Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor thesis

Lars Christian Schwensen

Embedded reactive systems - The impact of modern C++ standards on selected design patterns

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lars Christian Schwensen

Embedded reactive systems - The impact of modern C++ standards on selected design patterns

Bachelor thesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 04. April 2016

Lars Christian Schwensen

Thema der Arbeit

Eingebettete reaktive Systeme - Der Einfluss moderner C++ Standards auf ausgewählte Entwurfsmuster

Stichworte

Reaktive Systeme, Eingebettete Systeme, C++11, C++14, Entwurfsmuster

Kurzzusammenfassung

In diesem Dokument wird der Einfluss der Verwendung von mit C++11 und C++14 neu eingeführten Erweiterungen auf die in C++ implementierten Entwurfsmuster Fabrik Methoden Muster und Zustandsmuster untersucht und die Auswirkung von zeitlichem Verhalten, Speicherverbrauch und Code Qualität im direkten Vergleich zu etablierten Implementationsarten festgestellt.

Lars Christian Schwensen

Title of the paper

Embedded reactive systems - The impact of modern C++ standards on selected design patterns

Keywords

Reactive systems, Embedded systems, C++11, C++14, Design patterns

Abstract

Within this document the impact of the use of C++11 and C++14 facilities on the in C++ implemented design patterns factory method pattern and state pattern will be analysed to examine the effect on timing, memory consumption and code quality compared with the established ways of implementation.

Contents

List of Tables

List of Figures

Listings

1 Introduction

Desing patterns provide solutions for many challenges within software development. Embedded reactive systems show many of these challenges regarding execution time, memory consumption or safety critical behavior among others. Many sources offer well designed ways of implementations in C++ for commonly used design patterns. But most of these sources avoid the use of the new C++11 and C++14 standards with no named reason. The ambition of this document is the implementation of the factory method pattern and the state pattern as a representative set of design patterns which are commonly used within the development of embedded software with the use of the new C++ facilities to identify advantages and disadvantages of the use of the new C++ standards and find out whether it is beneficial or not.

1.1 Structure

The document's structure is devided into three main parts. The first part, including chapter 1 to 4, provides a general overview and introduction of the main topics for creating a same basis of knowledge regarding embedded reactive systems, embedded software development as well as design patterns.

Following this, the main body is divided into the factory method pattern and the state pattern. Each of these chapters explains several ways of implementation for both C++03 and C++11 including the particular advantages and disatvantages, an introduction of the used facilities of the new C++ standards and an analysis of the particular ways of implementation.

Chapter 7 and 8 finalize the work by summarize the results of the analysis and providing an opinion and a perspective based on the summary.

2 Embedded Reactive Systems

Our everyday life is supported by electronic components in almosed every aspect. A lot of these electronic components include an embedded reactive system. More than 95% of software systems are actually embedded. (cf. Oshana und Kraeling, 2013, 2) But what are the characteristics of an embedded system and what makes it reactive? Embedded systems are the counterpart of Personal Computer (PC), notebooks and workstations. An embedded system is a computing component which is, for the user almosed invisible, embedded within electronic devices. (cf. Gessler, 2014, 8). Unlike PCs, notebooks and workstations for embedded systems usually exist strict requirements for cost, energy consumption and size. The definition "a reactive system is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing or preventing events in the environment." (Wieringa, 2003, 5) gives an idea of how embedded systems and reactive systems merge to one powerful system. Furthermore a reactive system is defined with a number of characteristics:

- The system is continously interacting with its environment.
- The process by which the reactive system interacts with its environment is usually nonterminating.
- In its interaction with the environment, the reactive system will respond to external stimuli as and when they occur.
- Responses of the reactive system are dependent on the current state of the system and the external stimuli that it responds to.
- The response consists of enabling, enforcing or prohibiting communication of behavior in its environment.
- The behavior of a reactive system consists often of a number of interacting processes that operate in parallel.
- Often a reactive system must operate in real-time and is under stringent time requirements. (cf. Wieringa, 2003, 5-6)

So in one sentence, beeing reactive gives an embedded system the ability to interact with its environment in both directions.

The list of embedded reactive systems is endless. Beginning from the coffee machine in the morning, which receives the information what kind of coffee the user wants and grinds the right amount of coffee beans, over the traffic lights on the way to work which receive the request of passing the streets and stops the cars with a red signal, up to the television in the evening which receives the information of a remote control and switches the channel.

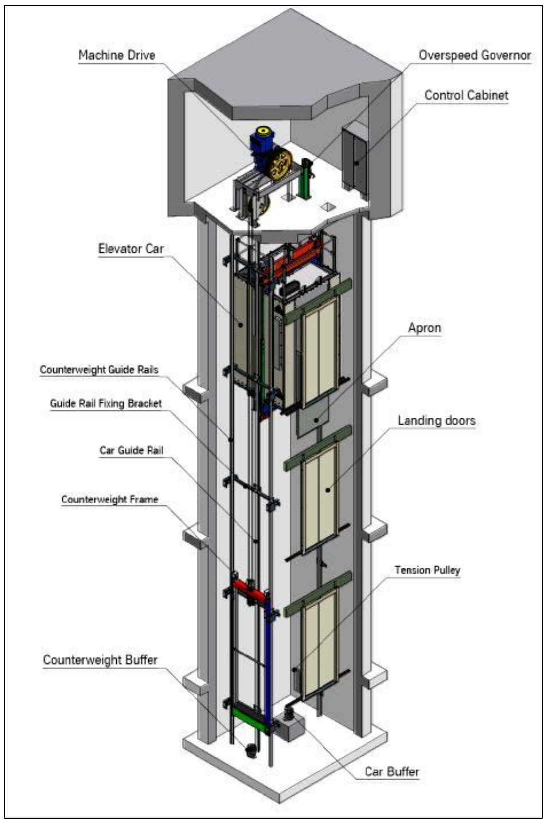

Figure 2.1: Components of an elevator (ElectricalKnowhow, 2013)

A reactive system can be divided into three basic parts. The system itself, the environment and the communication channel. The system is the processing unit which continously keeps the reactive system running. It receives messages and responds to them either internally or by sending messages. The environment is the part of the world which is relevant for the reactive system. For interacting with the environment both, the environment and the system, need an interface. And last but not least the communication channel connects these interfaces and transports the messages between the components. This could be a connection between the interface of the environment and the system as well as between several systems. (cf. Wieringa, 2003, 11-19) The example of an elevator gives a deeper understanding of the components of an embedded reactive system and how they do interact with each other. (Figure 2.1) The environment of the system includes the elevator car and the area around the elevator. Its interface is, of course the panel of buttons inside of the car and maybe a display which displays the current floor and the direction of the car. But the interface of the environment includes much more than just a panel and a display! A number of sensors are permanently observating the environment to detect for example doorblocking objects and other safety-critical aspects. The machine drive is also part of the environment's interface. For the user of the elevator not reachable is the system placed in a control cabinet. It usually contains a hardware interface for receiving and sending messages through the communication channels. This includes the wires to the interface of the environment as well as a communication bus between two or more elevator systems to work efficiently together. The mentioned characteristics of an embedded reactive system have an direct influence on the software architecture of the system.

2.1 Software Architecture

For designing a good software architecture, designers have to consider a lot of rules and principles. One major principle for every software designer is *Keep It Small and Simple*! a simple software architecture is a win situation for all stakeholders. It reduces unnecessary complexity, the maintainability and expandability increases and the sourcecode usually is smaller. Another rule that should be considered for a good design is a balanced expectation of changes. Time brings changes and changes bring new requirements. A software which can be easily changed is a good software. But it is not a good idea to design a "silver bullet" which fits into any possible case. The software designer has to identify the possibility of changes to find a good balance between customizability and fixedness. A third and no less important principle is the quality. Within the designing process the software designer has to consider possible failure situations and pre-define ways to avoid or manage them. (cf. Starke, 2015, 62 - 64) No person would use an elevator which has no strategy implemented in case when the cables break. But does the system also need a strategy in case when the display doesn't work? Probably not. Of course this is just a surficial overview of the considerations within a design process but it gives and idea of the complexity of designing a good software architecture for embedded reactive systems.

During the designing process of an embedded reactive system software- and hadwarearchitects must work hand in hand. This is necessary because within embedded reactive systems the hardware and software take responsibility for core functions of the system. (cf. Starke, 2015, 31 - 32) Usually an embedded reactive system has to meet high requirements. The user of an elevator assumes that brakes are working in the right moment, that the door opens when it should open and stays closed when it should not open. Systems with that kind of constraints in time are called real-time systems.

One concept of structuring the software of an embedded reactive system is the super loop architecture. The super loop architecture is a very common and straightforward way of implementing an embedded reactive system. Basically the super loop architecture can be divided into two phases. In the first phase the system will be initialized. After completing the initialization routines the system enters the second phase, the infinite loop. This loop contains every task and event. It processes each of them one by one and on the end it jumps back to the top of the loop where it starts to process the tasks again. A super loop architecture is only an option when the tasks and events can be processed in predictable time. Adapted variants of the super loop architecture like the power-safe super loop exist to make the task scheduling requirements more consistent with the loop execution time. (cf. Oshana und Kraeling, 2013, 24) As example names Oshana an embedded system with an average loop time of 1ms, which needs to check a certain input only once per second. In this example it is not advisable to execute the loop 999 times before the one time comes where the input is actually relevant. In this case a delay time will be added to the super loop which reduced the execution time to an appropriate level.

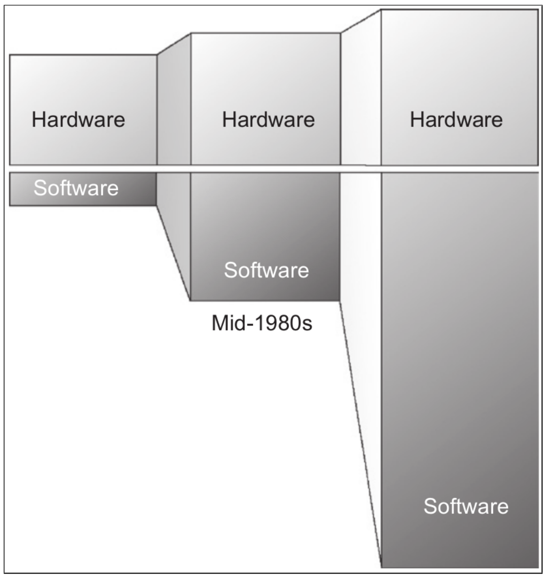

Figure 2.2: Design composition of embedded systems (Walls, 2012, 50)

In the last decades the hardware design of embedded systems has become more complex and the amount of software has increased drastically and takes up to 70 - 80% of the total design effort. (Figure 2.2) (cf. Walls, 2012, 50) The embedded systems model (also known as layered software architecture) (Figure 2.3) is an design concept which can be used for more complex hardware and software designs. The architecture contains three main layer. An hardware abstraction layer (HAL), a system software layer and an application layer. (cf. Noergaard, 2005, 12) Two basic functions of the HAL are decoupling the hardware from the application and providing legible sourcecode. The I/O interface of different hardware components are usually not consistent. Every hardware component brings its own unique interface. The HAL combines and encapsulates the I/O calls within methods. These methods are used by the application layer for work with the hardware.

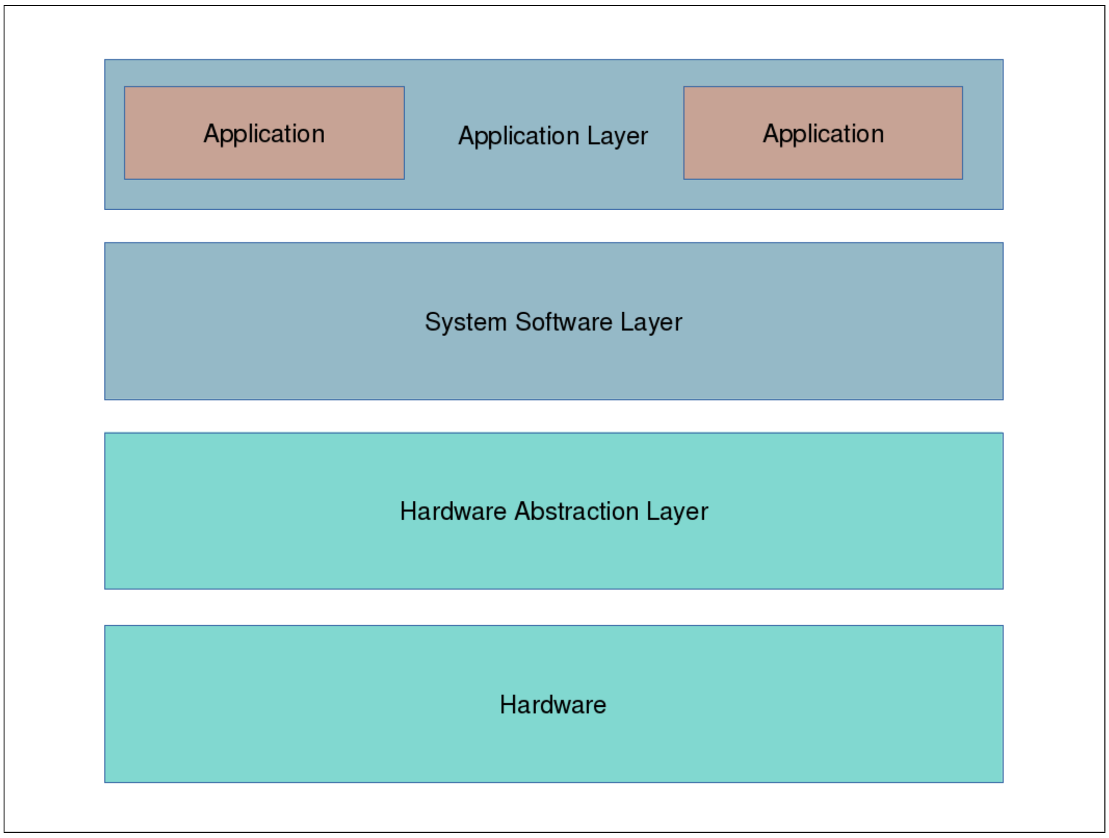

Figure 2.3: Embedded systems model (Noergaard, 2005, 12)

Changes of the hardware require an update of the HAL, but the whole application layer can stay without a single change. Or in other words, the portability of the application software increases. The system software layer allows software designs with more than one process running at the same time. Responsibilities and tasks can be divided and processed in parallel. For this, the system software layer must contain at least a simple scheduler. But also operating systems including ressource- and memory management and channels for interprocess communication are possible.

3 C++ for Embedded Software

The default programming language for embedded software development is C. Typical arguments agaist C++ were an excessive memory use and real-time overhead. The first C++ compiler, which name was Cfront and was written by Bjarne Soustrup, was just a preprocessor to convert the C++ code into C code. (cf. Stroustrup, 2016b) At this time these arguments may have been right, but nowadays the common compiler have improved in quality and functionality. So the arguments against C++ mostly became obsolete. (cf. Walls, 2012, P. 179) In fact, C++ brings a huge number of benefits which can't be named in every single detail. Some important benefits are

- **Object-Oriented Programming (OOP)**
  C++ is an extended variant of the C language. The extensions provide object-oriented programming facilities. The initial version of C++ was called "C with Classes". OOP opens the designer of the software new ways in managing the development process and structuring the sourcecode. As mentioned in chapter 2.1 the amount of software has increased drastically. Using Classes increases the maintainability and the reusability of the sourcecode.

- **Inheritance and Polymorphism**
  In combination with OOP the concept of inheritance allows to define new derived classes from already existing base classes. This increases the reusability of the sourcecode and reduces redundancy. Polymorphism extends the concept of inheritance with the ability of creating pointer pointing at derived classes which are type-compatible with pointer pointing at base classes. Member functions of the base class become virtual and can be redefined by the derived class. (cf. Kirch und Prinz, 2015)

- **Exception handling**
  The programming language C does not provide any facilities to deal with error conditions. The widely used convention of handling a global variable named *errno* has several problems. For example, checking *errno* for each library function call blows up the code and is impractical. The user of a library must be aware of when to check *errno*. C++

provides a simple exception handling. The function of a library detects an error and throws an exception. The exception handler catches the exception and reacts in an appropriate way. Of course the exception handling of C++ is not the solution for every possible problem. In fact, the exception handling was not specifically designed for use with embedded systems. Compilers which support exception handling in C++ tend to create additional code, regardless whether an exception occurs or not. This unexpected overhead could cause other problems for microprocessors with limited memory. (cf. Walls, 2012, P. 200 - 206)

3.1 Modern C++ Standards

C++ is standardized by ISO (International Organization for Standardization) with ISO/IEC 14882:1998 (C++98) as default standard in 1998. In 2003 C++03 replaced with minor changes the C++98 standard and was up to 2011 the relevant standard. in September 2011 the commitee released the ISO/IEC 14882:2011 (C++11) standard including a lot of new facilities for the C++ language which was replaced by C++14 in December 2014. (cf. Stroustrup, 2016a) For an easier distinction from now on applies the keyword C++11 representative for the C++11 and the C++14 standard within this document.

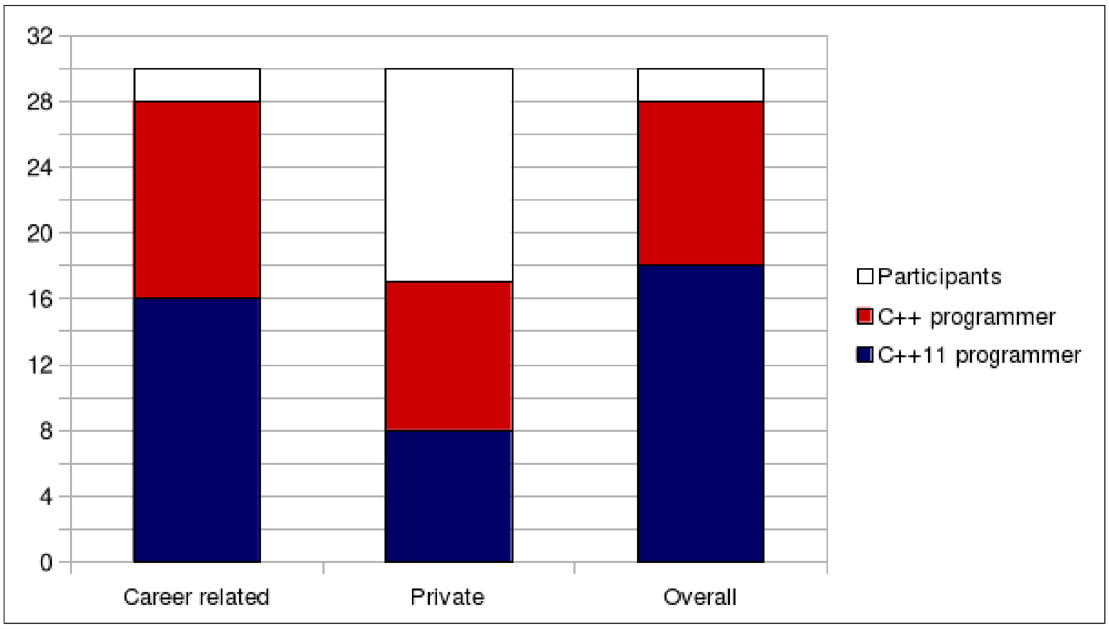

Figure 3.1: Use of C++11 facilities

The use of the C++11 standard within projects is still not common. A survey has shown that within career related software projects more than 40% of the C++ programmer haven't make use of the new C++ facilities yet (Figure 3.1) and within private software projects the less than 50% of the C++ programmer make use of C++11. A survey with 30 participants is not a representative result but it gives an idea of the tendency. Guidelines for C++11 recommend the use of smart pointers instead raw pointers. But not only smart pointers have an impact on the sourcecode. Also the use initializer lists, the type inference with the keyword *auto* and the range-based for-loop iteration changing the look of modern sourcecode. Switching from C++03 to C++11 within a big project could cause a huge workload of refactoring to avoid multiple programming styles. 9 of the 16 C++11 programmer mentioned they would spend working time for updating older sourcecode to the new standards.

4 Design Pattern

One benefit of C++ as an OOP language, which was not mentioned yet, is the suitability of using design patterns within the development process. The book 'Design Patterns: Elements of Reusable Object-Oriented Software' is a basic catalog of useful design patterns and was published by the authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The authors are often just known as Gang of Four (GoF). According to GoF every design pattern describes a problem and the core of the solution for the problem in a way where the solution can be reused million times without redundancy. (cf. Gamma u. a., 2015, 27 - 28) A design pattern contains four characteristics:

- A significant name which refers to problem, solution as well as the effect of the design patterns.
- The problem in which the use of the design pattern might be helpful. This problem defines specific constraints and requirements for architecture, behavior and algorithms.
- The solution for the problem describes an abstract definition of the architectural design for solving the problem.
- The consequences and effects of the usage of the design pattern. This includes advantages and disatvantages in many aspects like memory usage, timing, flexibility, portability, extensibility or reusability. Some of them are measurable and comparable, others are only visible within the work of the software architect.

Usually the use of design patterns effect a combination of advantages and disatvantages, so the software architect must decide whether the use is reasonable or not. As the name suggests, a design pattern is only a pattern. The solution does not describe or suggest a concrete implementation. No special facilities of particular programming languages are involved in the solution of the problem, which allows the use of design patterns for nearly every imperative programming language. In fact, unlike the first assumption of this chapter that the suitability of using design patterns would be a benefit of OOP languages, desing patterns are also available for the use with procedural programming languages like C. The book 'Design Patterns for

Embedded Systems in C', written by Bruce Powel Douglass, offers a deep insight of the use of design patterns for the programming language C. Nevertheless the beneficial connection between OOP languages and design patterns is ubiquitous. Not least because most of the design patterns presented by GoF assume an architectural design with objects. (cf. Gamma u. a., 2015, 29) GoF classifies design patterns in its purpose and its scope. The purpose can be categorized in:

- Creational patterns: with focus on the creation of new objects.
- Structural patterns: define the composition of classes and objects.
- Behavioral patterns: characterize the way of interaction between classes and objects.

In addition, the scope of a design pattern can be categorized in:

- Class based patterns: influence the relation between classes, which is static and already determined at compile time.
- Object based patterns: influence the relation between objects at runtime.

Within the survey, mentioned in section 3.1, the 30 participants were asked to name up to three of their most frequently used design patterns. The result shows a wide variety of design patterns with the singleton pattern as favourite. (Figure 4.1)

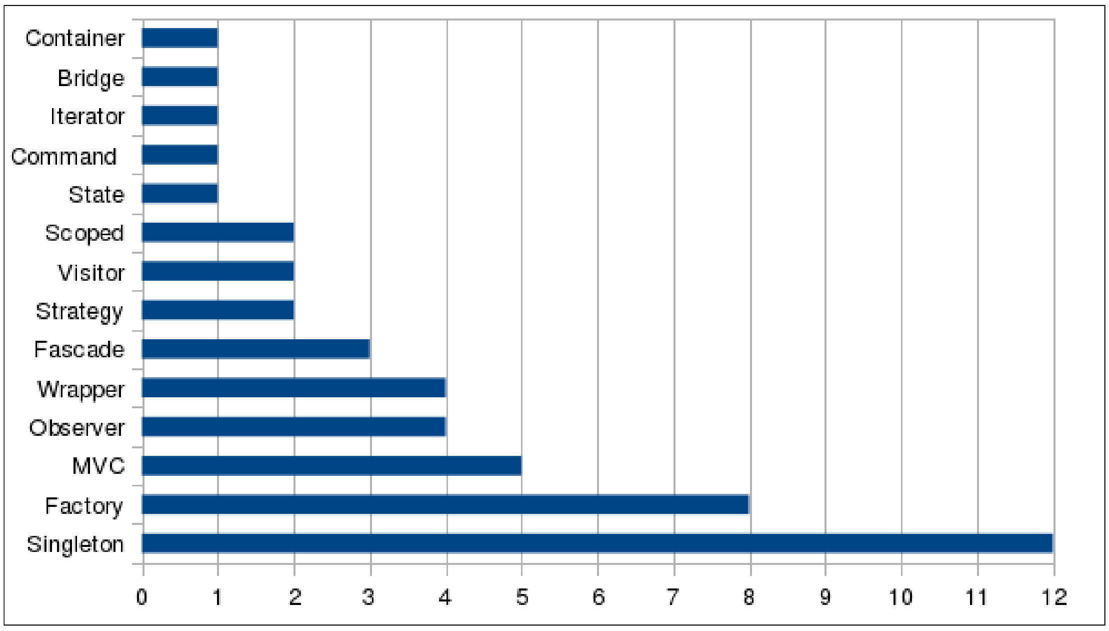

Figure 4.1: Frequently used design patterns

4.1 Design Patterns on Embedded Systems

Design patterns can improve the software architecture of embedded systems in many ways. As mentioned in chapter 2 an embedded system interacts with its environment. In many cases the behavior of the system depends on information received from the environment or other systems. During the development process the designer does only know what kind of messages could come in. Arrival time, occurrences and the order of the messages are unknown. The use of design patterns like the factory method pattern or the strategy pattern provide a more dynamic behavior of the system. Constraints in time or availability of memory require efficient software which can be reached with design patterns like the flyweight pattern. For many more problems and challenges within the design process of an embedded system appropriate design patterns are available.

4.2 Implementation in C++11

Literatur and the internet offer many examples of how to implement concrete design patterns in C++. Few of these examples use C++11 facilities. For example the actual release of GoF from 2015 also includes concrete implementations in C++ and non of them include the facilities released with C++11. A reason for this rarely representation within concrete examples is hard to find. The following selected design patterns shall give an deeper understanding of how does C++11 have an impact on the implementation of design patterns in C++ for finding an answer for the question whether C++11 is beneficial or not.

5 Factory Method Pattern

The factory method pattern is categorized as an creational-, object based pattern. One motivation for using the factory method pattern is when external information influence the demand of objects. During the designing process only the time when an object is needed is known, but not what kind of object. Without the factory method pattern the software designer could implement a conditional statement mechanism like shown in list 5.1 to cover every possible case.

Listing 5.1: Switch-case-selection

```
...
Shape* neededObj;
switch(externalSignal) {
    case 0:
        neededObj = new Triangle();
        break;
    case 1:
        neededObj = new Rectangle();
        break;
    case 2:
        neededObj = new Circle();
        break;
}
...
```

If this selection occurs only once in the whole sourcecode and the number of available objects is small, there is no further demand for refactoring. In projects where the selection occurs on several places, the switch-case-selection can be replaced by a factory call.

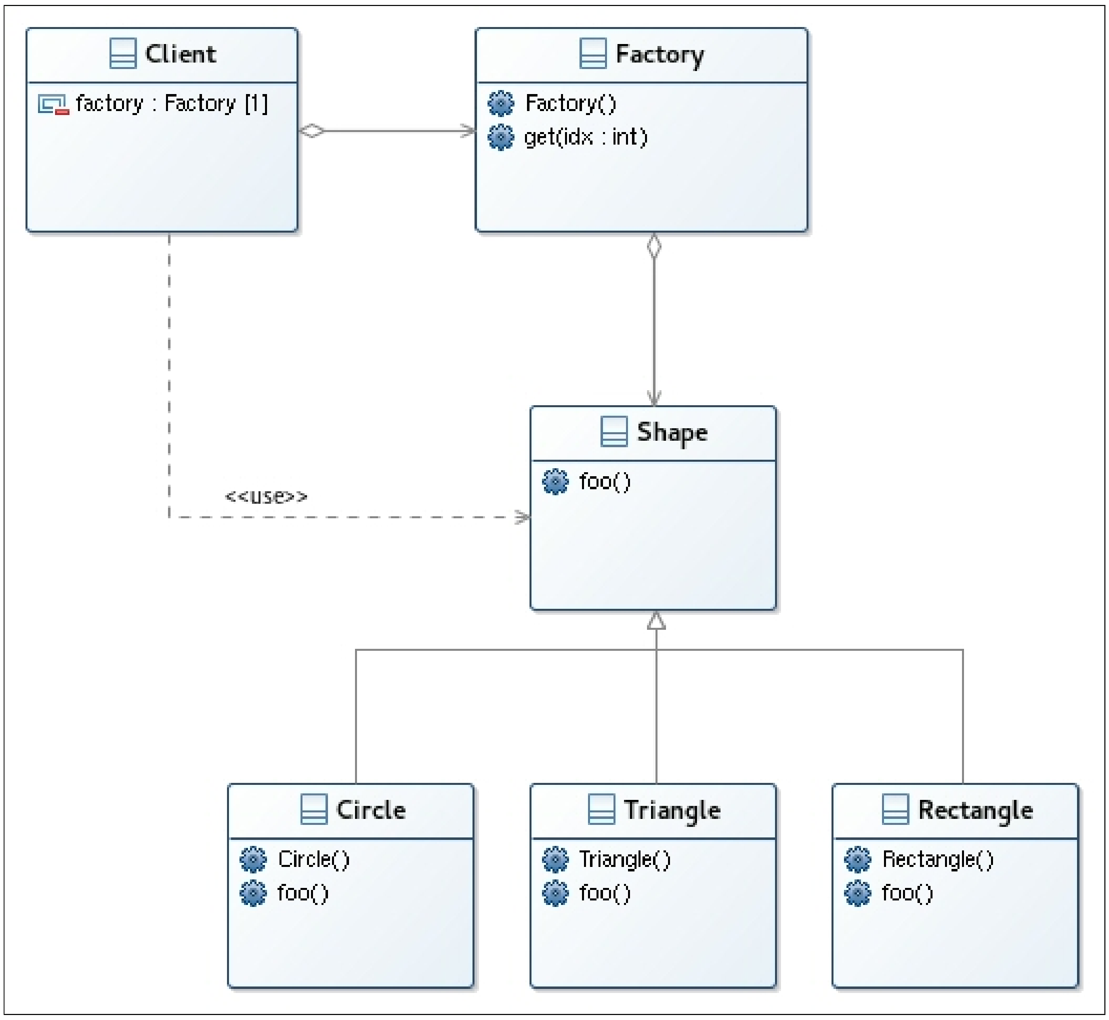

Figure 5.1: Factory method pattern UML

Instead of a direct instantiation, the client delegates the request of an object to the factory. The factory instantiates the particular object and returns it to the client. For the concrete classes either an interface or an (abstract) base class is mandatory like shown in figure 5.1. In this example Shape is the base class of every instantiable sub class. The factory method pattern can be implemented in various different versions. The most straightforward way is probably to move the switch-case-selection, shown in listing 5.1, to the factory.

5.1 Conditional Statement Implementation

Listing 5.2: Conditional statement factory C++03

```cpp
#include "Shape.h"

class Factory {
public:
    Factory() {}
    ~Factory() {}

    Shape* getInstance(int idx) {
        switch (idx) {
            case 0: return new Circle();
            case 1: return new Triangle();
            case 2: return new Rectangle();
            default: return 0;
        }
    }
};
```

This version of a factory has several advantages. Firstly, it is easy to read, to implement and to extend. Another advantage is the performance of a switch-case-statement. Many compilers compile the switch-case-statement to a jump table. The result is a performance of $T(n) = \Theta(1)$ for finding the requested branch. (cf. Ding, 2012) For getting a concrete object, the client uses the factory like shown in listing 5.3.

Listing 5.3: Factory call C++03

```cpp
int main(int argc, char** argv) {
    Factory factory;

    Shape* myObject = factory.getInstance(externalSignal);

    return 1;
}
```

The major disadvantage of this implementation is the hard-coded behavior which provides no flexibility and portability. The factory cannot be used with other classes without recompiling. If a projects needs more than one factory for different classes, each of these factories need to be implemented for its own. The implementation in C++11 of the conditional statement factory is very similar except for the replacement of all raw pointers by smart pointers. (Listing 5.4)

Listing 5.4: Conditional statement factory C++11

```cpp
#include "Shape.h"
#include <memory>

using namespace std;

class Factory {
public:
    Factory() {}
    ~Factory() {}

    unique_ptr<Shape> getInstance(int idx) {
        switch (idx) {
            case 0: return make_unique<Circle>();
            case 1: return make_unique<Triangle>();
            case 2: return make_unique<Rectangle>();
            default: return 0;
        }
    }
};
```

5.1.1 Smart Pointer

As mentioned in section 3.1 guidelines for C++11 recommend the use of smart pointers instead of raw pointers. The main reason for this is the tendency of getting memory leaks with raw pointers. Opportunities for getting memory leaks vary. For example if the software designer allocates memory with new and forgets to call delete before leaving the scope, only the pointer will be deleted. The allocated memory still exists in the heap. Another fatal situation can happen when an exception occurs before calling the delete. The software designer must take care of this situation. (cf. Kirch und Prinz, 2015, 848 - 849) With smart pointers such situations become more safe. Instead of directly allocating memory the user creates a smart pointer which manages the dynamic allocated objects or data types. C++11 provides three significant types of smart pointers. The Unique pointer, shared pointer and weak pointer.

Unique Pointer (unique_ptr)

Example: unique_ptr<int> myUniquePtr = make_unique<int>(42);

The destructor of a unique pointer deletes the managed object, regardless whether another pointer refers to that object or not. Therefore a unique pointer does not have a copy constructer, so that sharing the membership of the managed object is not possible. The only way to pass the membership is by using the move constructor. With the move constructor the current unique pointer loses the membership and the new unique pointer is the new unique membership holder of the managed object. Unique pointer are useful for managing ressources like text files. (cf. Kirch und Prinz, 2015, 850 - 853) The size of a unique pointer is 8 bytes[1] for the reference to the managed object.

Shared Pointer (shared_ptr)

Example: shared_ptr<int> mySharedPtr = make_shared<int>(42);

Unlike unique pointers, share pointers allow to share the membership of the managed object. To guarantee that the managed object will not be destroyed as long as at least one pointer refers to that, and will be destroyed when the last pointer stops refering to the managed object, the shared pointer uses a reference counter. The reference counter increases when a new pointer refers to the managed object and decreases when a pointer stops refering to the managed object. If the reference counter becomes zero, the object will be destroyed. A disadvantage of a shared pointer compared with a raw pointer is an overhead of allocation time because in addition to the memory for the managed object the shared pointer must also allocate memory for the reference counter. The *make_shared<>()* function reduces this overhead by allocating the memory in one block. (cf. Kirch und Prinz, 2015, 854 - 861) The size of a shared pointer is 16 bytes. 8 bytes for the reference to the managed object and 8 bytes for the reference to the reference counter. In addition to this the shared pointer also allocates 8 bytes for the reference counter.

Weak Pointer (weak_ptr)

Example: weak_ptr<int> myWeakPtr = mySharedPtr;

A weak pointer can only manage objects which are already managed by at least one shared pointer. It has no effect on the reference counter. This means if the last shared pointer stops

[1]Every memory size declaration within this document bases on a 64bit system.

refering to the managed object, the object will be destroyed and the weak pointer refers to unallocated memory. Weak pointers are useful when several objects have a direct or indirect reference to each other. In this case the reference counter cannot reach zero. Using weak pointer instead, removes the dependency of these objects. (cf. Kirch und Prinz, 2015, 862 - 863) The size of the weak pointer is 16 bytes. As already mentioned the weak pointer has no effect on the reference counter, but it provides the option for a cast to a shared pointer. In this case it is necessary to know about the reference counter. Therefore the weak pointer contains an reference to the reference counter.

5.1.2 Type Inference with auto

With C++11 not only the factory has changed. A look at the use of the factory (listing 5.5) shows that the explicit type declaration was replaced by the placeholder *auto*. *auto* is part of the generic programming facilities of C++. As long as the compiler can determine the correct data type at compile time, *auto* can be used for type declaration as well as for the declaration as a return-type. (cf. Kirch und Prinz, 2015, 210 - 211) Type inference benefits the maintaining and modifying process of the sourcecode.

Listing 5.5: Factory call C++11

```
int main(int argc, char** argv) {
    Factory factory;

    auto myObject = factory.getInstance(externalSignal);

    return 1;
}
```

5.1.3 Timing

Figure 5.2 compares the implementation of listing 5.2 with listing 5.4. Each factory manages 100 classes. For the instantiation of 10,000 objects the C++03 version needs about 0.72ms and the C++11 version needs about 5.1ms. For the instantiation of 100,000,000 objects the C++03 version needs about 4.3 seconds and the C++11 version needs about 35.9 seconds. The complexity of both versions is constant. $T(N) = \Theta(1)$.

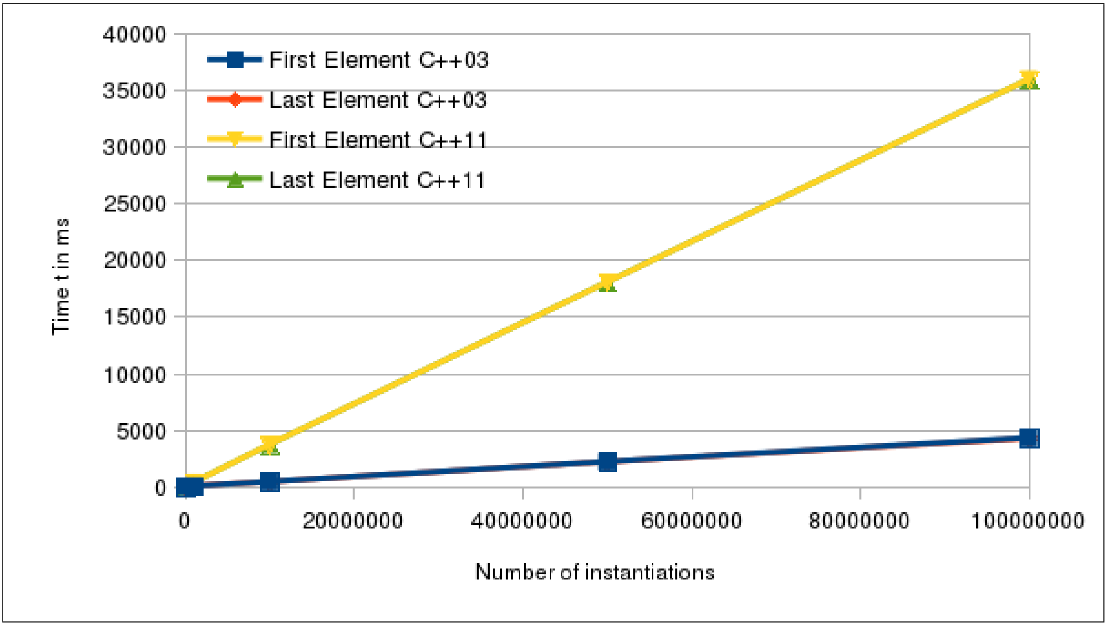

Figure 5.2: Conditional statement factory timing

Category	Percentage	Time in s
Factory	19.18	0.81
Shape	61.18	2.60
Main	17.27	0.73
Other	2.37	0.10

Table 5.1: Conditional statement factory timing distribution C++03

Category	Percentage	Time in s
Factoy	1.33	0.47
Smart pointer	82.28	29.53
Shape	2.87	1.03
Main	2.82	1.01
Other	10.7	3.84

Table 5.2: Conditional statement factory timing distribution C++11

The timing analysis, made with the C++ profiler gprof, gives an idea of the root of this overhead of execution time. As shown in table 5.1 with about 61% and 2.6 seconds for 100,000,000 objects Shape causes the biggest part of the C++03 implementation. Within the C++11 implementation table 5.2 shows that the use of smart pointers has a big effect on the execution time. With about 82% and 29.5 secondes for 100,000,000 objects the execution time for using smart pointers is about 10 times as much as the time for Factory, Shape and main together. The amount of operations within the C++11 implementation, which is not explicitly categorisable, takes about 10% and 3.8 seconds of the total execution time.

5.1.4 Memory Consumption

As shown in figure 5.3 neither the C++03 implementation nor the C++11 implementation of the factory cause an memory overhead. Depending on what kind of smart pointers will be used within th C++11 implementation the heap memory overhead for every instantiated object will be either 0 byte or 8 bytes.

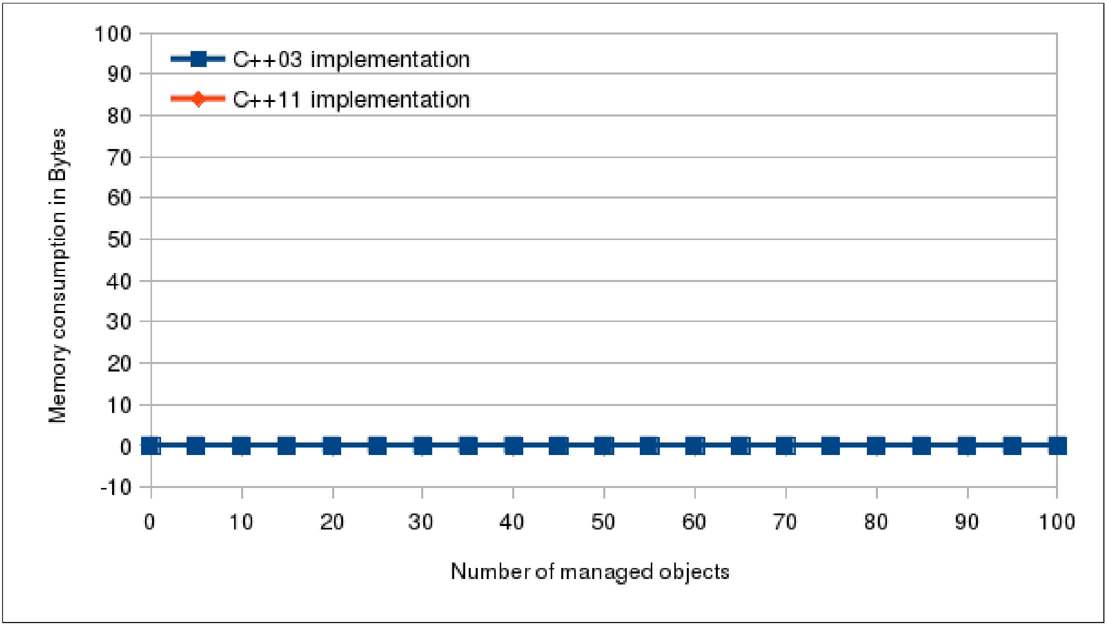

Figure 5.3: Conditional statement factory memory consumption

5.2 Clone Factory Implementation

A clone factory manages already instantiated objects. Therefore the factory needs a method for adding objects at runtime. (Shown in figure 5.4)

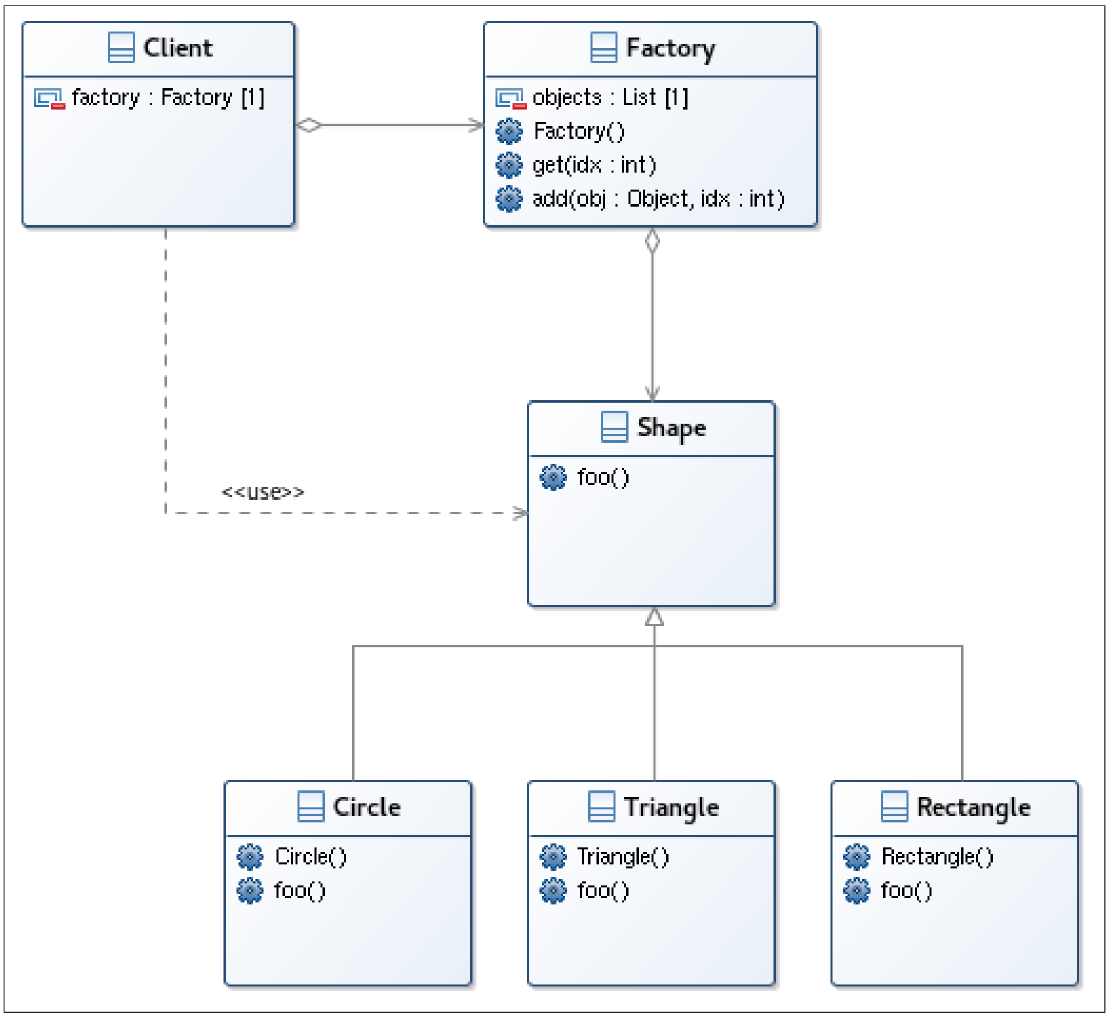

Figure 5.4: Factory method pattern UML including add() method

If the client requests an object, the factory creates a copy of its managed object and returns the copy. An advantage of this implementation is the flexibility. Objects can be added and removed at runtime. This allows to use the factory more dynamically and to instantiate more than one factories with different managed objects. A basic implementation is shown in listing 5.6.

Listing 5.6: Clone factory implementation C++03

```cpp
#include <vector>

using namespace std;

template<class Base>
class CloneFactory {
    class Cloner {
    public:
        virtual Base* clone() const = 0;
        const int idx;
        virtual ~Cloner() {}
    protected:
        Cloner(int idx) :
            idx(idx) {}
    };

    template<class T>
    class ClonerT: public Cloner {
    public:
        ClonerT(const int idx, const T& obj) :
            Cloner(idx), obj_(obj) {}

    private:
        virtual Base* clone() const {
            return new T(obj_);
        }
        const T obj_;
    };
public:
    ~CloneFactory() {
        for(unsigned int i=0; i<cloneList.size(); i++) {
            delete cloneList[i];
        }
    }

    template<class T>
    void add(const T& obj, const int idx) {
        cloneList.push_back(new ClonerT<T>(idx, obj));
    }

    Base* getInstance(const int idx) const {
        for(unsigned int i=0; i<cloneList.size(); i++) {
            if(cloneList[i]->idx == idx) {
                return cloneList[i]->clone();
            }
        }
        return 0;
    }

private:
        vector<Cloner*> cloneList;
};
```

This implementation works with a helper class (Cloner) for managing the objects. By adding an object the factory wrappes this object into a Cloner object and stores this within a vector list. The Clone factory requires a consideration of modifying the copy constructor. (Listing 5.7)

For developing exception-safe code it is neccessary to consider the ownership policy 'Rule of Three'. The Rule of Three says that if either the copy constructor, copy assignment operator or the destructor of a class had to be defined by the software designer, then all of these three parts have to be defined by the software designer. This makes copying objects error-prone because every modification of the object requires an update of the copy constructor, copy assignment operator and destructor. The software designer also has to decide wheter a dynamic allocated data type shall be copied as shallow copy or deep copy. A shallow copy copies the value of the pointer. So both objects, the original object and the copy, will have a pointer to exactly the same memory address. A deep copy allocates new memory and copies the value of the memory, where the pointer refers to, to the new allocated memory. So both objects are pointing to different memory addresses.

Listing 5.7: Rule of Three

```cpp
#include <iostream>
#include "Shape.h"

using namespace std;
class Circle: public Shape {
public:
    Circle() {
        dynInt = new int(42);
    }
    ~Circle() { //Destructor
        delete dynInt;
    }

    Circle(const Circle &obj) { //Copy constructor
        //deep copy:
        dynInt = new int(*obj.dynInt);

        //shallow copy:
        //dynInt = obj.dynInt;
    }

    void foo(void) {
        cout << "Circle calls foo()" << endl;
    }

    int* dynInt;
private:
    // Copy assignment operator
    Circle& operator=(const Circle&) & = default;
};
```

Because of the instantiated objects, the clone factory implementation has a higher memory consumption. The C++11 version of the clone factory (listing 5.8) implementation contains minor changes.

Listing 5.8: Clone factory implementation C++11

```
#include <memory>
#include <unordered_map>

using namespace std;

template<typename Super>
class CloneFactory {
    class Cloner {
    public:
        virtual shared_ptr<Super> clone() const = 0;
        const int idx;
        virtual ~Cloner() {}
    protected:
        Cloner(int idx) :
            idx(idx) {}
    };

    template<typename T>
    class ClonerT: public Cloner {
    public:
        ClonerT() {}

        ClonerT(int idx, const T& obj) :
            Cloner(idx), obj_(obj) {}

    private:
        virtual shared_ptr<Super> clone() const {
            return make_shared<T>(obj_);
        }
            const T obj_;
    };
public:
    template<typename T>
    void add(int idx, const T& obj) {
        cloneMap.emplace(idx, make_unique<ClonerT<T>>(idx, obj));
    }

    shared_ptr<Super> getInstance(int idx) const {
        auto found = cloneMap.find(idx);

        if (found == cloneMap.end()) {
            return nullptr;
        }
        return shared_ptr<Super>((found->second)->clone());
    }

private:
    unordered_map<int, unique_ptr<Cloner>> cloneMap;
};
```

As already seen and explained in chapter 5.1, all raw pointers were replaced by smart pointers. New is the use of an unordered map instead of a vector list.

5.2.1 Unordered Map

An unordered map is an unordered associative container. This means the unordered map manages the objects within a hash-table. (Figure 5.5)
The hash-table consists of entries, named buckets. Each bucket stores the managed objects in a linked list. A hash-function converts the key into a value of type *size_t*. This value defines in which bucket the object will be stored. With the linked list one bucket can store more than one object at the time. The efficiency of a hash-table depends on the distribution of the objects. In average each bucket contains only one object. In this case the complexity of seek time is T(n) = O(1). In worst case only one bucket contains all object. The seek time is linear so the complexity is T(n) = O(n). At the beginning the hash-table consists of a number b of buckets. If the loadfactor

$$loadfactor = n/b \tag{5.1}$$

exceeds a defined threshold with a number n of objects, the hash-table will be reorganized with a new number b of buckets. (cf. Kirch und Prinz, 2015, 828 - 829)

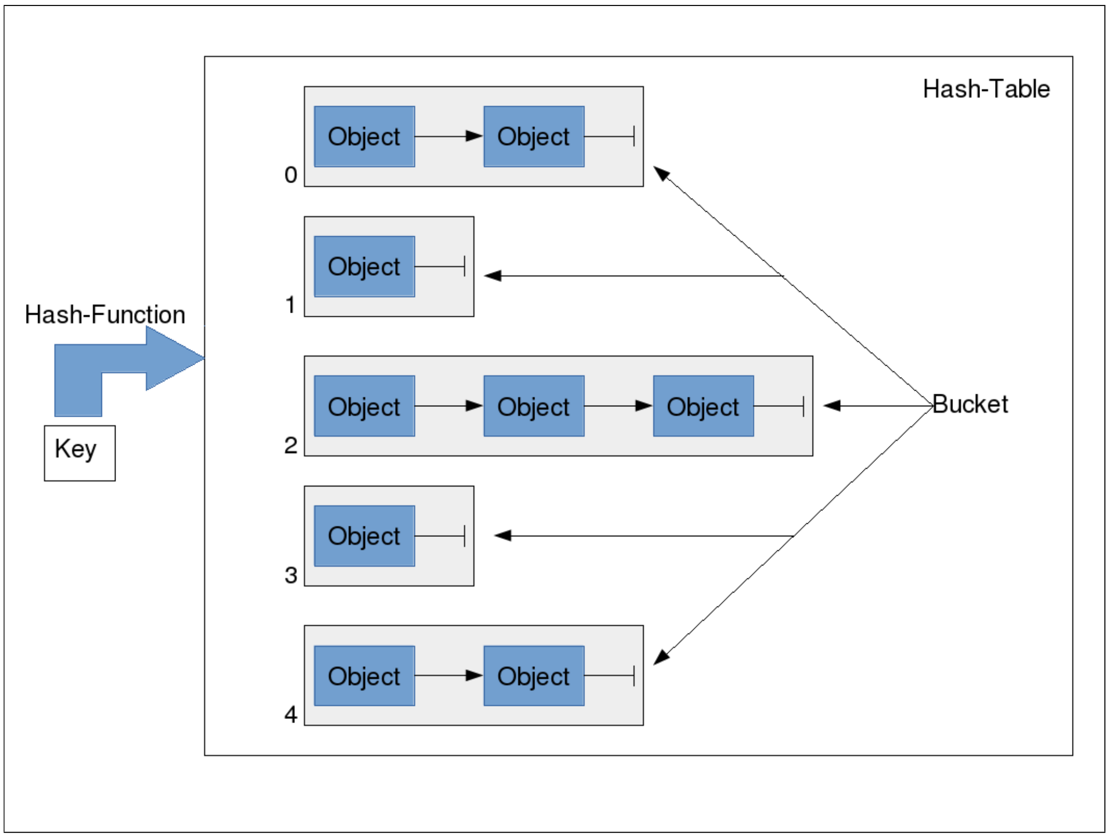

Figure 5.5: Hash-table (Kirch und Prinz, 2015, 828)

5.2.2 Rvalue Reference

C++11 provides the option to explicitly define rvalues. Unlike lvalues, rvalues do not have a memory address.

Listing 5.9: Lvalue example

```
string st = "Hello␣from␣an␣Lvalue";
foo(st);
```

Listing 5.10: Rvalue example

```
foo(string("Hello␣from␣an␣Rvalue"));
```

Listing 5.9 shows an example of a typical lvalue. The string does have a name and a memory address. Multiple modifications can be done. Listing 5.10 shows an example of an rvalue. It exists only temporarily and after passing it to the function, the client is not able to do any more with the string because it does not have a name and a memory address. Rvalues are not new. New is the rvalue reference.

Listing 5.11: Rvalue reference example

```cpp
void foo(string&& st) {
 ...
}
```

The function *foo* (listing 5.10) only accepts strings as rvalue as parameter. So the call of listing 5.9 would cause a compiler error because *st* is not an rvalue. An rvalue reference makes sure that a reference is not used anymore outside of the function. (cf. Pohmann, 2013, 32 - 33) For passing an lvalue as rvalue, C++11 provides the function move to cast an lvalue to an rvalue reference. (Listing 5.12) The function forward provides the option to cast an rvalue reference to an rvalue.

Listing 5.12: Move example

```cpp
string st = "Hello␣from␣an␣Lvalue";
foo(move(st));
```

as shown in listing 5.12 by casting an lvalue to an rvalue the string *st* can be passed to the function *foo*. The software designer must consider, the function implies that the parameter is an rvalue and not needed anymore outside of a function. Classes will be extended by a move constructor and a move assignment operator for managing rvalue references. These extend the Rule of Three ownership policy to the Rule of Five. The clone factory always keeps its managed objects. Therefore the visibility of move constructor and move assignment operator must be set to private. (Listing 5.13)

Listing 5.13: Move constructor and move assignment operator

```
#include "Shape.h"

using namespace std;
class Circle: public Shape {
public:
    Circle() {
        dynInt = new int(42);
    }
    ~Circle() { //Destructor
        delete dynInt;
    }

    Circle(const Circle &obj) { //Copy constructor
        //deep copy:
        dynInt = new int(*obj.dynInt);
        //shallow copy:
        //dynInt = obj.dynInt;
    }

    void foo(void) {
        cout << "Circle␣calls␣foo()" << endl;
    }
    int* dynInt;
private:
    // Move constructor
    Circle(Circle &&obj) = default;
    // Copy assignment operator
    Circle& operator=(const Circle&) & = default;
    // Move assignment operator
    Circle& operator=(Circle&&) & = default;
};
```

5.2.3 Timing

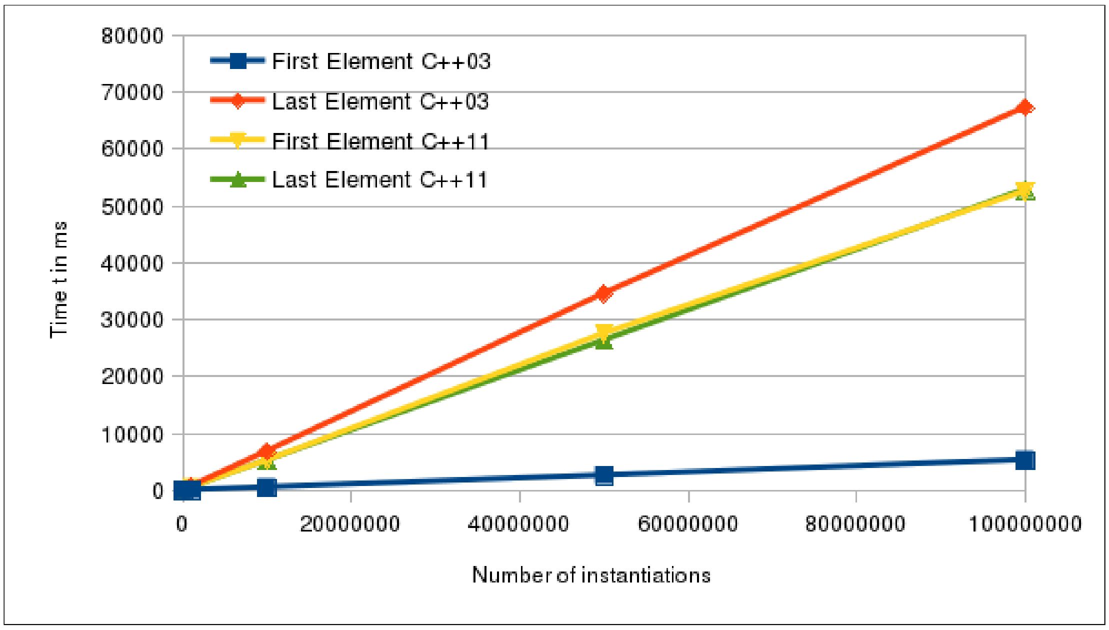

Figure 5.6: Clone factory timing

Figure 5.6 compares the implementation of listing 5.6 with listing 5.8. The seek complexity for finding the requested object within the vector with 100 entries is T(n) = O(n) for the C++03 implementation. Because of the hash-table, the unordered map has a seek complexity of T(n) = O(1). Nevertheless, in average the C++11 implementation is still slower than the C++03 implementation. For cloning 100,000,000 objects, where the requested object always is on the first place within the vector, the C++03 implementation needs about 5.3 second. The farther back the object is in the vector, the longer it takes to find it. So for cloning 100,000,000 objects, where the requested object always is on the last possible place within the vector, the clone factory needs about 67 second. More than 10 times as much as in best case. The C++11 implementation needs constantly 52 seconds for cloning 100,000,000 objects.

The profiling output (Table 5.3) shows the distribution of timing of the C++03 in the worst case scenario. With about 96% and 64.9 seconds the factory causes the biggest part of the execution time. The profiling output (Table 5.4) of the C++11 implementation shows that the execution time of the factory decreased. The main reason for this is the use of the unordered map. But the use of smart pointers increases the execution time by 38.6 seconds and makes about 73% of the total execution time. The amount of not explicitly categorisable operations takes about 6% and 3.24 seconds of the total execution time.

Category	**Percentage**	**Time in s**
Factory	96.55	64.90
Shape	2.59	1.74
Main	0.86	0.57
Other	0.00	0.00

Table 5.3: Clone factory timing distribution C++03

Category	**Percentage**	**Time in s**
Factory	18.01	9.51
Smart pointer	73.1	38.61
Shape	1.24	0.65
Main	1.35	0.71
Other	6.14	3.24

Table 5.4: Clone factory timing distribution C++11

5.2.4 Memory Consumption

Unlike the conditional statement implementation, a clone factory manages objects. For each object the factory allocates memory. Figure 5.7 shows the memory overhead of the factory depending on the number of managed objects. In general the C++11 implementation causes a bigger memory overhead than the C++03 implementation. This overhead is caused by the use of an unordered map including a hash-table. As mentioned in section 5.2.1 the hash-table of the unordered map consists of a number of buckets, depending on the loadfactor (Equation 5.1). When the loadfactor reaches a defined value, the unordered map allocates new memory for having more buckets. This allocation can be observed in figure 5.7 between 20 and 25, 45 and 50 as well as between 95 and 100 managed objects. The *std::vector* also allocates more memory than actually needed to reduce the number of allocations on growing. (cf. cppreference, 2014) Figure 5.7 also shows these allocations between 15 and 20, 30 and 35 as well as between 60 and 65 managed objects. The size of the particular objects has also has an impact of the memory consumption of the clone Factory.

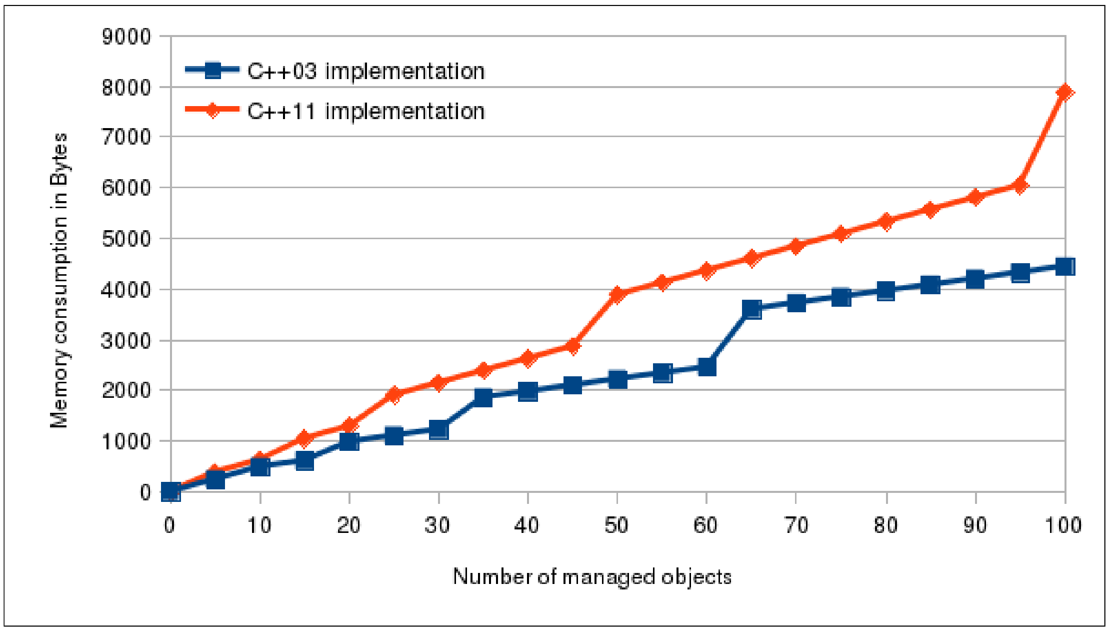

Figure 5.7: Clone factory memory consumption

5.3 Lambda Implementation

As mentioned in section 5.2 cloning objects is error-prone because of the demand for maintenance of destructor, copy constructor and copy assignment operator. In addition, depending on the size and the number of the managed objects the clone factory tends to have a high memory consumption. With establishing lambda expressions, C++11 provides an alternative implementation. (Listing 5.14)

Listing 5.14: Lambda implementation C++11

```
#include <memory>
#include <unordered_map>
#include <functional>

using namespace std;

template<typename Base>
class LambdaFactory {
public:
    LambdaFactory() {
    }

    shared_ptr<Base> getInstance(const int idx) {
        auto found = lambdaMap.find(idx);
        return found != lambdaMap.end() ? found->second() : nullptr;
    }

    template<size_t IDX, typename T, typename... Args>
    void add(Args... args) {
        lambdaMap.emplace(IDX,
            ([=]()-> auto {return make_shared<T>(args...);}));
    }

private:
    unordered_map<int,function<shared_ptr<Base>()>> lambdaMap;
};
```

5.3.1 Lambda Expression

A lambda expression is a local anonymous function with access to objects and variables in its environment. (cf. Kirch und Prinz, 2015, 914 - 915)

Listing 5.15: Lambda expression syntax

```
[capture](parameterlist) mutable noexcept -> return-type
{ function body }
```

The syntax of a lambda expression (Listing 5.15) consists of the following parts:

- **[capture]** defines the access to its environment. It can be either [] no access, [=] access by value or [&] access by reference. Also combinations and access to particular objects are possible. (Example: [=,&var] defines general access by value except for the variable *var*, which is passed by reference.)
- **(parameterlist)** contains the declaration of the function's parameter.
- **mutable** is optional and defines whether objects, passed by value, are declared as const or not.
- **noexcept** is optional and defines whether the function is able to throw an exception or not.

- **return-type** is optional and explicitly defines the return-type. Without defining it explicitly, the return-type can be determined from the return value.
- **function body** contains the implementation of the function.

The call of a lambda expression can be occur later than the declaration. Because of that, lambda expressions can be stored passed as a parameter. For this C++11 provides a generic wrapper named *function*. *function* is able to take a lambda expression. It gives the lambda expression a memory address. Line 20 and 21 of listing 5.14 shows the declaration of a lambda expression which instantiates a smart pointer refering to an object of a generic type T and stores it into a container. If the object of type T is actually needed, the stored lambda expression can be called which instantiates and returns the requested smart pointer.

5.3.2 Variadic Template

With variadic templates C++11 extends the generic type declaration. A variadic template is useful in case when the software designer does not know how many arguments are needed. (cf. Kirch und Prinz, 2015, 785)

Listing 5.16: Variadic templates syntax

```
template<size_t IDX, typename T, typename... Args>
void add(Args... args) {
    lambdaMap.emplace(IDX,
        ([=]()-> auto {return make_shared<T>(args...);}));
}
```

The variadic part of the template is declared with three dots and must always placed as last argument. The compiler analyses the use of the variadic templates and replaces them by the needed data types. Variadic templates also allow zero arguments. Listing 5.16 shows a snipped of the lambda factory implementation (Listing 5.14) for adding new classes to the factory. Within the factory the managed classes are unknown and so the number and types of the parameter. As result of the variadic template the factory is able to manage any type of class and gives no constrains for the constructor parameters.

5.3.3 Timing

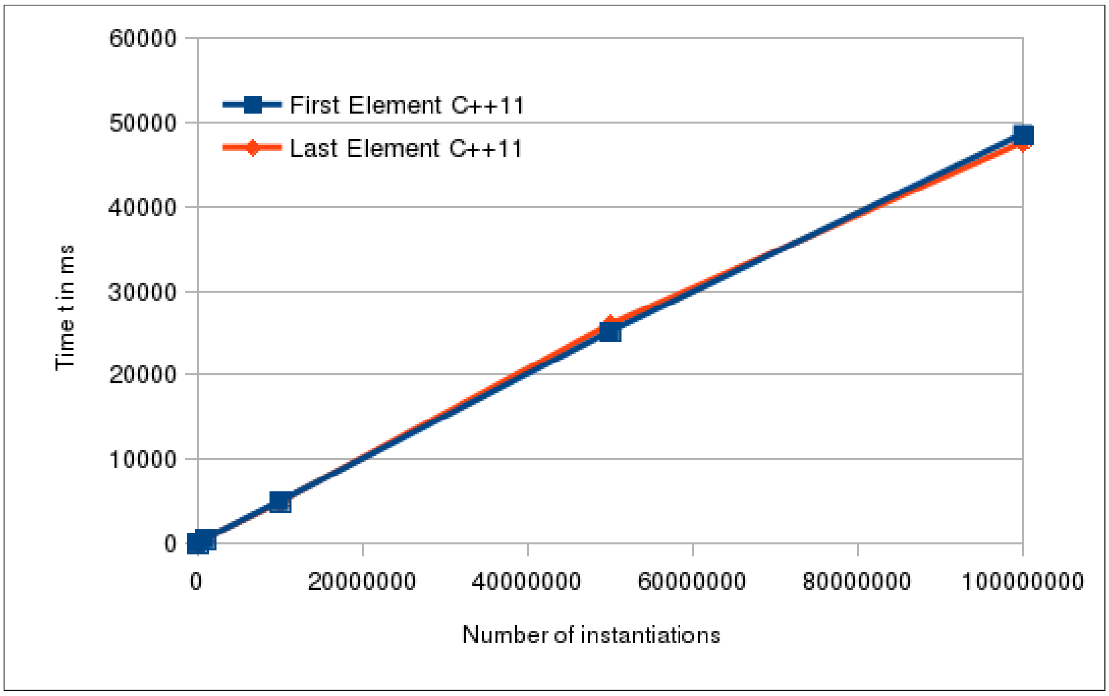

Figure 5.8: Lambda implementation timing

With about 48 seconds for instantiating 100,000,000 objects (Figure 5.8) the lambda implementation has a similar timing like C++11 implementation of the clone factory. Also the seek complexity is with T(n) = O(1) the same because of the use of an unordered map. The profiling output (Table 5.5) of the lambda implementation shows that the use of smart pointers has with about 64% and 30.7 seconds the biggest effect on the total execution time.

Category	**Percentage**	**Time in s**
Factory	24.80	11.78
Smart pointer	64.71	30.73
Shape	2.38	1.13
Main	3.63	1.72
Other	4.11	1.95

Table 5.5: Lambda implementation timing distribution C++11

5.3.4 Memory Consumption

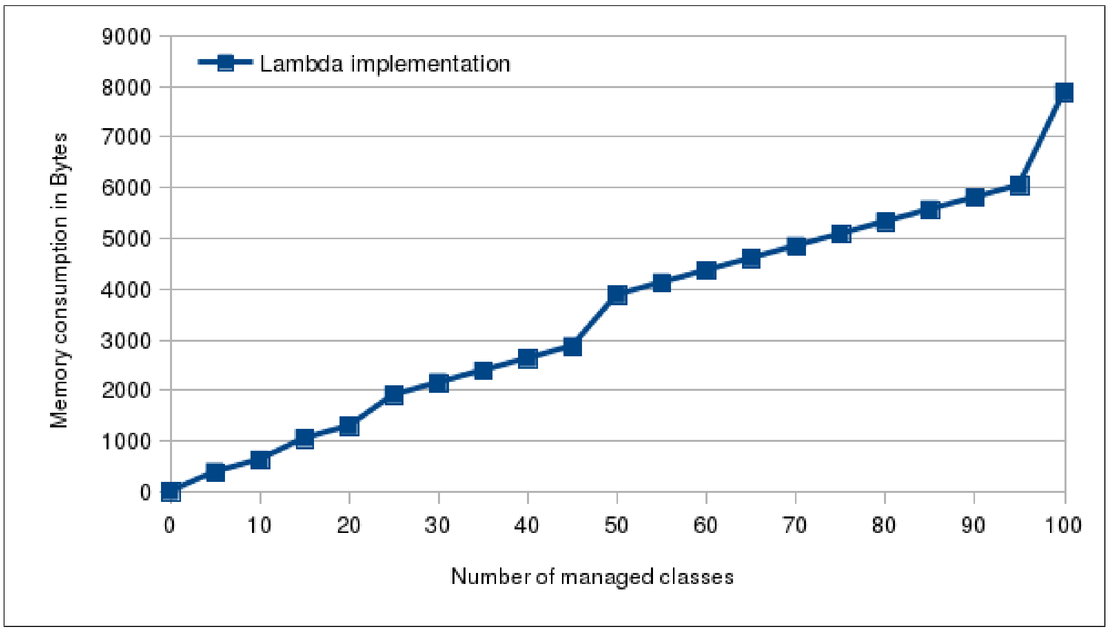

Figure 5.9: Lambda implementation memory consumption

On the first look, the lambda implementation of the factory shows exactly the same memory consumption (Figure 5.9) than the C++11 clone factory implementation (Figure 5.7). But this is only true in case when the managed classes have minimum size. One advantage of the lambda implementation, compared with the clone factory implementation, is the managemend of classes instead of objects. This makes the memory consumption of the factory independent of the size of the managed classes. So, in case when the size of the classes rise, the memory consumption of the clone factory implementation also rises, but the lambda implementation stays constant.

6 State Pattern

As mentioned in chapter 2, a characteristic of a reactive system is that responses of the reactive system are dependent on the current state of the system. Usually an embedded reactive system is able to change its state. A simple example of this is a system with a failure state. If everything meets the expectations, the system stays in its normal state. But if something unexpected occurs, the system switches to its failure state. Within the failure state the system tries to keep the essential subsystems running and minimizes the damages within the environment. Other stateful systems like a remote control extend the functionality of the system with a limited number of buttons. Depending on the state, the remote control is able to communicate with either the television, dvd player or another multimedia device.

Figure 6.1 shows a simple example of a state machine with three states and three signals. *Light Off* is the initial state. The system can change to the state *Light On* by the signal 'on' and to the state *Destroyed* by the signal 'destroy'. From state *Line On* the system can change back to the state *Light Off* or to the state *Destroyed*. *Destroyed* is a final state. When the light is destroyed, it is destroyed.

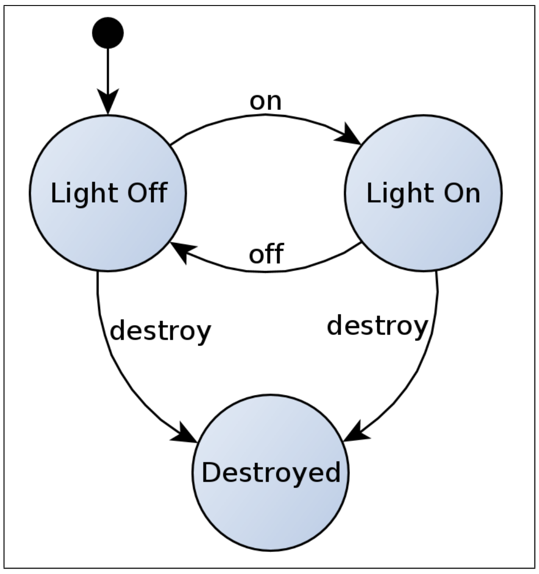

Figure 6.1: State machine of a light.

The state pattern is an object based pattern of the type behavioral. The problem is how to react depending on the current state and the external stimuli with a minimum of overhead in timing. Procedural implementations solve this problem by providing a matrix where the combination of the current state and the input determine behavior and the transition to the next state. Other implementations work with a number of switch-case-selections. The result of these implementations is difficult read and

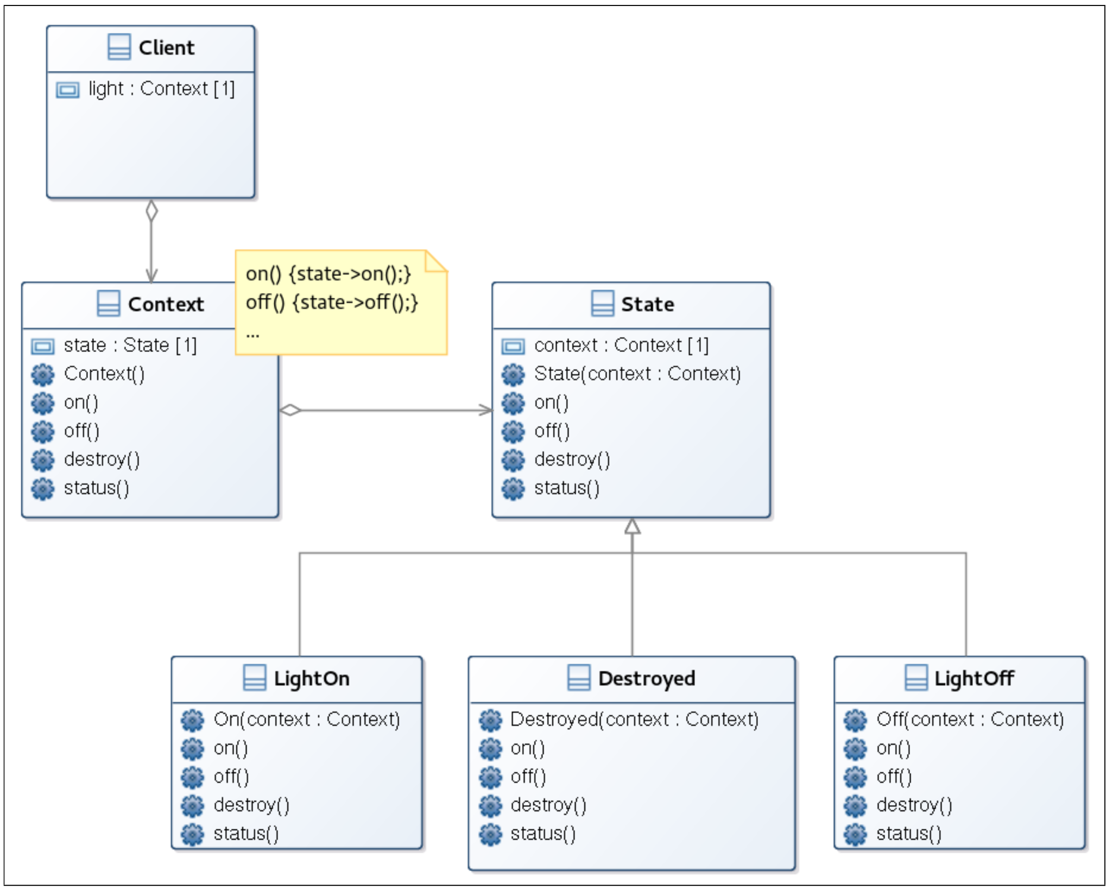

Figure 6.2: State pattern UML

to maintain. The base of the state pattern of GoF is a context class and an abstract base state class. (Figure 6.2) Concrete states derive from the base state class and implement the interface. *Context* is the connection between *Client* and the state machine. It contains an instance of the state machine and provides the same interface as the state machine for delegating the incoming signals. *Client* does only know *Context*. It uses *Context* as if it would be a concrete State class. The state machine has access to public member variables of *Context*. (cf. Gamma u. a., 2015, 372 - 377)

6.1 Polymophism Implementation

The example explained by GoF uses polymorphism. The concrete states use the interface of the abstract base state class (Listing 6.2) and implement the behavior. (Listing 6.3 and 6.4)

Context (Listing 6.1) contains an instance of *LightOff* as initial state. *Context* and *State* depend on each other. This circular dependency can be solved by forward declaration. Therefore it is necessary to declare *Context* once within the header file of *State* (Listing 6.2, line 1) and include the header file of *Context* later within the implementation of the concrete states. (Listing 6.4, line 4) Depending on the behavior of the state machine, similar dependencies occur between concrete states and can be solved on a similar way. *Context* also provides a dispatcher which receives external signals and calls the particular functions.

Listing 6.1: State pattern Context C++03

```cpp
#include "LightOff.h"
#include <map>

using namespace std;
class Context : State {
public:
    enum SIGNAL{ON=0, OFF=1, DESTROY=2};

    Context() {
        numOfChanges = 0;
        state = new LightOff(this);

        functionMap[ON] = &Context::on;
        functionMap[OFF] = &Context::off;
        functionMap[DESTROY] = &Context::destroy;
    }

    void call(int signal) {
        if(functionMap.find(signal) != functionMap.end()) {
            (this->*functionMap[signal])();
        }
    }

    void on() {state->on();}
    void off() {state->off();}
    void destroy() {state->destroy();}
    void status() {state->status();}

    State* state;
    int numOfChanges;
private:
    map<int, void (Light::*) ()> functionMap;
};
```

Listing 6.2: State pattern abstract State C++03

```
class Context;

class State {
public:
        enum INPUT{ON=0,OFF=1,DESTROY=2};

        State(){context = 0;}
        State(Context* context){this->context = context;}

        virtual void on() = 0;
        virtual void off() = 0;
        virtual void destroy() = 0;
        virtual void status() = 0;
        virtual ~State(){};

        Context* context;
};
```

Listing 6.3: State pattern LightOff declaration C++03

```
#include "State.h"

class LightOff : public State{
public:
    LightOff(Context* context): State(context) {}

    virtual void on();
    virtual void off();
    virtual void destroy();
    virtual void status();

    ~LightOff(){};
};
```

Listing 6.4: State pattern LightOff implementation C++03

```
 1 #include "LightOff.h"
 2 #include "LightOn.h"
 3 #include "Destroyed.h"
 4 #include "Context.h"
 5
 6 void LightOff::on() {
 7     delete context->state;
 8     context->state = new LightOn(context);
 9     context->numOfChanges++;
10 }
11
12 void LightOff::off() {}
13
14 void LightOff::destroy() {
15         delete context->state;
16         context->state = new Destroyed(context);
17         context->numOfChanges++;
18 }
19
20 void LightOff::status() {
21         cout << "current␣state␣is:␣OFF!"<< endl;
22 }
```

Listing 6.5: State pattern Client C++03

```
#include "Context.h"

int main(int argc, char** argv){
    Context* light = new Context();
    light->call(externalSignal);

    light->status();

    delete light;
    return 1;
}
```

When a transition occurs, the current instance of the concrete state will be replaced by a new concrete state. Lines 7 and 8 of listing 6.4 shows the transition from *LightOff* to *LightOn*.

The current state will be deleted and replaced by the new state. If transitions occur in a high frequency, it might be useful to keep the available states instantiated in a pool and reuse them. This reduces the time for allocating memory and instantiating objects but increases the consumption of memory. *Client* (Listing 6.5) instantiates *Context* and passes incoming signals to it.

The main change of the implementation with C++11 is the use of smart pointers instead of raw pointers. Within the polymorphism implementation the use of smart pointers shows two main difficulties. As seen in the C++03 implementation (Listing 6.1) *Context* passes the address of itself to the concrete state by using the keyword *this*. The *Client* of the C++11 implementation (Listing 5.4) uses a smart pointer for refering to *Context*. *Context* is not able to pass itself as smart pointer to the concrete state. Therefore *Context* provides the function *initialize* to provide the option of passing *Context* within a smart pointer from *Client* to the concrete state. *Client* must call *initialize* before using the state machine. (Listing 6.7)

Listing 6.6: State pattern Context C++11

```
#include "State.h"
#include <unordered_map>

using namespace std;
class Context {
public:
    enum SIGNAL{ON=0, OFF=1, DESTROY=2};

    Context() {
        numOfChanges = 0;
    }

    void initialize(weak_ptr<Context> context) {
        state = make_unique<On>(context);

        functionMap.emplace(make_pair(ON,
                            bind(&Context::on, this)));
        functionMap.emplace(make_pair(OFF,
                            bind(&Context::off, this)));
        functionMap.emplace(make_pair(DESTROY,
                            bind(&Context::destroy, this)));
    }

    void call(int signal) {
        auto found = functionMap.find(signal);
        if (found != functionMap.end()) {
            (found->second)();
        }
    }

    void on() {state->on();}
    void off() {state->off();}
    void destroy() {state->destroy();}
    void status() {state->status();}

    unique_ptr<State> state;
    int numOfChanges;
private:
    unordered_map<int, function<void()>> functionMap;
};
```

Listing 6.7: State pattern Client C++11

```
#include "Context.h"

int main(int argc, char** argv){
    auto light = make_shared<Context>();
    light->initialize(light);

    light->call(externalSignal);

    light->status();
    return 1;
}
```

The second difficulty occurs because of the circular dependency between *Context* and *State*. As mentioned in section 5.1.1 shared pointers do reference counting. In case of using shared pointers within the example the circular dependencies would cause that *Context* and *State* would not be deleted automatically.

As shown in figure 6.3 both, *Client* and *State*, refering to *Context* with a shared pointer. *Context* refers to *State* with an unique pointer. If *Client* disappears, the reference counter decreases

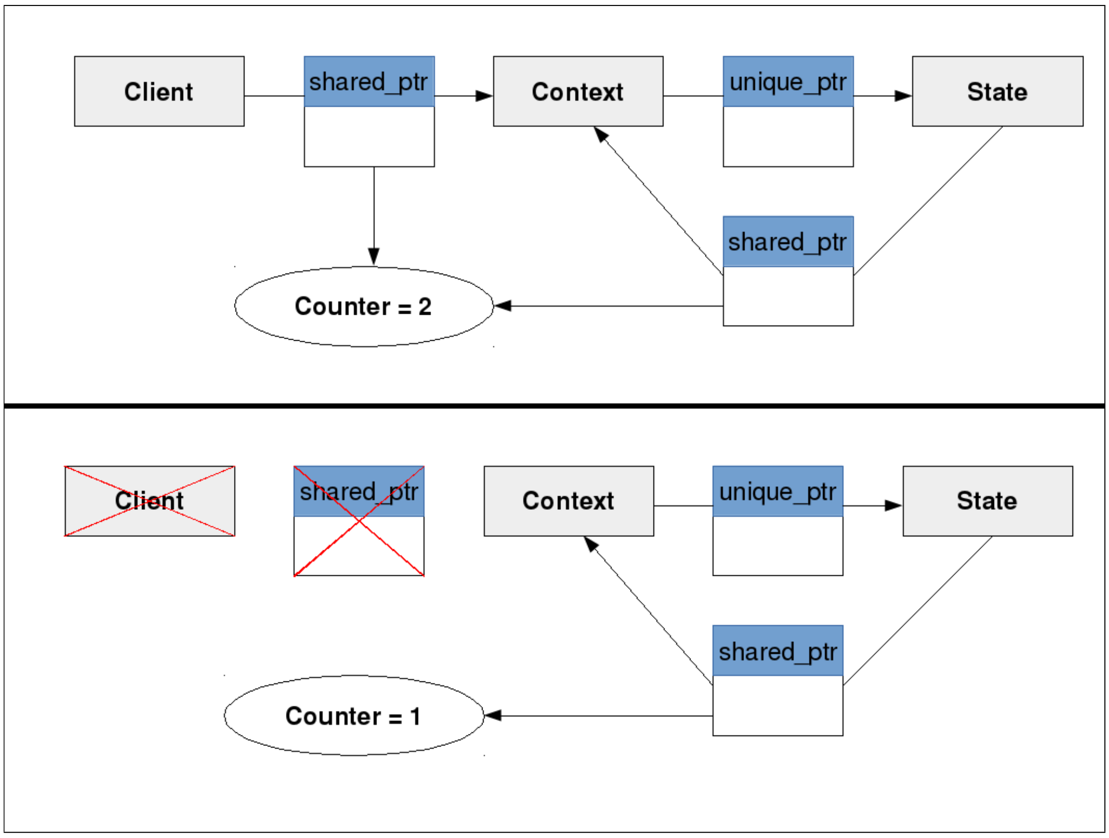

Figure 6.3: Shared pointer dependencies

but does not reach zero because *State* is still refering to *Context*. Therefore *Context* will not be deleted. When *Context* will not be deleted, *State* will also not be deleted because it is still referred by *Context*. A memory leak is the consequence. Deleting the unique pointer, which referes to *State*, manually would solve this problem. Another solution provides the use of weak pointer.
Figure 6.4 shows the same example with one difference. Instead of using a shared pointer, *State* only refers to *Context* with a weak pointer. The effect is that the reference counter reaches zero when *Client* disappears because weak pointers have no effect on the reference counter. When the reference counter reaches zero, *Context* will be deleted regardless whether *State* is still refering to *Context*. In this particular example this is no problem because the destructor of *Context* deletes the reference to *State* and *State* will also be deleted.

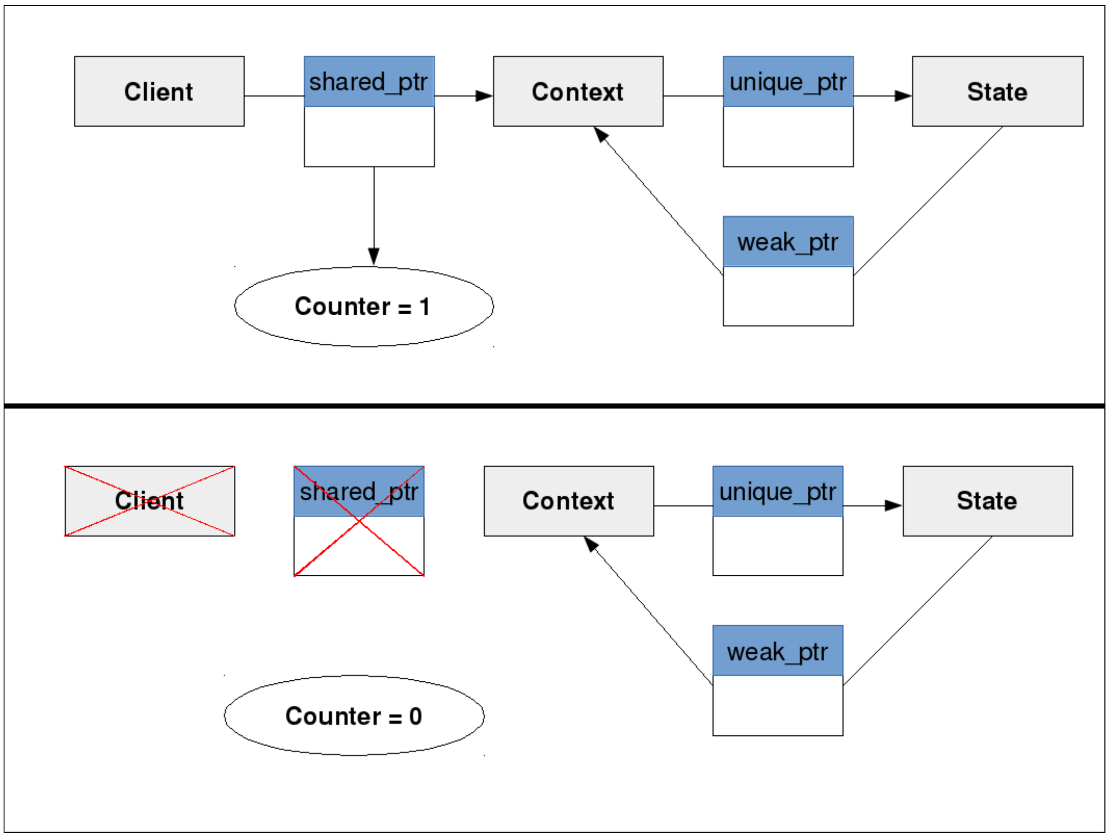

Figure 6.4: Weak pointer dependencies

Before calling a member or a function of an object referred by a weak pointer, the software designer must make sure that the weak pointer is pointing to a valid object. For this a weak pointer provides the function *lock*. *lock* casts the weak pointer to a temporary shared pointer which can be used to manipulate the object. (Listing 6.8) This temporary shared pointer temporarily increases the reference counter.

Listing 6.8: State pattern LightOff implementation C++11

```
#include "LightOff.h"
#include "LightOn.h"
#include "Destroyed.h"
#include "Context.h"

void LightOff::on() {
    if(auto tmp = context.lock()) {
        tmp->state = make_unique<On>(context);
```

```cpp
        tmp->numOfChanges++;
    }
}

void LightOff::off() {
    cout << "already␣off!" << endl;
}

void LightOff::destroy() {
    if(auto tmp = context.lock()) {
        tmp->state = make_unique<Destroyed>(context);
        tmp->numOfChanges++;
    }
}

void LightOff::status() {
    if(auto tmp = context.lock()) {
        cout << "current␣state␣is:␣OFF!" << endl;
    }
}
```

6.1.1 Bind

bind is a function which allows to adapt signatures of functions by predefining function parameters. (Listing 6.9)

Listing 6.9: Bind value to function parameter C++11

```cpp
int add(int a, int b) {
    return a + b;
}

auto addOne = bind(&add, 1, placeholders::_1);
auto addTwo = bind(&add, 2, placeholders::_1);
cout << addOne(3) << ",␣" << addTwo(5) << endl;
```

It also provides the option to bind function pointers of memberfunctions to particular instances. (cf. Pohmann, 2013, 169 - 172) Line 16 to 21 of listing 6.6 shows how to use *bind* in combination with *make_pair* to manage function pointers within a map without using raw pointers. *make_pair* is a wrapper which combines a key and a value to a map-compatible type.

6.1.2 Timing

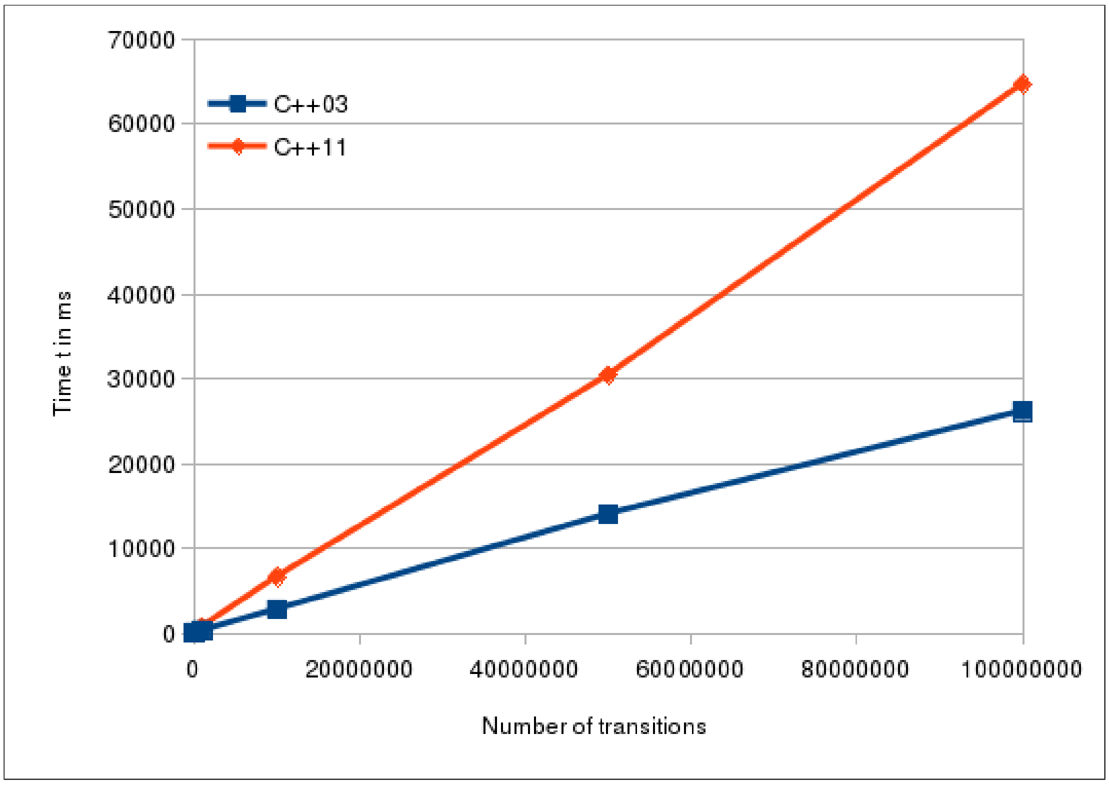

Figure 6.5: State pattern polymorphism timing

Figure 6.5 shows the differences between the C++03 implementation and the C++11 implementation of the state pattern. For changing the state 100,000,000 times, the C++03 implementaton needs about 26 seconds and the C++11 implementation needs about 65 seconds. Table 6.1 compared with table 6.2 shows a reduction of the execution time of about 6 seconds for *Context*. The reason for this is the use of an unorderd map with a seek complexity of T(n) = O(1) instead of the use of a normal map with a seek complexity of T(n) = O(log(n)). The dominating factor fo the execution time of the C++11 implementation is caused by the use of smart pointers. With about 35 seconds it is responsible for about 55.2% of the total execution time.

Category	Percentage	Time in s
Context	93.29	24.39
State	5.58	1.45
Main	1.12	0.29

Table 6.1: State pattern polymorphism timing distribution C++03

Category	Percentage	Time in s
Context	28.45	18.39
Smart pointer	55.28	35.76
State	5.11	3.30
Main	1.12	0.29
other	7.79	5.03

Table 6.2: State pattern polymorphism timing distribution C++11

6.1.3 Memory Consumption

For each transition the polymorphism implementation allocates memory for the new state. Figure 6.6 shows an allocation of 16 bytes for each transition for the C++03 implementation and 24 bytes for each transition for the C++11 implementation. The overhead of 8 bytes is caused by the use of a weak pointer within state for refering to *Context*. As mentioned in section 5.1.1, weak pointers also refer to a reference counter. One way to avoid this overhead is to move the weak pointer to *Context* and pass it as function parameter to *State* only when it is needed.

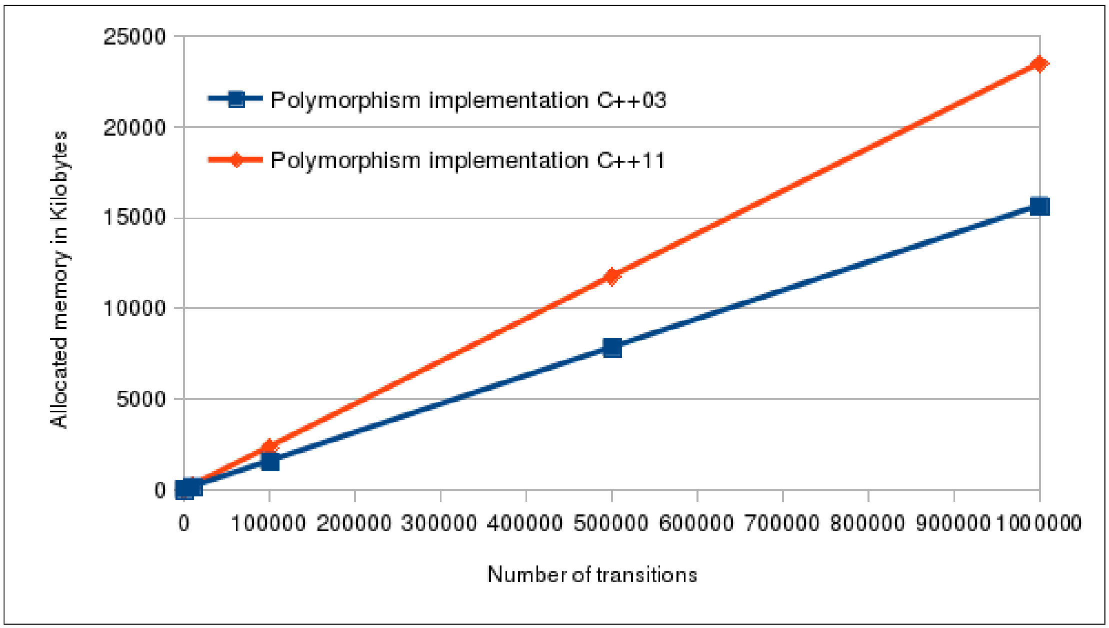

Figure 6.6: State pattern polymorphism memory consumption

6.2 Placement new Implementation

Dynamic memory allocation is unpredictable. It can take an undefined amount of time for finding a suitable chunk of memory. In worst case no memory is available. Especially for real-time systems the behavior of allocating memory is restricted. One solution C++ provides is the use of the *placement new* operator. The behavior of *placement new* is similar to the *new* operator with the difference that the *placement new* operator takes an already allocated memory address where the new object will be instantiated. (Listing 6.10, line 9)

Listing 6.10: Placement new operator C++11

```
1  #include "LightOff.h"
2  #include "LightOn.h"
3  #include "Destroyed.h"
4  #include "Context.h"
5
6  void LightOff::on() {
7      cout << "turn_on!" << endl;
8      if(auto tmp = context.lock()) {
9          new(this) On(context);
10         tmp->numOfChanges++;
```

```
    }
}

void LightOff::off() {
    cout << "already off!" << endl;
}

void LightOff::destroy() {
    cout << "Destroy!" << endl;
    if(auto tmp = context.lock()) {
        new(this) Destroyed(context);
        tmp->numOfChanges++;
    }
}

void LightOff::status() {
    if(auto tmp = context.lock()) {
        cout << "current state is: OFF!" << endl;
    }
}
```

In this particular case the combination of smart pointers and the use of the *placement new* operator causes no problems. The change of the object will not be noticed by the smart pointer. However, it is discussable whether the use of the *placement new* operator is still acceptable when the use of the *new* operator is replaced by smart pointers. C++11 does not provide a direct compensation for the *placement new* operator with the same behavior. But with *allocate_shared* C++11 provides an option to use an own memory management.

6.2.1 Allocate_shared

The *allocate_shared* operator is a variant of the *make_shared* operator. Unlike *make_shared*, *allocate_shared* takes an own allocator as parameter for allocating memory. Writing an own allocator which is compliant to the standard allocator is not trivial. Therefore it is recommended to use already existing allocators. Line 40 of listing 6.11 shows the instantiation of a pool allocator with the name *alloc*. In line 15 *alloc* is used for instantiating a new shared pointer refering to an object of class *On*. *alloc* will also be used for transitions within a particular state. (Listing 6.12, line 8) A pool allocator uses a memory pool with a pre-allocated number of fixed sized memory chunks. (Figure 6.7) If memory is requested, the pool allocator finds the next free memory chunk, and returns the address. In case when a memory chunk is not needed anymore by the requestor, the pool allocator declares this chunk as free to use by other requestors.

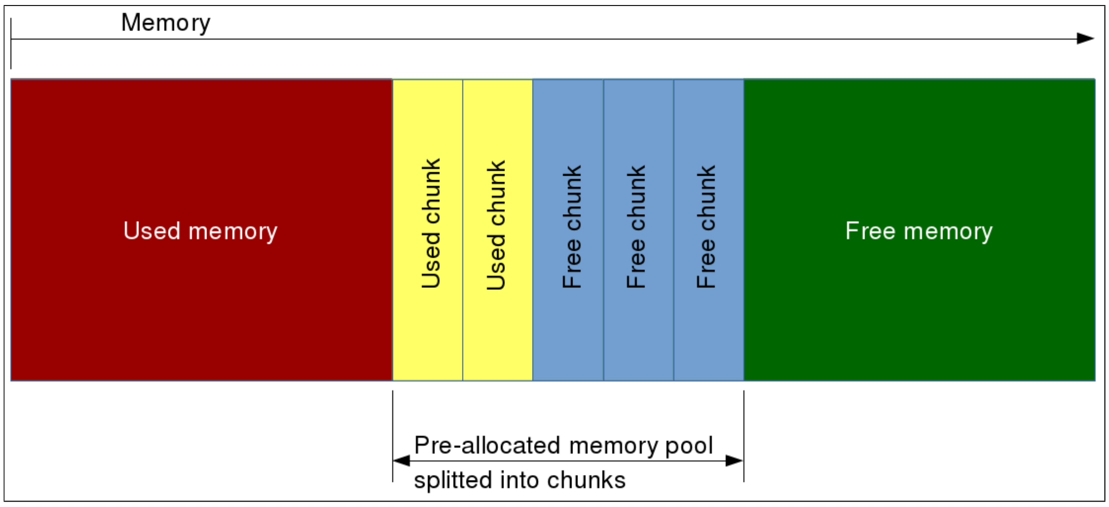

Figure 6.7: Memory pool

Listing 6.11: State pattern pool allocator Context C++11

```cpp
#include "State.h"
#include <unordered_map>
#include <boost/pool/pool_alloc.hpp>

using namespace std;
class Context {
public:
    enum SIGNAL{ON=0, OFF=1, DESTROY=2};

    Context() {
        numOfChanges = 0;
    }

    void initialize(weak_ptr<Context> context) {
        state = allocate_shared<On>(alloc,context);

        functionMap.emplace(make_pair(ON,
                                      bind(&Context::on, this)));
        functionMap.emplace(make_pair(OFF,
                                      bind(&Context::off, this)));
        functionMap.emplace(make_pair(DESTROY,
                                  bind(&Context::destroy, this)));
    }

    void call(int signal) {
        auto found = functionMap.find(signal);
        if (found != functionMap.end()) {
            (found->second)();
        }
    }

    void on() {state->on();}
    void off() {state->off();}
    void destroy() {state->destroy();}
    void status() {state->status();}

    shared_ptr<State> state;
    int numOfChanges;

    boost::pool_allocator<State> alloc;
private:
    unordered_map<int, function<void()>> functionMap;
};
```

Listing 6.12: State pattern pool allocator LightOff implementation C++11

```
#include "LightOff.h"
#include "LightOn.h"
#include "Destroyed.h"
#include "Context.h"

void LightOff::on() {
    if(auto tmp = context.lock()) {
        tmp->state = allocate_shared<On>(tmp->alloc,context);
        tmp->numOfChanges++;
    }
}

void LightOff::off() {
    cout << "already_off!" << endl;
}

void LightOff::destroy() {
    if(auto tmp = context.lock()) {
        tmp->state = allocate_shared<Destroyed>(tmp->alloc,context);
        tmp->numOfChanges++;
    }
}

void LightOff::status() {
    if(auto tmp = context.lock()) {
        cout << "current␣state␣is:␣OFF!" << endl;
    }
}
```

6.2.2 Timing

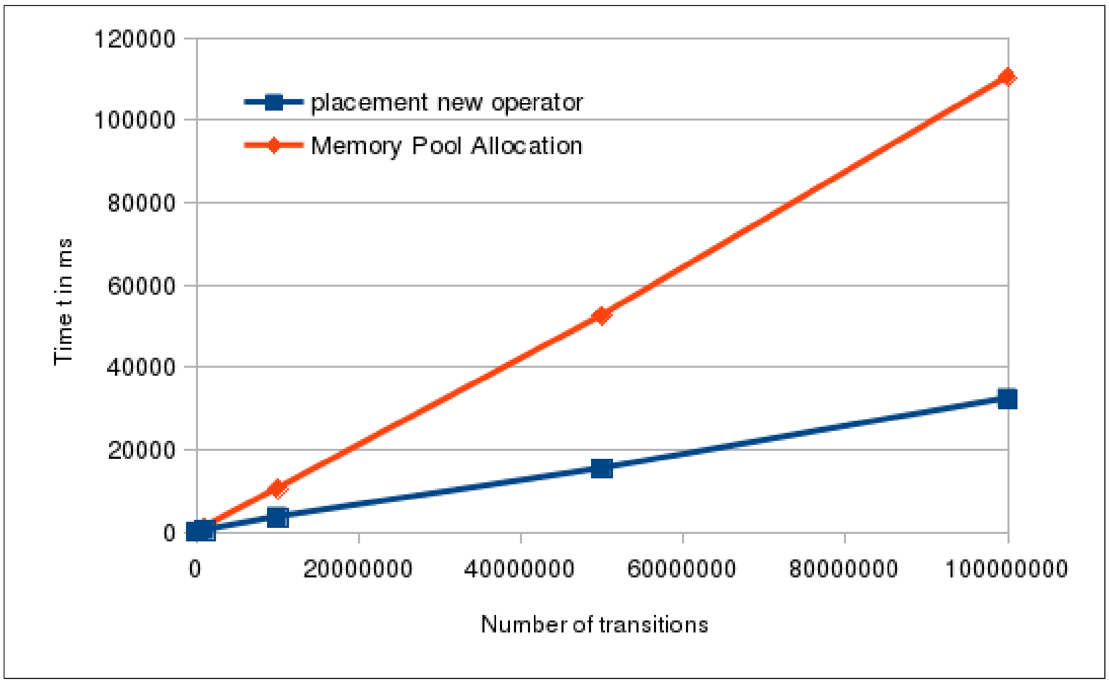

Figure 6.8: State pattern placement new timing

Using *allocate_shared* including an own memory management is more expensive in timing than using the *placement new* operator. Figure 6.8 shows the differences in timing between the *placement new* implementation and the *allocate_shared* implementation. For changing the state 100,000,000 times, the *placement new* implementation needs about 32 seconds and the *allocate_shared* implementation needs about 110 seconds. The comparison of profiling output of the *placement new* implementation (Table 6.3) with the profiling output of the C++11 polymorphism implementations (Table 6.2) shows a reduction of the execution time for smart pointers. The *placement new* operator does not instantiate a new smart pointer. It just instantiates a new refered object on the same memory address where the already existing smart pointer is refering to. This causes a reduction of about 24 seconds for changing the state 100,000,000 times.

Category	**Percentage**	**Time in s**
Context	39.34	12.73
Smart pointer	35.92	11.62
State	6.57	2.12
Main	6.21	2.01
other	11.95	3.86

Table 6.3: State pattern placement new timing distribution C++11

For every transition the *allocate_shared* implementation creates a new smart pointer. With 37,8% of the total execution time the use of *allocate_shared* is responsible for about 41 seconds. (Table 6.3) In addition to that, the used memory pool contributes to the execution time with about 23 seconds.

Category	**Percentage**	**Time in s**
Context	20.55	22.69
Memory pool	21.55	23.79
Smart pointer	37.83	41.77
State	4.20	4.63
Main	1.52	1.67
other	14.32	15.81

Table 6.4: State pattern pool allocator timing distribution C++11

6.2.3 Memory Consumption

Both, the *placement new* implementation and the *allocate_shared* implementation, have the advantage that transitions cause no memory allocation. Figure 6.9 shows a constant memory consumption independent of the number of transitions. Because of the memory pool, which allocates a number of memory chunks, the *allocate_shared* implementation has a slightly higher memory consumption than the *placement new* implementation. This overhead is dependent on the choice of the allocator. The choosen allocator of listing 6.11 does not provide the option to declare the number of pre-allocated chunks. For the state pattern only two chunks are needed.

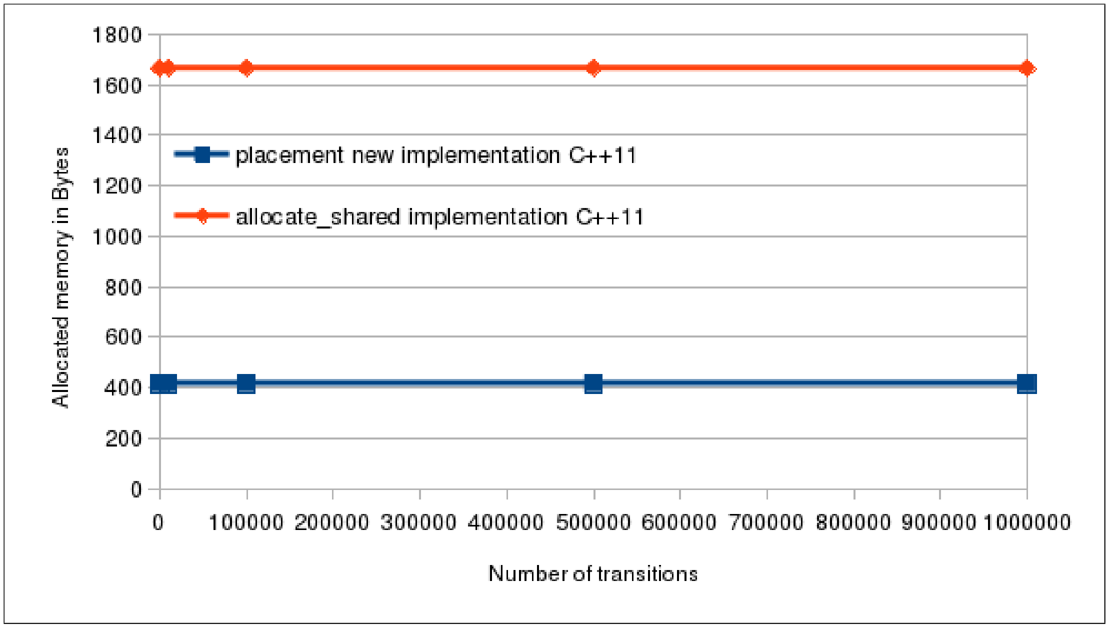

Figure 6.9: State pattern placement new memory consumption

7 Conclusion

Within the scope of this bachelor thesis it was the task to analyse the impact of the new C++ facilities, introduced since C++11, on a selection of design patterns which are commonly used within embedded software development.
With three differend versions of implementation for the factory method pattern as one of the two analysed design patterns, the profiling output has shown that using C++11 facilities mainly causes an overhead of execution time and memory consumption. As advantage the analyse has shown that C++11 provides simple ways of implementation with constant seek complexity independent of the number of managed classes and a memory consumption independent of the size of the managed classes.
The profiling output of two versions of implementation for the state pattern as the second analysed design pattern, confirms the overhead in execution time and memory consumption of the C++11 facilities. In addition to this, the analyse has shown how the wrong use of smart pointers causes memory leaks and how to combine the use of smart pointers with an own memory management.
An analysis of just two design patterns can only provide idea of how C++11 does impact design patterns. In general it has shown that C++11 does not increase the performance but it increases the quality of the sourcecode. This includes reusability as well as maintainability. Nevertheless C++03 does still have a right to live. After all, my opinion is to keep C++03 and especially raw pointers as possible option for projects with a limited amount of resources. But for most of the modern embedded systems with an adequate amount of resources, I think the cost effectivness of the use of C++11 facilities is appropriate and will benefit the project.

8 Perspective

Tha analysis has shown that modern standards more focus the quality of sourcecode and less the performance. A view into the list of the features probably released with the C++17 standard shows that this trend will continue. Many design patterns more show a tendency of improvement with the new standards. I think spending time in implementing more established design pattern with modern C++ facilities is the right way to be prepared for the futures requirements on software development.

Bibliography

[cppreference 2014] CPPREFERENCE: *std::vector*. 2014. – URL http://de.cppreference.com/w/cpp/container/vector. – Zugriffsdatum: 2016-03-21

[Ding 2012] DING, Zuoliu: *Something You May Not Know About the Switch Statement in C/C++*. 2012. – URL http://www.codeproject.com/Articles/100473/Something-You-May-Not-Know-About-the-Switch-Statem. – Zugriffsdatum: 2016-02-28

[ElectricalKnowhow 2013] ELECTRICALKNOWHOW: *Basic Elevator Components*. 2013. – URL http://www.electrical-knowhow.com/2012/04/basic-elevator-components-part-one.html. – Zugriffsdatum: 2016-03-28

[Gamma u. a. 2015] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. Bd. -. mitp Verlags GmbH und Co. KG, 2015. – ISBN 978-3-8266-9700-5

[Gessler 2014] GESSLER, Ralf: *Entwicklung Eingebetteter Systeme*. Springer Vieweg, 2014. – ISBN 978-3-8348-2080-8

[Kirch und Prinz 2015] KIRCH, Ulla ; PRINZ, Peter: *C++ Lernen und Professionell anwenden*. Bd. 7. mitp Verlags GmbH und Co. KG, 2015. – ISBN 978-3-95845-028-8

[Noergaard 2005] NOERGAARD, Tammy: *Embedded Systems Architecture*. Elsevier Inc., 2005. – ISBN 978-0-12-382196-6

[Oshana und Kraeling 2013] OSHANA, Robert ; KRAELING, Mark: *Software Engineering for Embedded Systems*. Bd. 1. Elsevier Inc., 2013. – ISBN 978-0-12-415917-4

[Pohmann 2013] POHMANN, Peter: *C++11 - Praxiswissen zum neuen Standard*. Bd. -. entwickler.press, 2013. – ISBN 978-3-86802-106-6

[Starke 2015] Starke, Gernot: *Effektive Softwarearchitekturen.* Bd. 7. Carl Hanser Verlag Muenchen, 2015. – ISBN 978-3-446-44361-7

[Stroustrup 2016a] Stroustrup, Bjarne: *The C++ Programming Language.* 2016. – URL http://www.stroustrup.com/C++.html#guidelines. – Zugriffsdatum: 2016-02-07

[Stroustrup 2016b] Stroustrup, Bjarne: *When was C++ invented?* 2016. – URL http://www.stroustrup.com/bs_faq.html#invention. – Zugriffsdatum: 2016-02-07

[Walls 2012] Walls, Collin: *Embedded Software The Works.* Bd. 2. Elsevier Inc., 2012. – ISBN 978-0-12-415822-1

[Wieringa 2003] Wieringa, Roelf J.: *Design methods for reactive systems.* Elsevier Science and Technology, 2003. – ISBN 978-1-55860-755-2

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 04. April 2016 Lars Christian Schwensen