Uppsala Master's Theses in
Computing Science 178
Examensarbete DV3
2000-11-01
ISSN 1100–1836

Design Patterns for Simulations in Erlang/OTP

Ulf Ekström

Information Technology
Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

This work has been carried out at
Sjöland & Thyselius Telecom AB
Sehlstedtsgatan 6
SE-115 28 Stockholm
Sweden

Abstract

The growing field of *Design Patterns* offers hope of controlling the complexities associated with the development of large software applications. The architecture of a system can be expressed at a higher level of abstraction, which allows the designer to concentrate on the specifics of the application rather than having to deal with low-level issues. *Simulations* are today an essential tool for research and training in areas such as national defense, where simulations are a cheaper (and sometimes the only possible) alternative. This work has focused on finding design patterns for simulation software and provides an implementation of the discovered patterns in *Erlang/OTP* as something called *behaviours*. The main simulation software in this study is *Sim94 - A concurrent simulator for plan-driven troops*, written in the Erlang programming language at the Computing Science Department of Uppsala University in 1994. This paper also investigates patterns in general as well as the relationship between behaviours and design patterns.

Supervisor: Lennart Öhman
Examiner: Richard Carlsson

Passed:

Contents

List of Figures

1 Introduction

This thesis is part of an effort initiated by the *Swedish Defense* with the purpose of gathering as much information as possible about *Design Patterns*.

The main goal of this work has been to create a catalogue of design patterns for simulations and to provide the implementation of these patterns in the form of Erlang behaviours. As the primary aid in the search for simulation patterns, Sim94 has been used. Sim94 is a concurrent troop simulator written in Erlang at the Computing Science Department of Uppsala University in 1994.

This report provides the following results:

- An investigation of behaviours and design patterns in general.
- A comparison of behaviours versus published design patterns.
- The implementation of one published design pattern as a behaviour.
- A catalogue of design patterns for simulations.
- The implementation of the design pattern catalogue as behaviours.
- A programming manual for the new behaviours.

The report is divided in two major parts. The first part contains information about Design Patterns, Simulations and Erlang/OTP. The second part contains the actual results.

All code is developed for Erlang/OTP (R6B) on a Linux machine running RedHat 6.2. This report is written in Emacs [1], using the LaTeX typesetting system. [2]

I would like to thank my supervisor Lennart Öhman at Sjöland & Thyselius Telecom AB, my examiner Richard Carlsson at the Computing Science Department of Uppsala University, and especially the people responding to my questions sent to the *erlang-questions@erlang.org* mailing list.

2 Design Patterns

2.1 Introduction

Design patterns are gaining more and more acceptance in the software community and are becoming an important building block in modern software development[1]. A design pattern is an abstract solution to a recurring problem in a specific domain. Design patterns capture good existing practice by documenting the assumptions, structure, dynamics and consequences of a design decision. The primary purpose of a pattern is to communicate design insights.

The general picture that most people have is that design patterns are an object-oriented thing, which only is applicable in object-oriented designs and languages. This is due to the fact that most of the work done in this field has been done using object-orientation. Even the definition of design patterns proposed in *Design Patterns* [3] talks about communicating objects and classes. However, design patterns are not restricted to the object-oriented paradigm and can be used with equally good results in any other context. The original definition of patterns proposed by Christopher Alexander has nothing to do *(as you will see in section 2.2 on page 6)* with object-oriented programming.

This section starts with the history of design patterns and then goes on to explain what design patterns really are and ends with a discussion about why every programmer should learn and use them.

2.2 The History of Design Patterns

Design patterns originate from the work of an architect named Christopher Alexander. In the late 1970s, he wrote two books about patterns for urban planning and building architecture, *A Pattern Language* [4] and *A Timeless Way of Building* [5]. Alexander wanted to create structures that are good for people and have a positive influence on them by improving their comfort and their quality of life. Christopher Alexander asked himself,

> *What is present in a good quality design that is not present in a poor design? What is present in a poor quality design that is not present in a good design?*

Alexander studied these questions by making many observations of buildings, towns, streets and other aspects of living spaces that human beings have built for themselves. He discovered that, for a particular architectural creation, good constructs had things in common with each other. He named these similarities *patterns* and defined them as *a solution to a problem in a context*:

> *Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over; without ever doing it the same way twice.*

[1] Industrial Experience with Design Patterns,
http://www1.belllabs.com/user/cope/Patterns/ICSE96/icse.html

The four components required in every pattern description, according to Alexander are:

- The name of the pattern.
- The purpose of the pattern, the problem it solves.
- How we could accomplish this.
- The constraints and forces we have to consider in order to accomplish it.

The following is an example of a pattern from Alexander's famous book *A Timeless Way of Building* [5]:

The Courtyard Pattern

A courtyard, which is properly formed, helps people come to life in it.

Consider the forces at work in a courtyard. Most fundamental of all, people seek some kind of private outdoor space, where they can sit under the sky, see the stars, enjoy the sun, perhaps plant flowers. This is obvious.

But there are more subtle forces too. For instance, when a courtyard is too tightly enclosed, has no view out, people feel uncomfortable, and tend to stay away . . . they need to see out into some larger and more distant space.

Or again, people are creatures of habit. If they pass in and out of the courtyard, every day, in the course of their normal lives, the courtyard becomes familiar, a natural place to go . . . and it is used.

But a courtyard with only one way in, a place you only go when you want *to go there, is an unfamiliar place, tends to stay unused . . . people go more often to places which are familiar.*

Or again, there is a certain abruptness about suddenly stepping out, from the inside, directly to the outside . . . it is subtle, but enough to inhibit you.

If there is a transitional space - a porch or a veranda, under cover, but open to the air - this is psychologically half way between indoors and outdoors, and makes it much more simple, to take each of the smaller steps that brings you out into the courtyard . . .

When a courtyard has a view out to a larger space, has crossing paths from different rooms, and has a veranda or a porch, these forces can resolve themselves. The view out makes it comfortable, the crossing paths help generate a sense of habit there, the porch makes it easier to go out more often . . . and gradually the courtyard becomes a pleasant customary place to be.

In the early 90s, some experienced software developers looked at Alexander's work in patterns and wondered if what was true for architectural creations would also be true for software design.

- Were there problems in software that occur over and over again that could be solved in somewhat the same manner?

They concluded that this was certainly the case.

Many people were involved in the study of design patterns in the early 90s, but it was not until 1995 that a book that had great influence on the community was published. The book was called *Design Patterns: Elements of Reusable Object-Oriented Software* [3] and is still the most popular book in the field. The authors of the book are known as the *Gang of Four*.

2.3 What is a Design Pattern?

A general and widely accepted definition of a pattern is as follows:

> A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts. [6]

A pattern is more than just a battle-proven solution to a recurring problem. The problem occurs within a certain context, and in the presence of several competing concerns. The proposed solution involves some kind of structure, which balances these concerns in the manner most suitable for the given context. Using the pattern form, the description of the solution tries to capture the essential insight so that others may learn from it, and make use of it in similar situations. The pattern is also given a name to facilitate the discussion of the pattern.

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, we encounter many problems that have occurred, and will occur again. The question is how we are going to solve them this time. Documenting patterns is one way that you can reuse and share the information that you have learned about how it is best to solve a specific design problem.

To remove every possible misunderstanding of the pattern concept the following discussion will explain several different things that a pattern is not.

2.3.1 A design pattern is not an algorithm

Algorithms and data structures may be used in the implementation of patterns, but algorithms and data structures generally solve more fine-grained computational problems like sorting and searching. Algorithms are usually concerned with optimizing space or time and are rarely concerned with compromises and tradeoffs regarding things like maintainability, adaptability and (re)usability of the design.

2.3.2 A design pattern is not a framework

Design patterns may be used in both the design and the documentation of a framework. A single framework typically contains several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns. Despite the fact that they are related in this manner, it is important to recognize that frameworks and design patterns are two distinctly separate

things: a framework is implemented source code, whereas design patterns represent knowledge and experience about software. Hence, both frameworks and design patterns are abstract designs that solve a given problem.

2.3.3 A design pattern is not an idiom

Idioms are on a much lower level than design patterns and are programming language specific. An idiom describes how to implement particular aspects of components or relationships between them. The most famous publication of idioms is James Copliens book, *Advanced C++ Programming Styles and Idioms* [7]. One example of an idiom is the Counted-Pointer idiom which makes memory management of dynamically allocated shared objects in C++ easier. It introduces a reference counter to a body class that is updated by handle objects. Clients access body class objects only through handles via the overloaded operator→().

2.4 Elements of a pattern

There exist many different formats for the documentation of design patterns. The most widely used format is the one suggested by the *Gang of Four* in Design Patterns [3]. Therefore, a description of what is commonly referred to as GoF-format is presented next:

2.4.1 Pattern Name

The pattern must have a meaningful name since it allows us to refer to the pattern with a single word or short phrase, which simplifies the discussion of the pattern.

2.4.2 Intent

Describes the intent of the pattern, what it does and which design problem the pattern is supposed to solve.

2.4.3 Also Known As

Other known names used for this pattern in the literature.

2.4.4 Motivation

A scenario that illustrates a problem and how the pattern solves it.

2.4.5 Applicability

In which situations one should use the pattern. Typically, this section contains a couple of examples.

2.4.6 Structure

Contains a graphical representation of the different objects in the pattern and the relationships between them.

2.4.7 Participants

The different objects and their respective responsibilities.

2.4.8 Collaborations

How the participants work together in order to perform their responsibilities.

2.4.9 Consequences

Describes both the trade-offs and the positive results of using the pattern.

2.4.10 Implementation

What to think about when you are implementing the pattern: pitfalls, hints and techniques.

2.4.11 Sample Code

Sample code that illustrates how an implementation of the design pattern might look like. The code should be thoroughly explained.

2.4.12 Known Uses

Describes known occurrences of the pattern and its applicability within existing systems. This confirms that it really is a recurring pattern. To make sure that the pattern really is recurring the *Rule of Three* is applied. The rule of three tells the pattern writer that he must show three occurrences of the pattern in three different software applications.

2.4.13 Related Patterns

Relationships between this pattern and others within the same pattern language[2] or system. This includes patterns that this pattern is using and other patterns using this pattern in their implementation. If there exists any similar patterns, they should be described here as well.

2.5 Why use Design Patterns?

For the same kind of reason that you should reuse good code: Benefiting from the knowledge and experience of other people who have put more effort into understanding contexts, forces, and solutions than you have done. Further, patterns can be more reusable than code, since they can be adapted so that you can build software to address particular circumstances that cause you to be unable to reuse an existing component. Tradeoffs and alternatives, which the pattern reader might never discover alone, are already documented.

[2] A pattern language is a collection of patterns but unlike a pattern catalogue it includes rules and guidelines that explain how and when to apply its patterns to solve a problem that is larger than any individual pattern can solve. A pattern language may be regarded as a lexicon of pattern plus the grammar that defines how to put them together into valid sentences.

2.5.1 Improved communication among developers

Typically, when several software developers are discussing various potential solutions to a problem, they can use the pattern names as a precise and concise way to communicate complex concepts efficiently.

2.5.2 Patterns are extracted from working designs

Each pattern is extracted from existing, working designs and not created without experience. The design patterns capture the essence of working designs in a form that makes them usable in future work, including specifics about the context that makes the patterns applicable or not.

2.5.3 Records and encourages the reuse of best practice

This is especially important for helping less-experienced developers produce good designs faster. A collection of design patterns in handbook-form is useful for teaching software engineering. However, a design pattern is not a rule to be followed blindly, but rather it should serve as a guide to the designer or provide alternatives when being applied to a particular situation.

2.5.4 Gives a higher-level perspective on the problem

This is, according to *Pattern Oriented Design* [8], the greatest of the reasons to study design patterns. They mean that it frees you from the tyranny of dealing with the details to early.

2.6 The Dark Side of Design Patterns

There are several things in the world of patterns that are not as positive as the opinions from the previous section. First of all, design patterns have no generally accepted definition and no standardization exists. This leads to, among other things, that misconceptions arise and design patterns written by different authors can be as different as night and day. Secondly, no central organization, to which design pattern proposals can be sent and reviewed by, exists. Neither does a place where programmers can find all published design pattern. As it is today, one pattern can be found in a book, another pattern can be found on an unknown web site and yet another pattern might not even be found even though it exist. Further, the Internet contains many different versions of each pattern, which makes it almost impossible for novices in the design pattern field to know which pattern to use or even if a pattern is correct.

Introducing design patterns to the programmers in a company is a major task. For programmers to take advantage of design patterns they must all learn many patterns and how to use them. It is not sufficient if one of the programmers becomes an expert on design patterns. To take full advantage of the communication speedup among developers everybody involved in the implementation of a system must know the relevant design patterns.

Some people complain about the fact that design patterns must be (re)implemented each time they are used. This is the certainly the case.

However, this is the way it must be since an implementation of a design pattern would loose in generality and thus applicability.

An expert programmer that have been implementing a special kind of software for several years will not gain any advantages by using design patterns since he writes solutions to his problems much faster and better without them. He knows the special requirements and problems within his domain better than anybody else. This is the kind of person that should write design pattern.

Sometimes it is better to use a simpler solution than the appropriate design pattern. [9] investigates the question whether it is useful to design programs with design patterns even if the actual design problem is simpler than that solved by the design patterns, i.e., whether using patterns which overkill the problem at hand is useful, harmful, or neutral. They found that all of this can be the case, depending on the situation.

Design Patterns are often designed for object-oriented languages, i.e. either the implementation section assumes that you are working with C++/Java or the problem itself is occurring only in object-oriented languages.

3 Erlang/OTP

3.1 Erlang - The Programming Language

Erlang is a declarative programming language which was designed for programming concurrent, real-time, distributed fault-tolerant systems. The Erlang programming language has been developed at Ericsson and Ellemtel Computer Science Laboratories [10]. The development started in the early eighties as an investigation of whether modern declarative programming paradigms could be used for programming large industrial telecommunications switching systems. Erlang has ideas from Prolog and the syntax clearly resembles that of an untyped ML. With the recent development of the Internet startup company BlueTail one must say that Erlang really became a success. At the time of this writing BlueTail, which works with Internet traffic management in Erlang, has been sold for over a billion SEK.

Applications written in Erlang are divided into modules. Modules are composed of functions and each function consists of clauses. Functions are either only visible inside a module or exported, i.e. they can also be called by functions in other modules. Different functions can have the same names as long as they have different number of arguments or exist in different modules. This is due to the fact that functions are distinguished by module:name/arity.

```
-module(list).
-export([reverse/1]).

reverse(Lst) ->
   reverse(Lst, []).

reverse([X|Xs], Rev) ->
   reverse(Xs, [X|Rev]);
reverse([], Rev) ->
   Rev.
```

In the example above the module list exports the *reverse/1* function. *reverse/2* is hidden and no other modules are allowed to call it. As you can see in *reverse/2* Erlang supports pattern matching. This means that different clauses are invoked by different inputs. In *reverse/2* above, a nonempty list matches the first clause while empty lists matches the second clause.

Variables in Erlang must start with a capital letter and are assign once variables. This means that once a variable has been assigned a certain value the variable can never change again. This approach makes careless mistakes less frequent. Apart from the basic data types, Erlang also provides tuples and lists. Tuples are of fixed size and are created by writing:

```
Tuple = {Variable1, Variable2, ...}.
```

while lists are of variable length and are created like

```
List = [Variable1, Variable2, ...].
```

The work in a system is done by lightweight processes. The processes communicate with each other by messages passing. A process is started through

```
Pid = spawn(Module, Function, [Arg1, Arg2, ...]).
```

where the arguments describe which function the process should run. The variable Pid now contains the process identifier of the newly created process and messages can be sent to it by writing

```
Pid ! Message.
```

A process can receive messages by using the receive command. Further, a process can use pattern matching to choose which messages to receive. Messages not yet received are stored at the process in a FIFO queue.

```
receive ->
   {message1, From} -> ...;
   {message2, From} -> ...
end.
```

Erlang has features for error handling and recovery like exception handling. When you are programming in Erlang it is possible to change the code of a running program. This feature is very important for real-time systems that are impossible to stop for maintenance.

The characteristics of the Erlang programming language according to Seved Torstendahl [11]:

- Very high-level functional/declarative language
- Symbolic data representation
- Support of massive lightweight concurrency
- Support for distribution
- Permits *tailored-to-fit* fault recovery schemes in distributed systems
- No pointers, no memory leaks
- No fixed sizes of limits
- Easy to interface other software and hardware
- Permits software to be updated while running
- Modular concept for structuring applications
- Easy to create reusable libraries

3.2 OTP - Open Telecom Platform

The Erlang Open Telecom Platform is designed for the development of telecommunication applications. Therefore, OTP supports the requirements of typical telecommunication systems like robustness, smooth software upgrades, distribution and real-time functionality. The purpose of OTP is to reduce the time-to-market.

OTP applications adhere to the structure of fig 1 on page 15. The bottom layer contains commercial operating systems and hardware. The middle layer contains support for telecommunication. The top layer contains applications.

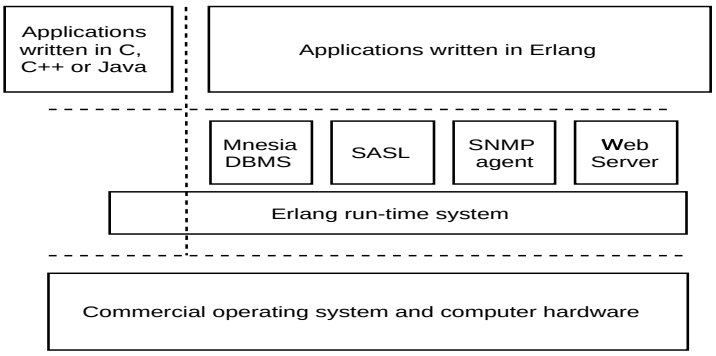


Figure 1: The OTP system architecture

The system architecture provides the following three interfaces: the interface to the OTP software, the interface to the operating system and the interface between applications written in different languages.

OTP provides, among others things, the following set of tools and building blocks:

Erlang The programming language, the compiler, the debugger and the virtual machine. The compiler produces platform independent byte code.

SASL The *S*ystem *A*rchitecture *S*upport *L*ibraries contains the necessary functionality for building fault-tolerant and distributed systems.

Behaviours Formalizations of design patterns that can be used to build new applications.

Mnesia A real-time fault-tolerant distributed database management system.

SNMP Simple Network Management Protocol.

Appmon A program for application monitoring.

C interface generator This tool creates the necessary stubs for communication between C and Erlang.

Coverage tester Makes sure that all code in a module has been executed.

Cross-reference tool To build graphs and find out module dependencies.

Erlang mode for Emacs Helps the programmer by providing syntax highlighting, code formatting etc.

Profiler A profiler is used to detect the "hot spots" in your code.

Kernel This is an application that is running at all times. The Kernel provides facilities like authorization, error logging, rpc, etc.

OS monitoring Services for controlling available disk space and free memory.

Sockets Interface to communication protocols.

3.3 Behaviours

The way in which something functions or operates.

- Webster Online Dictionary

To explain the concept of behaviours lets first introduce something that in Erlang/OTP is called *Applications* [12].

Applications are the building blocks of systems. The idea is that applications should free the user from dealing with the internal details of an application and instead focus on the relationships between them. Applications are named collections of objects that encapsulate the different system components. Each application can be named, loaded and unloaded, as well as started and stopped. Applications must obey certain rules and follow certain protocols in order to provide a uniform interface to the Erlang system. For example, they have to be programmed so that *code change*[3] can take place during runtime. Most of the time, an application is written as a supervision tree. A supervision tree is a hierarchical tree of supervisors and workers. The supervisors supervise the workers, which perform the actual computations. This creates a fault-tolerant system where the supervisors detect errors among their children and restart them whenever necessary.

To relieve the programmer of the burden of always writing code that follow these laws and protocols, he/she can make use of behaviours instead. Behaviours are described as *formalizations of design patterns* by the Erlang/OTP Users Guide [12]. A behaviour solves a particular problem (in a general way so that it can be used in many different contexts). The meaning is that systems can be built by combining several of these behaviours. As a result, all applications will behave in the same way. A behaviour is implemented as a callback module, which means that in order to use a behaviour the user writes a module that exports a specific set of functions, which the behaviour calls in order to complete its task. Applications that are written using behaviours all provide functionality for debugging, handling the termination of a parent, presentation of error information and code change. This makes it easy to use and understand an application written by someone else.

[3] In, for example, the telecom business there exists applications that are impossible to shut down in order to install a new version of the software. For these situations, the Erlang system provides a mechanism that lets the code of an executing program be replaced by a new version.

Lets take a look at the *gen_server* behaviour, which provides a standard way of writing Client-Server applications. The only thing the programmer has to do in order to write a server application is to implement the following callback functions:

- `init(Args)` → `Return`
- `handle_call(Request, From, State)` → `CallReply`
- `handle_cast(Request, State)` → `Return`
- `handle_info(Info, State)` → `Return`
- `terminate(Reason, State)` → `ok`
- `code_change(OldVsn, State, Extra)` → `{ok, NewState}`

When the programmer has implemented the functions above the application automatically follows the required protocols. Another positive effect that the use of behaviours introduces is that the application will be more reliable and have fewer bugs since it is built on top of a framework that has been thoroughly tested and used in several commercial applications.

The Erlang system currently provides the following set of behaviours:

- *application* - defines how *applications* are implemented.
- *sup_bridge* - is used to connect a process to a supervision tree.
- *supervisor* - defines how to write fault-tolerant supervision trees.
- *gen_server* - defines the generic server described above.
- *gen_event* - is used for event handling mechanisms.
- *gen_fsm* - is used for finite state machine programming.

4 Simulations

The technique of imitating the behaviour of some situation or system (Economic, Mechanical etc.) by means of an analogous model, situation, or apparatus, either to gain information more conveniently or to train personnel.

- Oxford English Dictionary

4.1 Introduction

To model a system is to imitate it with something that is simpler and easier to study, and which at the same time is equivalent to the real system in all important aspects. Simulations can be used to gain knowledge about existing systems, to predict the behaviour of some imaginary system and for teaching purposes. One of the key powers of simulation is the ability to model the behaviour of a system as time progresses.

Simulation has become an indispensable technique for system studies and product development in today's industry. This is due to the fact that simulations can be used when it is too difficult, too dangerous, or too expensive to observe the real system. For these reasons simulations are frequently used by military organizations in order to train their soldiers in military leadership[4], teach their pilots how to fly using flight simulators, etc. These activities would otherwise require much personnel and/or be extremely dangerous to perform.

Computers have been used to simulate systems for more than four decades and the field of computer simulations has been the subject of much research. Several different types of simulations have emerged from this research and they will be briefly explained in the next section.

4.2 Different Kinds of Simulations

Depending on the system to be simulated different kinds of simulation principles are applicable. The three most commonly used simulation methods are *Continuous-, Discrete-,* and *Monte Carlo*-simulations. Simulation methods are often classified by how time is treated in the simulation, see figure 2. Continuous and discrete simulations are separated by how they handle time, whereas in Monte Carlo simulations; time is not even a part of the model.

4.2.1 Continuous Simulation

In cases where the systems state changes all the time, not just at the time of some discrete event, continuous simulation is appropriate. In this kind of simulation, the model is often a set of differential equations solved with numerical methods where time is one of the free variables. Continuous simulation generally causes problems with speed, numerical accuracy, and statistical accuracy [14]. These problems come from the algorithms used in the computation of random numbers as well as the algorithms used for solving numerical integrals.

[4]This is the purpose of Sim94 - A concurrent simulator for plan-driven troops. [13]

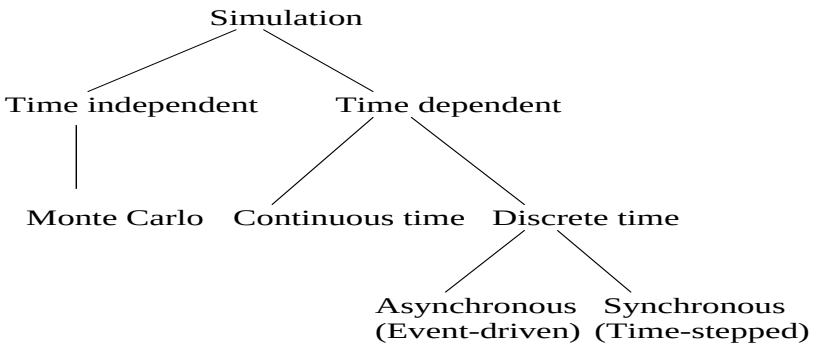


Figure 2: Classification of simulation methods

4.2.2 Discrete Simulation

Discrete simulations are simulations where time is incremented in discrete steps. These simulations can be further divided into two groups, *event driven simulations* and *time stepped simulations*. In event driven simulation, the events that occur within the system control the simulation while in time stepped simulation, the time control the simulation. Of these two, event driven simulation is the more efficient one. When programming discrete event simulations you have a queue of events that are scheduled to happen at certain times in the future. The reason that discrete event simulation is faster is that it lets the time jump in non-uniform steps, that is directly to the next scheduled event. Time stepped simulation on the other hand jumps in fixed size steps and therefore visits times when no events are scheduled to happen. Even though discrete event simulation is more efficient, there are times when time stepped simulation is preferable. Often when a simulation is constructed for teaching purposes, time stepped simulation is the primary choice. In such cases you do not want the time to change in non-uniform steps due to user interaction.

4.2.3 Monte Carlo Simulation

A technique with great impact in many different fields of computational science is called *Monte Carlo Simulation*. This technique derives its name from the casinos in Monte Carlo - a Monte Carlo simulation uses random numbers to model some sort of process. It works especially well when the process is one where the underlying probabilities are known but the results are difficult to determine. Several of the fastest computers in the world are performing Monte Carlo simulations. This is because we can write down the fundamental laws of physics but cannot analytically solve them for problems of interest. One of the most famous examples of a Monte Carlo simulation is the computation of π [15].

4.3 Sim94

4.3.1 Introduction

The following text is a shortened version of the abstract taken from the Sim94 technical report [13].

"Sim94 is a simulation system for plan-driven troops developed at the Computing Science Department, Uppsala University, 1994. Traditionally, military leadership training have consisted of operations involving much personnel. This is expensive and therefore simulation is needed. In order to provide computer-controlled units in military leadership training, Sim94 uses agent-based simulation. An agent is an entity, representing a unit that can make decisions and act accordingly. The agents may interact with other agents, the user, and/or the environment. Sim94 implements a simulator for military leadership training of battalion chiefs. The active entities in the simulation represent troops of 150 infantry soldiers. Sim94 is based on the client-server model, where the server operates independently, i.e. the simulation can go on without any connected clients. Clients are able to connect for inspection and manipulation of the simulation. The agents are controlled by a set of rules and placed in a dynamic terrain where they interact with friendly and hostile troops. The digital terrain is dynamic and can be manipulated during the simulation. The simulator can import geographical data from common GIS formats e.g. ARC/INFO."

4.3.2 Simulation in Sim94

The simulation model that is used differs from the usual framework common for simulations. The usual approach is discrete event simulations where the events control the running of the simulation. Instead, Sim94 is built using a kind of distributed discrete simulation where the active entities are concurrent Erlang processes. This approach forces the simulator to use a central *world* process in order to keep the entities in the simulation synchronized. This module also keeps track of the time and provides a view of the entire simulation for the connected clients to see.

Agents The agents are controlled by a state machine with memory. The memory is necessary to keep track of strength, fatigue, position and other parameters.

A message from the world process containing enemy agents within the reconnaissance radius marks the start of a time interval. After that, the agent first decides what to do. Second, it takes appropriate action, simulating an amount of time equivalent to the length of the interval, to achieve the goals decided upon. Third, it reports the new state, including the new position, to the world process. Finally, it waits for the next interval.

The actions may involve communication with the nationality process to inform the friendly agents about enemies and forbidden areas. If the agent is moving, there is also communication with the terrain process to check that no forbidden areas are entered. If the agent needs movement route planning, the terrain process is consulted.

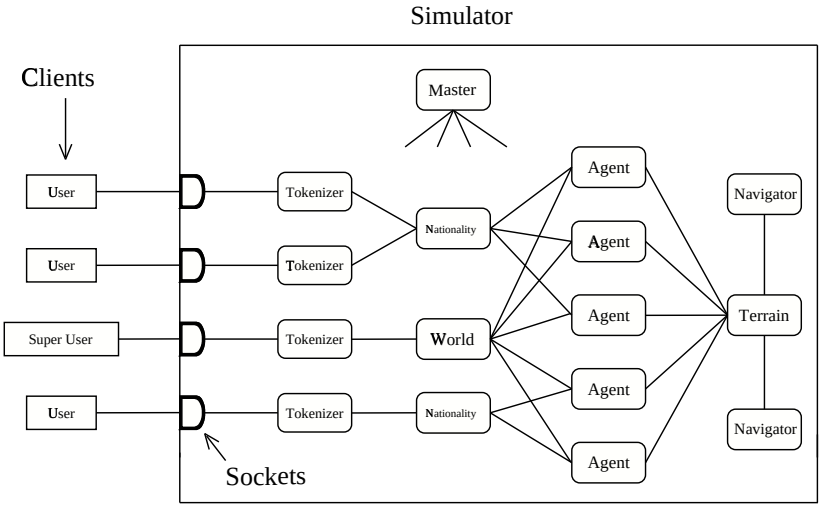


Figure 3: The structure of the Sim94 simulator

World The world process holds a database with records for all agents. This database is used to calculate which agents are in each other's vicinity. The interesting parts of the result are sent to the agents, which marks the start of a new interval. Then the world process collects information from the agents to update the database.

All agents wait until the interval starts and the world process waits until all agents have finished their actions before starting a new interval. This synchronizes the agents. If the simulation is paused, the world process simply refrains from starting a new interval.

If there is an agent attacking another agent, the world process takes care of battle judgment to determine the losses.

Terrain The geographical data is pre-processed to an internal format with the GIS-system GRASS. The terrain is a 25×25 km area divided in 100×100 m squares. Water is approximated only on the edges delimiting the squares. Roads are approximated between the centers of adjacent squares crossing the edges in a straight angle.

The representation of a square contains a terrain type and information about which edges are water and which are crossed by a road. A water edge can only be crossed if there is a road on the same edge. There is no altitude information.

The terrain is represented as a 250-element tuple of tuples where the inner tuples are 250 element tuples of squares. This is a common representation of a two-dimensional array in Erlang.

When the agents move along their routes, the terrain process is frequently queried about the geography along the route.

Route planning requests from the agents are replayed to the navigator processes through the terrain process.

Navigator The navigator processes plan routes from point A to point B given certain constraints. There are time constraints of the form *it may not take more than three hours to go from A to B*. There are also geometric constraints of the form *I want to go no closer to C than 3 km while going from A to B*.

The navigator processes use agent based planning. A search-agent starts at point A and tries to find a safe way to point B that satisfies the constraints. The search is guided in B's direction. In each square local choice is made of where to go next. If the choice is too hard, the search-agent may split in two to explore different alternatives.

If a path is found that satisfies the constraints it is returned to the querying agent, otherwise a "no path found" answer is returned.

Master The master process controls the simulator, creates and destroys processes. If the simulator is distributed the master process is in charge of the distribution.

The master process allocates a UNIX port for client connections and handles the communication with the clients via a socket.

Logging of the system status for robustness is frequently done. A saved status can be reloaded and the system restarted.

Clients and tokenizers The clients are implemented for X windows using Scheme Tk (STk). STk is a scheme interpreter integrated with the GUI toolkit Tk, usually used together with TCL (Tool Command Language). As part of the project, the socket library is interfaced with STk for communication with the simulator.

The tokenizer processes are internal representations of the clients in the simulator. A tokenizer process receives messages from a client. These messages can be requests to e.g. create a new agent, destroy an agent, get information about an agent and issue orders.

A tokenizer process has a view of the simulation, in terms of agents and forbidden areas, given by the world process for the super user and nationality process for normal users.

The clients subscribe to events through their tokenizer process. Events are e.g. new positions for an agent, discovery of a forbidden area and creation of a new agent. The tokenizer process sends a subscription request to the process where the event will occur. When the event occurs the process sends a message to the tokenizer process, which relays the message to the client. This simplifies the client's task, as there is no need to constantly ask for information.

5 Design Patterns versus Behaviours

5.1 Design Patterns compared to behaviours

Something closely related to design patterns are software frameworks:

> *A software framework is a reusable mini-architecture that provides the generic structure and behaviour for a family of software abstractions, along with a context of metaphors, which specifies their collaboration and use within a given domain.* [16]

A framework implements the context, while making the abstractions open-ended by designing them with specific *plug-points*. These plug-points, implemented using callbacks or polymorphism, enable the framework to be adapted and extended to fit varying needs. A framework is *not* a complete application since it lacks the application-specific functionality. Instead, an application may be constructed from one or more frameworks by inserting this missing functionality into the plug-points provided by the frameworks. Thus,

> *A framework supplies the infrastructure and mechanisms that execute a policy for interaction between abstract components with open implementations.* [16]

The definitions above might just as well be about behaviours. When you are using behaviours to program your application, you must implement a bunch of functions in order to specialize the behaviour. This is exactly how you use software frameworks. The difference is that frameworks often are written in object-oriented languages where you use inheritance or implement abstract methods in order to specialize the behaviour of the frameworks.

Behaviours are just as frameworks implemented in code while design patterns are not. Only examples of design patterns can be written down in code. This makes patterns more general than behaviours. On the other hand, one of the strengths of frameworks/behaviours are that they can be reused directly.

Every time you use a design pattern, you have to (re)implement it in a programming language. Thus, behaviours are language dependent (Erlang/OTP) while design patterns are language independent[5].

Both behaviours and design patterns contain well-proven solutions to common problems and are intended to help the programmer to produce better programs faster. To produce better programs faster is actually the main reason behind the development of Erlang/OTP.

When you are using behaviours as well as design patterns in your programs, you get a lot of the documentation for free. The behaviours are rigorously documented and every good pattern description are as well. Design patterns also explain the intent, trade-offs, and consequences of a design.

[5]This is not entirely true since there exists design patterns that are of use only in object-oriented languages. However, most of the existing design patterns are language independent.

A design pattern is a solution to a single problem. Frameworks, and behaviours in particular, contains so much more. Frameworks are often an implementation of one or more design patterns and behaviours gives you functionality for debugging, handling the termination of a parent[6], the ability to change code during runtime and the presentation of error information.

Many design patterns are impossible to implement as behaviours. One example of such a pattern is the Facade pattern [3]. The facade provides a single and simplified interface to a set of interfaces in a subsystem, minimizing the communication and coupling between the client and the subsystem. The structure and content of the facade object is different for each implementation and depends entirely on the specific application. Thus, the code is completely different and no general structure can be extracted to form a meaningful behaviour.

Further, some of the patterns that exist in the literature are only designed for object-oriented languages since they solve object-oriented problems. These patterns have no meaning in a language like Erlang.

5.2 The *gen_fsm* behaviour vs. the *FSM* patterns

A Finite State Machine is a black box that responds to external events. Seen from the outside, the device appears as if it can occupy any of a finite number of states. Depending on the current state, a given input causes the device to issue a particular output and then enter a new state. The change from one state to another is called a transition and there can be several transitions from any state. The events decide which transition to take. Finite State Machines are often described by transition diagrams. The telephone system in figure 4 taken from [12] is an example of a Finite State Machine described by such a diagram. Implementing a Finite State Machine gives the programmer a set of recurring problems to solve. To address these problems, Yacoub and Ammar have developed a design pattern catalogue for FSM's [17]. This catalogue contains 13 patterns and as you will see, the gen_fsm behaviour has made use of most of them in its implementation. This again shows that a behaviour is more than just an implementation of a single design pattern.

The gen_fsm behaviour provides a way of writing Finite State Machine processes. In order to write a FSM the programmer has to write functions for the transitions. These functions must have the following format:

```
StateName(Event, StateData) ->
   ... code for actions here ...
   {next_state, StateName', StateData'}
```

This means that if the FSM is in state *StateName* when it receives the event *Event* it performs some action and makes a transition to the state *StateName'*. If, for example, one were to implement the FSM from figure 4, the function for the idle state would look like,

[6]This is something specific for Erlang/OTP applications structured in a supervisor tree.

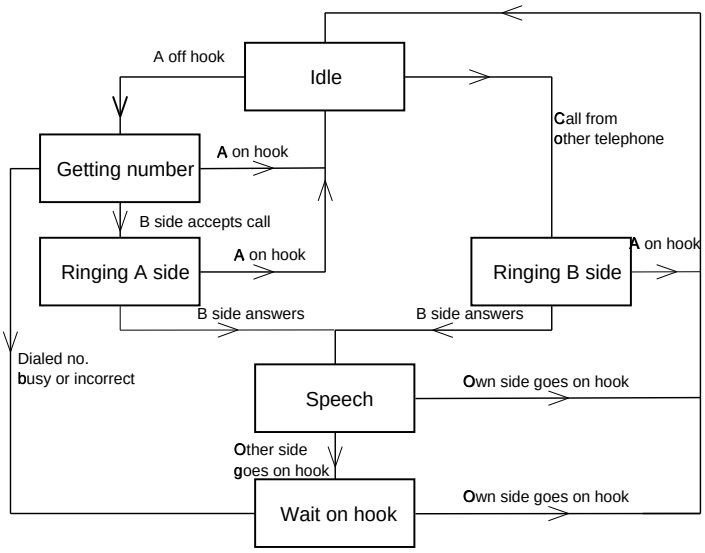


Figure 4: A FSM describing the "Plain Ordinary Telephony Service"

```
idle({off_hook, A}, A) ->
   {next_state, getting_number, {A,[]}};
idle({seize, A}, B) when A /= B ->
   {next_state, ringing_b_side, {B, A}};
idle(_, A) ->
   {next_state, idle, A}.
```

were *getting_number* and *ringing_b_side* are other states represented by the same kind of transition function. Note that the programmer has to write some more functions in order to make the behaviour complete, i.e. *init/1*, *code_change/3* and *terminate/3*, but the *StateName/2* function is the most crucial. The rest of the implementation of the FSM is taken care of by the behaviour. As usual with behaviours, gen_fsm supports the general things that all Erlang applications must support like the presentation of error information etc. *(see section 3.3 on page 16)*

Lets look at how the published FSM patterns fit in this picture. The following section will state each pattern together with a description of it and then explain how/why they are related to the gen_fsm behaviour.

Name: State Object Pattern [3] [18]

Problem: How can you get different behaviour from an object if it changes according to the objects state?

Solution: Create states for the object, describe its behaviour in each state, attach a state to the object and delegate the action from the object to its current state.

Relation: This is exactly what the gen_fsm does. The FSM object has one function for each state that describes its behaviour in means of events, actions and transitions. The FSM has one state associated with it at all times. The difference between this use of the pattern and the use intended by the authors is that they designed the pattern for object-oriented languages in which each state is implemented as a new class.

Name: Basic Finite State Machine Pattern [19] [18]

Problem: Your objects state changes according to events in the system. The state transitions are determined from the object specification. How can you implement the object behaviour in your design?

Solution: Use the state object pattern and add state transition methods in response to state transition events.

Relation: This pattern is also incorporated in the implementation of the gen_fsm. The functions for each state, as described above, also serve as transitions into other states in response to state transition events.

Name: State-Driven Transition FSM Pattern [17] [18]

Problem: How to hide the state transition logic from the object class in the state machine pattern?

Solution: Have the states of the object initiate the transition from self to the new state in response to the state-transition event.

Relation: This problem only occurs in the object-oriented solution where the FSM object holds a reference to the current state class inviting the user to implement the state transition inside the state object. This leads to that every event is processed twice, once in the FSM object and another in the state class. This problem does not occur in the gen_fsm solution since the only place where the programmer handles transitions is in the *StateName* function. These functions actually initiate the transitions to the new states.

Name: Layered Structure FSM Pattern [19] [17]

Problem: You are using a FSM pattern, how can you make your design maintainable, easily readable and eligible for reuse?

Solution: Organize your design in a layered structure that de-couples the logic of state transitions from the object's behavior defined in terms of actions and events.

Relation: The gen_fsm does not follow this design pattern since the *StateName* functions contain actions for the current state in response to a given event and transitions to the next state. However, the gen_fsm behaviour nevertheless has a maintainable design and is easily readable.

Name: Interface Organization FSM Pattern [17]

Problem: How can other application entities communicate and interface to an object whose behavior is described by a finite state machine?

Solution: Encapsulate states and logic inside the machine and provide a simple interface to other application entities that receives events and dispatches them to the current state.

Relation: This pattern is used in the gen_fsm behaviour. States and logic are encapsulated in the module implemented by the programmer and the behaviour exports a set of functions, for other entities to use, which makes up the interface talked about in the pattern.

The FSM catalogue contains other patterns as well, but they must be used or unused by the programmer since they are alternatives to each other. These patterns deal with things like the type of the FSM (Meally, Moore or Hybrid), if the machine should be exposed or encapsulated and if the state instantiation should be static or dynamic. Thus, these design patterns are not possible to implement in a behaviour.

The gen_fsm behaviour made use of five out of six of the patterns that were possible to implement in a behaviour.

5.3 The *gen_server* behaviour vs. the *Client-Server* pattern

The client-server pattern identifies five elements in the pattern: Client, Server, request, reply, and Connection. The client is responsible for generating a request that is sent to the server which, in turn, performs its service and delivers a response in the form a reply. The responsibility for conveying the requests and replies between the client and server is assigned to an intermediary known as the connection. The client and server each collaborate directly only with the connection (but only indirectly with each other). The collaboration between the elements is defined by the sequence of events beginning with the generation of a request by the client and its transmission to the server followed by the generation at the server of a reply and its transmission to the client.

Notice that the pattern does not specify the nature of the service provided, it could be a name service, a time service, a location service, a file service, a security service or any other. Neither does the pattern specify how the connection should be implemented. The connection could be a memory buffer connecting two procedures within the same process, a memory buffer connecting two different processes on the same machine, or a network link between two processes on different machines. While these details vary, the pattern remains the same.

Some of the positive consequences of the client-server pattern are that the client and server may be implemented on different machines allowing each to take advantage of local specialized hardware or software resources, the client and server may be totally or largely unaware of and insensitive to the actual location of each other, and the server may be made available to many clients at the same time. Two negative consequences of the client-server pattern are that the client may be left hanging if its request or reply is lost or if the server

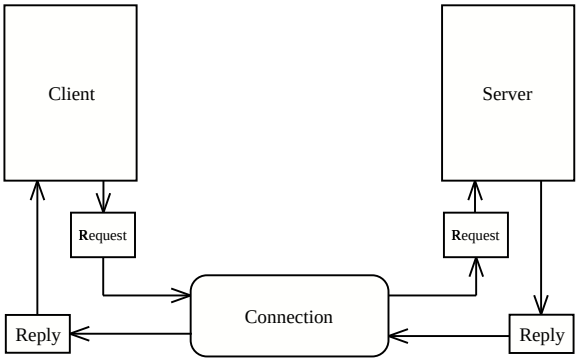


Figure 5: The structure of the client-server design pattern

crashes, and the client cannot demand or control the service from the server - it can only request such service.

The gen_server behaviour provides the general structure for building client-server applications. For a detailed description see section 3.3 on page 17. The gen_server helps the programmer to construct the server in the client/server model. The gen_server does not specify the nature of the service provided since it is general enough to let the programmer implement the functionality specific for the current service.

The implemented server can be called by clients either from the same host or from a host located on the other side of the earth. The user does not have to worry about the specifics of the connection since this is handled by the Erlang/OTP system. The two negative consequences described in the pattern holds for the behaviour as well. However, Erlang/OTP provides functionality to increase the robustness and availability of the server. Nevertheless, hardware failures such as broken wires, are impossible to be completely protected against.

From the above one can conclude that the gen_server behaviour follows the client/server design pattern exactly.

6 A Pattern Catalogue for Simulations

6.1 Introduction

This section presents a catalogue of design patterns to be used in the development of simulation software. The design patterns are mostly for discrete event simulations, but some of them are suitable for other kinds of simulation as well.

This compilation of patterns is taken from the ideas of several authors. Some of the patterns are written specially for simulations, while others are general-purpose patterns [3] that are applicable in the simulation domain. Further, the author himself wrote some of the patterns. These latter patterns are created from the study of Sim94 [13] along with ideas from many simulation researchers, of which Moshe A. Pollatschek [20] and Wolfgang Kreutzer [21] [22] are the two most famous.

The *communicator*, *tokenizer* and *synchronizer* patterns are ideas extracted from the design of Sim94 [13]. The *FSM* patterns, useful for state programming, are written by Yacoub and Ammar [17]. The *time driven, event driven, random event generation* and *tally* patterns are created from general simulation principles [20], [21]. The famous *Gang of Four* patterns [3]; *Command, Observer, Mediator, Singleton, State, Strategy, Template Method* and *Memento* are all useful in the implementation of some simulation aspects. Finally, five design patterns for simulations created by Miguel Vargas [23] are presented in compact form.

The patterns in this catalogue are not presented in the format suggested by the *Gang of Four* in their Design Patterns book [3]. Instead, another format which is closer to the original form proposed by Alexander is used since it is more appropriate for describing non object-oriented design patterns.

6.2 Design Patterns from Sim94

6.2.1 The Communicator Design Pattern

Problem: Communicate the state of each agent in a simulation to every other agent.

Context: When developing software for simulations, a recurring problem is that the agents in the system need to know the state of each other. This pattern gives the programmer an efficient way of communicating the state between arbitrarily many agents in a distributed environment.

Forces: The simple and naive approach would be to have every agent send its state to every other agent, see figure 6. However, if the system consists of n agents, the system will be flooded with $n \times (n-1)$ messages, $O(n^2)$. As the size of the simulation grows, this design can lead to extreme redundancy in storage and computation.

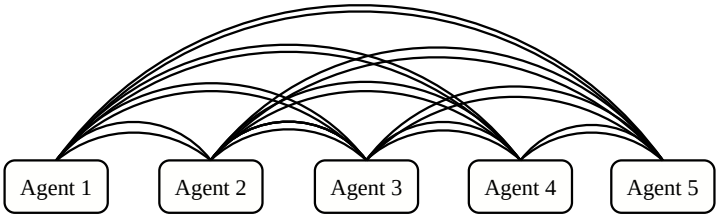


Figure 6: The messages in a system using a naive approach

Further, this approach introduces tightly coupled agents, as each agent is required to know the internal representation of data structures and interfaces to agents of different types. Each agent also has to know the locations of the other agents in order to send and receive state information.

Often, the computations performed on the received states will be the same for each participating agent. Thus, wasting valuable CPU-time.

Solution: The solution introduces a many-to-many dependency that goes through a central node, the communicator. When an agent wants to communicate its state to the others, it simply sends it to the communicator. The communicator collects the states from all the agents and assembles a message containing the state of all agents in the system. Then the communicator broadcasts it to every agent. This pattern may be used in several ways depending on the application. One way is to construct the communicator so that it queries each agent for their states and then distributes the combined state back to the agents. Hence, the communicator is in control. Another way is to create the agents so that they send their states to the communicator whenever they feel like it and the communicator kindly waits until it has received the state from each agent. The communicator

can be configured to send the state immediately to the agents or to perform some computation on the states and pass on the result. The latter approach significantly reduces the computations in the system compared to the alternative of each agent computing this itself.

Example: Two of the modules in Sim94 make use this pattern, i.e. the *world* and the *navigator*.

- Since the agents need to know each other's positions, the world module keeps track of them. At the beginning of each interval the world module calculates which agents are in each other's vicinity, sends a message with the result to each agent and at the end of the interval collects the new positions.
- Information about discovered enemies and forbidden areas is communicated to the friendly agents, which raises the same communication problem as for position updates. The nationality process is created to handle this information. There is one nationality process for each force to which the agents report their discoveries.

Resulting context: This solution requires only $2 \times n$, $O(n)$ messages for a system of n agents, see figure 7. This will drastically decrease the network traffic for systems with many communicating agents. Even though

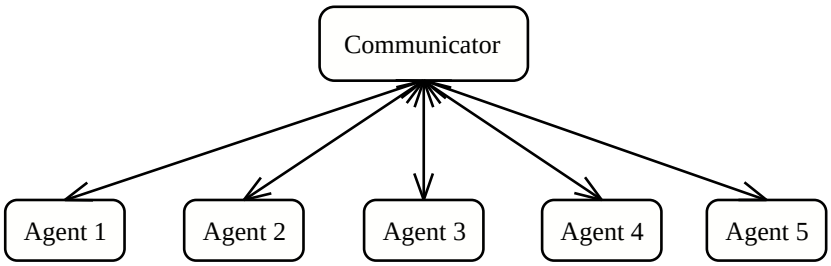


Figure 7: The messages in a system using the communicator

the messages sent from the communicator to the agents are larger than before, this solution will be more efficient since (especially with increasing transmission rates) the overhead of communication (packet assembly/disassembly, error codes, communication setup etc.) is significantly reduced. An additional benefit of the centralized design is its promotion of modularization and information hiding since each agent is not required to know the inner functionality of the others, given that communication takes place only via the central communicator. This pattern is of even more use if the communicator computes results on the collected information that otherwise would have been computed by the individual agents. This approach saves computer resources. However, if the computer where the communicator is placed should be the slower of the involved computers or if it is choked with work, there is no point in using this pattern since the communicator becomes the bottleneck on which the agents must wait.

The major drawback of this pattern is its sensitivity to network failure. If the node where the communicator is located crashes, the whole system goes down. A possible solution to this problem is replication. Working with Erlang gives the programmer the ability to build applications that automatically recovers from crashes.

Related patterns: The observer pattern from the *"Gang of Four"* is similar to the communicator pattern. However, the observer pattern is one-to-many while the communicator pattern is many-to-many. This pattern can be used with great results in conjunction with the synchronizer pattern described in section 6.2.3 on page 34.

Known uses: The communicator is found in the world and navigator modules of Sim94, described above.

6.2.2 The Tokenizer Design Pattern

Problem: Provide a programming language independent interface, which has a minimal interference with the simulation.

Context: In order to control and/or monitor the simulation one needs an interface to the simulation. This is true for all simulations that interact with users. Often, the implementation of a GUI is very closely dependent upon the current platform while the kernel can run (perhaps with minor modifications) on several different platforms. This is because the operating systems on the market use completely different graphical systems. If you are programming for Unix/Linux, your GUI's are written for the X-windows system while in Win95/98/NT interfaces are written using the Microsoft API.

Forces: A GUI must be updated whenever something changes inside the simulation. This can be accomplished in two ways, either the GUI uses polling to check the state of the interesting variables, or the GUI subscribes to events from the simulation. The latter approach is of course the only realistic solution since it requires less resource and we have the observer pattern to use.

The solution must not disturb the simulation. The simulation must continue running whether or not any clients are connected to it. The performance of the simulation must not decrease significantly due to client specific code in the simulation.

As always, one wants to have a clear and decoupled solution that makes it easy to change the involved components. The interface must be programming language independent. This is because the simulation might in the future be part of some larger system that is written in *"who knows which"* programming language. By making the interface programming language independent, no changes are needed in this part in order to communicate with a client written in, for example, Java, C/C++, Ada or even Visual Basic.

Solution: The solution introduces three new components that will take care of the communication between the simulation and any connected clients.

The three components, the Buffer, the Tokenizer and the Socket can be schematically explained as in figure 8. When the GUI wants to communi-

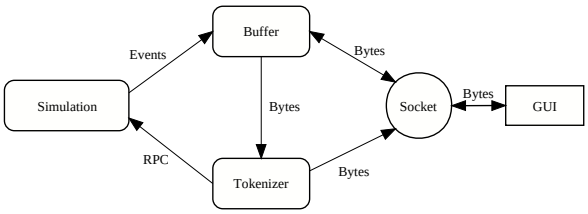


Figure 8: The structure of the Tokenizer Design Pattern

cate with the simulation, it sends the byte representation of the command it wants to execute to the socket. Every programming language *(almost)* has support for socket communication over TCP/IP. The buffer process then reads from the socket and sends this to the Tokenizer process. The Tokenizer converts the byte representation of the command into something that is understandable for the specific programming language used to implement the simulation. In the case of Erlang, the tokenizer converts the byte representation into Erlang messages that will be sent to the simulation. The tokenizer uses synchronized message passing in order to communicate with the simulation. This is almost identical to what is called RPC, i.e. Remote Procedure Call. If the GUI wants to receive updates whenever an event of some type occurs within the simulation the Tokenizer sends a message about this in the same way, as described above, to a subscription mechanism in the simulation, implemented with the observer design pattern. When events occur they are sent to the buffer for immediate transmission to the socket and thus the client. This way the simulation makes no difference between observers within the simulation and clients written in other languages.

Resulting Context:

- Allows use in a heterogeneous network
- Minimal interference with the simulation
- Robust
  - The buffer works even if the tokenizer hangs, which means that events will continue to arrive to the client.
  - No deadlocks (in the buffer process) due to missed messages since we use asynchronous communication where unwanted messages are thrown away.
  - Bad messages from either the client or the simulation are ignored.
- Introduces a lot of overhead compared to the solution of direct communication with the simulation.

Example: The communication between the interface written in Scheme Tcl/Tk and the Sim94 kernel is done through the use of this pattern.

Known Uses: Sim94 [13].

6.2.3 The Synchronizer Design Pattern

Problem: Provide a synchronization mechanism for the agents participating in a simulation.

Context: This pattern is applicable in a simulation where the agents are represented as concurrently executing processes. Synchronization is needed to make sure that;

- the different agents have the same notion of time and
- the agents get to execute the same amount of time.

Forces: When the involved agents are executing concurrently on different machines, the time it takes one agent to perform its task might be completely different than for another agent. This is due to things like, the speed of the CPU, the amount of available memory or the current load of the machine.

The usual way of programming simulation systems is to use a time queue to achieve synchronization. Every simulating entity is simulated one at a time as a discrete event. The first event in the time queue is always executed, (the time queue is sorted by time), and after it is finished it is put back into the time queue at its proper place, given which time its next action is planned to happen. This repeats until the time queue is empty or the simulation is explicitly stopped. This approach is not possible in this context since the agents often interact with each other and therefore sequential execution is out of the question. Moreover, to maximize the utilization of processor power the agents must execute concurrently.

Solution: Agent synchronization is achieved by dividing the simulation time into discrete intervals. These intervals are separated by ticks. A tick is a message sent to all entities, informing them that a new time interval has begun. Between two consecutive ticks, no synchronization of agents occurs. The synchronizer initiates each interval by sending a tick to each entity. An interval is completed when all entities have notified the synchronizer that they have completed their tasks for the current time interval.

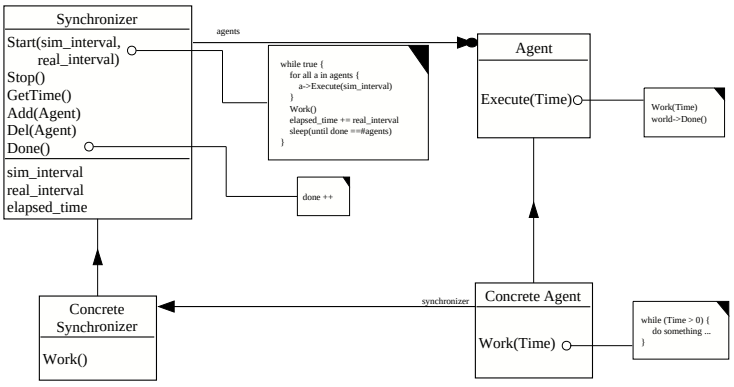


Figure 9: The structure of the Synchronizer Design Pattern

Example: The World module in Sim94 is a synchronizer.

Resulting context: This solution introduces, just like the communicator pattern, a failure sensitive system. See the description of the communicator on page 30 for further details. Another negative effect shows up when this pattern is applied to a distributed environment. Either the agents executing on computers that are slower than the average will not be able to do as much work during their execution, or implemented another way, the speed of the entire simulation will depend on the slowest computer in the simulation.

Related patterns: The synchronizer pattern could be used together with the communicator pattern for increased efficiency. This way, the synchronization could be the state gathering messages and the work performed each step by the synchronizer could be to compute some interesting results on the state of the simulation. This is how the World module of Sim94 is implemented.

Known uses: This pattern is found in Sim94 [13].

6.3 General Simulation Patterns

6.3.1 Time-Driven Execution

Problem: Provide a way of controlling the time and execution of a simulation.

Context: This pattern answers the question of how to implement a Simulation. The solution described has been used for building simulations since the early 50's. The pattern is designed to help novices in simulation programming. The solution is simple and straightforward to implement.

This design pattern can be used to develop simulations where time is part of the model. It is not a pattern for use in Monte Carlo simulations. The pattern should be used in simulations where events are frequently occurring. A time-driven simulation is ideal in the case where we know that at least one event happens at every step.

Forces: Simulations that are interacting with users must use the time-driven execution model since time cannot jump forward to the next scheduled event. This is because the user might give some input (between two scheduled events) that needs an immediate response or action.

Solution: In a "time-driven" simulation, we have a variable that holds the current time, and we increment that time by some fixed amount at every step of the simulation. After each time step, we check all possible event types to determine if an event of that type happened at the current time, and handle it if it does. The simulation stops either when time reaches a specific value (allowing us to answer questions like "how far can we get in that much time", for example) or when a specific state is reached in the system (allowing us to answer questions like "how long would it take to get as far as this", for example). For each time step, we collect statistics from the current state and at the end of the simulation, we use these statistics to compute the results of the entire simulation.

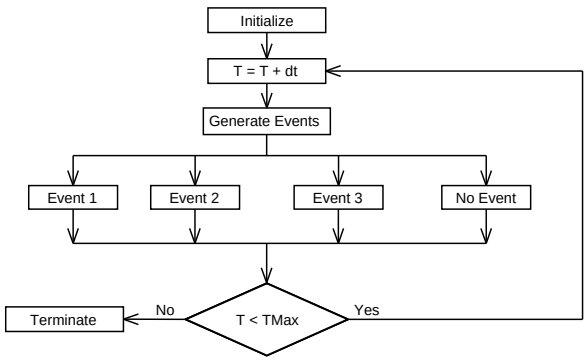


Figure 10: The structure of the Time-Driven Design Pattern

Initialization Initial preprocessing, i.e. state initialization, random number initialization, time initialization and event queue initialization.

Time Keeps track of the time, increments it each step and checks whether it has reached the end or not.

Event generation Creates the necessary events according to the current state and behaviour of the simulation.

Event An occurrence at a point in time that may change the state of the system.

Event queue A queue holding every event that is scheduled to happen.

Statistics gathering Compute statistics on the current state, events and other.

Termination Necessary post processing, computing the results and cleaning up.

Example: The following is a simple example in the C programming language where customers arrives and departs to a simulated store.

```
initialization();
while (clock <= closetime) {
    if (depart_event.time == clock) process_departure(...);
    if (arrival_event.time == clock) process_arrival(...);
    clock = clock + 1;
}
termination();
```

Resulting context: This approach of programming simulations can be extremely inefficient if there are few events occurring in the system. Few events in the system means that most of the time the simulator will just be waiting, thus wasting valuable time. The problem with a discrete time-driven simulation is that it can waste a lot of computing time just advancing the clock tick by tick to the next event. The problem is worse when the events occur infrequently compared to the granularity of the clock. Moreover, the numerical accuracy of this approach is only as good as the size of the time steps. Increasing the time steps leads to faster simulations, but at the same time they become less accurate. If we instead decrease the time steps, the accuracy of the simulation increases, but instead the simulation becomes slower and slower as the size of the time step decreases.

Related Patterns: As an alternative to this pattern, the event-driven execution pattern can be used. When the time between two events is varying, event-driven simulation is preferable since it jumps directly to the time of the next event. Thereby becoming much more efficient.

Known Uses:

HyperReal One (HR1) [24] HyperReal One is an experimental platform for Hard-Real-Time (HRT) systems, which has been developed in the context of the HyperReal project. It is focused on the experimentation of architectural abstractions related to configuration and timing for deeply embedded applications.

A Compiler by IBM for the PowerPC [25] The list-scheduling algorithm executes a time-driven simulation that dispatches instructions in each cycle that are expected to dispatch on the target processor during the equivalent cycle.

DSP Design Toolset The MathWorks Inc. announced a new version of The MathWorks DSP Workshop, a single integrated software environment comprised of MATLAB, Simulink, DSP Blockset, and the Signal Processing Toolbox.

6.3.2 Event-Driven Execution

Problem: Provide a way of controlling the time and execution of a simulation.

Context: We know what we want to simulate and are about to implement our ideas as a simulation. As long as time is part of our model this pattern should be applicable.

Forces:

- The solution must be efficient since we do not want to wait forever for the simulation to complete.
- At the same time the simulation must be accurate and not loose in precision due to the efficient execution.
- The two goals above often interfere with each other.
- Further, the solution must be maintainable and therefore easy to understand and as simple as possible.

Solution: The central concept in this pattern is event scheduling. Occurrences that may directly or indirectly change present or future states are called events. An important implication of this definition of an event is that between two consecutive events, the state does not change. Lets say that the first event occurs at time 10. The next event processed happens to be at time 15, and we would simply just jump to that time because that is what happens next. From a philosophical standpoint, nothing happens in the universe that interests us so we just skip over those times. We make use of a queue to hold events that are going to happen, and we order that queue by time - so that events that are supposed to happen occur in the order they would in the natural system. When we know that predictable events are going to occur, we can insert an event into the queue at the correct point to allow the event to occur at the specified time. As an event occurs, it can spawn other events. These subsequent events are placed into the queue as well. Execution continues until all events have been processed. The purpose of *initialize* is to set up the initial state and generate the initial events. Without those initial events the simulation would terminate immediately. The figure 11 below shows the structure of event-driven simulation.

Resulting context: There is a certain elegance in all this that you will notice when you have a finished simulation running: no matter what the events occurring are or how complex they may be, the simulator itself is simply peeling events off the end of a queue and feeding them to a set of routines

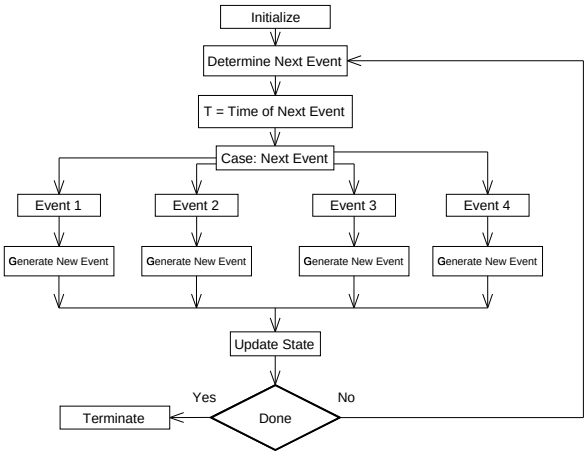


Figure 11: The structure of the Event-Driven Design Pattern

to process whatever the events are supposed to do. Provided those routines insert future events as appropriate, the simulator simply chugs along, and the world naturally follows what the natural system being modeled would do.

- Very precise and accurate handling of the time and therefore also good accuracy in the gathered statistics.
- This approach is very efficient and has just a minimal overhead for the queue operations.
- The fact that the time jumps in non-uniform steps makes this solution inappropriate for certain kinds of simulations that involves user interaction.
- Structuring the simulation according to this pattern makes it easy to modify. Since this is the way that most simulations are implemented following this pattern makes it easy for other developers to quickly understand the code.

Related patterns: To be used in conjunction with event-driven execution the patterns tally, event queue, random events, patterns from Sim94 [13] and patterns by Vargas [23] are given later in this catalogue. The time-driven design pattern is an alternative to this pattern.

Known uses: Event-driven simulation is a popular simulation technique which is the most common way of implementing simulations today. Almost every text-book about simulations teaches event-driven simulation. This is the case in both [20] and [21]. Sjöland & Thyselius have been involved in about 15 large simulation projects in which event-driven simulation have been used almost exclusively. [26]

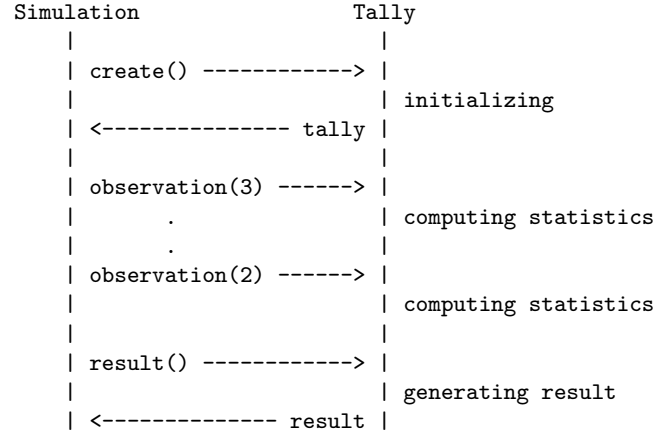
6.3.3 The Tally Design Pattern

Problem: Simulations are often constructed in order to monitor the size/space/number of a certain entity in a system. It could be a store wanting to know how large parking lot they need or a shop wanting to know how much shelf space is needed for their inventories. What we most often are interested in is the minimum, maximum and average values of the monitored entity. Most simulations involve several entities that needs to be monitored. The problem is how to record changes and compute statistics on the entities in a discrete event simulation.

Solution: The solution introduces a tally for each of the monitored variables. A tally is designed to keep track of the changes of a certain variable. A tally provides the three following methods.

- *create()* - This method is called only once and is used to initialize the internals of the tally.
- *observation(value)* - This method is called whenever a change of the variable should be registered.
- *result()* - This method computes and returns the result.

A simulation program creates a new tally by issuing $t = new\ tally()$, and displays the result at the end of the simulation by calling *t.result()*. To record observations during the simulation the programmer simply calls *t.observation(current_value)* repeatedly. The nice thing here is that if the programmer decides that he wants to include a histogram or perhaps compute the statistical variance of the observed entity, changes are needed only to the tally itself. Not a single line in the simulation program needs modification. The following diagram shows the interaction between a simulation and a tally:



Resulting context: Since a tally is called only through the three methods described above, what it computes can change without changing a single line in the simulation. For example, the internals of the tally can change from computing just the min and max of the observed variable to computing the average, the standard deviation or even an histogram of the changes in the variable without affecting the simulation even the slightest. It is

important that the implementation of the tally is efficient since it will be called extremely often when the simulation is executing. The performance of the entire simulation might be dragged down by an inefficient tally implementation. The use of tallies gives the simulation a clean and easy to understand structure. The alternatives often give the simulation a more cluttered look.

Example:

```
/** An example implementation of a tally class that computes
    the minimum, maximum and average of the observed entity.  */

public class tally {

     private double initial_time, last_observation,
                    last_time, min, max, sum;

     public tally() {
         this.initial_time = time(),
         this.last_observation = 0,
         this.last_time = 0,
         this.min = 0,
         this.max = 0,
         this.sum = 0,
     }

     public observation(double value) {
         if (value > max) max = value;
         if (value < min) min = value;
         sum += (time() - last_time) * last_observation;
         last_observation = value;
         last_time = time();
     }

     public result() {
         double time = last_time - initial_time;
         double average = sum / time;
         System.out.println("Statistics for time = " + time + ":");
         System.out.println("Min = " + min +
                            ", Max = " + max +
                            ", Average = " + ".");
     }

}
```

Known Uses: Tallies have been used for decades in the simulation community. In "System Simulation: Programming styles and languages" [21], W. Kreutzer talks about the use of tallies. The SSS simulation library written by Moshe A. Pollatschek and distributed with his book "Programming Discrete Simulations" [20] is another use of tallies.

6.3.4 The Random Event Generation Pattern

Problem: How to organize the creation and scheduling of random events.

Context: Simulation is to imitate the real thing with something that is simpler and easier to study, and which at the same time is equivalent to the real system in all important aspects. Therefore, we need a mechanism that allows events to occur as they would in the real world. This is why random distributions are frequently used in simulations. A distribution is a probability law according to which a random variate is generated. The distribution law states mathematically which values have a greater chance than others in a lottery. In discrete distributions, the random variable can have only integer values; in continuous distributions, any (real) value in a given range is possible. *Binomial* and *Poisson* are discrete distributions; *Beta*, *Gamma*, *Erlang*, *Exponential*, *Weibull*, *Normal*, *Log-Normal*, *Uniform* and *Triangular* are continuous distributions. You have a random generator that acts according to the distribution needed in your simulation. How do you make the events happen according to the distribution?

Forces:

- Creating all the events at once and inserting them into the event-queue makes the queue enormous and thus very memory consuming and slow to work with. If we do not know the duration of the simulation it is even impossible since we are unaware of the number of events needed.
- In a time-stepped simulation we could generate and schedule the events at certain intervals but in event-driven simulations we only visit those times where events occur.

Solution: Create one initial event scheduled to occur according to the distribution of your choice. Then, delegate the job of creating the next event to the function assigned to handle events of this type. The event handling routine handles the arriving event and thereafter calls the random distribution to get the time until the next occurrence. The amount of time returned from the distribution is added to the current simulation time and the event is scheduled according to this. This way, events will happen according to the random distribution.

Resulting context: A minimal amount of work is needed in the event handling routines to accomplish the generation and scheduling of the next event. However, this way the size of the queue of events waiting to happen will be kept to a minimum since only one random event, of each type, will be stored in the queue at any time. This allows for good efficiency and reduced memory consumption.

Known uses: This is the usual way of generating random events in simulation software. Programming Discrete Simulations [20] by Moshe A. Pollatschek teaches simulation according to this principle.

6.4 GoF-Patterns for Discrete Event Simulations

6.4.1 Introduction

This section is put together from *A Case Study: Designing a Discrete Event Simulation* by David Blackledge [27]. It shows how to use several of the GoF-patterns in the implementation of discrete event simulations.

6.4.2 Design Problems

We will examine several problems with discrete event simulation design:

Event Queuing We would like multiple types of events to occur, but we need our events queued in a simple, easy to understand, and easy to process manner.

Multiple Views of Simulation Data Beyond simply seeing a display of all our entities, which we may want in more than one window, we would like to see some statistics information updated with each event as well. All of these views must share the data from the individual watched entities.

Choice of Action Based on Condition and Event Our entities are not simple stimulus-response pair. They will choose the appropriate action that is best for them at the particular time depending on their condition and what is going on around them.

Actions Done independently of Condition and Event Even with the above, the entities can have some particular behaviors that affect actions independently of the current conditions, which may differ between entities.

Allowing Apparent Parallel Action on Events One major issue with this kind of DES is that results from one entity's actions may have bearing on what another entity decides to do, so we want to have all entities act only based on what was true at the end of the last event even though by the time a particular entity gets its turn to act on the new event, many other entities have already acted, changing the current environment.

We discuss these problems in the sections that follow. Each problem has an associated set of goals plus constraints on how we achieve those goals. We explain the goals and constraints in detail before proposing a specific solution. The problem and its solution will illustrate one or more design patterns.

6.4.3 Event Queuing

The backbone of the simulation is the Event Generator. It will create a stream of events that shape how and when the entities act. Each event, however, will require significant processing to occur before it is complete, so we would like to queue the generated events to be processed later. Also, we would like to have multiple kinds of events rather than simple step execution. We would like the queuing to work simply and independently of what events we have in the queue and have it be general enough that we can decide to add more kinds of events to the simulation later on without having to change a lot of code. The way to do this is to allow the queue to work with a generic event object so it does not

need to know what kind of event it is, and new events would be of the same super-class, so the queue would automatically work with them as well.

We can solve this issue with the *Command pattern* [3] which objectifies code, allowing it to effectively be passed around at will. The Event Generator will generate, somewhat at random, different events from the list of commands it has available, and place them in a queue. A separate function will de-queue and execute the commands when processing of previous commands has completed. We might, for instance, have a *NormalEvent* concrete command that simply allows each entity to go about its business for another step. We might also have a *BadEvent* concrete command that, depending what subclass of bad it is, chooses particular entities to affect negatively, and contrastingly, we could have a *GoodEvent* that affects entities positively.

This also allows us to use threading or multiple processes at once: one queuing events, another extracting and executing them.

6.4.4 Multiple Views of Simulation Data

To make the simulation interesting, we would like to watch it as it progresses, viewing areas of the *world* and all of the entities in it. We would also like a list of running statistics to give an overall view of the system. All of these things will need to share the data from any given entity, but will only be interested in a change in that data. They will commonly only be interested in certain entities at a time and within the current area of the world.

Each of our views needs to be updated when an entity they are watching changes. However, since many entities will affect the data of other entities around them, many changes may take place for the same entity during the same event. These complex interactions are such that we would like to update the views only when all the changes have taken place to a particular entity rather than repeatedly updating the view every time the entity changes.

Three patterns make their way into the solution of these problems. The *Observer pattern* [3] is used by each of the views. It allows the subjects being watched to notify the list of observers that a change has taken place so that they may update themselves. Because of the complex interactions causing us to delay the update until all changes have taken place, the *Mediator pattern* [3] can be used to strategize the decision of when the update actually takes place. Finally, we only want one Mediator, and we want it to be available globally to all of the appropriate objects. The pattern that facilitates this is the *Singleton pattern* [3].

6.4.5 Choice of Action Based on Condition and Event

Each entity has a mind of its own, so to speak. An entity decides which its next action will be based on its current state, its goals, and its perception of the world around it. We would like to have changes, in what an entity decides to do next, to be transparent to the rest of the program. This allows requests to an entity to look the same to the requester, no matter what the entity's current state and plans are.

We will solve this problem two-fold by using the patterns *State* [3] and *Strategy* [3]. The State pattern will allow an entity to change its entire set of

functions, while keeping the same interface, by simply changing what State object it currently contains. The State object contains functions oriented towards whatever goal is most appropriate for that state.

The second part of the solution, the Strategy Pattern, allows an entity, within the current State, to choose different methods of meeting its goals. If one method of getting from point A to point B is not working, it can change its Strategy for the next try. Different strategies may be appropriate for different events as well.

6.4.6 Actions Done Independently of Condition and Event

Each entity can have its own, independent "ideals" that it feels it must act on no matter what state it is in or what is going on around it. For example, an entity with a "propensity towards violence" may feel that it necessary to attack every other entity around itself before continuing. However, we would like the entity to have a general form that does not have to cater directly to those ideals. We would like to allow the sub-classes of an entity to decide what to do before and after the requests without directly affecting the code of the super-class.

The perfect solution to this problem is the *Template Method* [3] pattern. Using this pattern says that when you write the main function, you have it execute some otherwise empty operations prior to and/or following (or even in the middle of) the main function's execution. You then allow sub-classes to redefine those two functions, adding any additional personal behaviors they like without changing the definitions of the base functions of movement and action.

6.4.7 Allowing Apparent Parallel Action on Events

During the invocation of an event in the simulation, we would like to keep a copy of the current state of each entity because the next entity that does its work for the same event needs to base its choices on what was true for the first entity when the event began rather than the future state of the first entity. The problem is that we do not want to violate encapsulation when we keep that copy. We should not look at the internals of the entity, but we need to keep the information that is stored there without knowing what it represents.

This is a complex problem, but it can be solved with the *Memento pattern* [3]. With this pattern, we will ask the entity to give us a copy of itself. The entity itself will decide which data it needs to keep track of so it may reinstate itself in the current form. It then places that data in a structure that nobody else knows how to examine. When the entity has finished its actions for the current event, we get another memento so we may give the entity its new state when the entire event has ended. Finally, we give the entity the first memento so it is returned to its pre-event state, allowing the proper actions to be performed by all of the entities that still need to execute for this event. When all entities have done their work, they are all restored with the saved mementos of their newly calculated states and the event execution has completed.

6.4.8 Summary

We have applied eight different GoF-patterns to the DES design:

Command to allow different and easily queueable events, *Observer* to allow multiple views of the same information, *Mediator* to prevent excessive updates of

Observers, *Singleton* to assure a single, global Mediator, *State* to allow actions based on the current state, *Strategy* to allow different options to perform an action, *Template Method* to allow independent behavior attached to actions, and *Memento* to allow apparent parallel actions. There are many, many aspects to the Discrete Event Simulation, and all parts included could incorporate even more design patterns. Many of the ideas applied in this example are also easily applied to numerous other applications. This was only meant as an overview of the kinds of uses one can make of design patterns.

6.5 Design Patterns by Miguel Vargas

6.5.1 Introduction

The following design patterns are taken from a report written by Miguel Vargas titled *Patterns for simulation* [23]. Due to limited space the patterns are given in a more compact form than in the original document. For a more detailed and complete description of the patterns see [23].

6.5.2 Achievable Goal Pattern

Problem How should the goal of the simulation be stated?

We are going to use simulation for measuring the performance of a new software solution. Either we have come out with a new software solution or are willing to figure one out. The system where the solution is intended to run is very large, or not available, or not suitable for making tests on. We might also be dealing with a heuristic solution.

Projects without clear goals continue forever and are eventually terminated when the funding runs out.

Solution Specify clear numbers instead of qualitative words such as low, high, rare, small, etc. Set reasonable and realizable numerical values for the requirements. Apply our experience. Take into account all possible outcomes of the simulation.

Now we have a quantified goal, which we can use for verifying whether the software meets the requirements or not. Based on the goal, we know the scope of our work and can focus on issues relevant to our target. Our simulation has an end point.

6.5.3 Level Of Detail Pattern

Problem How to make an appropriate abstraction of a real system in such a way that its real sensitiveness can be modeled and tested by simulation?

We are going to test a new design solution which is intended to work in a large-scale system. To generalize, we can think of the solution as an algorithm. Generally, simulation is also required when the problem, which is being solved by the algorithm, is just an approximation to the optimal solution. At this stage, we already have identified all the variables that interact in the system and have an idea of their impact on the system behaviour.

- If we include fewer variables in the model, the development is easier but the simulation is less realistic.
- A simulator can be in construction forever, as we can always enhance our simulation in order to approximate it to the reality.
- A more detailed simulation requires more time to be developed.
- A more detailed simulation is more liable to contain bugs, so the debugging time increases.
- A more detailed simulation requires more computer time to run.

- Sometimes, the behaviour of certain variables of the system is unknown and making assumptions is not reliable.

Solution Instead of starting with a detailed model of the system, we start with a less detailed model, get some results, study sensitivities, and introduce details in the areas that have the highest impact on the results. We iterate until the level of detail is such that the variables included in the simulation are enough for accomplishing the goal. [7]

By including the strictly necessary variables we are optimizing the development, debugging, and running time of the simulation.

At this point, it does not matter how realistic our model is as soon as we are modeling all we need for accomplishing our goal.

There is nothing we can do when modeling unknown variables besides making assumptions based on prior experience.

6.5.4 Divide and Simulate

Problem How to structure a large-scale simulation tool?

- Incorrectly structured large-scale simulation tools are highly liable to fail, producing undetectable misleading results. The obtained results are of vital importance for claiming that our solution will be efficient in the real world.
- Simulating real systems is always a complicated task, difficult to maintain and enhance. Real systems change with time and new models of simulation are required. A scalable and maintainable structure is desired.
- The simulation tool must be independent of the algorithm that is being simulated. However, making an algorithm-dependent simulation tool may facilitate its development.

Solution Have one class responsible for generating the framework where the simulation is going to act.

Have another class responsible for managing a list of events but delegate the actions triggered by the events to a set of classes, which model the simulated object.

Delegate the simulation time control to a class dedicated exclusively to keep track of the time.

Have another class dedicated to polling the set of simulated objects in order to obtain their current state periodically.

- *EventScheduler:* Keeps a list of events waiting to happen. The Scheduler allows the events to be manipulated in various ways:
  - Schedule event X at time T
  - Cancel a previously scheduled event X
  - etc...

[7] According to Magnus Sjoland, one of the hardest problems with simulation design is to remove what is unnecessary in order to simplify the reality.

- *SimulationTime:* This class implements the mechanism for keeping track of the simulated time.
- *SimulatedObject:* This class models an independent object of the system and implements the mechanisms that are going to be tested on the simulator.
- *SystemState:* This class keeps track of the SimulatedObjects and implements the methods for consulting the variables of the system.
- *Framework:* This class is responsible for generating the structure and initial conditions of the system. The structure is characterized by the number of SimulatedObjects and the relationship between them. This class participates at the beginning of each iteration of the simulation.

6.5.5 Initial Conditions Pattern

Problem How to choose realistic initial conditions for the simulation?

As experts in the context of the algorithm we are going to test, it is implied that we have at least an intuition of the current conditions of the system but want to make sure that our intuition is a good approximation to the reality.

- The results are more accurate if measured in a stable state of the simulation, that is, when the initial conditions are not affecting the course of the simulation any more; this state is also called the *steady state*. Therefore, we want to initialize the system with the most realistic set of conditions.
- Though the developers of the tool are supposed to have knowledge of the system being modeled, they cannot know for sure how the simulation have to be initialized.

Solution We use the results of previous simulations, if there are any. If it is possible, make measurements in the real world that will help us to figure out the initial conditions of our simulation. Otherwise, we have a brainstorming with the people knowledgeable with the system, and make sure that all of us agree as much as possible with the initial conditions.

If measurements in the real world are possible, the initial conditions will be almost perfect but unfortunately, in many cases there are no such systems, and the best we can do is to compile knowledge from members of the developing team.

Most of the time, we are dealing with huge systems, composed of many independent objects, each in different states, which keeps changing with time. The best we can do is an approximation, which accuracy depends on previous simulations or on the abilities and experience of the designers. Even if we can make measurements in the real world, it could be too difficult (or event impossible) to get a verbatim copy of the real system conditions.

6.5.6 Long Enough Simulation Pattern

Problem How long a simulation has to be in order to get reliable results?

- Short simulations can produce undetectable misleading results. The obtained results are of vital importance for claiming that our solution will be efficient in the real world.
- At the beginning, the simulator starts running with initial conditions, this stage is called *transient state*. Transient state results are not reliable so we need to get rid of them.
- The results are urgent. Moreover, if the simulation is too long, we will unnecessarily be wasting resources.
- If the simulation was initiated with proper initial conditions, the simulation is in a state close to the expected *steady state*.
- Generally, variability of results during the steady state is less than that during the transient state.

Solution Observe the variability of the results. We might assume that when the variability of the results are stable the simulation has reached the steady state. Once the steady state has been identified we can discard all earlier results and only consider the new ones.

If the simulation reaches a steady state, our problem has been solved. Many simulations will never reach a steady state, in which case, it is still unknown when to stop the simulation.

7 The implemented behaviours

7.1 Installing a new behaviour

In order to add a new behaviour to your Erlang/OTP installation a number of things needs to be done. These instructions all assume that you have Erlang/OTP installed in your */usr/local/lib/erlang/* directory.

7.1.1 otp_internal.erl

To extend Erlang with, for example *the observer behaviour*, two of the functions in the *otp_internal.erl* file, located in */usr/local/lib/erlang/lib/stdlib-1.8.1/src*, must be rewritten from

```
behaviour_info() ->
    [application, gen_server, gen_event, gen_fsm,
     supervisor, supervisor_bridge].
```

to the following code where the observer atom is added last in the list,

```
behaviour_info() ->
    [application, gen_server, gen_event, gen_fsm,
     supervisor, supervisor_bridge, observer].
```

and from

```
behaviour_info(application) ->
    [{start,2}, {stop,1}];
behaviour_info(gen_server) ->
 ...
```

to the following code where a new clause is added.

```
behaviour_info(observer) ->
    [{init,0}, {code_change,3},
     {terminate,2}, {aspect,2}];
behaviour_info(application) ->
    [{start,2},{stop,1}];
behaviour_info(gen_server) ->
 ...
```

The first function, *behaviour_info/0*, makes it possible for Erlang to recognize the new behaviour and the second function, *behaviour_info/1*, tells Erlang which functions must be exported from the call-back module using the behaviour.

7.1.2 Where to place the files

To make it possible for Erlang to load the new behaviour, the behaviour specific files must be placed in directories know by the Erlang system.

The source and binary files of the behaviour must therefore be placed in */usr/local/lib/erlang/lib/stdlib-1.8.1/src/* and */usr/local/lib/erlang/lib/stdlib-1.8.1/ebin/*.

7.1.3 Startup scripts

The last change needed before one can start programming with the new behaviour is to add the name of the behaviour to the following two files, located in */usr/local/lib/erlang/bin/*:

- start.script
- start_sasl.script

Now the Erlang/OTP system is capable of loading the new behaviour.

7.2 The Observer as a Behaviour

This is an implementation of the observer design pattern in the form of an Erlang behaviour. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically, see [3] for details. The observer pattern is one of the most known and used design patterns, especially in the implementation of graphical user interfaces.

The observer behaviour provides a standard way of writing the *subjects* [3] from the observer design pattern. The subject is a process with a state which other processes are interested in monitoring. By using the observer pattern to build the subject, *observers* can register with the subject and thereafter receive updates whenever the state of the subject changes.

7.2.1 Exported functions

The observer behaviour provides the following functions:

start(Name, Mod, Aspects) → ignore | {ok, Pid} | {error, Reason}
This function starts an instance of the subject. *Name* is the name by which the subject will be registered. *Mod* is the name of the callback module that has implemented the callback functions described in section 7.2.2 on page 53. *Aspects* is a list containing the different aspects that an observer can subscribe to. The return value is a tuple containing the atom ok, indicating that the behaviour started correctly, along with the *Pid* of that process.

start_link(Name, Mod, Aspects) → ignore | {ok, Pid} | {error, Reason}
Same as the function above except that the calling process will be linked to the subject process. This means that if any of the two processes dies, both will be terminated.

stop(Name) → ok This function terminates the subject process. Before exiting the subject informs each of its observers that it is shutting down. This is done by sending the {*terminating, Name*}-tuple to the observers, where *Name* is the name of the subject. The name has to be included in the message since an observer might subscribe to aspects from several different subjects.

attach(Name, Aspect) → ok Calling this function adds the calling process to the list of processes subscribing on the aspect *Aspect* from subject

Name. After invoking this function the process will receive a notification each time the *Aspect* changes. The subscribing process will now also receive the {*terminating, Name*}-tuple, described above, when the subject dies.

detach(Name, Aspect) → ok When the observer process no longer wants to receive any updates of a given aspect from a given subject, this is the function to call.

notify(Name, Aspect) → ok Whenever this function is called, each process subscribing to the given aspect gets notified. A notification looks like: {*update, Name, Aspect, State*} where *Name* is the name of the subject, *Aspect* is the aspect that has changed and *State* is the changed information. This function should be called by the module using the behaviour.

get_state(Name) → State Returns the current State of the subject process. This state is in the format chosen by the programmer when he implemented the *init/0* function described in section 7.2.2 on page 53.

set_state(Name, NewState) → ok Changes the internal state of the subject named *Name* to *NewState*.

7.2.2 Callback functions

The observer behaviour requires that the following functions are exported from the callback module in order to function properly:

init() → {ok, State} | {stop, StopReason} This function is called once when the subject is starting. The purpose of this function is to create the initial state and perform other initializations required for the particular application. If the initializing procedure fails, the reason is supplied as StopReason with the {stop, StopReason} return value.

terminate(Reason, State) → ok This is called just before the subject is shutting down. Here the programmer should do all the necessary cleaning up. *Reason* contains the reason to why the program is terminating and *State* is the current state of the subject.

code_change(OldVsn, State, Extra) → {ok, NewState} This function is called when a code change is performed, which implies that the internal data structures of the server has changed. This function is supposed to convert the old state to the new one. OldVsn is the vsn attribute of the old version of the module. If no such attribute was defined, the atom undefined is sent. Extra is an optional term which is typically defined in the release upgrade script [28].

aspect(Aspect, State) → StateInfo Whenever the notify function is called the observer behaviour calls the aspect/2 function with the given aspect. The purpose of aspect/2 is to compute the information that the observers, subscribing to the current aspect, should receive. *State* is the internal representation of the state in the subject process. The callback module using the observer module must provide one clause for each aspect it provides. See the next section for details.

7.2.3 How to use the observer behaviour

As said before, the observer behaviour is provided to simplify the implementation of a one-to-many dependency between a subject and a set of observers. This section describes how to, with the help of the observer behaviour, implement a subject and its observers.

7.2.4 Implementing an observer

When the observer behaviour have been used to implement a subject it is extremely easy to write an observer process. Lets say that you want an observer that subscribes to time updates from a clock subject. The following simple code is all that is needed to receive an update every time the clock changes minute:

The first thing an observer needs to do is to attach itself to the subject. This is done by issuing the following command:

```
observer:attach(clock, minute)
```

In order to handle the updates the process needs a loop that receives two kinds of messages. The loop can be written as:

```
loop(State) ->
    receive
        {update, clock, minute, Update} ->
            do something interesting
            ...
            ;
        {terminating, clock} ->
            do something interesting
            ...
    end,
    loop(State).
```

One last thing is needed when the observer no longer wants to receive updates from the subject, and that is to detach itself from the subject:

```
observer:detach(clock, minute)
```

7.2.5 Implementing the subject

Lets say that the people behind "The Search for Extraterrestrial Intelligence" needs to notify people around the world when they eventually discover life in outer space. To accomplish this they can implement a seti-subject like the following, which allows computers around the world to attach themselves and then receive updates whenever contact to outer space is established.

```
init() -> {ok, []}.
terminate(_, _) -> ok.
code_change(_, State, _) -> State.

aspect(hostile_aliens, State) -> {warning, hostile_info(State)};
aspect(friendly_aliens, State) -> {success, friendly_info(State)}.
```

```
start() ->
    observer:start(seti, seti,
                   [hostile_aliens, friendly_aliens]),
    loop().

loop() ->
    case query_radar() of
        hostile_aliens ->
            observer:notify(seti, hostile_aliens);
        friendly_aliens ->
            observer:notify(seti, friendly_aliens);
        _ ->
            loop()
    end,
    loop().
```

7.3 The Communicator as a Behaviour

This is an implementation of the communicator design pattern in the form of an Erlang behaviour. It defines a many-to-many dependency between objects so that objects that need to share state can go through the communicator instead of sending their state directly to all the other agents. Further, the communicator can compute something on the collected states and then distribute the result among the agents. See section 6.2.1 on page 30 for details about this pattern.

The communicator behaviour provides a standard way of writing communicators like the navigator and world modules of the Sim94 simulator.

7.3.1 Exported functions

The communicator behaviour provides the following functions:

start(Name, Mod, Timeout, Agents) → ignore | {ok, Pid} | {error, Reason}
This function starts an instance of the communicator. *Name* is the name by which the communicator will be registered. *Mod* is the name of the callback module that has implemented the callback functions described in section 7.3.2 on page 56. *Timeout* is the number of milliseconds that the communicator will wait for replies from the involved agents. *Agents* is a list containing the different process identifiers of the agents participating in the simulation.

start_link(Name, Mod, Timeout, Agents) → ignore | {ok, Pid} | {error, Reason}
Same as the function above except that the calling process will be linked to the communicator process. This means that if any of the two processes dies, both will be terminated.

stop(Name) → **ok** This function terminates the communicator process. *Name* is the name of the subject and it has to be included since their might be several different communicators in the simulation.

add(Name, Agent) → **ok** Calling this function adds the *Agent* to the list of Agents that are participating in the simulation. *Agent* is the process identifier of the agent that wants to join and *Name* is the name by which the

communicator is registered. This function performs no checks to determine whether or not the agent already has joined. It is left to the programmer to make sure that this is the case. This is for increased efficiency.

del(Name, Aspect) → ok When an agent no longer want to be part of the simulation it can be removed by calling the del/2 function. The arguments are as in add/2 above. This function assumes that the agent to be removed actually is in the list.

go(Name) → ok Whenever this function is called, each agent process receives a request to send their states to the communicator. The Communicator gathers the replies from every agent and then calls the call-back function of the module that is implementing the communicator behaviour. The result returned by that function is sent back to all of the agents.

7.3.2 Callback functions

The communicator behaviour requires that the following functions are exported from the callback module in order to function properly:

init() → {ok, State} | {stop, StopReason} This function is called once when the communicator is starting. The purpose of this function is to create the initial state and perform other initializations required for the particular application. If the initializing procedure fails, the reason is supplied as StopReason with the {stop, StopReason} return value.

terminate(Reason, State) → ok This function is called just before the communicator is shutting down. Here the programmer should do all the necessary cleaning up. *Reason* contains the reason to why the program is terminating and *State* is the current state of the communicator.

code_change(OldVsn, State, Extra) → {ok, NewState} This function is called when a code change is performed, which implies that the internal data structures of the server has changed. This function is supposed to convert the old state to the new one. OldVsn is the vsn attribute of the old version of the module. If no such attribute was defined, the atom undefined is sent. Extra is an optional term which is typically defined in the release upgrade script [28].

compute(AgentStates, State) → Result When the communicator behaviour has collected each of the agents states it calls the compute function in the call-back module. *AgentStates* is a list of {State, Agent}-tuples, one for each agent. *State* is the state used by the callback module to store relevant information initially created by init/0. This function should compute some interesting result based on the *AgentStates*-list and the internal State. The returned result is sent by the communicator to every agent in the simulation.

7.3.3 How to use the observer behaviour

This section describes how to, with the help of the communicator behaviour, implement a communicator and its agents.

7.3.4 Implementing a communicator

```
init() -> {ok, []}.
terminate(_, _) -> ok.
code_change(_, State, _) -> {ok, State}.

compute(AgentStates, State) ->
    compute something interesting with AgentStates
    ...
```

7.3.5 Implementing the agents

An agent that is working with a communicator must receive two special kinds of messages. First of all, the {*get_state, From*}-message must be received. It is a request from the communicator that it wants the agent to send its state. The agent must respond to this message by sending the {*state, self(), State*}-tuple to the communicator. The second message that an agent must receive is the {*set_result, Result*}-message. Where *Result* contains anything that the callback module using the communicator behaviour has computed.

```
loop(State) ->
    receive ->
        {get_state, From} ->
            From ! {state, self(), State};
        {set_result, Result} ->
            do_something interesting with Result
            ...
    end,
    loop(State).
```

7.4 The Tokenizer as a Behaviour

This is an implementation of the tokenizer design pattern in the form of an Erlang behaviour. It defines an interface between a program implemented in Erlang and programs written in other programming languages. For details about the tokenizer design pattern see section 6.2.2 on page 32.

The gen_tokenizer behaviour provides a way of specifying the interaction between an Erlang process and connected clients implemented in other languages. It also provides facilities that enables clients to subscribe on events generated by modules inside the Erlang program by using the observer behaviour, gen_observer.

7.4.1 Exported functions

The tokenizer behaviour provides the following functions:

start(Port, Mod) → ignore | {ok, Pid} | {error, Reason} This function starts an instance of the tokenizer. *Port* is the port through which the communication should take place and *Mod* is the name of the callback module implementing the specific functionality.

start_link(Port, Mod) → ignore | {ok, Pid} | {error, Reason} Same as above except the fact that the calling process gets linked to the tokenizer.

stop() → ok This function terminates the tokenizer process.

7.4.2 Callback functions

The tokenizer behaviour requires that the following functions are exported from the callback module in order to function properly.

init() → {ok, State} | {stop, StopReason} This function is called once when the tokenizer is starting. The purpose of this function is to create the initial state and perform other initializations required for the particular application. If the initializing procedure fails, the reason is supplied as StopReason with the {stop, StopReason} return value.

terminate(Reason, State) → ok This is called just before the tokenizer is shutting down. Here the programmer should do all the necessary cleaning up. *Reason* contains the reason to why the program is terminating and *State* is the current state of the subject.

code_change(OldVsn, State, Extra) → {ok, NewState} This function is called when a code change is performed, which implies that the internal data structures of the server has changed. This function is supposed to convert the old state to the new one. OldVsn is the vsn attribute of the old version of the module. If no such attribute was defined, the atom undefined is sent. Extra is an optional term which is typically defined in the release upgrade script [28].

request(Message, From, State) → {reply, Reply, NewState} | {noreply, NewState} This is the most important function of this behaviour. Its purpose is to handle requests from clients written in other languages. *Message* is a message from a client in string representation. *From* is the pid of the buffer representing the client application. Messages sent to *From* are immediately delivered to the client application. These messages must be on the form {*reply, Message*}. But the real purpose of giving the *From* pid is not to provide a way of sending replies to the client, since this can be done with the return value of this function. Rather, the purpose is that this pid can be given to an observer and this way the client gets updated each time the subject notifies its observers. *State* is the internal state of the callback module. If a response should be sent to the client application this is done by returning the {*reply, Reply, NewState*}-tuple and if no response is required, {*noreply, NewState*} is returned.

7.4.3 Example usage

This example shows how the tokenizer can be used to communicate between Java and Erlang. Lets assume that the a Java client need a way of getting the state of a simulation and a way of telling the simulation to attack its enemies. For this purpose the two messages “get_state” and “attack” are used. The Erlang code looks like:

```
-module(test).
-behaviour(gen_tokenizer).

init() ->  {ok, []}.
terminate(_, _) ->  ok.
code_change(_, State, _) ->  {ok, State}.

start() ->
    gen_tokenizer:start(5678, test).

request("get_state", From, State) ->
    {reply, "state", State};

request("attack", From, State) ->
    attack(),
    {reply, "ok", State};

request(Other, From, State) ->
    {noreply, State}.
```

And the Java code on the client side might look like:

```
socket = new Socket(InetAddress.getLocalHost(), 5678);
out = socket.getOutputStream();
in = socket.getInputStream();

String request1 = "get_state";
String request2 = "attack";

byte reply[] = new byte[100];

out.write(request1.getBytes());
in.read(reply);
System.out.println(new String(reply));

out.write(request2.getBytes());
in.read(reply);
System.out.println(new String(reply));
```

The output created by executing this java client when the tokenizer is up and running is:

```
state
ok
```

7.5 The Synchronizer as a Behaviour

This is an implementation of the synchronizer design pattern in the form of an Erlang behaviour. The gen_sync provides the ability to synchronize a set of agents represented as erlang processes.

The gen_sync behaviour provides a standard way of writing a synchronizer process to which agents can be added and thereafter become synchronized with the other agents in the simulation.

7.5.1 Exported functions

The following functions are exported in order to control the synchronizer.

start(Mod, Agents, Start, Stop, Step, Timeout) → ignore | {ok, Pid} | {error, Reason}
This function starts an instance of the synchronizer. *Mod* is the name of the callback module. *Agents* is a list of pids of the agents that will participate in the simulation. *Start* is the starting time of the simulation, *Stop* is the finishing time and *Step* is the amount of time (simulated time) the agents will execute each interval. *Timeout* is the number of milliseconds that the synchronizer will wait for the agents to finish (real time).

start_link(Mod, Agents, Start, Stop, Step, Timeout) → ignore | {ok, Pid} | {error, Reason}
Same as above, except that the calling function gets linked to the synchronizer.

pause() → **ok** pause/0 puts the simulation to sleep. The synchronizer and the agents will kindly wait for the resume function.

resume() → **ok** resume/0 starts a simulation after that it has been suspended by the pause/0 function .

stop() → **ok** Terminates the simulation. The agents participating in the simulation will now receive the *synchronizer_stopping*-message.

get_time() → **Time** Returns the current simulated time.

set_time(Time) → **ok** Sets the current simulated to *Time*

add_agent(Agent) → **ok** Adds a new agent to the set of agents currently being synchronized. *Agent* is the process identifier of the joining process.

del_agent(Agent) → **ok** Removes an agent from the synchronizer. *Agent* is the process identifier of the leaving process.

7.5.2 Callback functions

Besides the usual *init/0*, *terminate/2* and *code_change/3*, which are implemented as usual, the following functions must be supplied:

work(TimeStep, State) → **{ok, State}** | **{stop, Reason}** *work/2* is called each interval by the synchronizer in order to let the callback module perform some work of its own. *TimeStep* is the length of the interval and *State* is the internal state of the callback module.

timeout(Agents, State) → **{ok, State}** | **{stop, Reason}** If, for some reason, some agents were unable to complete there intervals and the synchronizer timed out, this functions is called with the pids of the unfinished agents and the internal state of the callback module.

7.5.3 Example usage

The agents must receive two kinds of messages, {*execute, From, Period*} and *terminate*. The first message tells the agents to execute their intervals. *From* is the pid of the synchronizer and *Period* is the amount of time the agents are allowed to execute. When an agents has done its work for the current interval it must respond to the synchronizer by sending the {*done, self()*}-message. The terminate message is sent to every agent when the simulation has ended. An example loop might look like the following:

```
loop() ->
    receive
        {execute, From, Period} ->
            do something ...
            From ! {done, self()},
            loop();
        terminate ->
            clean up ...
            ok
    end.
```

The structure of the callback module should look like:

```
-module(sync).
-behaviour(gen_sync).

init() ->
    {ok, []}.

terminate(Reason, State) ->
    ok.

code_change(_ ,State, _) ->
    {ok, State}.


work(Period, State) ->
    do something ...
    {ok, State}.


timeout(Agents, State) ->
    handle the non responding agents in Agents ...
    {ok, State}.


go(Agents) ->
    gen_sync:start(sync, Agents, 0, 10000, 10, 5000).
```

7.6 The gen_sim behaviour

7.6.1 Introduction

The gen_sim behaviour provides a general framework for the construction of discrete event simulations (DES). The gen_sim behaviour is built using the gen_server behaviour and therefore provides all the behaviour specific stuff discussed through out this report. This behaviour is constructed from the following patterns:

- The Event-Driven Execution Pattern
- The Tally Pattern
- The Random Event Generation Pattern

To summarize, gen_sim provides the following simulation specific functionality:

Time: The behaviour handles all aspects of time, i.e. the current time, the elapsed time, the elapsed "real" time and the finish time. It also increments the time according to the events and checks when the simulation has reached the end.

Event handling: The behaviour takes care of events by placing the generated events in an internal queue sorted in time-stamp order. The behaviour then executes the events one after another, inserting new events as they occur, until the queue is empty and the simulation ends.

Tallies: Tallies are a construction for computing statistics on variables in the simulation. Tallies are created for each variable we want to observe. Each time a variable changes we call the tally, which updates the statistics for that variable.

Random distributions: One of the most fundamental concepts in simulation design are random numbers. In order to correctly model phenomenon from the world, simulations make use of different random distributions. The gen_sim behaviour provides the following distributions: *Binomial, Poisson, Beta, Gamma, Erlang, Exponential, Weibull, Normal, Log-Normal, Uniform and Triangular.*

Queues: The behaviour also provides queues that the programmer may use for storing entities and other values. Three different types of queues are supported, i.e. FIFO, LIFO, PRIO.

Statistics: During the execution of the simulation the behaviour gathers statistics from the tallies, the queues, the simulated time and the real time. When the simulation has reached the end, the behaviour prints a statistical report.

The next two sections will describe the functions that must be implemented by the callback module and the functions exported from the behaviour. The last section provides two examples of "real" problems solved using this behaviour.

7.6.2 Exported functions

The gen_sim behaviour provides the following functions:

start(Mod, Start, Stop) → ignore | {ok, Pid} | {error, Reason} This function starts an instance of the simulator. *Mod* is the name of the callback module that has implemented the callback functions described in section 7.6.3 on page 64. *Start* is the starting time of the simulation and *Stop* is the finish time.

start_link(Mod, Start, Stop) → ignore | {global, Pid} | {error, Reason} Same as start/3 except the fact that the calling process gets linked to the simulation.

stop() → ok Immediately terminates the simulation. The simulation produces a statistical report with the finish time set to the current time.

enqueue(Queue, Element, Time) → ok This function adds an element to a queue. *Queue* is the name of a queue constructed in the *init/0* of the callback module. The queue can be any of the types: LILO, FIFO or PRIO. *Element* is any Erlang variable and its place in the queue is decided by the type of the queue. *Time* is the current time and is supplied for statistical purposes.

dequeue(Queue, Time) → Element This functions removes an element from the queue named *Queue*. Which element is removed from the queue is decided by the type of the queue. *Time* is the current time and is supplied for statistical purposes.

queue_len(Queue) → Length Returns the number of elements currently in the queue named *Queue*.

elapsed() → Time Returns the amount of time units that have passed since the simulation begun.

current() → Time Returns the current time of the simulation.

tally(Name, Value, Time) → ok This function corresponds to the tally design pattern and is used to observe a value through the lifetime of the simulation. *Name* is the name given in the *init/0* function of the callback module. *Value* is the current value of the observed variable and *Time* is the current simulated time.

statistics() → ok Calculates a statistical summary of the simulation so far and prints it.

ra() Uniform distribution between 0 and 1.

un(I, C) Uniform distribution between I and C.

ex(M) Exponential distribution with mean M.

tr(I, B, C) Triangular distribution between I and C with mode B.

np(M) Poisson distribution with mean M.

er(M, K) Erlang distribution with K stages and mean K*M.

ga(M, K) Gamma distribution; ga(M,K) same as er(M,K) for integer K.

be(W, U) Beta distribution with mean W/(U+W).

rn(M, S) Normal distribution with mean M and standard deviation S.

rl(M, S) Log-normal distribution with mean M and standard deviation S.

bi(N, P) Binomial distribution with P probability of success in each of N trials.

we(M, U) Weibull distribution; we(M,1) is the same as ex(M).

7.6.3 Callback functions

The gen_sim behaviour requires the following functions to be exported from the callback module in order to function properly:

To use the gen_sim behaviour one must implement (besides *terminate/2* and *code_change/3*, which are implemented as usual) the two functions *init/0* and *handle_event/3*, which are described below:

init() → {ok, {State, Events, Tallies, Queues}} | {stop, StopReason}
This function is called once when the simulation is starting. The purpose of this function is to create the initial state and the different components of the simulation. If the initializing procedure fails, the reason is supplied as *StopReason* with the {*stop, StopReason*} return value. *State* is the internal state of the callback module. *Events* is a list of initial events. The syntax of this list:

```
[{EventName1, Time1}, {EventName2, Time2}, ... {EventNameN, TimeN}],
```

where Time1 ... TimeN are the points in time when the events are scheduled to happen. Next, *Tallies* is a list of tallies, described above. The tallies are created as

```
[{TallyName1, Init1}, {TallyName2, Init2}, ... {TallyNameN, TimeN}],
```

where

```
Init = {sample} | {sample, Value} | {time} | {time, Value}.
```

The *sample* keyword is used for gathering statistics on variables that are time independent and *time* is used, of course, for gathering statistics on time dependent variables. Both of these can be created with or without an initial value. The last thing *init/0* does is to create a list of queues. The syntax of the list is:

```
[{QueueName1, Type1}, {QueueName2, Type2}, ... {QueueName3, Type3}],
```

where

```
Type = fifo | lifo | prio
```

The queues will be subject to statistical calculations, which will be printed in the summary at the end of the simulation.

handle_call(Event, Time, State) | {State', Events} The second function needed by the behaviour is the *handle_event/3* function which is called by the gen_sim behaviour with arguments *Event*, *Time* and *State*. *Event* is the event occurring. *Time* is the current time of the simulation. *State* is the internal state of the callback module. The purpose of this function is to handle events, take appropriate actions and then schedule new events by returning the tuple {*State', Events*} where *State'* is the new internal state and *Events* are a list of events as described for the *init/0* function.

7.6.4 Examples using the gen_sim behaviour

Introduction

This section shows some actual examples of how to implement simulations with the help of the gen_sim behaviour. The examples are taken from *Programming Discrete Simulations* [20] by Moshe A. Pollatschek. These examples are chosen since [20] also provides implementations of the examples, using the SSS-library, which allows us to compare our solution to the solution suggested in [20].

Each section contains a problem, the solution to the problem using the gen_sim behaviour and a discussion about the solution.

Camera

Problem

"Simulate the inventory of a single type of camera at a photo shop. There is a customer for the camera every 2 days on average. The shop gets new cameras every 30 days, so the shop starts each new 30-day period with 15 cameras on the shelf, no matter how many have been sold. Simulate three 30-day periods to evaluate the number of requests that cannot be satisfied and the average stock."

Solution

```
-module(camera).
-behaviour(gen_sim).

init() ->
    {ok, {{15, 0},
          [{customer, 0}, {cameras, 30}, {cameras, 60}],
          [{stock, time}, {unsatisfied, samples}], []}}.

handle_event(customer, Time, {0, U}) ->
    gen_sim:tally(unsatisfied, U+1, Time),
    {{0, U+1}, [{customer, gen_sim:ex(2)}]};

handle_event(customer, Time, {C, U}) ->
    gen_sim:tally(stock, C-1, Time),
    {{C-1, U}, [{customer, gen_sim:ex(2)}]};

handle_event(cameras, Time, {C, U}) ->
    gen_sim:tally(stock, C+15, Time),
    {{C+15, U}, []}.
```

Discussion

As you can see this solution requires no more than 15 lines of Erlang-code while the solution in [20] requires 27 lines of C-code.

init/0 creates the state {C, U}; *C* for *Cameras* and *U* for *Unsatisfied Customers*. These are initialized to {15, 0} since we have 15 cameras to begin with and zero unsatisfied customers. Next *init/0* creates three events; one customer event and two camera events. The first customer is scheduled to arrive at time zero and the camera events are scheduled to occur on day 30 and day 60. The

last thing in *init/0* is the creation of two tallies; stock, which is time dependent, and unsatisfied, which is independent of time. *handle_event/3* has three different clauses. Clause one and two handles the arrival of customers. Clause one handles the case where the cameras in stock are zero in which case we increment the number of unsatisfied customers, record the observation and schedule the next customer to arrive in approximately 2 days. This is an example usage of the *random event generation pattern* from page 6.3.4. Clause two handles the case where we still have cameras in the store, in which case we decrease the number of cameras, record the observation using the *tally pattern* from page 6.3.3 and schedule the next customer in the same manner as clause one. The third clause handles the arrival of new cameras in which case we increase the number of cameras by 15 and record the observation.

This code is then executed by calling *gen_sim:start(camera, 0, 90)* which executes the simulation for 90 days and creates the following output:

```
The simulation took 22.0000 milliseconds.

stock - Statistics for 90 time units:
Average = 11.3412, Standard Deviation = 3.80722
Minimum = 3, Maximum = 22

unsatisfied - No statistics collected!
```

Parking

Problem

"Determine the satisfactory size of a parking lot for a shop, assuming that at peak hours the arrival rate is 30 customers per hour and the customers spend an average of 20 minutes in the shop. Assume each customer arrives in one car and the cars are of approximately the same size."

Solution

```
-module(parking).
-behaviour(gen_sim).

init() ->
    {ok, {0, [{arrival, 0}], [{parking, {time, 0}}], []}}.

handle_event(arrival, Time, C) ->
    gen_sim:tally(parking, C+1, Time),
    {C+1, [{arrival, gen_sim:ex(2)},
           {departure, gen_sim:ex(20)}]};

handle_event(departure, Time, C) ->
    gen_sim:tally(parking, C-1, Time),
    {C-1, []}.
```

Discussion

The solution requires no more than 11 lines of Erlang-code while the solution in [20] requires 49 (!) lines of C-code.

init/0 creates the initial state C, where C is the number of customers in the parking lot. It is initialized to zero. *init/0* then creates the initial event which is a customer arrival, scheduled to happen at time zero. Finally *init/0* creates a time dependent tally initialized to zero for observing the customers in the parking lot. The *handle_event/3* function has two clauses; one for the arrival of new customers and one for the departure of customers. The first clause increments the number of customers in the parking lot, records the observation with the tally function and schedules two new events. The first event is the arrival of the next customer which will be scheduled in approximately 2 minutes. This way customers will arrive in 2 minute intervals which corresponds to the arrival rate of 30 customer per hour. The other event is the departure of the arriving event which is scheduled to occur in about 20 minutes. The second clause handles customer departures. This is done by decrementing the number of customers and recording this observation by using the tally function.

This code is executed by calling *gen_sim:start(parking, 0, 40)* which executes the simulation for 40 minutes and creates the following output:

```
The simulation took 23.0000 milliseconds.

parking - Statistics for 40 time units:
Average = 6.19630, Standard Deviation = 2.08790
Minimum = 0, Maximum = 10
```

8 Conclusion

Design patterns can be a valuable tool in the development of software applications. They capture working ideas, explain the pros and cons of their solutions, and improve the communication between developers. Still, the design pattern community can be improved by better organization and a common standard for pattern writing. Other deficiencies of design patterns includes the relatively large amount of work needed to be fluent in applying patterns, patterns might be overkill when applied to a simple problem, and since they often are written for object-oriented languages it is hard for programmers using other paradigms to adapt the patterns to their languages.

Behaviours are formalizations of design patterns, which means that they are implemented in a programming language. A behaviour can be seen as a framework, i.e. software implementing the context while keeping the specific functionality open by using callback functions. Behaviours are as general as written code can be, but are still less general then the corresponding design pattern. The two behaviours that have corresponding published design patterns, the gen_server and the gen_fsm, follows their patterns very closely. Further, behaviours provide more than design patterns. This includes specific functionality for debugging, code change, presentation of errors, etc. Behaviours significantly increase the speed of software development since they decrease the amount of code that needs to be written and therefore also the number of bugs.

Providing a catalogue of design patterns for simulations can be of great use, especially for novices in the area of simulations. Experts, having implemented simulations for several years, will not gain anything from a catalogue of design patterns. Rather they are the kind of people suitable for writing design patterns. Behaviours however, might be of even more use since they can be used by both novices and experts, to speed up their daily programming.

This report has demonstrated a catalogue of design patterns for simulations. These patterns include four patterns for concurrent agent simulation [13], four patterns for general discrete simulations, several patterns by Miguel Vargas [23] handling everything from how to decide the goals of a simulation to its implementation, and finally a discussion about how to use eight of the GoF-patterns [3] in the implementation of a simulation.

The design patterns for concurrent agent simulations have been implemented as three unique behaviours, i.e. gen_communicator, gen_tokenizer and gen_sync. For discrete event simulations a single behaviour named gen_sim has been developed. Gen_sim provides as much as possible of the corresponding design patterns and makes the implementation of a simulation relatively easy. This report shows two examples of using the gen_sim behaviour, which results in programs with half the size of the same programs written in C using Pollatschek's SSS-library [20].

This report also provides an implementation of the famous observer pattern as the behaviour gen_observer.

References

[1] Richard Stallman. *GNU Emacs Manual, Sixth Edition, Version 18*. Free Software Foundation, 1987.

[2] L. Lamport. *LaTeX: A Document Preparation System. Second Edition.* Addison-Wesley, Reading, Mass., 1994.

[3] Johnson Gamma, Helm and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[5] Christopher Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.

[6] Dirk Riehle and Heinz Zullighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1), 1996.

[7] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[8] Alan Shalloway and James R. Trott. *Pattern Oriented Design: Using Design Patterns From Analysis to Implementation*. 2000.

[9] Walter F. Tichy Lutz Prechelt, Barbara Unger and Peter Brössler. A controlled experiment in maintenance comparing design patterns to simpler solutions. *Fakultät für Informatik, Universität Karlsrube, D-76128 Karlsrube, Germany*, March 9 1998.

[10] Wikström Armstrong, Virding and Williams. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.

[11] Seved Torstendahl. Open telecom platform. Ericsson Review No. 1, 1997.

[12] Ericsson Telecom AB. *Open Telecom Platform User's Guide*, 1996.

[13] Björn Carlsson and Samuel Tronje. Sim94 - a concurrent simulator for plan-driven troops. Technical report, Computing Science Department, Uppsala University, 1994.

[14] I. Mitrani. *Simulation techniques for discrete event systems*. Cambridge Computer Science Texts 14, 1982.

[15] Thomas Huber. Introduction to monte carlo simulation. http://physics.gac.edu/∼huber/envision/instruct/MonteCar.html, 1997.

[16] Brad Appleton. *Patterns in a Nutshell, The 'bare essentials' of Software Patterns*. http://www.enteract.com/∼bradapp/, 1999.

[17] Sherif M. Yacoub and Hany H. Ammar. Finite state machine patterns. Technical report, Computer Science and Electrical Engineering Department, West Virginia University, Morgantown, West Virginia, WV26506, 1998.

[18] Paul Dyson and Bruce Anderson. *Pattern Languages of program design 3*, chapter 9, page 125. Addison-Wesley, 1998.

[19] Robert Martin. *'THREE-LEVEL FSM', In Jim Coplien, Douglas Schmidt, etc., Pattern Languages of Program Design*, chapter 19, page 383. Addison-Welsey, 1995.

[20] Moshe A. Pollatschek. *Programming Discrete Simulations: Tools for modeling the real world*. R&D Books, Lawrence, Kansas 66046, 1995.

[21] Wolfgang Kreutzer. *System Simulation: Programming styles and languages*. Addison-Wesley, 1986.

[22] Wolfgang Kreutzer. Towards a family of pattern languages for simulation software design. In *Proceedings OOIS'96*, pages 387–400, Berlin: Springer, 1996.

[23] Miguel Vargas-Martin. Patterns for simulation. Technical report, School of Computer Science, Carleton University, mvargas@scs.carleton.ca, December 1999.

[24] Flavio De Paoli and Francesco Tisato. Architectural abstractions and time modeling in hyperreal. Seventh EUROMICRO Workshop on Real Time Systems, 1995.

[25] Ibm compiler for the powerpc. http://www.chips.ibm.com/techlib/products/powerpc/manuals/compiler/issues3.html, May 2000.

[26] Magnus Sjöland. Simulering. Lecture for the staff about simulations, 23 August 2000.

[27] David Blackledge. A case study: Designing a discrete event simulation. Internet, 2000.

[28] Ericsson Telecom AB, http://www.erlang.org. *Online documentation of Erlang/OTP*, August 2000.