#) in the 1000 % area can be obtained (try using an ArrayList with integers and compare its performance to a List<int>. Since no boxing / unboxing has to occur anymore the performance is much better).

Delegates: (only C#)

Try to define delegate types as public members of classes that use them. Only define them at namespace scope when not possible otherwise (e.g. when being used by an interface). In this case create a code file with only the delegates and name it "Delegates". Do not add namespace-level delegates to an existing code file.

Methods:

CamelCase with meaningful names. Parameter names should also have meaningful names and start with a lower case letter followed by CamelCasing. If you run into a conflict between a parameter name and a field or property name, remember that you can give them both the same name and differ the field name by prefixing it using the keyword "this" (e. g. this.myField = myField;). In static classes where this is not possible use the prefix "local" in case of a name conflict.

For virtual methods do not forget to mention in the comments if overriding methods must or should call the base method and if they do, what will happen. Sometimes it is also important when the base method is called (beginning or end of the overridden method).

Avoid long methods. If a method doesn't fit on your screen anymore, you should break it up into multiple methods.

Constructors:

Always define constructors to initialize a class. They can also be protected if the class is abstract so to only allow derived classes to call it.

Remember that you can use the “this” and “base” keywords to access other constructors. If your class can be derived from, do not forget to mention in the comments if derived classes must call the base constructor for certain initialization steps.

Properties:

For external data access always use properties or indexers since they don't break encapsulation. In Java always use the "get" and "set" prefixes for getter and setter methods. In C# use methods instead of properties if long or complex operations are involved in the "get" or "set" process.

Give properties meaningful names. If the property wraps a local variable, give it the same name as the underlying variable differing by a prefix like my or "_" (underscore). You may also decide differing only by case but this breaks CLS compliancy and makes the property unusable in some other languages.

Events: (only C#)

Events are most often defined as public members of a class. They get normal CamelCase names.

When you define an event you should name it after the logical event that it fires upon (e. g. "Load" or "Click").

To fire the event itself you should wrap it into a method which is protected and virtual with the "On" prefix. This is according to Microsoft Guidelines and helps when you need to reimplement or refactor a system.

Fields:

Never use public fields except in structs. Fields are named starting with a lower case letter followed by CamelCasing.

Operator Overloading: (only C#)

When overloading operators, wrap the methods into a region "oper" or "Operators" at the top level and indicate the use of operator overloading in the class comment.

Reflection:

Reflection is a very powerful tool but it is also relatively slow. So do not use reflection just to find implementations of an internal interface when you could also keep them in a dictionary instead.

Lambda Expressions:

Avoid complicated and large lambda expressions. Remember that a method larger than your screen is considered to be too big. If a lambda expression doesn't fit in one line inside the character width of your screen, consider making it a class method instead.

Optional Parameters: (only C#)

Try to avoid creating methods or constructors with optional parameters if you do not have unambiguous default values. If the default value is not a constant and you set it to null or some constant and let the method replace it with some instance if no other value is provided, document this in the method comments. Example:

```
public SimpleEntity(string name = null)
{
    if(string.IsNullOrEmpty(name)) this.name = (instanceCounter++).ToString();
}
```

For this method you must point out in the comment that an empty name will be replaced by some auto-incrementing number, otherwise the user might try to create an instance with the name "" (empty string) and will not be able to find it again.

New Paradigms and Constructs:

Often when a new language feature is introduced, programmers tend to overuse it. If new features are made available, try to fully understand them before you use them. Always weigh the advantages of the new feature against the readability and understandability of the code. Do not create "job security

antipatterns".

Inheritance:

Never use protected class variables. Allowing this from the language is a flaw that breaks encapsulation. If you need that behavior, define a private field and let derived classes access it using a protected property.

If you provide several interchangeable implementations of an interface, consider introducing an abstract class that implements the basic behavior of the interface while offering sufficient extension points to customize it. This leads to less code duplication and still allows to completely redefine an implementation should you need to do so.

In C# consider grouping together override or shadowed methods (with the override or new keywords) into a common region on a high level to be able to find them quickly. Be extremely careful with shadowing (avoid it if possible) and document such cases thoroughly.

Resource Management:

If your class holds onto unmanaged resources, implement the IDisposable interface on it.

This helps as you can release the resources when needed to conserve resources. Also, consider using the “using” idiom to automatically release resources for classes that implement IDisposable in all circumstances including unhandled exceptions.

For normal (file) resources use a "Resources" directory in the project folder.

General Project Setup:

In the projects code directory create a new repository with the name of the project. The top level directories should be

- ⑩ trunk
- ⑩ branches
- ⑩ tags

for high level branching/merging of versions. In the trunk directory start with the following directories:

- ⑩ data
- ⑩ source
- ⑩ design
- ⑩ binaries
- ⑩ externals
- ⑩ dependencies
- ⑩ documentation

Other big data and image files should not be versioned in a code repository. Only those images and data files which are necessary to build and run the code should be checked in. (Usually in the Resources directory of the respective project)

In Source create the solution file for the project. The individual project / code files should be created in subdirectories of Source, even if there is only one code project in the solution (yet).

Individual project / module assemblies and files should have the project name followed by the module name (or even multiple names). All code files should be placed where their namespace points. (in Java this is the only possibility because the package structure, directory structure and namespace structure are the same) Namespaces always contain the Project name and flow logically.

Place all binaries that get built into the "binaries" top level directory with appropriate Module / Configuration prefixes (e. g. binaries\MyCompany.MyProject.MyBinaryFile\Debug).

Precompiled dependencies live in the Dependencies directory (e. g. 3rd party components). If feasible they can also be imported as SVN vendor branches to achieve a better synchronization with an external project. This should always be done for files that depend on another project in the same company / developer group.

If a dependency is used by more than one project and is a 3rd party dependency, it makes sense to create a separate project for it housing the binaries and / or any changes to the code and its build environment, so that those can be imported as vendor branches into other projects making them appear as internal projects.

For example the DirectX SDK could live in a Project "DirectX" with its binaries and documentation and then be imported into every dependent project's dependency directory as a vendor branch (even though we can't track DirectX itself as a vendor branch). This way company / developer group wide dependency updates can be handled easily.

The documentation folder should contain reference and manual documentation and in commercial projects also use case and requirement analysis documents.

The design folder is meant for UML and other design documents like Enterprise Architect Project Files (use XML storage format for versioning).

The data folder is meant for other files. But remember that big databases and other very large files are not meant for versioning in a code repository and should be left out. In those cases versioning / archiving should be done separately.

Compiling:

For Debug builds always activate Code Analysis, your project should pass that without warnings / errors. Warnings are not there to be switched off. If you ever need to use code which causes a warning, disable the warning only locally and document the case so that others can understand what you did and why you did it. No warnings or errors must exist at build time. Plan your implementation and refactoring operations in small steps so that you can always check in a version that builds at the end of the day.

Internal Deployment:

If possible use a build server. Otherwise follow the steps below:

Provide a script in the top level directory that

- ⑩ Tags the code with the current revision
- ⑩ Builds the project
- ⑩ Builds the documentation
- ⑩ Builds the data files
- ⑩ Copies all output (data, documentation, binaries and other required files) into a specified directory making the application runnable from there alone (XCOPY deployment)
- ⑩ Optional: that builds an installer for the project

⑩ Optional: that releases the project

The script syntax should look like this: "Build d:\target Release Win32". Always make sure your project(s) build(s) cleanly and flawlessly with that setup.

For .NET one can leverage MSBUILD/XNA for those build scripts. Since Visual Studio Solutions and Projects are MSBUILD scripts themselves, one can easily include them.

Documentation:

Generate an XML Documentation file (JavaDoc / NDoc) for every project. Also, comment on private and internal items, even though it will not be used in the reference documentation. Do not write trivial comments like "Gets / Sets the SomeInternalField value" for a property. If the property really doesn't do anything else except for wrapping the field, at least describe the purpose of the field. On empty constructors do not only write "empty constructor" but provide information about how the class is initialized if necessary. Otherwise write "no further initialization necessary" or something along those lines.

Testing and Profiling:

Always profile your code prior to a release. Use unit tests to cover your code as best as possible and run them every time you want to commit your changes, otherwise you might discover errors only after others have built on them.

10.3 License

The software on which this work is based is licensed under the Artistic License 2.0 [Perl Foundation 2000]. Please consider that parts of the documentation may also fall under this license.

Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed. The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the

entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose

a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty: THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10.4 UML Class Documentations

This section contains UML diagrams of the most important parts of the simulation and optimization frameworks. A number of single class diagrams are contained in the last subsection.

10.4.1 Interfaces

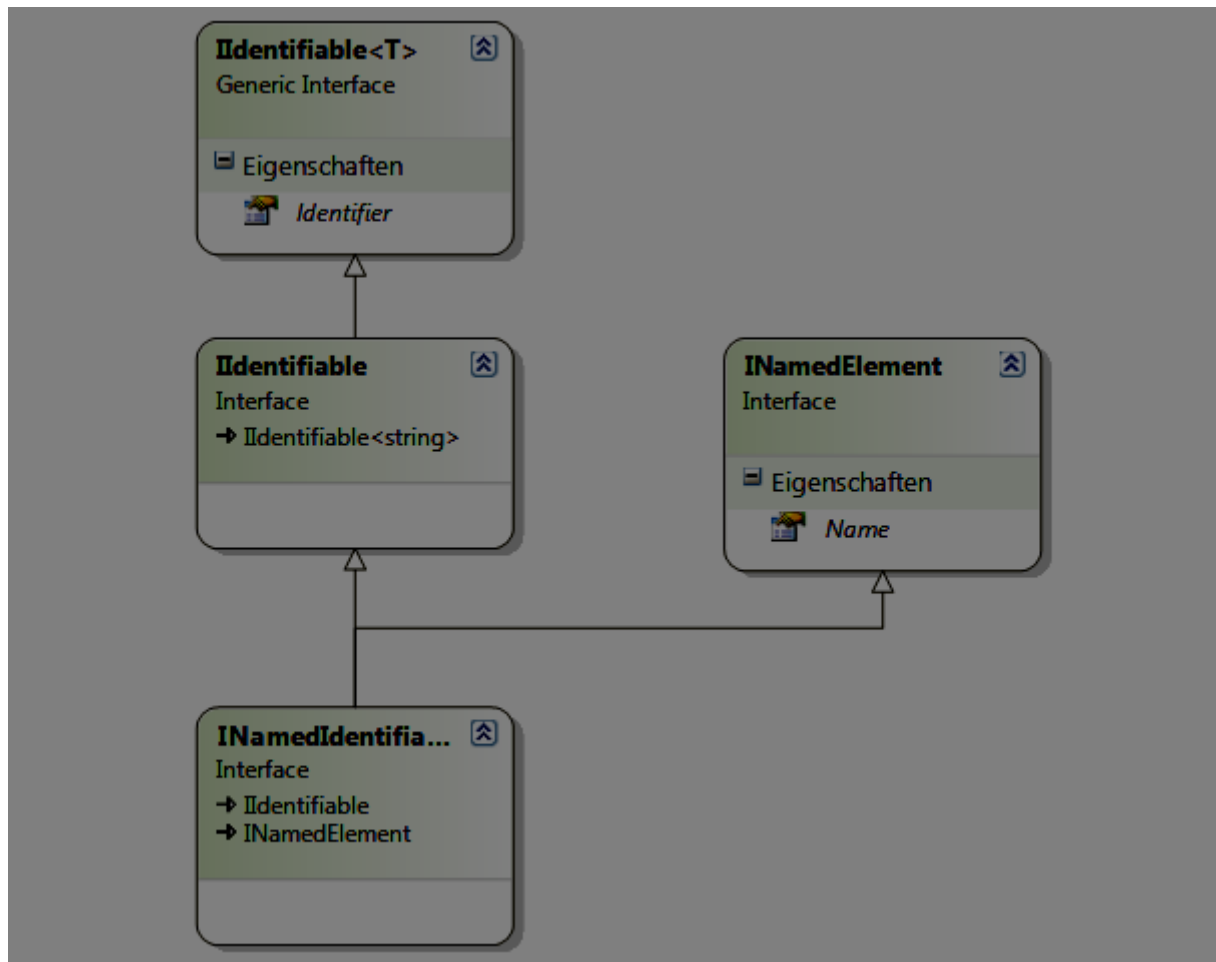


Figure 5: Interfaces for identifiable and named entities.



Figure 6: Interfaces for resource management.

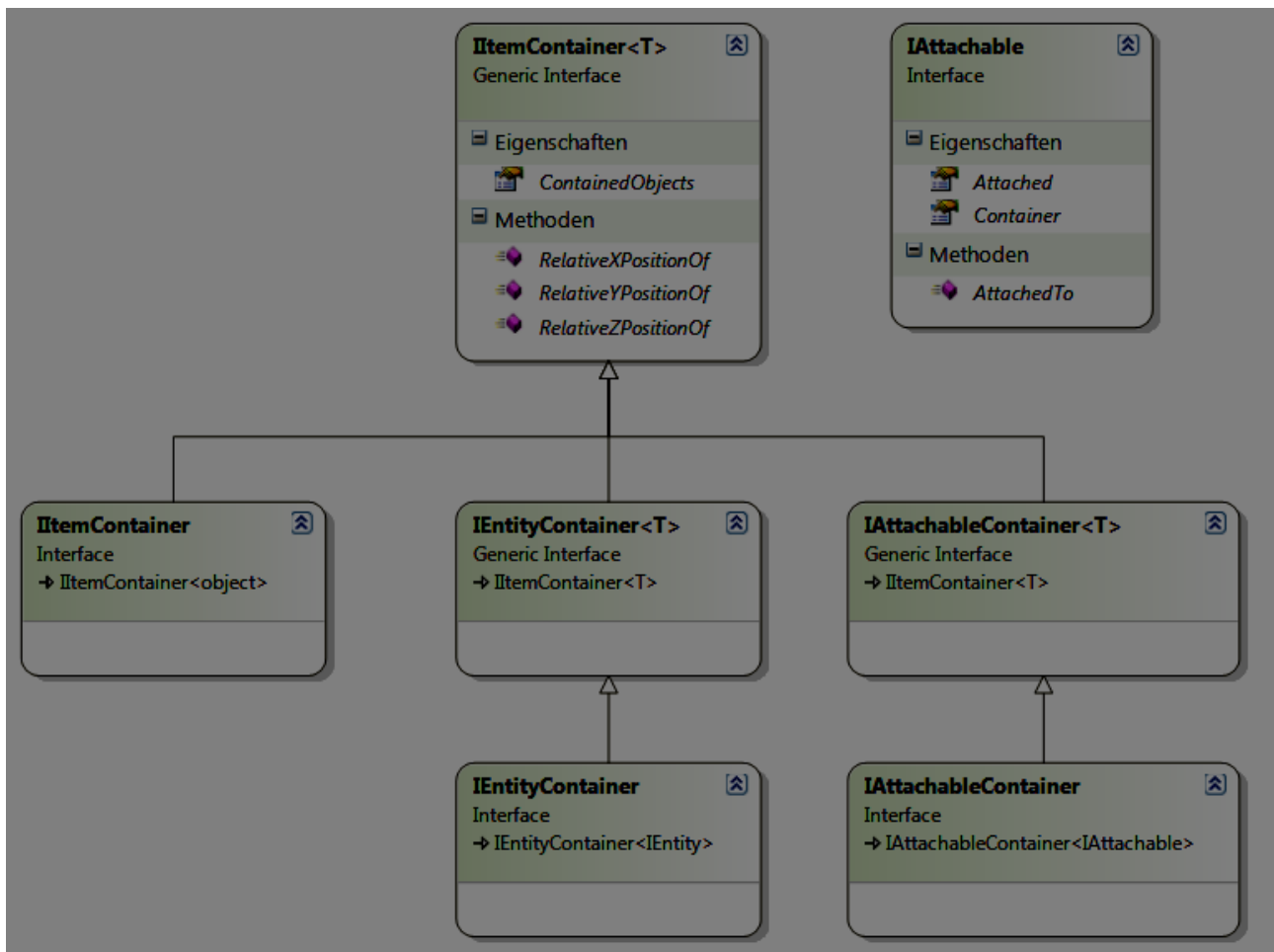


Figure 7: Interfaces for containers and attachment.

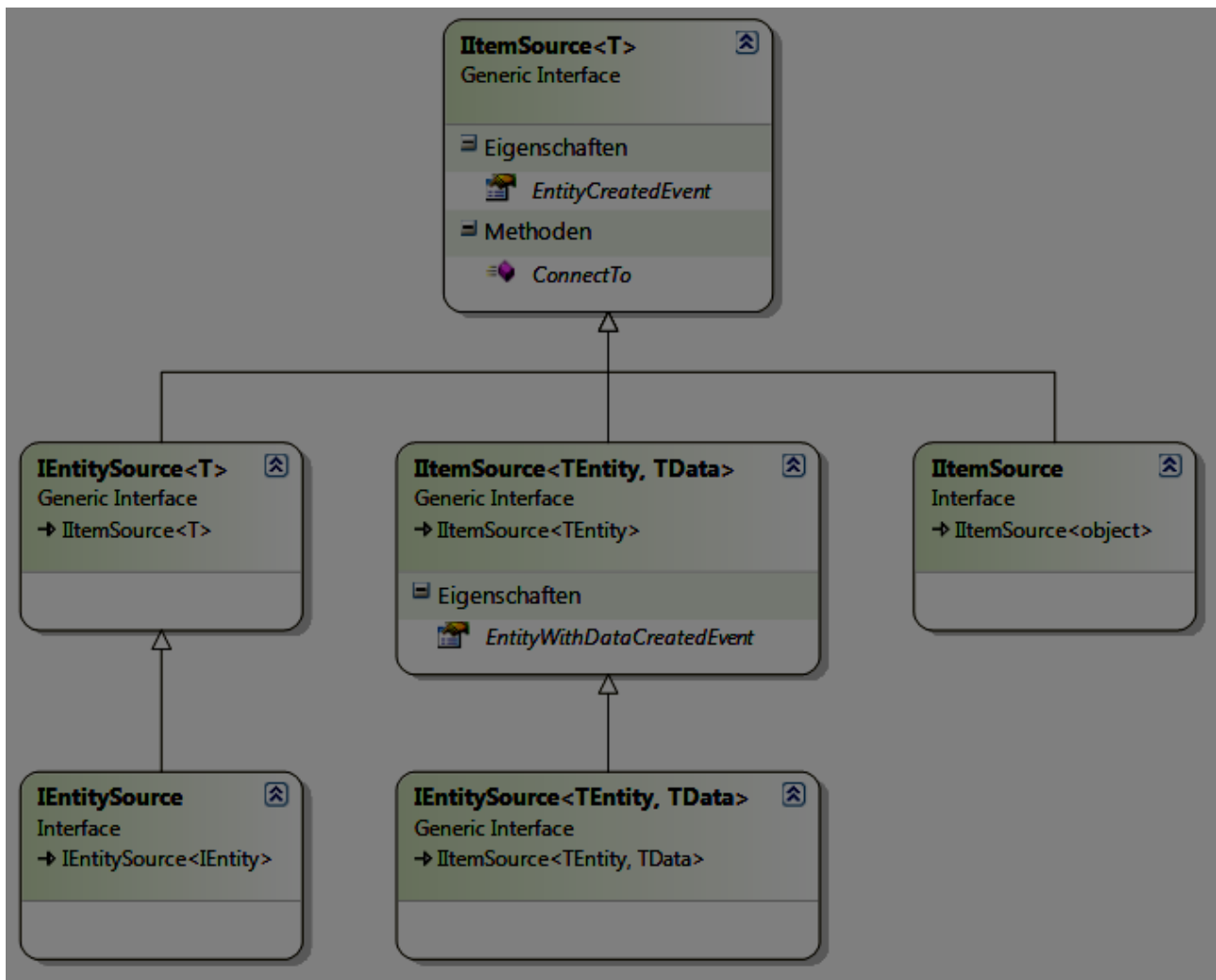


Figure 8: Interfaces for item sources.

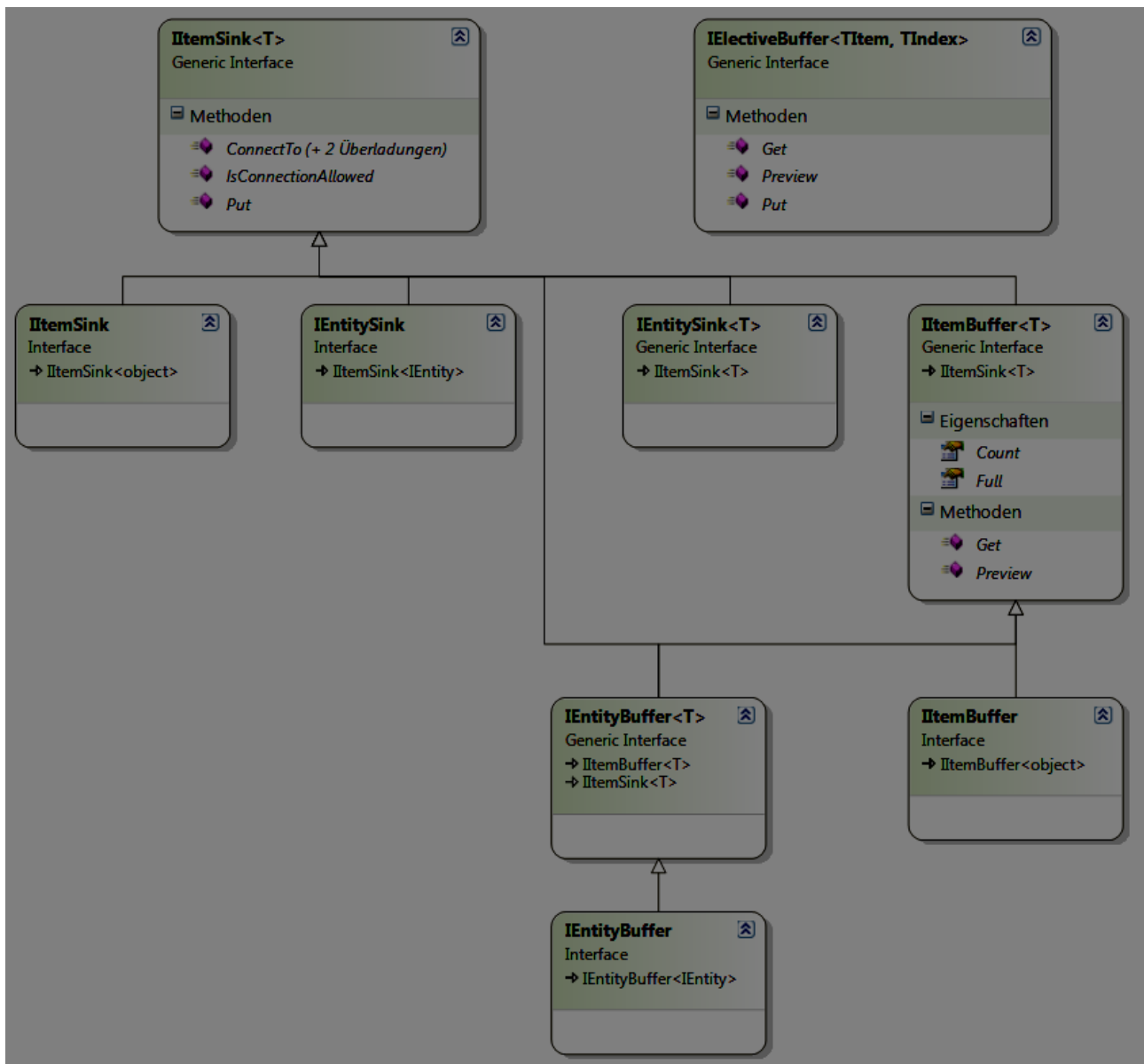


Figure 9: Interfaces for sinks and buffers.

10.4.2 Base Entity Framework



10.4.3 Entity Templates

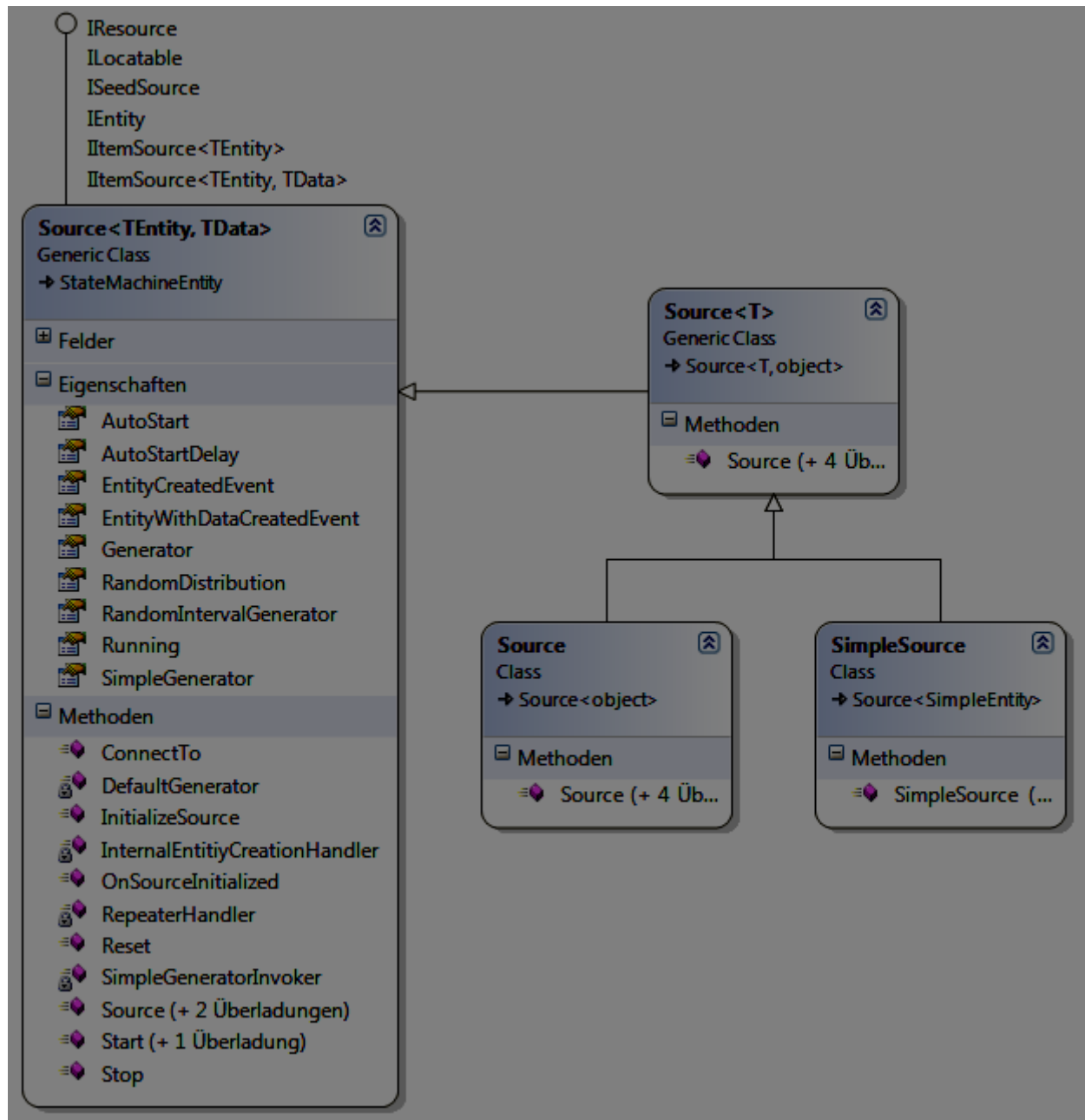


Figure 11: Source implementations.

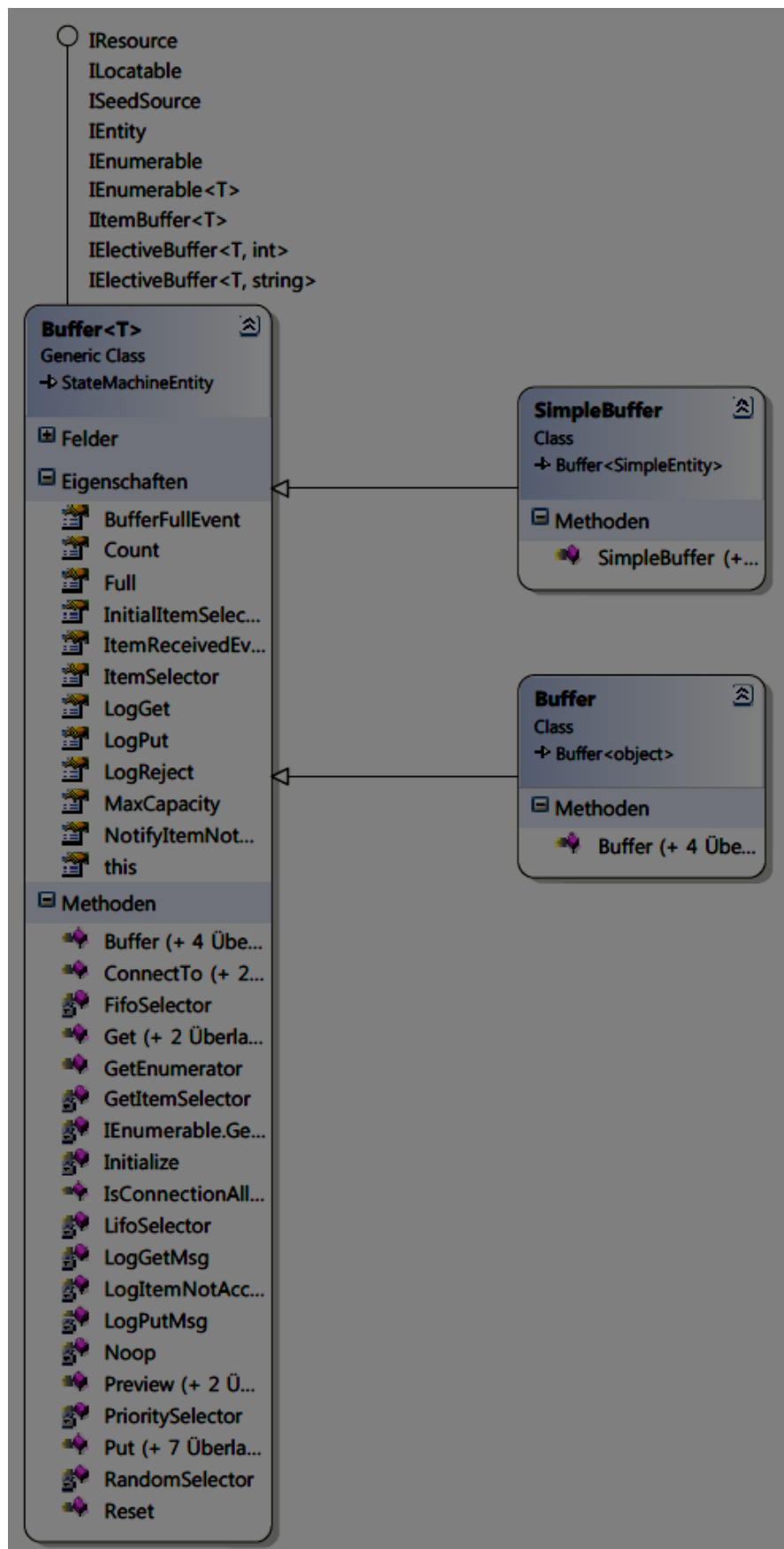


Figure 12: Buffer implementations.

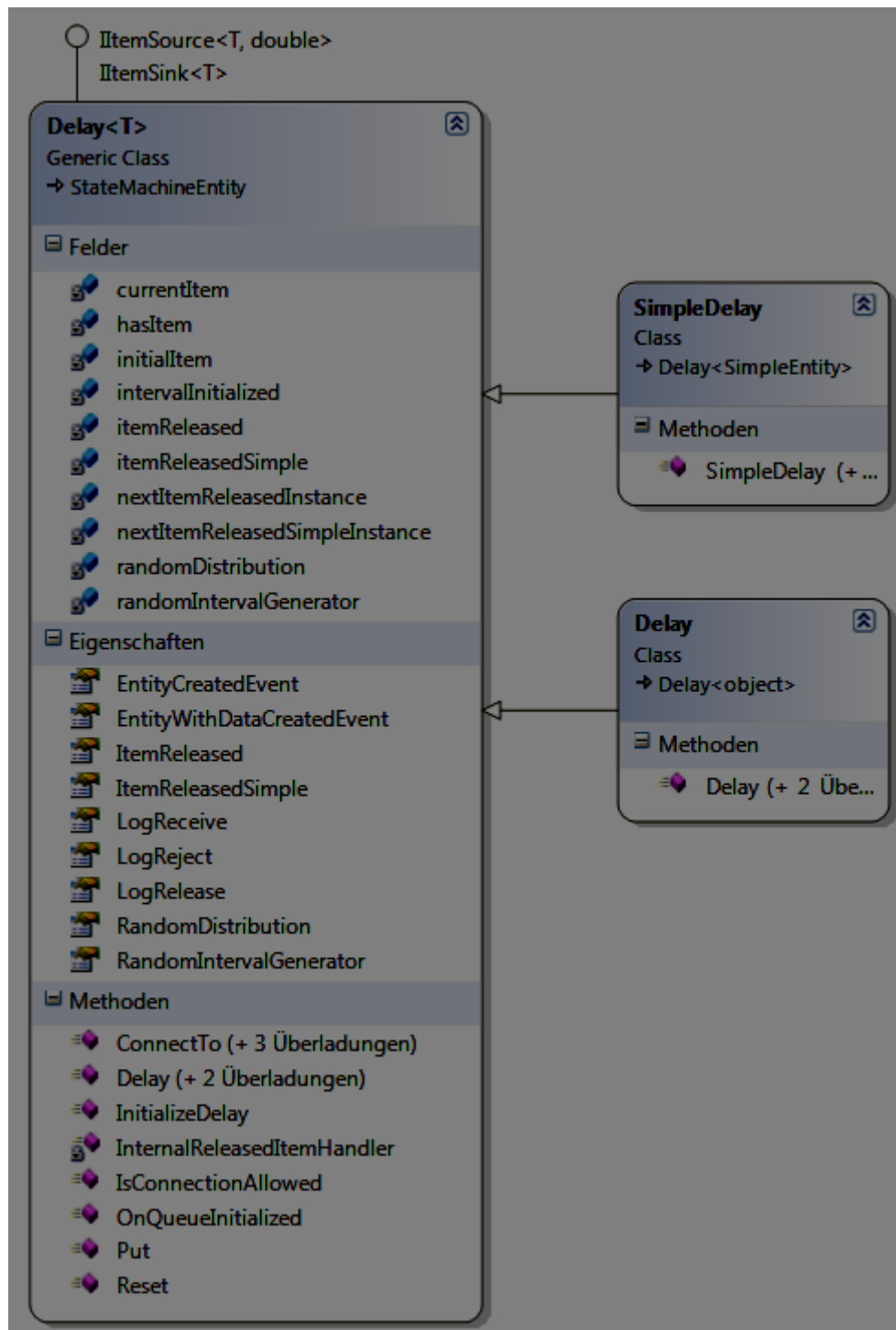


Figure 13: The delay entity templates.

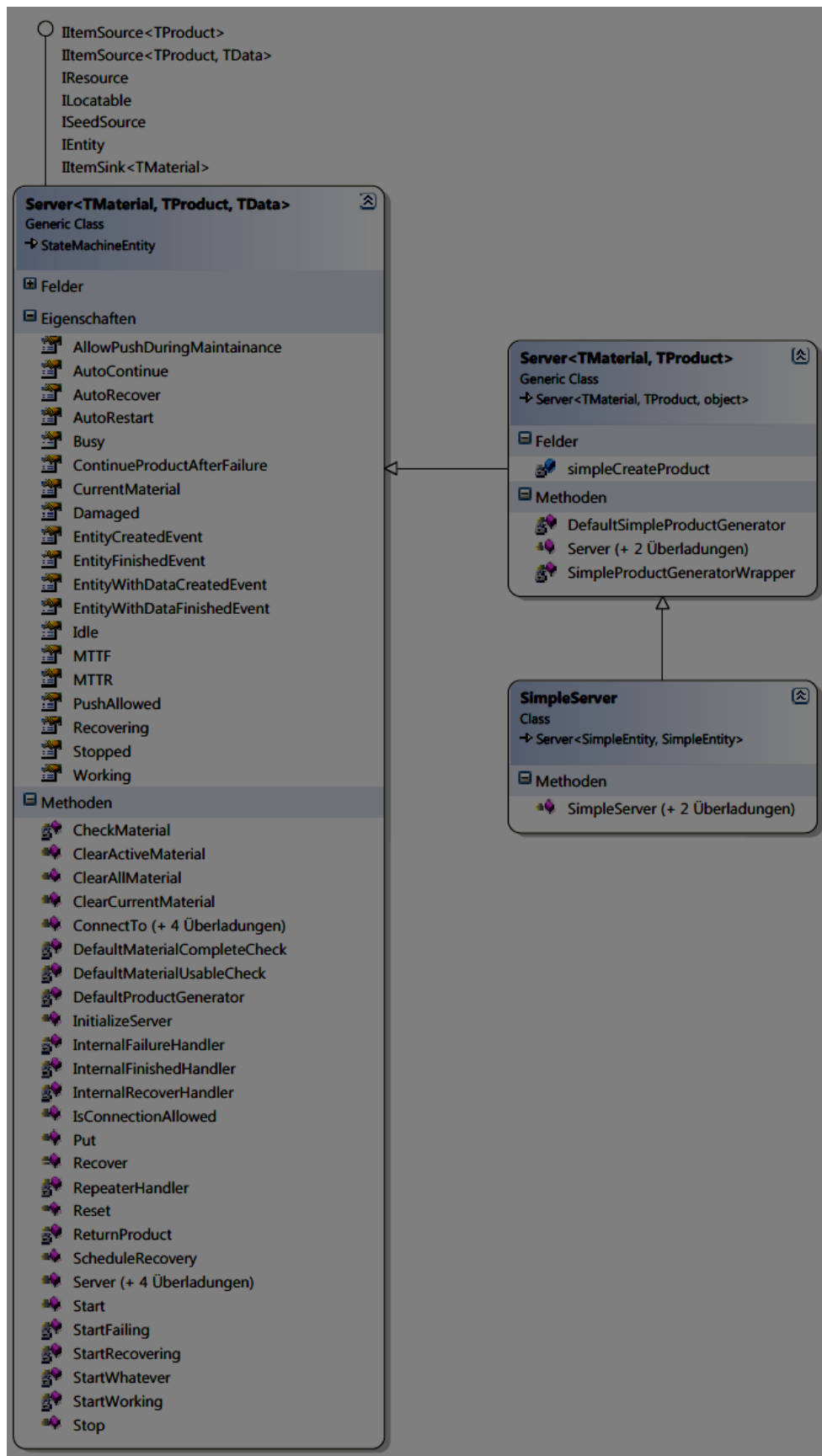


Figure 14: The server entity templates.

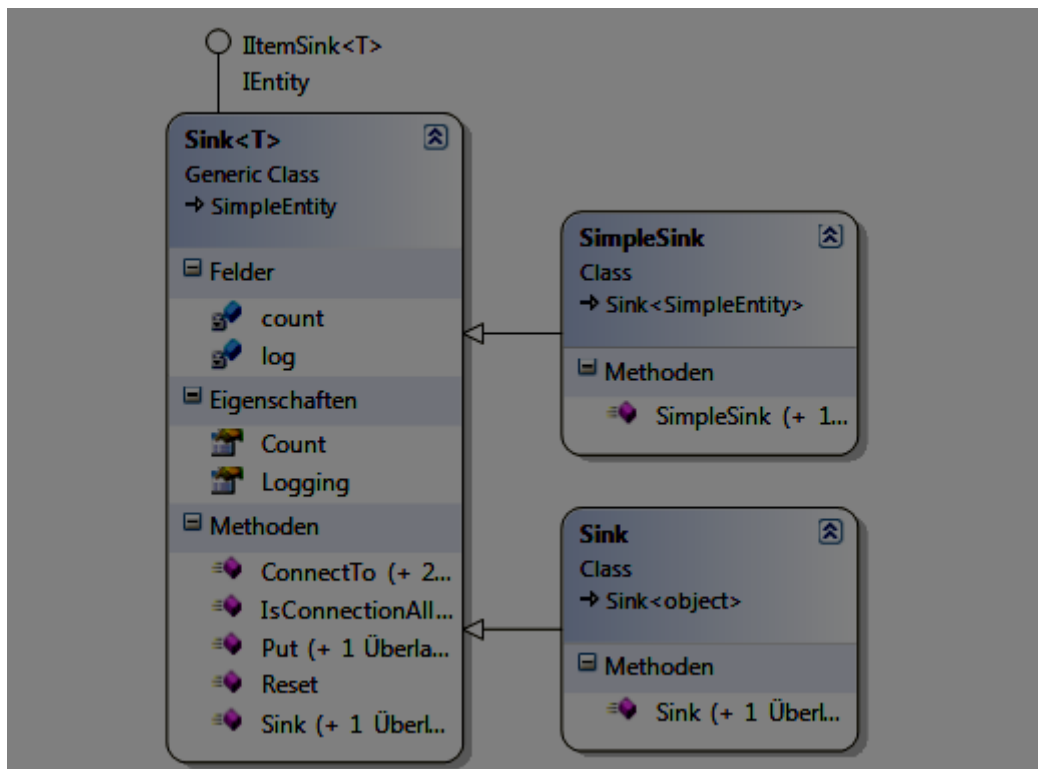


Figure 15: The sink entity templates.

10.4.4 Logging Framework

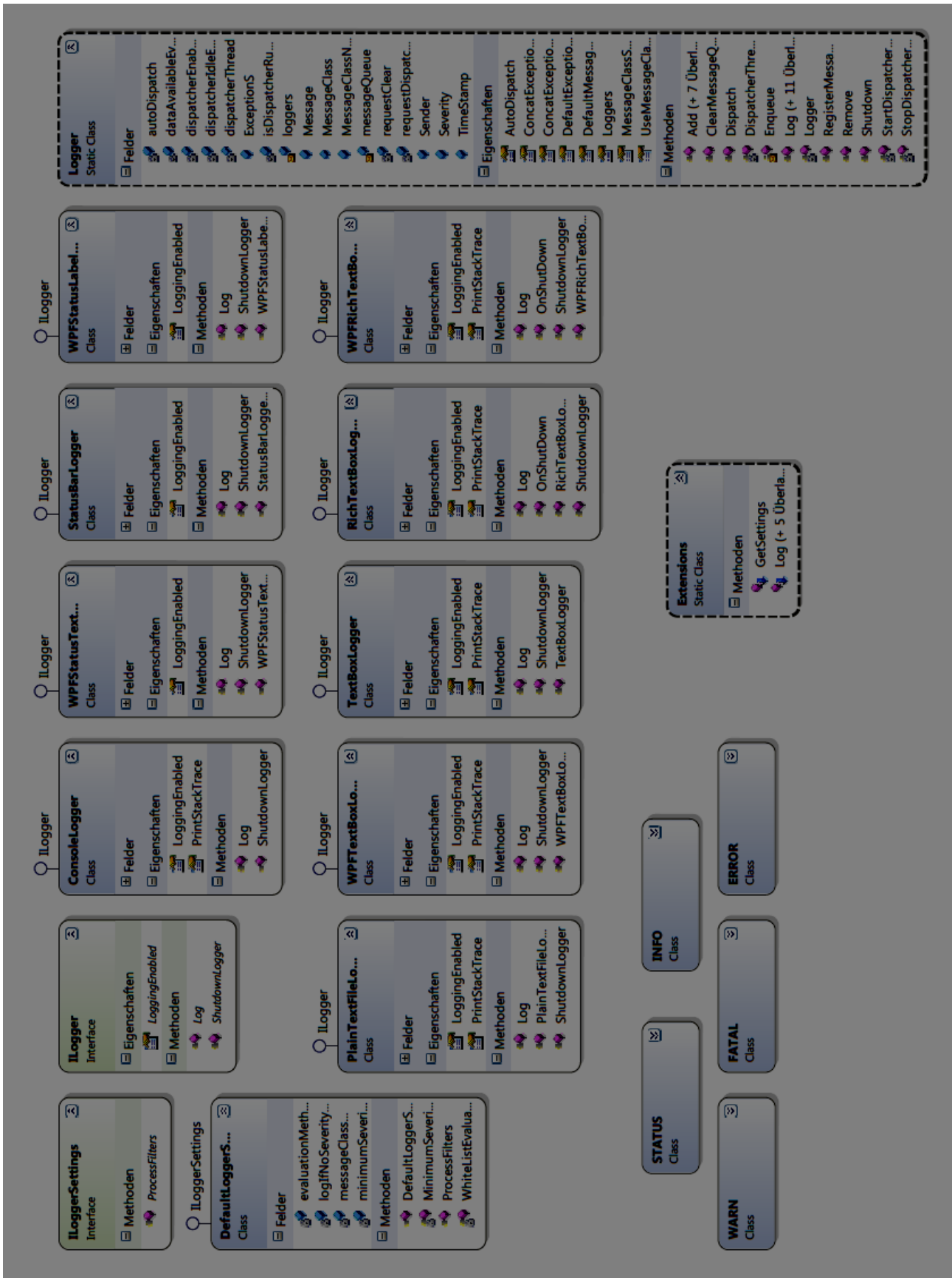


Figure 16: Architecture of the logging framework.

10.4.5 The Optimization Framework

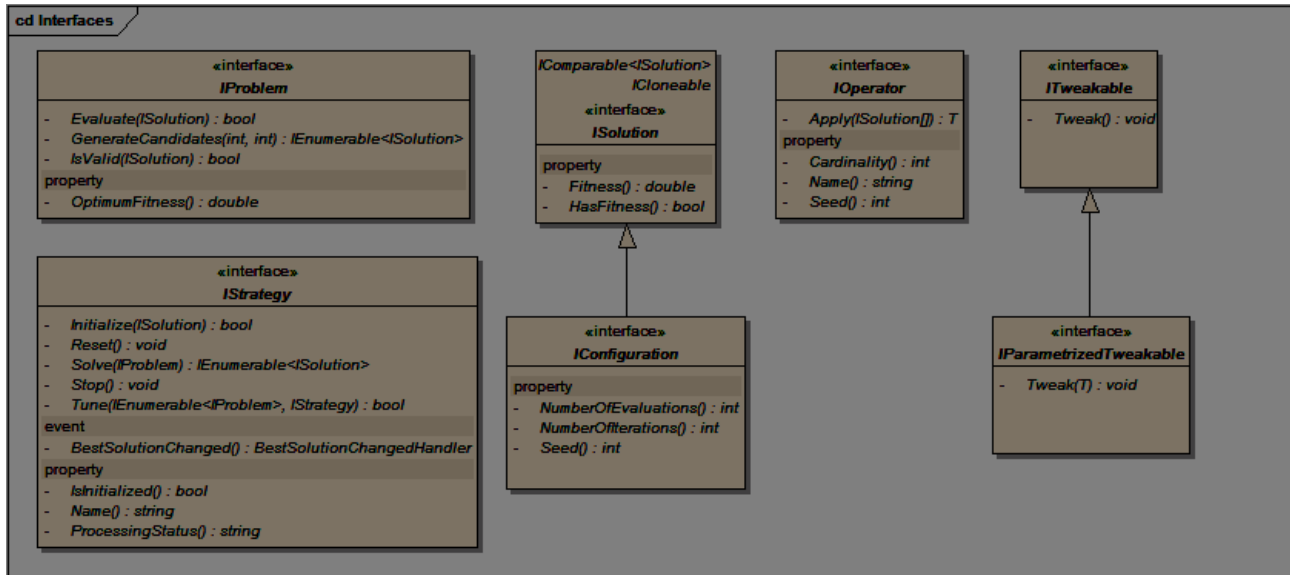


Figure 17: Interfaces for Optimization

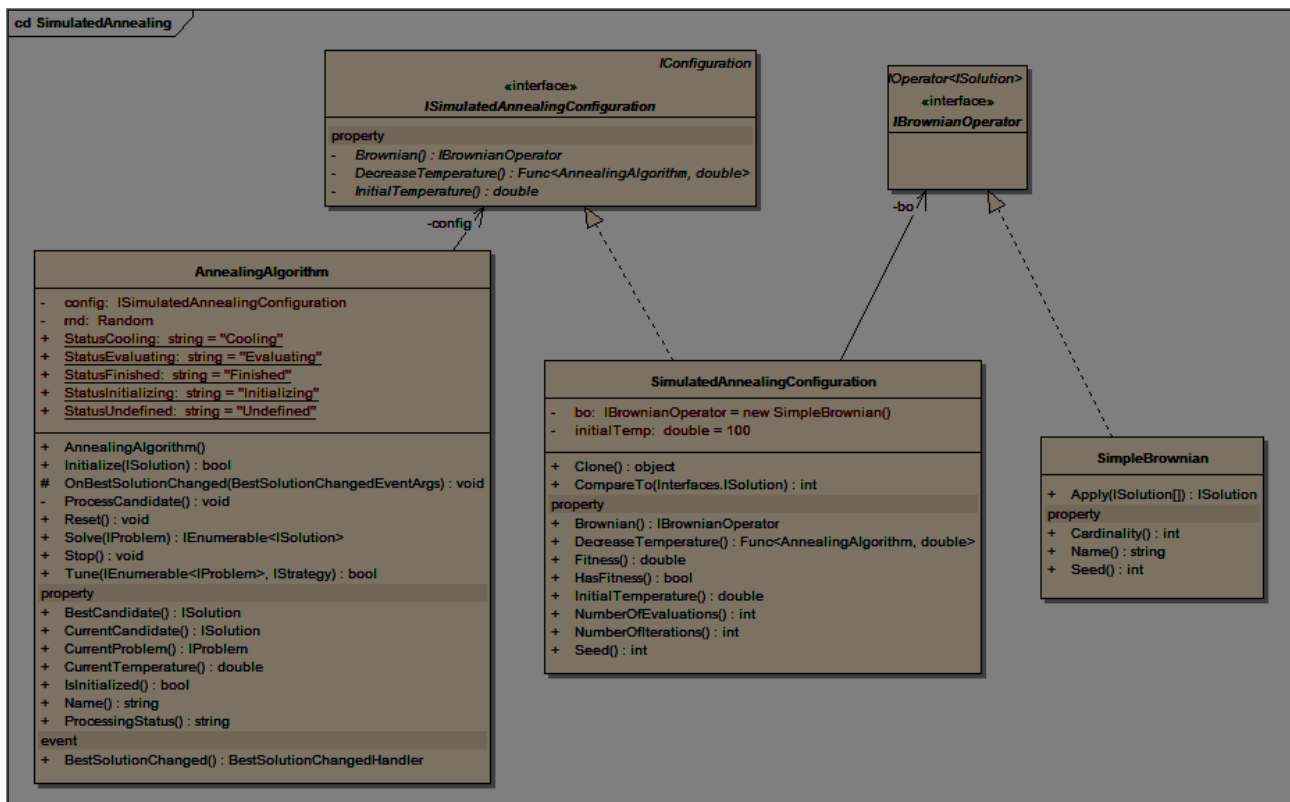


Figure 18: The Simulated Annealing Implementation

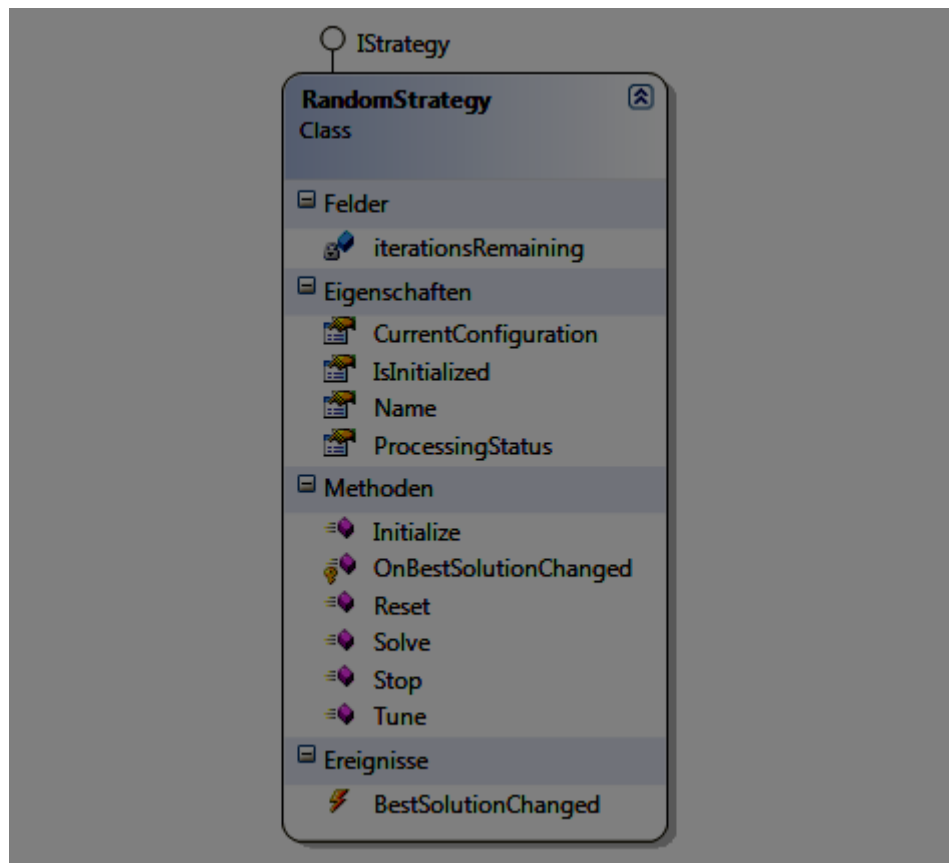


Figure 20: The Random Strategy Implementation

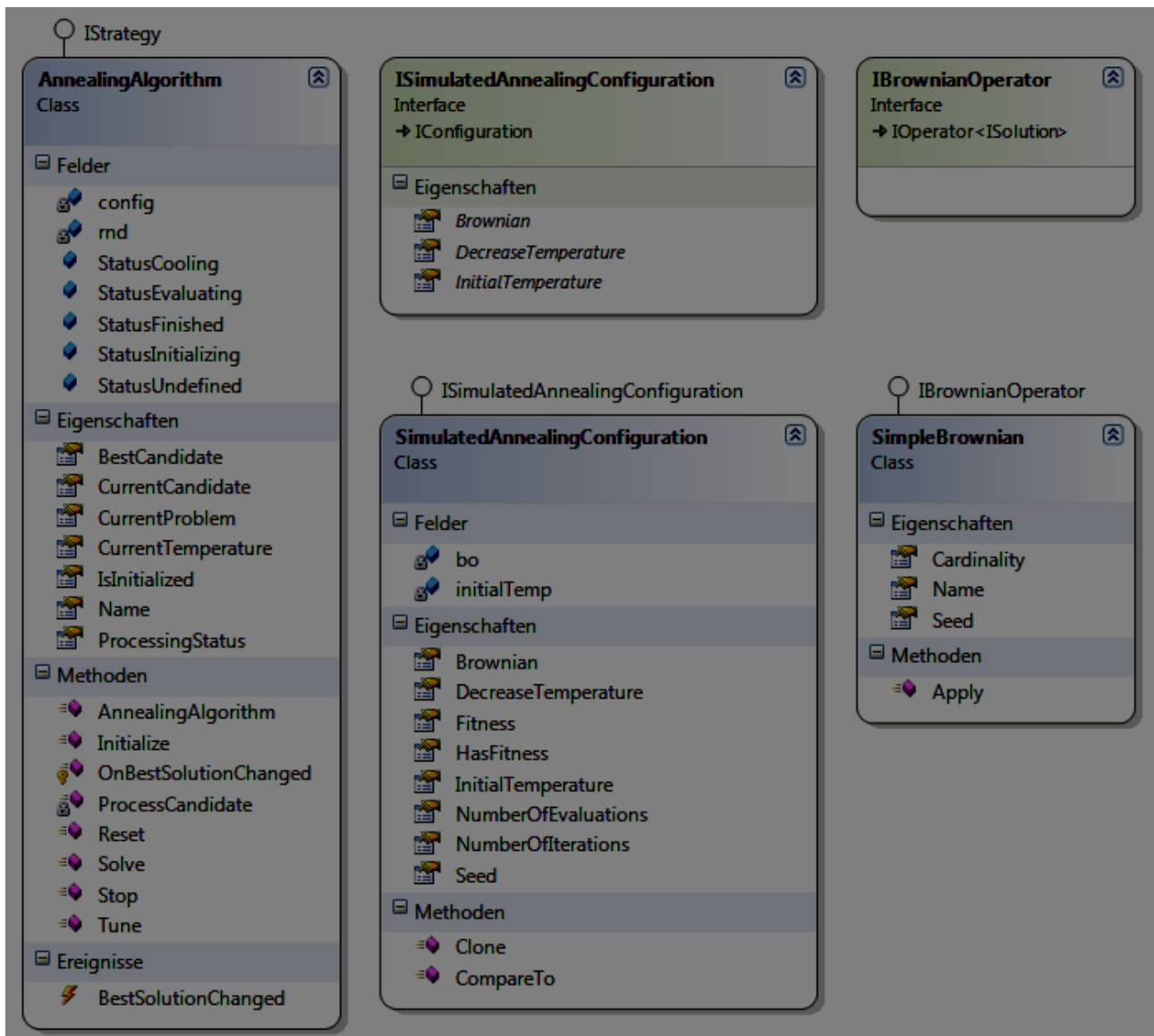


Figure 21: The Simulated Annealing Implementation

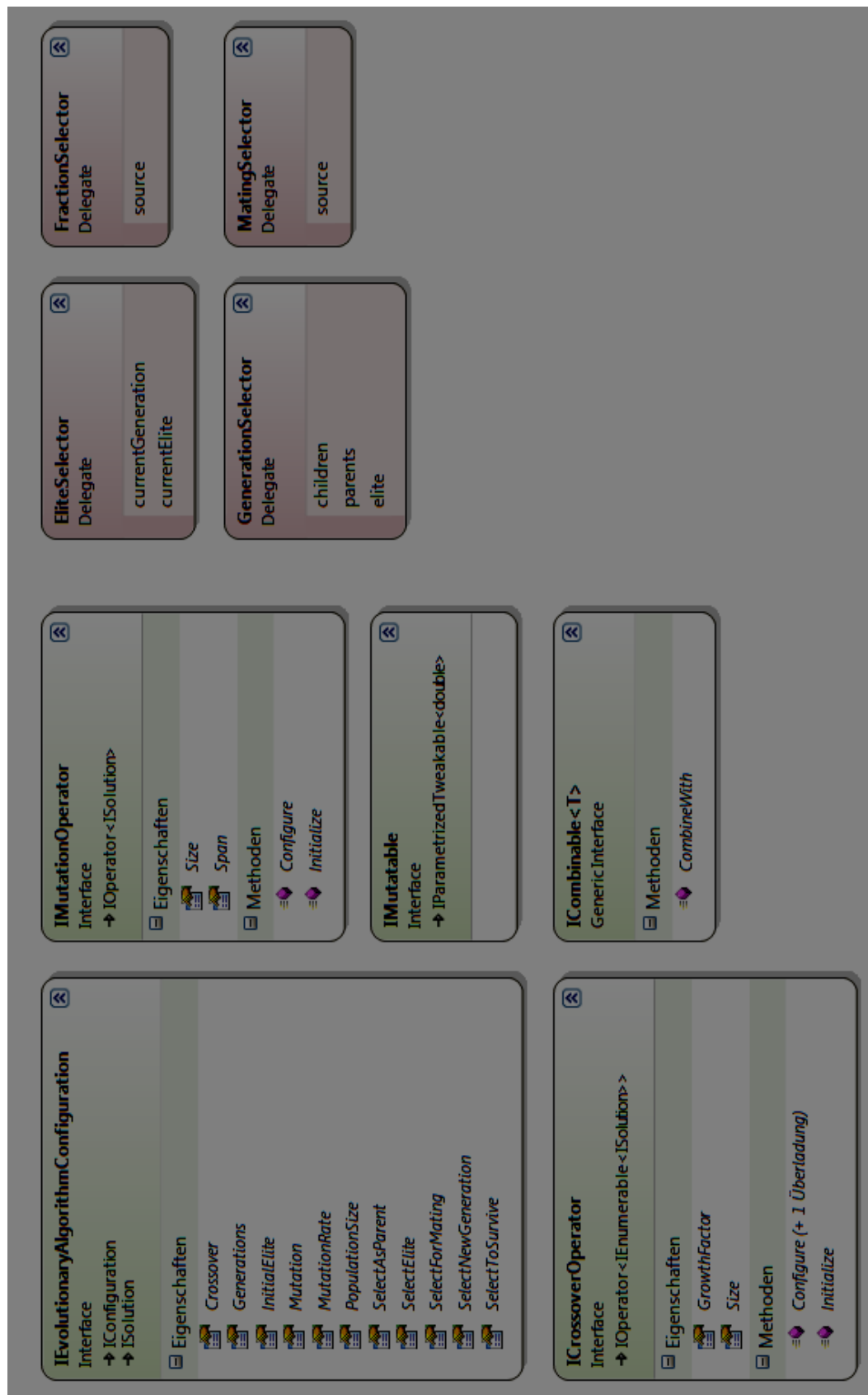


Figure 22: The Evolutionary Algorithms Infrastructure

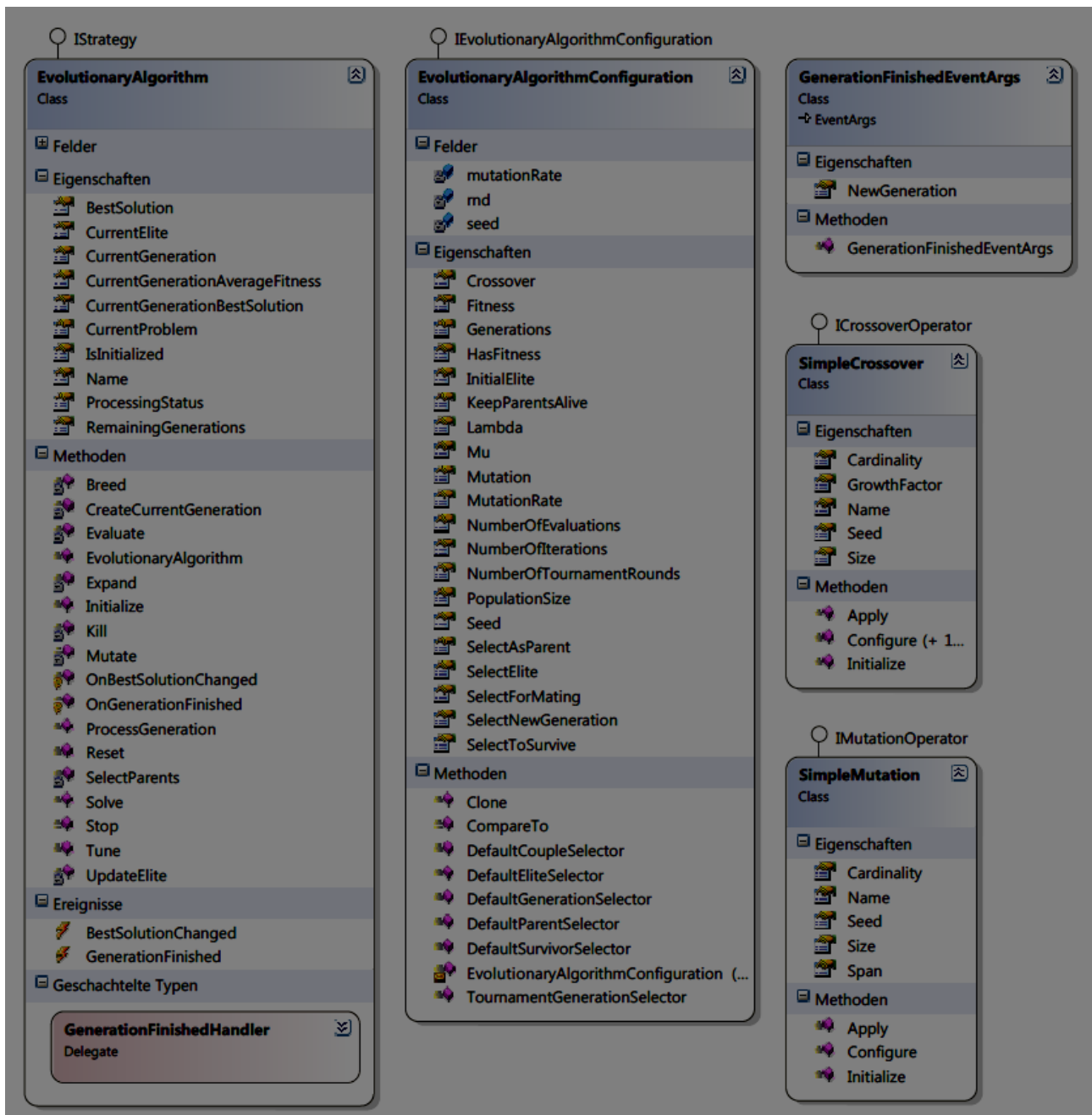


Figure 23: The Evolutionary Algorithms Implementation

10.4.6 Further UML Diagrams

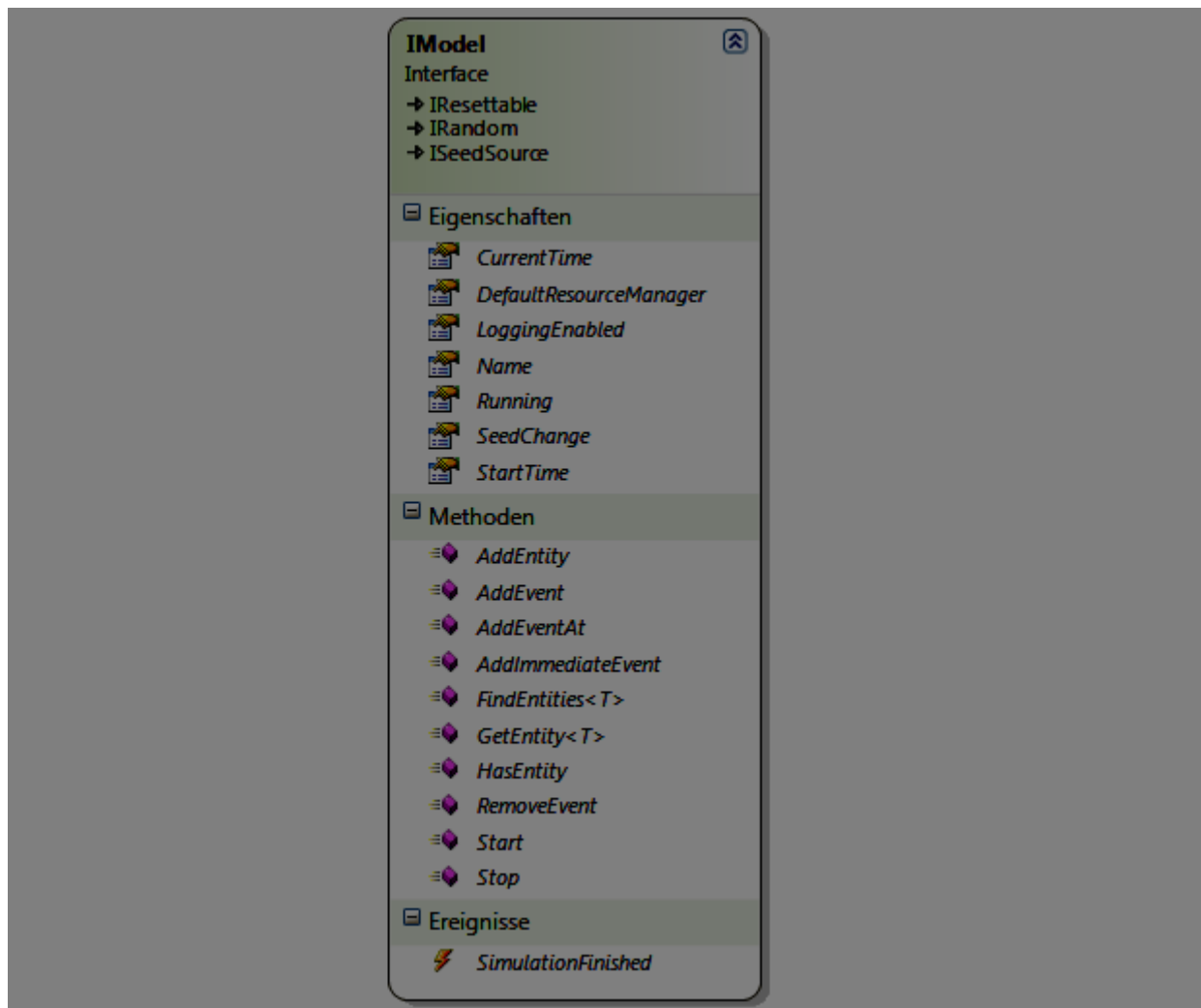


Figure 24: *MatthiasToolbox.DiscreteSimulation.Engine.IModel* interface

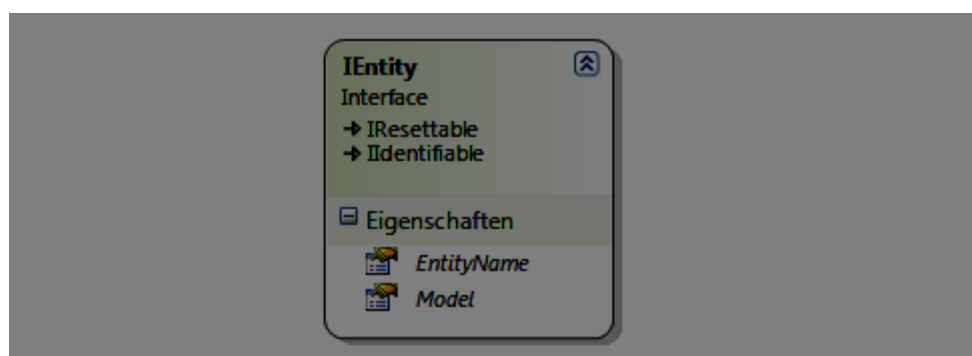


Figure 25: *MatthiasToolbox.DiscreteSimulation.Engine.IEntity* interface

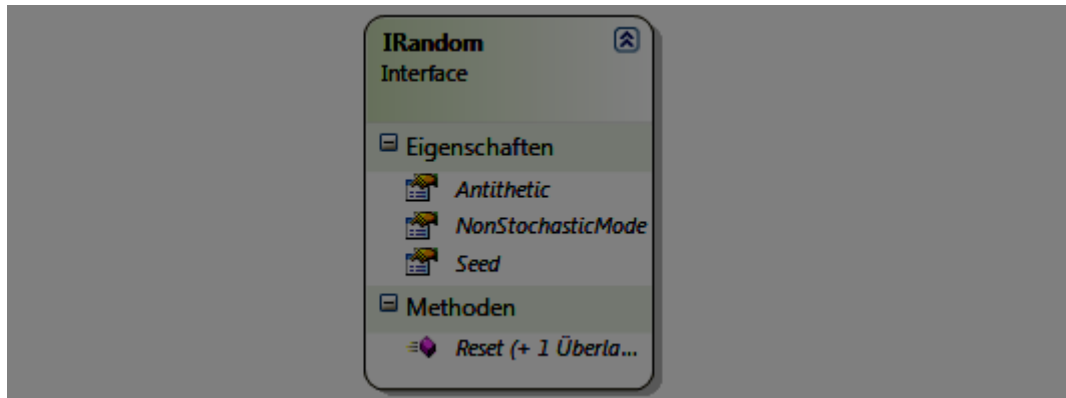


Figure 26: *MatthiasToolbox.DiscreteSimulation.Engine.IRandom* interface

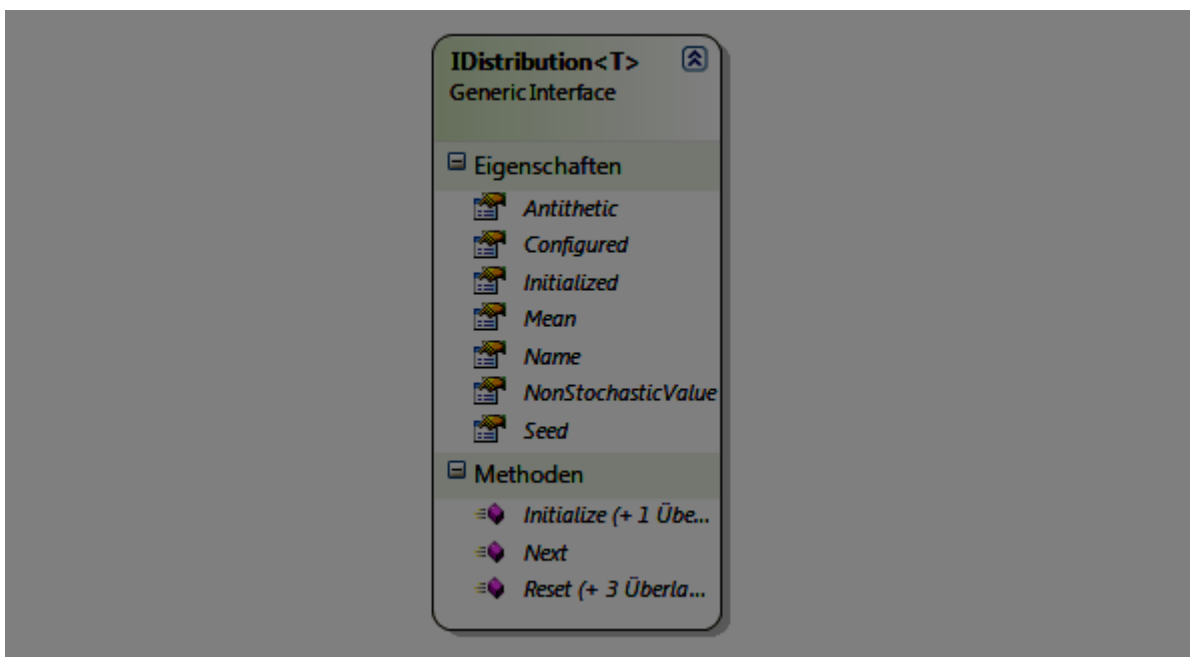


Figure 27: *MatthiasToolbox.Mathematics.Stochastics.Interfaces.IDistribution<T>* interface

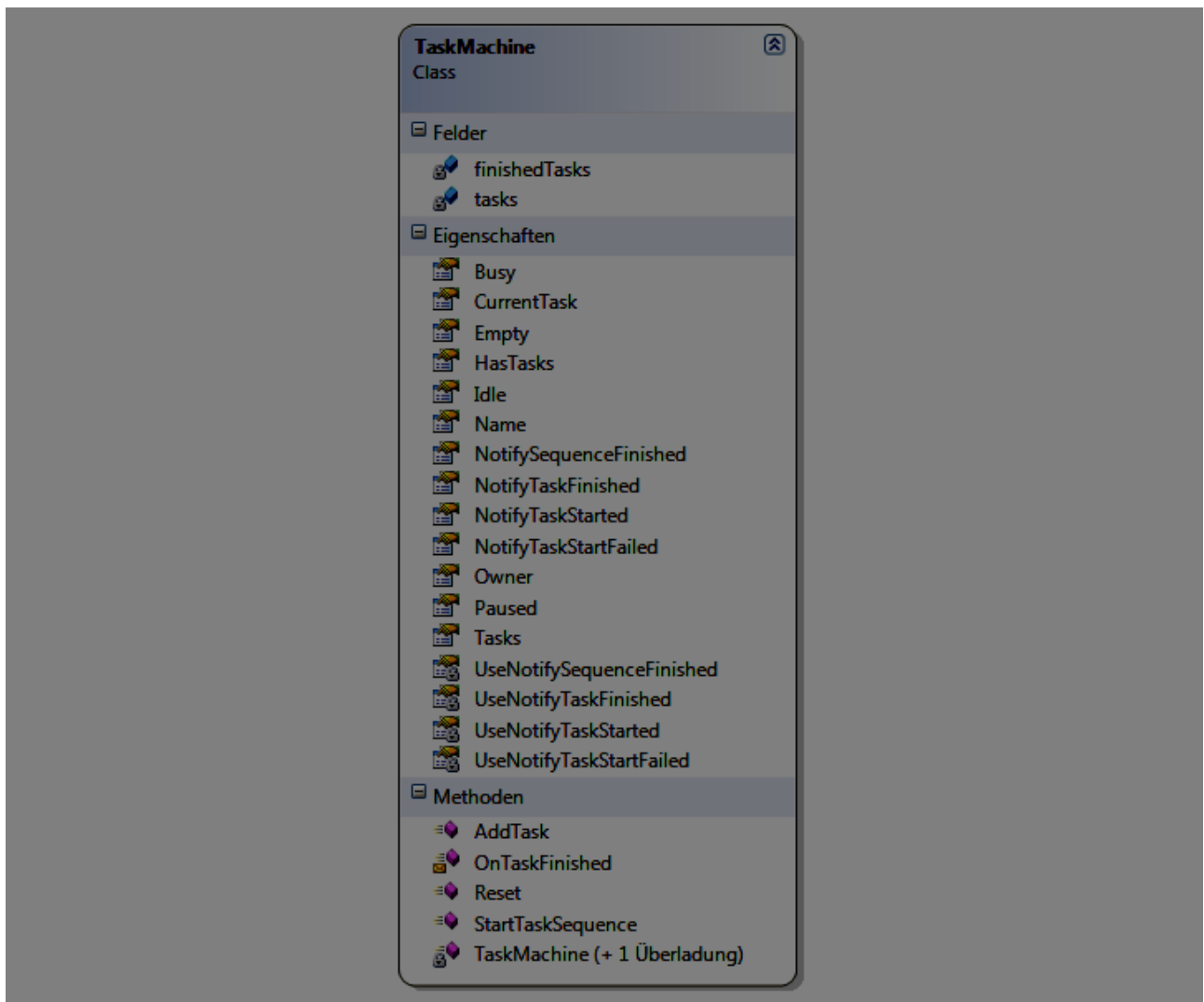
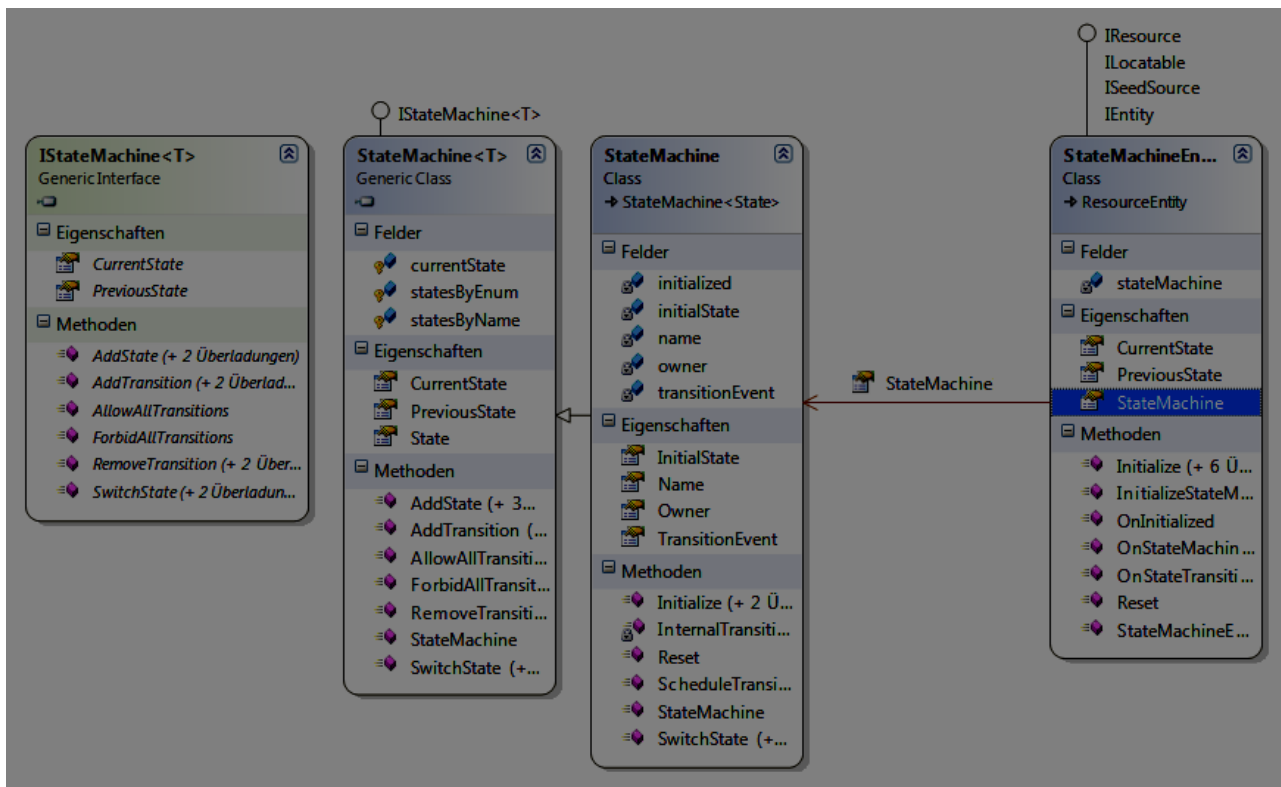


Figure 28: The task machine class.



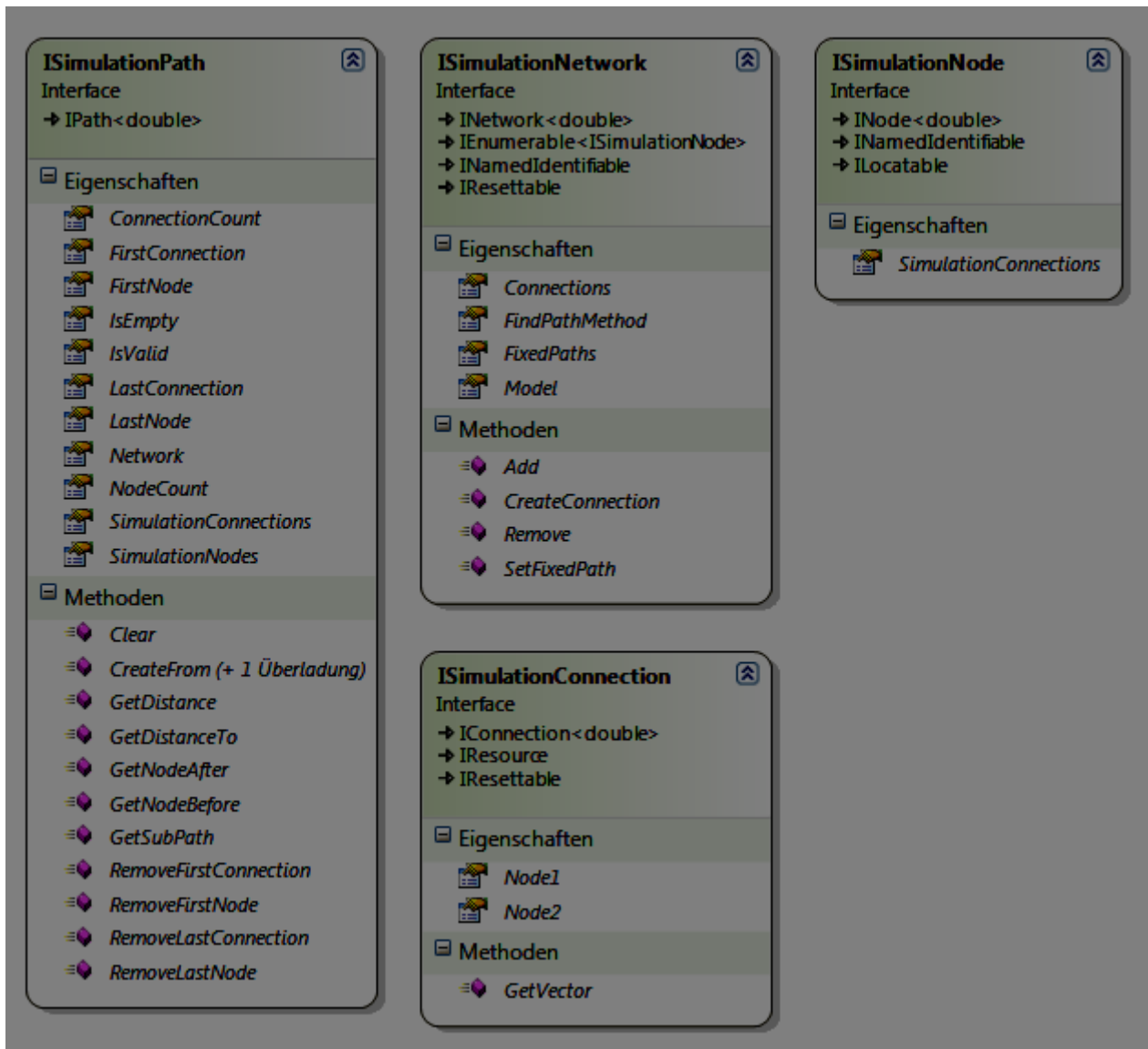


Figure 30: Simulation Network Interfaces

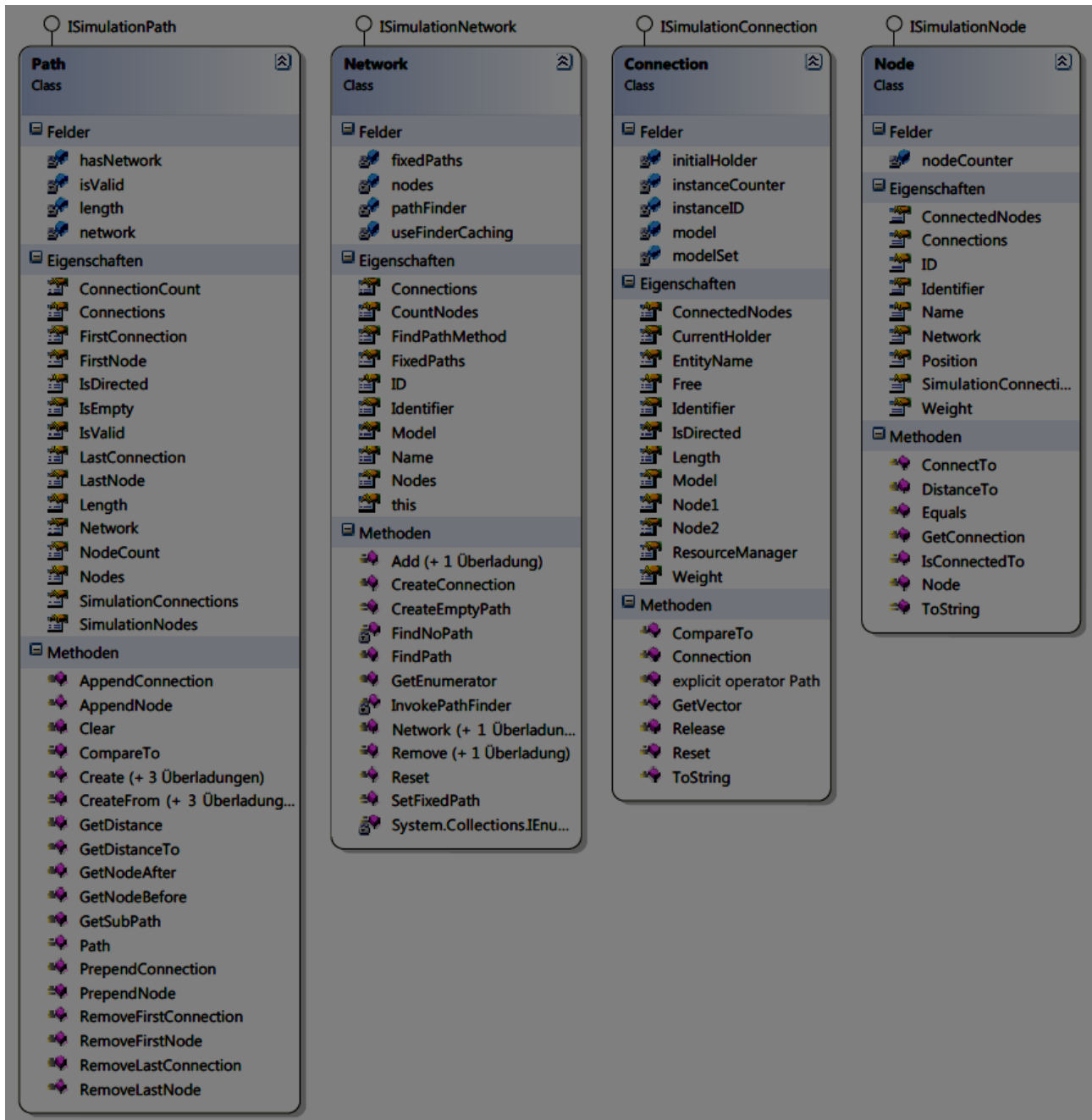


Figure 31: Simulation Network Infrastructure

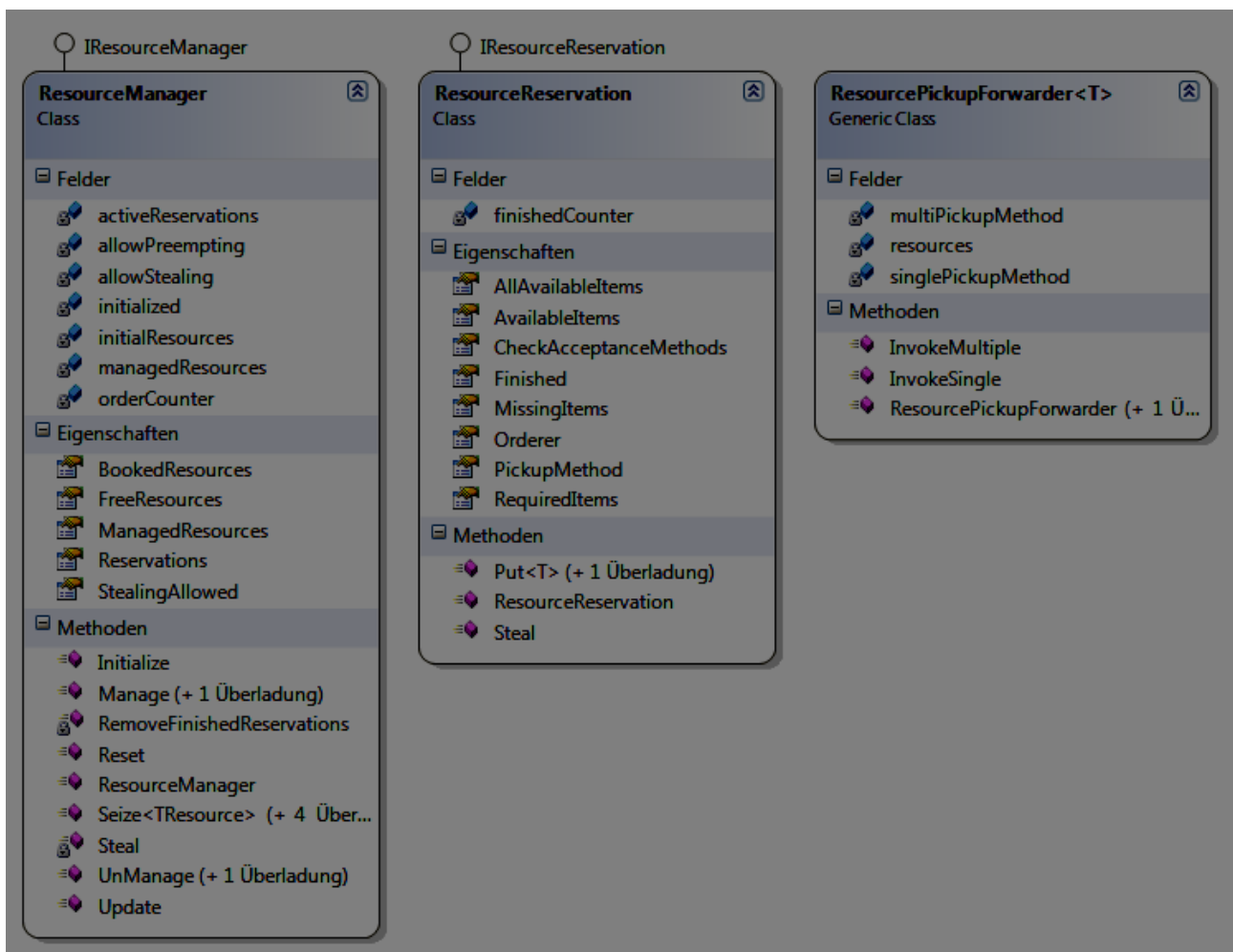


Figure 32: Resource Manager and Reservation

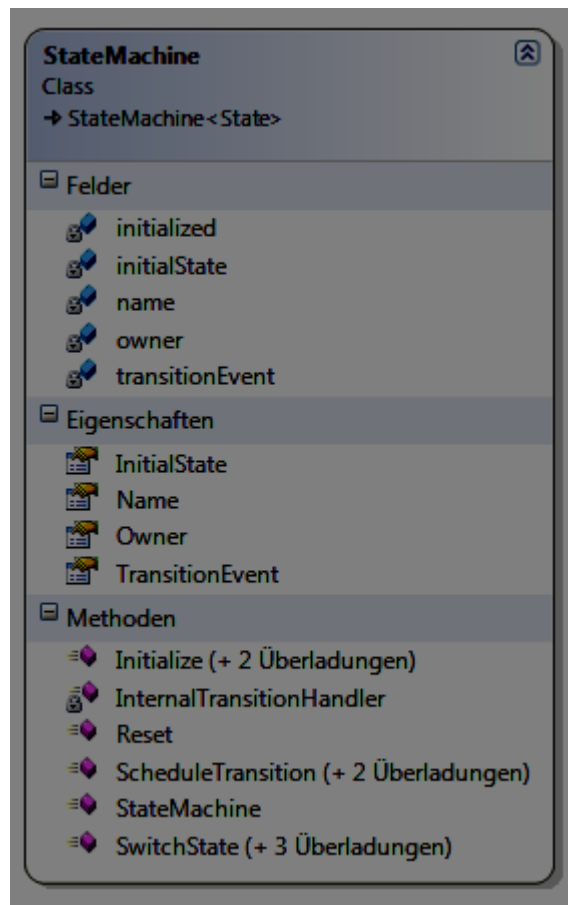


Figure 33: State Machine Implementation

10.5 Code Excerpts

The following sections contain C# source codes for some parts of the framework. Usually further classes and structures are required for these codes to be executed.

10.5.1 The SQSS Model

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using MatthiasToolbox.Logging.Loggers;
using MatthiasToolbox.Mathematics.Stochastics.Distributions;
using MatthiasToolbox.Simulation;
using MatthiasToolbox.Simulation.Engine;
using MatthiasToolbox.Simulation.Entities;
using MatthiasToolbox.Simulation.Enum;
using MatthiasToolbox.Simulation.Templates;

namespace MatthiasToolbox.SQSSModel
{
    /// <summary>
    /// The Source-Queue-Server-Sink model is considered a showcase model for simulation
    /// software. It is one of the simplest possible models in which still non-linear
    /// effects typical for discrete simulation may occur.
    ///
    /// The model consists of a SimpleSource instance which creates entities in normal
    /// distributed time intervals using a Generator function. The source is connected to
    /// a Queue (alternatively a delay can be put in between them). The queue is a
    /// SimpleBuffer with a FIFO rule and a capacity for 15 items.

```

```

///
/// The next item is a server, which pulls items from the queue in constant time
/// intervals. The server passes on the finished items to the last simulation object
/// in the chain which is the sink. The sink does nothing except for counting the
/// items it receives.
/// </summary>
public class Simulation
{
    #region cvar

    private static int productCounter = 0;

    private GaussianDistribution gauss;
    private ConstantDoubleDistribution c0nst;

    private DateTime startTime = DateTime.Now;
    private TimeSpan sourceIntervalAverage = new TimeSpan(0, 0, 2, 0, 0);
    private TimeSpan sourceIntervalDeviation = new TimeSpan(0, 0, 0, 10, 0);
    private TimeSpan sourceStartDelay = new TimeSpan(0, 0, 1, 30, 0);

    #endregion
    #region prop

    public Model Model { get; private set; }

    public SimpleSource Source { get; private set; }
    public SimpleBuffer Queue { get; private set; }
    // public SimpleDelay delay;
    public SimpleServer Server { get; private set; }
    public SimpleSink Sink { get; private set; }

    #endregion
    #region ctor

    /// <summary>
    /// Create the SQSS Model
    /// </summary>
    /// <param name="seed"></param>
    /// <param name="logTextBox"></param>
    public Simulation(int seed, RichTextBox logTextBox)
    {
        Model = new Model("SQSS", seed, startTime);
        Model.LogStart = true;
        Model.LogFinish = true;
        Simulator.RegisterSimulationLogger(new RichTextBoxLogger(logTextBox));
    }

    #endregion
    #region hand

    private void ProductFinished(IEntity sender, SimpleEntity product)
    {
        if (Model.LoggingEnabled)
            sender.Log<SIM_INFO>(product.ToString() + " finished.", Model);
    }

    private void SourceHandler(IEntity sender, SimpleEntity entity)
    {
        // if (productCounter > 20) source.Stop();
        // this.Log<SIM_INFO>("Queue has " + queue.Count.ToString() + " items.", model);
    }

    private void ItemReceived(IEntity sender, SimpleEntity item)
    {
        if (Queue.Count > 1 && Server.Idle) Server.Start();
    }

```

```

}

private void QueueRejectedItem(SimpleEntity entity)
{
    if (Model.LoggingEnabled)
        this.Log<SIM_WARNING>(entity.EntityName +
            " was not accepted because the queue is full.", Model);
}

private void QueueFull(IEntity entity)
{
    if (Model.LoggingEnabled) this.Log<SIM_WARNING>("Queue just ran full!", Model);
}

#endregion
#region impl

/// <summary>
/// Initializes all entities and builds the model.
/// </summary>
public void BuildModel()
{
    // distributions
    gauss = new GaussianDistribution(sourceIntervalAverage.ToDouble(),
        sourceIntervalDeviation.ToDouble());
    c0nst = new ConstantDoubleDistribution(sourceIntervalAverage.ToDouble() * 4,
        false);

    // the source
    Source = new SimpleSource(Model, gauss, ProductGenerator, name: "TheSource");
    // source = new Source(model, c0nst, ProductGenerator, name: "TheSource");
    Source.EntityCreatedEvent.AddHandler(SourceHandler);
    Source.EntityCreatedEvent.Log = true;

    //// the delay
    //delay = new SimpleDelay(model, c0nst, name: "TheDelay");
    //delay.LogReceive = true;
    //delay.LogRelease = true;
    //delay.LogReject = true;
    //delay.ConnectTo(source);

    // the queue
    Queue = new SimpleBuffer(Model, QueueRule.FIFO,
        name: "TheQueue", maxCapacity: 15);
    Queue.ItemReceivedEvent.AddHandler(ItemReceived);
    Queue.BufferFullEvent.AddHandler(QueueFull);
    Queue.NotifyItemNotAccepted = QueueRejectedItem;
    Queue.ConnectTo(Source);

    // the server
    Server = new SimpleServer(Model, c0nst, name: "TheServer",
        checkMaterialUsable: CheckMaterialUsable,
        checkMaterialComplete: CheckMaterialComplete);
    Server.EntityFinishedEvent.AddHandler(ProductFinished);
    Server.AutoContinue = true;
    // server.PushAllowed = true;
    // server.ConnectTo(source);
    Server.ConnectTo(Queue);

    // the sink
    Sink = new SimpleSink(Model, name: "TheSink", log: true);
    Sink.ConnectTo(Server);
}

/// <summary>

```

```

    /// Start the simulation.
    /// </summary>
    /// <param name="hoursToRun"></param>
    public void Start(int hoursToRun)
    {
        Source.Start(sourceStartDelay);
        Model.Run(startTime.AddHours(hoursToRun));
    }

    /// <summary>
    /// Stop the simulation.
    /// </summary>
    public void Stop()
    {
        Model.Stop();
    }

    private SimpleEntity ProductGenerator()
    {
        productCounter += 1;
        return new SimpleEntity(Model, "Product " + productCounter.ToString(),
            "Product " + productCounter.ToString());
    }

    private bool CheckMaterialUsable(SimpleEntity material)
    {
        bool result = Server.CurrentMaterial.Count < 2;
        if (!result && Model.LoggingEnabled)
            Server.Log<SIM_INFO>(material.ToString() + " was discarded.", Model);
        return result;
    }

    private bool CheckMaterialComplete(List<SimpleEntity> material)
    {
        return material.Count == 2;
    }

    #endregion
}

```

10.5.2 Helper Functions for the Evolutionary Strategy

```

private List<ISolution> UpdateElite(List<ISolution> individuals,
    List<ISolution> currentElite)
{
    return config.SelectElite.Invoke(individuals, currentElite).ToList();
}

private List<ISolution> SelectParents(List<ISolution> potentialParents)
{
    return config.SelectAsParent.Invoke(potentialParents).ToList();
}

private IEnumerable<ISolution> Breed(List<ISolution> parents)
{
    foreach (Tuple<ISolution, ISolution> couple in config.SelectForMating(parents))
    {
        foreach (ISolution kid in config.Crossover.Apply(couple.Item1, couple.Item2))
            yield return kid;
    }
}

private List<ISolution> Mutate(List<ISolution> individuals)

```

```

{
    List<ISolution> result = new List<ISolution>();
    foreach (ISolution individual in individuals)
    {
        if (!rnd.ExecuteConditionally(
            () => result.Add(config.Mutation.Apply(individual)), config.MutationRate))
            result.Add(individual);
    }
    return result;
}

private List<ISolution> Kill(List<ISolution> parents)
{
    return config.SelectToSurvive(parents).ToList();
}

private double Evaluate(List<ISolution> individuals)
{
    CurrentGenerationBestSolution = null;
    double result = 0;
    double counter = 0;
    string oldStatus = ProcessingStatus;
    ProcessingStatus = StatusEvaluating;
    foreach (ISolution individual in individuals)
    {
        if(!individual.HasFitness) CurrentProblem.Evaluate(individual);
        if (BestSolution == null || BestSolution.Fitness <= individual.Fitness)
        {
            previousBestSolution = BestSolution;
            BestSolution = individual;
            bestSolutionChanged = true;
        }
        if (CurrentGenerationBestSolution == null ||
            CurrentGenerationBestSolution.Fitness <= individual.Fitness)
            CurrentGenerationBestSolution = individual;
        result += individual.Fitness;
        counter += 1;
    }
    ProcessingStatus = oldStatus;
    return result / counter;
}

```

10.5.3 Default Evolutionary Delegate Implementations

```

/// <summary>
/// This selects subsequent pairs from the given parents. In case
/// of an odd number of parents the first individual is used twice.
/// In case of only one item, a tuple with both items being the same
/// individual will be returned. If no candidates were provided
/// this will return an empty enumerable.
/// </summary>
/// <param name="parents">The parents to process.</param>
/// <returns>A list of couples for mating.</returns>
public IEnumerable<Tuple<ISolution, ISolution>> DefaultCoupleSelector(
    List<ISolution> parents)
{
    if (parents.Count == 0) yield break;
    ISolution firstParent = parents[0];
    if (parents.Count == 1)
    {
        yield return new Tuple<ISolution, ISolution>(firstParent, firstParent);
        yield break;
    }
}

```

```

    for (int i = 1; i <= parents.Count; i += 2)
    {
        if (i == parents.Count)
        {
            yield return new Tuple<ISolution, ISolution>(parents[i - 1], parents[0]);
        }
        else
        {
            yield return new Tuple<ISolution, ISolution>(parents[i - 1], parents[i]);
        }
    }
}

/// <summary>
/// Simply selects the first  $\mu$  (MU) items from the current generation.
/// </summary>
/// <param name="currentGeneration">The current generation.</param>
/// <returns>A list of parents.</returns>
public IEnumerable<ISolution> DefaultParentSelector(List<ISolution> currentGeneration)
{
    int max = Math.Min(Mu, currentGeneration.Count);
    for (int i = 0; i < max; i += 1)
    {
        yield return currentGeneration[i];
    }
}

/// <summary>
/// This will not kill anyone.
/// </summary>
/// <param name="currentParents">The parents in the current generation.</param>
/// <returns>The complete contents of currentParents.</returns>
public IEnumerable<ISolution> DefaultSurvivorSelector(List<ISolution> currentParents)
{
    foreach (ISolution s in currentParents) yield return s;
}

/// <summary>
/// This will select the next generation using the  $\mu$  (MU) and  $\lambda$  (LAMBDA) parameters.
/// If KeepParentsAlive is set to true, it will select  $\mu$  parents and  $\lambda$  children,
/// otherwise only a maximum of  $\lambda$  children will be returned. The elite will be ignored.
/// </summary>
/// <param name="children">A list of children to select from.</param>
/// <param name="parents">A list of parent individuals to select from.</param>
/// <param name="elite">Will be ignored.</param>
/// <returns>A new generation with no elitism.</returns>
public IEnumerable<ISolution> DefaultGenerationSelector(
    List<ISolution> children,
    List<ISolution> parents,
    List<ISolution> elite)
{
    int i = 0;
    if (KeepParentsAlive)
    {
        // return a maximum of  $\lambda$  children
        foreach (ISolution s in children)
        {
            i += 1;
            yield return s;
            if (i == Lambda) break;
        }

        // return a maximum of  $\mu$  parents
        i = 0;
        foreach (ISolution s in parents)

```

```

        {
            i += 1;
            yield return s;
            if (i == Mu) break;
        }
    }
else
{
    // return a maximum of  $\lambda$  children
    foreach (ISolution s in children)
    {
        i += 1;
        yield return s;
        if (i == Lambda) break;
    }
}
}

/// <summary>
/// This will select the next generation using the  $\mu$  (MU) and  $\lambda$  (LAMBDA) parameters.
/// If KeepParentsAlive is set to true, it will select  $\mu$  parents and  $\lambda$  children,
/// otherwise only a maximum of  $\lambda$  children will be returned. The elite will be ignored.
/// Tournament selection will be used with NumberOfTournamentRounds as t parameter.
/// </summary>
/// <param name="children">A list of children to select from.</param>
/// <param name="parents">A list of parent individuals to select from.</param>
/// <param name="elite">Will be ignored.</param>
/// <returns>A new generation with no elitism.</returns>
public IEnumerable<ISolution> TournamentGenerationSelector(
    List<ISolution> children,
    List<ISolution> parents,
    List<ISolution> elite)
{
    if (KeepParentsAlive)
    {
        // return a maximum of  $\lambda$  children
        for (int i = 0; i < Lambda; i++)
        {
            yield return children.TournamentSelect(rnd, NumberOfTournamentRounds);
        }

        // return a maximum of  $\mu$  parents
        for (int i = 0; i < Mu; i++)
        {
            yield return parents.TournamentSelect(rnd, NumberOfTournamentRounds);
        }
    }
    else
    {
        // return a maximum of  $\lambda$  children
        for (int i = 0; i < Lambda; i++)
        {
            yield return children.TournamentSelect(rnd, NumberOfTournamentRounds);
        }
    }
}

/// <summary>
/// Returns only the current elite
/// </summary>
/// <param name="currentGeneration">Will be ignored</param>
/// <param name="currentElite">Will be returned</param>
/// <returns>The current elite.</returns>
public IEnumerable<ISolution> DefaultEliteSelector(
    List<ISolution> currentGeneration,

```

```

    List<ISolution> currentElite)
{
    foreach (ISolution s in currentElite) yield return s;
}

```

10.6 Logging Framework

The logging framework is a very lightweight, generic and flexible tool for relaying debug and failure information. A small number of predefined "Loggers" (log message consumers) is provided (e. g. for logging to text files, to the console and to various GUI controls in Windows Forms as well as Windows Presentation Foundation). No setup or initialization whatsoever is required to use the framework.

The following code shows how to use the framework to log string messages to a RichTextBox inside a windows forms application assuming the RichTextBox is named "richTextBox1":

```

Logger.Add(new RichTextBoxLogger(richTextBox1));
// ...
this.Log<INFO>("Hello World!");

```

The output of this code will be similar to the following:

2010-05-10 19:09:25 - INFO @ SomeProject.Form1, Text: Form1 - Hello world!

Please note that "this.Log..." only works in non static classes. In static classes use "Logger.Log..." instead and provide a sender as parameter if you want to give a context (an example of this is shown further below).

The first line is used to register a logger. All loggers are registered centrally and can be enumerated using Logger.Loggers. A known logger can also be removed again using the Remove method. There are a number of overloads allowing certain kinds of settings to be configured when adding a logger. These will be described further below.

The second line is used to write "Hello World!" to the logging framework. Log messages are usually grouped into classes of severity or other. The following message classes are predefined:

Message Class	Default Severity
STATUS	0
INFO	1
WARN	2
ERROR	3
FATAL	4

The severity is a float value to allow configuring a severity setting in between existing values. But the values can also be overridden and it is furthermore possible to add your own message classes and remove or replace the predefined values as shown below:

```

Logger.RegisterMessageClass<MyMessageClass1>(0.5);
Logger.RegisterMessageClass(typeof(MyMessageClass2), 5);
Logger.MessageClassSeverity[typeof(MyMessageClass3)] = 3.5f;
Logger.MessageClassSeverity.Remove(typeof(INFO));
Logger.MessageClassSeverity[typeof(WARN)] = 100;
Logger.RegisterMessageClass<FATAL>(10);

```

Removing a message class or not providing a severity value does not prevent its usage though. For unknown message classes the framework will assume zero as severity.

It is also possible to prevent the framework from using the defined severity values by setting `Logger.UseMessageClassAsSeverity` to false. In this case you must explicitly provide a severity value when logging or the severity will be set to zero for the given message:

```
this.Log<ERROR>("Hello world!", Severity => 20f);
```

The expression in the second parameter will be automatically parsed by the framework. Note though that predefined parameters like "Severity" are case sensitive and values do not automatically cast, therefore 20f will work but 20 (int) or 20.1 (double) may not be correctly recognized. Finally, if you provide a Severity value in spite of a predefined value being available, the predefined value will be overwritten.

The logging framework will store the sender as object and the predefined loggers will try to use the sender's `ToString()` method to retrieve the sender name. However, if a sender of type string is provided manually as shown below, the value of sender will be used instead:

```
Logger.Log(MyStaticClass.Instance, "Hello World!");  
Logger.Log("MyNamespace.MyStaticClass", "Hello World!");  
Logger.Log("Hello World!", Sender => "MyNamespace.MyStaticClass");  
this.Log("Hello World!", Sender => "MyNamespace.MyStaticClass");
```

The code in the last line will result in the logger using "MyNamespace.MyStaticClass" as sender no matter what "this" is. As you can see in the examples above it is possible to use the Log function with or without a generic parameter. If no generic parameter is used, the framework will use `Logger.DefaultMessageClass` (default is INFO) or `Logger.DefaultExceptionClass` (default is ERROR) if an exception is provided as shown below:

```
this.Log("Hello Exception!", someException);
```

The above line will use "ERROR", though both settings can be changed of course. If – as shown above – an exception and a message are provided, the framework will append the exception message to the given message if `Logger.ConcatExceptionMessage` is set to true (default is true). The two strings will be separated using `Logger.ConcatExceptionMessageSeparator` (default is space).

The Logging Process

Normally logging is done in a separate thread. That means that if you log messages, your code will continue to be executed after the message has been passed on to the framework. This may result in messages being lost if your application crashes. However, you can call `Logger.Dispatch()` at any time to force the framework to process all currently queued messages. Furthermore you can set `Logger.AutoDispatch` to true (default is false) to force the framework to always process the messages before handing back execution control to your own code. This can result in your application being slowed down but may sometimes still be desirable, especially if the logging is used to provide feedback to the user (for example in splash screens).

If you call `Logger.Shutdown()` or `Logger.Shutdown(false)` this will also result in the remaining messages being processed. However, you only need to call shutdown if one of your loggers requires this (the predefined loggers do not require a shutdown call). `Logger.Shutdown(true)` will force the framework to shutdown the dispatcher thread immediately. This may result in messages being lost.

Adding and Configuring Loggers

When adding a logger to the framework there are a number of ways these can be configured. First of

all, using one of the following methods a number of message classes can be provided. The logger will only be notified of these classes:

```
Logger.Add<STATUS>(SomeStatusLogger);  
Logger.Add<ERROR, FATAL>(LoggerForBadThings);  
Logger.Add(LoggerForHarmlessThings, INFO, STATUS, WARN);
```

Alternatively a minimum severity can be provided:

```
Logger.Add<STATUS>(SomeStatusLogger, 2f);
```

The default minimum severity (used by DefaultLoggerSettings) is one. It is also possible to provide a delegate which will filter messages in its own way:

```
Logger.Add(SomeBiasedLogger, data => ((string)data["Message"]).Contains("something"));
```

Finally you can provide an arbitrary ILoggerSettings instance. If you do not want to implement your own logger settings class, you can use DefaultLoggerSettings which allows roughly the same settings as the above methods. Otherwise you just have to Implement ILoggerSettings which only requires one method.

To create your own loggers, use the ILogger interface.