DIPLOMARBEIT

A new Tabu Search Framework and it's Application

Ausgeführt am Institut für

Informationssysteme (184),
Database and Artificial Intelligence Group (184/2)

der Technischen Universität Wien

unter der Anleitung von: O.Univ.Prof. Dr. Georg Gottlob
Univ.Ass. Dr. Nysret Musliu

durch

Michael Mörz

Hans Kutra Gasse 10/2
1210 Wien

7, November 2006

Michael Mörz

Abstract

In the area of computer science heuristics are very important when it comes to solving NP hard problems. Though they neither provide an accurate method of finding a solution nor a provable approximation like Approximation algorithms, they nevertheless find reasonable good solutions to NP hard problems usually while consuming a reasonable amount of computation time and memory.

A well known class of Heuristic methods are Local Search techniques. Those techniques focus on starting with a solution, creating a neighbourhood for the solution by applying small changes to the solution, evaluating the solutions of the neighbourhood and choosing a solution of the neighbourhood for the next iteration. The algorithm terminates when the termination condition is fulfilled.

Whenever a problem is solved with Tabu Search in a program the structural and algorithmic definitions of Tabu Search have to be implemented. A solution to avoid reimplementing Tabu Search again and again, is to create a framework providing a basic design for Tabu Search. To remove the cumbersome process of defining those structural and algorithmic definitions the framework of this thesis provides an object oriented design for Tabu Search. In order to utilise it the developer has to subclass four classes and implement their missing functionality. Basically those four classes provide the following problem dependent information to the framework: how to store a possible solution for the given problem, how to evaluate such a solution regarding its fitness, how to represent a move operation on a solution and how to generate a neighbourhood by creating move operations. The rest of the Tabu Search framework can be used like a black box, taking the four classes as an input, and solving the given problem.

The state of the art research already provides us with a set of Modern Heuristic frameworks. Some of those frameworks are either already specialized on Tabu Search or they provide a limited, but functional Tabu Search algorithm. Neither of them provides a design for frequency memory. Therefore an enhancement of the framework in this thesis is to remove the negligence of frequency memory. So the framework of this thesis provides an approach for storing and querying frequency information. In addition it also provides the developer with tools for managing a list of elite solutions.

In order to verify the frameworks capability to handle Tabu Search algorithms, a NP-complete problem from literature has been chosen and an example program has been successfully implemented which solves the selected problem.

German Kurzfassung

Im Gebiet der Informatik sind Heuristiken besonders wichtig, wenn es um die Lösung von NP harten Problemen geht. Obwohl diese weder eine akurate Methode noch eine approximative, wie Approximative Algorithmen, zum Finden einer Lösung sind, können sie dennoch verhältnismäßig gute Lösungen innerhalb einer akzeptablen Zeit und mit akzeptablem Zeit- und Speicher-Verbrauch finden.

Eine bekannte Klasse von Algorithmen der Heuristic ist die Technik der Lokalen Suche. Diese basiert auf folgenden Schritten: Beginn mit einer möglichen Lösung, Erzeugung deren Nachbarschaft durch das Anwenden von kleinen Änderungen an der Lösung, Evaluierung der Lösungen und dem Neubeginn mit einer besseren Lösung. Wenn keine bessere gefunden werden kann, oder eine ausreichend gute Lösung gefunden wurde, terminiert der Algorithmus. Ein bekanntes Beispiel aus der Literatur für die Lokale Suche ist die Tabu Suche und mit dieser beschäftigt sich diese Diplomarbeit.

Sobald ein Program mittels Tabu Search ein Problem lösen soll, muß die Struktur und der Algorithmus von Tabu Search implementiert werden. Das Framework dieser Diplomarbeit erleichtert diesen Prozeß durch ein objektorientiertes Design, welches die Basis der Struktur und den Algorithmus für Tabu Search definiert. Um dieses Framework zu verwenden, müssen vier neue Klassen von vier abstrakten Klassen des Frameworks abgeleitet werden, damit folgende problemspezifische Informationen an das Framework weitergegeben werden: wie eine mögliche Lösung für das gegebene Problem gespeichert wird, wie eine derartige Lösung bezüglich ihrer Fitness evaluiert wird, wie eine Move Operation repräsentiert wird und wie eine Nachbarschaft durch Move Operationen erzeugt wird. Der Rest des Frameworks kann als Black Box gesehen werden, welche die vier Klassen als Input nimmt und als Output die Lösung des Problems liefert.

Der heutige Stand der Technik stellt uns bereits einige Moderne Heuristik Frameworks zur Verfügung. Einige dieser Frameworks sind entweder auf Tabu Suche spezialisiert oder stellen einen funktionellen aber eingeschränkten Tabu Suche Algorithmus bereit. Keines dieser enthält ein Design für das Speichern von Informationen über Auftrittshäufigkeit. Deshalb stellt das Framework dieser Diplomarbeit als Erweiterung im Vergleich mit dem heutigen Stand der Technik eine Möglichkeit für das Speicher und Abfragen von Frequenz Informationen bereit. Zusätzlich wird auch noch ein Design zum Verwalten von elitären Lösungen zur Verfügung gestellt.

Um die Anwendbarkeit dieses Frameworks für Tabu Suche zu überprüfen, wurde ein NP-complete Problem aus der Literatur gewählt, welches durch ein Beispiel Programm mit Hilfe des Frameworks erfolgreich gelöst worden ist.

Acknowledgements

Firstly, I want to express my thanks towards my supervisors Prof. Dr. Georg Gottlob and Ass. Dr. Nysret Musliu. Without their help including but not limited to their assistance, constructive criticism and invaluable support, this diploma thesis would not have been possible.

I would also like to thank the people of the university in general, especially the staff members I came in contact with during my computer science education. Their variety of understanding in the area of computer science heavily influenced the way I perceive computer science nowadays, which helped me creating this thesis as it is.

Last but by no means least I want to thank my family. My parents for their unfaltering support. Without their guidance and advice my life so far would not have been as gratifying. My brother for his help by reviewing my work and his helpful suggestions. My mother in law for looking after my little daughter giving me the time to create this thesis. My wife for her support, encouragement and guidance whenever I started to get distracted with less important things than my thesis. My little daughter just for being and giving her love freely whenever I needed a little day off time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays computer science is an ever growing area which influences common day life more and more. One of it's most prominent questions is to be found in theoretical computer science: The relationship between the complexity classes P and NP. This is studied in computational complexity theory which is concerned about the required resources for the computation of a solution for a given problem. The most interesting resources for research are:

- Memory consumption, inquiring how much memory it will consume to solve the problem.
- Time, measuring how many steps it takes to solve a problem.

In the context of the computational complexity theory the class P can be explained according to Ausiello et. al [1] as: Class P consists of all those decision problems that can be solved on a deterministic Turing machine in an amount of time that is polynomial in the size of the input.

Ausiello et. al [1] also detail class NP that consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently whose solution can be found in polynomial time on a non-deterministic Turing machine.

So one of the biggest open questions in theoretical computer science concerns the relationship between those classes:

Is P equal to NP?

Actually this is a question that is not answered yet. Decades of research just have proven that it is a very difficult question, and today some computer scientist even believe that the question may be independent of the currently accepted axioms, and therefore impossible to prove or disprove. Afterall this diploma thesis cannot answer this question, yet it is concerned with methods for solving NP hard problems as explained below.

As Ausiello et al. [1] show: in complexity theory, the NP-complete problems are the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. The reason is that if you could find a way to solve a NP-complete problem quickly, then you could use that algorithm to solve all NP problems quickly.

As presented NP-complete problems are not directly solvable in an exact manner by computers within a feasible amount of time. To alleviate this problem other ways to find a solution are needed. Therefore other approaches like randomized search, approximation algorithms, heuristics or metaheuristics have been devised.

According to Vazirani [36] an approximation algorithm solves NP hard problems in polynomial time, but normally provides non-optimal solutions, with a provable solution quality and within provable run time bounds.

A heuristic is a replicable method or approach for directing one's attention in learning, discovery, or problem-solving. It is originally derived from the Greek "heurisko", the verb which Archimedes' famous exclamation of "eureka" was derived from, which means "I find". In contrast to approximation algorithms, heuristics are only concerned to find reasonably good solutions in a reasonable amount of time, but offer no performance guarantee.

A metaheuristic is a heuristic method for solving a very general class of computational problems by combining user-given black-box procedures, usually heuristics themselves, in a hopefully efficient way. Actually *meta* is again a Greek word which means "beyond".

An advanced and well known modern metaheuristic search technique class is Local Search. That is a collection of several methods that have a special structure for solving problems. According to Michalewicz and Fogel [28] Local Search techniques have the following steps in common:

1. Pick a solution from the search space and evaluate it's merit. Define this as the *current* solution.

2. Apply a transformation to the current solution to generate a new solution and evaluate its merit.

3. If the new solution is better than the current solution, exchange it with the current solution; otherwise discard the new solution.

4. Repeat steps 2 and 3 until no transformation in the given set improves the current solution.

While this structure shows a basic Local Search algorithm, there exist more sophisticated local search algorithms like tabu search, simulated annealing and others. They have in common that they step from one solution to another in the space of candidate solutions, which is also called the *search space*.

The class of problems solvable by Local Search can be formulated as finding a solution maximising or minimising a criterion among a number of candidate solutions.

A mathematical notation expressing a broad class of the solvable problems of Local Search algorithms is now given as a basis for a discussion of Local Search and Tabu Search features. As proposed by Glover and Laguna [16] the class of problems can be characterised as minimising or maximising a function $f(x)$ subject to $x \in X$, where $f(x)$ may be linear or non-linear, and the set X summarises constraints on the vector of decision variables x. The constraints may include linear or non linear inequalities, and may compel all or some components of x to receive discrete values.

Taking that definition of a problem and utilising Local Search, each $x \in X$ has got a neighbourhood $N(x) \subset X$, and each solution $x' \in N(x)$ is reached by

an operation called *move*. When applying that mathematical notation to the presented structure of Local Search algorithms the resulting structure definition is:

1. Pick a $x \in X$ to start the search
2. Define $N(x)$
3. Find the best x' where $x' \in N(x)$
4. If no such x' can be found or time bound is exceeded terminate, otherwise go to 2.

This Thesis is concerned with Tabu Search, which is a heuristic algorithm and a Local Search technique, offers several useful features as outlined in detail by chapter 2.1. Some of the most useful features are for example:

- determinism - starting at a specific point, the search will always make the same steps and end in the same solution. Therfore the path of search can be reconstructed just by a giving a starting solution.
- dynamic neighbourhood - even when a solution is visited more than once, it's neighbourhood is very likely to be different than the last times, since the list of tabu moves will be different.
- avoids cycles when the tabu tenure is set properly.

Anyway the basic structure of Tabu Search is always the same according to the nature of heuristics. So whenever a problem is solved with Tabu Search the complete structure and algorithm has to be written from scratch. Therefore everyone implementing Tabu Search has to spend time on creating a design and implementing it. Afterwards, the implementation has to be tested and errors have to be fixed. So creating a Tabu Search implementation is known to be an error prone task like any other programming task. Moreover, one cannot benefit from the design and implementation ideas of others easily. This is neither a technological nor economical feasible situation. To alleviate that situation, software reuse is utilised. Two important forms of software reuse are libraries and frameworks.

Still, there is a big distinction between libraries and frameworks, since the former is normally concerned about function reuse and the latter is about design reuse. Therefore they provide different sets of capabilities as outlined later in chapter 2.2. A very important feature concerning reuse is how much time the developer has to spend on learning how to use the reusable software component properly. According to Mohamed Fayad et. al [10] frameworks need more time to be learned using them compared to libraries. Although that is a disadvantage, frameworks are much more flexible since they try to provide the user with a generic design avoiding to limit the user to a specific functionality. Anyway the line between libraries and frameworks is rather fuzzy, therefore it is hard to distinguish if a software is one or the other, because it might offer features of both approaches.

That leads to the interesting question whether it is possible to reduce the work needed to create a Tabu Search algorithm by providing some sort of library. Because Tabu Search has got a basic structure and algorithm, it might imply

that an implementation as a library or framework might be beneficial. Actually it is, as the sheer number of state of the art frameworks for heuristics might suggest. Their details are outlined in chapter 2.3.

The intention of this diploma thesis is not to simply design and implement an other framework for Tabu Search, but to give a framework for Tabu Search that has been extended with a feature not yet seen in any of the state of the art frameworks. That feature is implemented in the form of a new design part which has not yet been used by state of the art frameworks. This is accomplished by providing the developer with tools for extracting frequency information. The developer is given the possibility to query how often a specific move or solution has been visited in the past of the search process.

To ease legibility personal pronouns are limited to the male gender throughout the text, although female gender is also referred by them.

This thesis starts with an explanation of the Tabu Search algorithm and its features. The basic structure and algorithm of Tabu Search is outlined. Since several people contributed and influenced the evolution of this algorithm the most important ideas are explained. Afterwards the topic "frameworks" and their significance is discussed along with a general overview about the design of frameworks. Then necessary tools are explained like object orientation, which is used in frameworks, and the Unified Modelling Language, that is frequently utilised to describe object oriented designs. Then chapter 2 is concluded with an overview about the state of the art heuristic and metaheuritic frameworks. Some of those frameworks are solely committed to Tabu Search, some of them provide additional algorithms besides Tabu Search and some of them are metaheuristic frameworks without an explicit implementation of Tabu Search.

Following the presentation of all those details, chapter 3 concentrates on the object oriented design of the framework of this thesis. It starts with the initial design requirements that have been proposed prior to creating the design. Those are then compared with the state of the art frameworks and similarities and differences are outlined. Then an overview about the general interaction between the classes is given. This is followed by an explanation of the interfaces used. Afterwards each class is explained in detail by describing it's attributes, methods and the way they are expected to interact with other classes. As a conclusion the implementation specifics regarding C++ and linux are given.

To verify that the framework is truly capable of handling Tabu Search applications, a program has been written solving a NP-hard problem by utilising the framework of this diploma thesis. The problem solved by the program is a NP-complete problem from literature, which is explained in chapter 4 along with the implementation specifics of the program. Afterwards the results of the program are given and compared to literature. Additionally the most important parts of the source code of the program can be found in Appendix A along with a in depth discussion.

The finishing touch is given at the end of the thesis when the conclusion and perspectives are discussed. Additionally improveable areas of the framework are pointed for possible future work.

Chapter 2

Basics and state of art

The following chapter starts with a description of the modern heuristic algorithm called Tabu Search. It is followed by a characterisation of the general principles of framework design. Afterwards an overview of the state of art work in the combined area of both topics will be given.

2.1 Tabu Search

The modern heuristic, called Tabu Search, derives from Glover [15]. Others have contributed to the algorithm by suggesting modifications:

- Reactive Tabu Search as explained by Battiti and Tecchiolli [2]
- strict Tabu Search introduced by Glover and Laguna [14]
- robust Tabu Search defined by Taillard [34].

According to Reeves [32] those contributions to the algorithm are still influencing the evolution of the method and they are also responsible for the growing number of successful applications. Therefore Tabu Search has become a powerful modern heuristic technique.
The Oxford dictionary defines 'tabu' as:

Its name tabu is derived from a Polynesian language where it is used to indicate things that cannot be touched because they are sacred.

However, the most important connection to traditional use is the fact that tabus are transmitted by means of social memory and can change over time. And that is, what makes Tabu Search so different from other search techniques - the search process is guided by the known past of it.

2.1.1 A short description of the algorithm

Before any specific details are given, the term "solution" has to be defined to avoid any confusion, since in the context of this text that term is used with relaxed definition. Therefore a "solution" does not necessarily mean to be a final solution of the problem. A "solution" is just an element of the space of candidate solutions presented in chapter 1.

Tabu Search is a local search technique and therefore follows the local search technique's common structure as presented in chapter 1. Since the following section gives an in depth discussion of the relationship of Tabu Search and Local search, the common structure is listed again for a better understanding:

1. Create initial solution
2. Create neighbourhood for the chosen solution
3. Pick best solution in neighbourhood
4. If solution is not good enough, then continue at 2.; otherwise terminate search.

Step 1 contains the challenging task to choose an initial solution. There are two commonly used methods for finding such a solution. On one hand there is randomisation which provides a completely random solution. On the other hand there is the construction of a solution that satisfies a set of constraints. This can be beneficial when the move operations are designed in a way that they don't disturb the already satisfied set of constraints, since it reduces the number of computations necessary for finding a solution. Anyway it is impossible to always fulfil all constraints by this method according to the nature of NP hard problems.

As the structure shows, some part of it is repeated. In computer science such a repetition of a process within a computer program can be called iteration according to the Dictionary of Computer Science [23].

The constructed initial solution is the starting point for the iterative algorithm. For a better understanding the starting solution of an iteration is called the *current solution*. Therefore the first current solution must be the initial solution. For each loop a neighbourhood of the current solution is created by applying small changes with so called moves to the current solution. When a move is applied, a look up in the memory of the Tabu Search takes place. If the move is found in the tabu list there, it'll be tabu, otherwise it's non tabu. In case it's tabu, the move is ignored as long as there is no aspiration criteria that overrides the tabu status. Since aspiration criteria is a rather complex topic it is discussed in detail below in Aspiration Criteria 2.1.5. At the moment it's just necessary to know that aspiration criteria can override the tabu status and that makes them useful for escaping local optima.

While creating the neighbourhood, a search for the best solution is conducted within it. That solution is found by evaluating the solutions in respect to the constraints that should be satisfied. When a better solution is found, the algorithm starts over with that new solution. If there is no better solution, further computations will be done which are outlined in greater detail in chapter 2.1.5.

Normally the iteration terminates when a threshold is reached like a maximum number of iterations, by finding a really good solution or by exceeding some time limit.

An important difference of Tabu Search compared to other Local Search techniques is it's deterministic behaviour. A good definition for deterministic is given by the Dictionary of Computer Science [23] as: *deterministic means permitting at most one next move at any step in computation.* That implies that

a deterministic computation in computer science is a computation that starting with a specific initial state will always produce the same final state when given the same input. Actually this is an important feature of Tabu Search that has to be kept in mind when creating moves, constructing the neighbourhood and choosing the next solution.

2.1.2 Moves

Creating the neighbourhood of a solution is essential for Local Search techniques in general. That makes it an important topic that needs to be discussed in depth.

In Tabu Search small changes are applied to the current solution in order to construct its neighbourhood. How and where those small changes are induced, is defined in operators that are so called *moves*. Therefore a *move* is making small changes to a solution, which creates a new solution in turn.

Such moves are not atomic and can be broken down into so called *move attributes*. According to Michalewicz and Fogel [28] an attribute of a trial move from x^{now} to a tentative solution x^{trial} can encompass any aspect that changes as a result of the move. So a move attribute can be a specific change in the solution's features. For example that might be the change of a variable from one value to an other. Therefore a move can consists of one or more attributes. Attributes themselves may be a set of other attributes, though that doesn't necessarily mean that they contain more information that can be used to contribute for a better search process.

Any attribute of a move may be broken down even further into so called *component attributes* called from-attribute and to-attribute as shown by Fogel and Michalewicz[28]. Those component attributes are taken from the attributes of the x^{now} or the x^{trial} solution for the from- or the to-attribute accordingly. This information is important when storing tabu information as described in the following paragraphs.

Either a complete move is stored and it's reversal is then forbidden, or the move attributes that compose the move are stored and their reversal is forbidden. Forbidden moves are called *tabu*. Storing the tabu state of the attributes of a move gives the tabu evaluation a finer granularity, because moves that have never been done before, may also be tabu. Forbidden move attributes are said to be *tabu active*. The tabu status of those moves is caused by having one or more tabu active attribute. This stems from the fact that they might contain one or more move attribute that are tabu active.

2.1.3 Tabu memory

According to Glover [15] the memory structure in Tabu Search operates by reference to four principal dimensions: quality, influence, recency and frequency.

- Quality stands for identifying elements that are common to good solutions or paths that lead to them.
- Influence is about the impact of a choice to make a certain move. So it records information about the impact to change a specific element.
- Recency stores the elements that have been changed in the past of the ventured search path.

- Frequency stores for each element, that has been modified in the past, how often it has been changed.

There are two styles of memory used in Tabu Search and they are called *explicit* and *attributive*. *Explicit* memory records complete solutions, which is normally used for remembering very good solutions visited during the search process. Those very good solutions are also called "elite solutions". The purpose of *explicit* memory is to expand the search by providing information that can be analysed and applied to step to unvisited parts of the search space.

Quite in contrary *attribute* memory records the changes that where made while advancing from one solution to the next. That information is used to guide the search because it is utilised for inhibition or encouragement of following a certain search direction.

So *explicit* and *attribute* memory is used to construct the short term and the long term memory of Tabu Search. As their names suggest one stores short term information and the other long term data about the search process. Anyway both influence the generation of a modified neighbourhood $N^*(x)$ which is $N^* \subset N(x)$ where $N^*(x) = \{x' \in N(x) | \forall x' \text{ that are not tabu}\}$.

That shows the importance of the tabu classification which is normally stored in short term memory when a simple Tabu Search algorithm is used. Once an element is declared tabu and stored in short term memory, it modifies the neighbourhood. To avoid blocking an element forever, the so called tabu tenure has been introduced. That is a boundary that defines how many iterations an element has a tabu classification. Accordingly this is a parameter that heavily influences short term memory, since it defines how many elements it will contain. For example a tabu tenure of 5 suggests that a chosen element is tabu for the next 5 iterations. Therefore a maximum of 5 elements will be in the short term memory. It is important to note that short term memory stores information in the recency dimension of the four principal dimensions of Glover [15] mentioned above. In contrast long term memory is used to store frequency information and therefore operates in the frequency dimension.

2.1.4 Intensification and Diversification

Intensification is the encouragement of moves and solution features that have been found good in the history of the search. It concentrates on examining the neighbours of elite solutions, where neighbours are not only solutions that are reachable by standard moves, but also solutions that have been created by the components of good solutions. Since intensification needs the history of elite solutions it is tightly linked to explicit memory.

The counterpart to intensification is diversification, since it focuses on exploring unvisited regions of the search space and on generating solutions significantly different to those seen before. This technique is important when trying to escape from a local optimum.

A possible approach to the computation of a moves intensification or diversification value is to analyse elite solutions, as already defined in chapter 2.1.3 and the moves that lead to them. When comparing the moves by breaking them down into so called move attributes and comparing those, specific patterns can be recognised about the move attributes as listed here:

1. move attributes that often lead to elite solutions can be said to have a high intensification.

2. move attributes that don't lead to elite solutions have a bigger diversification.

2.1.5 Aspiration

When no better solution can be found within the neighbourhood of non tabu solutions, the search is expanded into the space of tabu solutions as shown by Glover and Taillard [17]. Aspiration criteria define the part of tabu solutions that are examined; e.g. those might be removing tabu restriction when a move yields a solution that is better than the best obtained so far.

Of course different aspiration criteria can lead to different tabu restrictions to be removed and in turn different solutions could become non tabu. To decide which of them takes precedence some sort of measurement is needed. One possibility is to create a static order of evaluation of aspiration criteria. An other possible approach is to measure the significance of an aspiration criteria at a specific point in the search by calculating its influence. Fogel and Michalewicz[28] show a precise definition for the influence of aspiration criteria: *Influence measures the degree of change induced in solution structure or feasibility.* They add that high influence with it's high degree of change is important for escaping from local optima. So choosing between aspiration criteria also involves the current search strategy regarding intensification and diversification. It may even go beyond that by inducing a change to that strategy in favour of diversification.

Aspiration criteria can be split up into two categories: One so called *move aspiration* revokes a move's tabu restriction. The other is named *attribute aspiration* meaning the revocation of a move attribute's tabu active status.

Actually aspiration criteria are usually dynamic and adapt themselves to the current situation of the search. For example an aspiration criteria using influence tolerates moves of lower influence until the gain seems negligible. If that is the condition and no improving moves can be found, the aspiration criteria should shift towards influential moves.

When looking at tabu attributes and taking aspiration criteria into account, a third state can be introduced as defined by Fogel and Michalewicz [28]: A pending tabu attribute is an attribute where aspiration is satisfied and that is otherwise tabu-active.

2.2 Frameworks

Nowadays reuse is a very important aspect not only in software development but in system development in general according to Gamma et. al [11]. This stems from the requirement to shorten the development time and money needed in creating systems. The basic idea of reuse is to create a system by utilising components that can be used again when building similar systems. Since those components only need to be built once, the development time can be reduced significantly, when the components have been created and documented properly.

Therefore the concept of "reuse" helps in software development, when it saves time and effort by removing the ordeous task to recreate software that already has been built once.

Even when looking at simple software systems an inherent complexity can be seen. According to Champlain and Patrick [5] two major paradigms have dominated the software landscape to tackle that complexity.

The first and older paradigm divides development work into two distinct parts:

- Identification of real world entities and mapping as structures or records (data).
- Writing of subprograms to act upon the data (behaviour).

This method of development is called procedural approach. The primary disadvantage of this method is the separation of data and behaviour. This happens when subprograms utilise global variables for sharing data and that leads to a distribution of behaviour between those subprograms. Therefore difficulties can arise when testing, debugging or maintaining procedural applications.

The second paradigm divides the development work into two very different tasks:

- Identification of data and behaviour of real world entities and subsequent encapsulation into a single structure called class.
- Creation of objects from the classes and interaction between those objects for providing a solution to the given problem.

Therefore this paradigm is known as the object oriented approach, which can provide reuse and it's benefits.

2.2.1 Object orientation

As already explained, Object orientation itself is concerned with classes which are structures that relate to real world entities. According to Booch and Grady [3] detail how objects and classes interact and interrelate as given below. An object is created from a class, which is also referred to as an *instantiation* of the class. It consists of so called *features*, which can be split into attributes and operations. The attributes represent the data of the real world entities and the operations contain the behaviour of the real world entities. However, object orientation does not only revolve around objects and classes, but also utilises abstraction, inheritance, polymorphism and encapsulation for further describing aspects of the object. Important parts are message sending, associations and associations for detailing interactions and relationships between the classes and objects.

Abstraction is to filter the object's attributes and operations until only the ones you need are left.

Inheritance is a special relationship between two classes, one referred to as superclass, the other as subclass. The subclass inherits the features of the superclass, meaning the subclass has got the attributes and operations of the superclass.

Polymorphism only concerns operations with the same name throughout different classes, when those operations differ in their internal working. For example, open() can be an operation for a book, newspaper or a window. For

a newspaper and a book they are similar, but compared to a window, it's quite different and therefore polymorphism is needed to override the open() operation.

Encapsulation is the process of information hiding. Its intention is to hide the way how an operation internally works. The advantage of this approach is to reduce the amount of work needed when methods of a class malfunction due to coding errors. Since the functionality is encapsulated within a method of an object, most likely only the malfunctioning operation needs to be fixed and any other classes interacting with that operation don't need to be changed. The set of operations a class presents to the outside can be called "interface" of the class.

Message sending is: Objects work together by sending messages to each other. So one object sends a message to an other object which contains a request to perform an operation and the other object executes it.

Associations are relationships between objects that can be uni- or bi- directional. An important attribute in this context is the multiplicity of an association which defines how many objects of a class relate to one object of an other class. Commonly used multiplicities are one-to-one and one-to-many.

Aggregation is a special kind of association, which describes that a set of classes compose an other class. For example a car is consists of tires, the gear stick, trunk, etc. and therefore the car is an aggregation of them. A strong relationship between an aggregate object and it's components, where the components only exist within the composite object, is called composition.

Object Orientation was first introduced in a computer language by Smalltalk. Since performance has always been crucially important, other languages that aimed at performance were more widely accepted and used than Smalltalk. For example C is such a language that was created by Dennis Ritchie in the early 1970s, but it doesn't provide an object oriented approach. So Bjarne Stroustrup designed C++ which is based on C and extends it with object orientation. Nevertheless C++ doesn't provide a complete "everything is an object" approach, like Smalltalk did. In C++ it's rather an option that can, but doesn't need to be utilised. An even newer language is Java, which is known for it's Just in Time compilation feature, that allows execution of a program on a variety of platforms and is mostly independent of its hardware features. Although Java was built with object orientation in mind and forces the user to utilise classes and objects, it still has got gaps in the object orientation like data types that aren't objects. A very recent language is C# utilising the "everything is an object". Since computer hardware also has reached a certain speed, the penalty caused by C# is going to be neglectable. Therefore C# is standing a good chance to be widely accepted and utilised as a modern computer language. An other important factor of course is Microsoft, which has developed and introduced C# and therefore has a high interest of making it a standard.

Not even the most powerful and perfect computer language can guarantee the success of any software project nowadays, since they don't focus on programming anymore. In the early days of software development drafts were merely more than something written on the back of a napkin, and programming the software started right away. Today it's crucially important that the client can see the direction of the development and can correct it when the needs of the client aren't properly satisfied. An other important role plays the developer who needs to know how he should do his work and also needs an understanding of the "big picture" and how he should fit in his work. Therefore the client, who

orders the software project and gives insights into his business process, talks to the analyst. The analyst in turn creates a design and a documentation that needs to be read by the client and by the programmer. Needless to remark that this documentation needs to be understood by the client and the programmer in the first place, otherwise the outcome wouldn't be likely to be satisfying to the client.

According to Schmuller [33] before the Unified Modelling Language (UML) was defined, there has been no standard defined about how to design, create and document a model for a software project.

As computer technology advances, software systems with higher complexity can be implemented. So the questions arises how to get your hands around the growing complexity. Schmuller [33] explains that the key is to organise the design process in a way that the analyst, the client, the programmer and others involved in system development understand and agree on it. The Unified Modelling Language provides that organisation.

2.2.2 Unified Modelling Language (UML)

Grady Booch, James Rumbough and Ivar Jacobson are the initial creators of UML. Each of them started their work independently from each other and improved it by borrowing bits and pieces from each other's methods. As employees of Rational Software Corporation they started together their drafts about UML. Those drafts found quite a good response in the software development area and created feedback which induced changes to the draft. At that point many companies found the idea of UML promising and so they founded the UML consortium. Members of that consortium were Hewlett Packard, Microsoft, Oracle, Texas Instruments, Rational and others. That consortium produced version 1.0 of the UML and all subsequent version.

UML itself is a modelling language that is composed of a variety of different diagrams described below. Those diagrams can present different views of a system describing a model of the system. Therefore UML provides a good way to describe systems in general. The parts of UML presented below not only give an overview about the capabilities of UML, but also try to go into detail whenever it is necessary for understanding the description of the diploma thesis framework design later on.

Class Diagram

A class diagram is used to model object oriented design. A class is represented by a rectangle, where the top contains it's name, the middle it's attributes and the bottom it's operations. The attribute or operation section can be left empty or not present at all, but that doesn't imply that the class necessarily lacks attributes or operations. It just expresses that they're not shown. Because that might be confusing to the reader the attribute or operation section might be terminated with a ... in order to signal that not all attributes or operations are shown. That process is called an *ellipsis* and the verb for it is to *elide*. A good example for a class is presented in figure 2.1 (a). It shows a single car class, with attributes for the number of doors, the vendor and the gasoline load and methods for starting/stopping the engine, breaking and steering. Whereas in figure 2.1 (b) an ellipsis of the same class can be seen.

As the explanation of object orientation already has shown, it not only consists of classes, but also several other parts. So their definition in UML is detailed in the following paragraphs.

Inheritance which is also called generalisation in UML, because the superclass generalises the subclass. So a child or so called subclass inherits the attributes and operations from a parent or so named superclass. In the diagram this is represented by a line running from the child to the parent that ends with a triangle pointing to the parent as shown in figure 2.2.

A special case is a class that has got no parents and therefore it's called a *base* or *root* class. An other special case is a class lacking children, which is called *leaf* class. In addition to those cases a differentiation between *single inheritance* and *multiple inheritance* is also conducted, the former for classes having got only one parent, the later for classes having more than one parent. A further special case is a so called *abstract* class, which is a class that doesn't provide objects.

A different kind of relationship compared to the ones explained so far is the so called *dependency*. A dependency shows that the signature of an operation is using an other class. In the diagram it is represented by a dashed line leading from the operation to the class ending with an arrow head pointing to the class.

As previously mentioned *associations* can be uni- or bi- directional. Further bi-directional associations can be split into uni-directional ones as shown in figure 2.3 by (a) as the bi-directional and (b) the uni-directional case for the relationship between a tee-shirt and it's sleeves. Additionally the association can have a name expressing the relationship, so that it can be read like: a tee-shirt has got a sleeve or a sleeve belongs to a tee-shirt. The multiplicity of an association can be shown by writing it's numbers at the ends of the association, e.g.: A customer of a garage can have one or more cars (one-to-many) and a car can have one customer (one-to-one). One or more is a special case that is encoded with a * in UML. Figure 2.3 (c) shows that customer - car association in an UML diagram.

An *aggregate* class's association to a set of component classes is also called "whole part" association in UML. It is represented by a line running from the component class to the "whole part" class ending with a diamond as can be seen in figure 2.4 (a).

As already defined, a composite is a strong type of aggregation, which is shown by the same symbol as the aggregation with the exception that the diamond is filled as can be seen in figure 2.4 (b).

An *interfaces* consists of a set of operations, which define specific behaviour and are accessible by other classes. The relationship between an interface and a class is called realisation, because the class has to "realise" the behaviour of the operations of the interface by implementing them. In the class diagram this is expresses by a dashed line between the class and the interface ending with a triangle pointing to the interface.

Every class feature has a level of visibility, which defines the accessibility of the feature by other classes.

1. The **public** level is accessible by all classes.
2. The **protected** level only by classes inheriting from the original class.
3. The **private** level is only accessible by the original class itself.

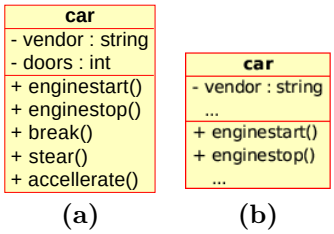

Figure 2.1: UML class diagram

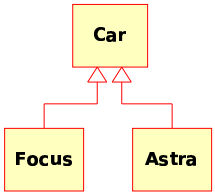

Figure 2.2: UML class diagram with inheritance

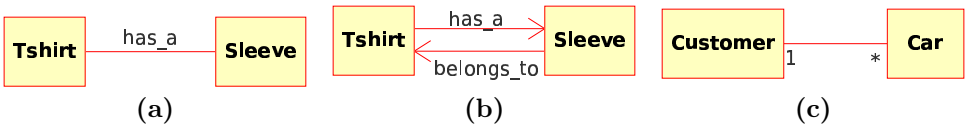

Figure 2.3: UML class diagram associations

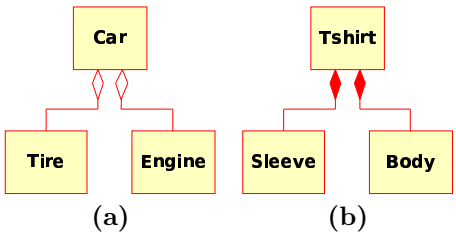

Figure 2.4: UML class diagram aggregation and composition

In the diagrams **public** attributes and operations are prefixed by a '+', **protected** ones by a '#' and **private** ones by a '-'. An example for the different levels of accessibility can be seen in figure 2.1 which only shows private attributes, one protected method called and the rest of the operations are public. Following list of notation conventions concerning the name construction for attributes and operations, which are introduced in Schmuller [33], have been used throughout the paper:

- The name of a class begins with an uppercase letter
- A multi word class name runs all the words together, and each word begins with an uppercase letter.
- The name of a feature (attribute or operation) begins with a lowercase letter
- A multi word feature name runs all the words together, and each word begins with an uppercase letter except for the first one.
- A pair of parentheses follows the name of an operation.

This naming schema allows an easier and quicker distinction between classes and their features, which is perhaps overkill in a diagram, but when the implementation in a programming language starts, it improves legibility.

Object Diagram

As we already know, an object is an instance of a class. Figure 2.5 shows the representation of an object in UML. It's a rectangle with an underlined name where the left part before the colon is the object name and the right part is the class name. Anonymous objects can be defined by providing the class name only as seen in figure 2.5 with the Tires class. The necessity for *object diagrams* might not be directly visible, because *class diagrams* might provide the same information. Still they can show information that cannot be represented within *class diagrams* by describing a special case and the way the objects interact with each other.

Use Case Diagram

A *use case* diagram describes a system from the user's point of view. Therefore it gathers information about a system from the user's standpoint. That explains its importance for building a system when focusing on usability, which expresses how good a user can use the system. Figure 2.6 shows the *use case* within the rectangle and the little stick figure is the so called "actor". In general the actor can be a person or another system that initiates the *use case*. Important to note is that the "actor" is always outside the use case.

State Diagram

State diagrams capture the state of objects and the transition between the states. For example figure 2.7 shows the starting, ignition and running of the engine.

Sequence Diagram

Since objects interact with each other in a running system, those interactions need to be described too. This can be done by utilising *sequence diagrams*. They capture the sequence of the messages getting sent between the objects. A message sent from one object to an other is represented by an arrow leading from the sending object to the receiving one. The time line is encoded in the vertical dimension of the diagram. Processes, consuming time inside a class, are drawn as rectangles on the time line of the corresponding object. As an example, figure 2.8 shows some of the interactions of the electronics, the gasoline pipe and the engine.

When putting a scenario into a sequence diagram, a so called *instance sequence diagram* has been created. When including all of a use case's scenarios while creating the diagram, it is named *generic sequence diagram*. The UML 2.0 standard permits labelling of a sequence diagram by encapsulating it in a box, tagging it with its name. This opens up the opportunity to replace repeating sequences by a single box with an inscribed sequence name in sequence diagrams.

Communication Diagram

Describing how elements of a system work together, can not only be done by utilising the so called sequence diagram shown in figure 2.8, but also by presenting the information in communication diagrams. As presented in figure 2.9 the communication diagram shows the same information as the previously mention sequence diagram, but instead of encoding time in a vertical direction, the order of the messages is marked by numbers.

As the sequence and the communication diagram display show interactions between objects, UML summarises them by the term "interaction diagrams".

Component Diagram

Today software development breaks software systems into components in order to create the possibility of semi independent development. Since this is rather important for team based development, it should be used when more than one developer is going to implement the design described in UML. The figure 2.10 shows an example component.

Deployment Diagram

To present the physical structure of a computer based system, a deployment diagram can be used. For example a system with two HP Proliant servers and a NetApp storage machine can be seen in figure 2.11.

Conclusion

The Unified Modelling Language has been presented as a modern tool for creating and describing designs so that the analyst, the client, the developer and others have a common basis for talking about a design of a system.

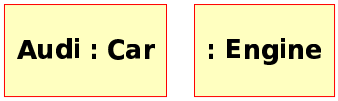

Figure 2.5: UML object diagram

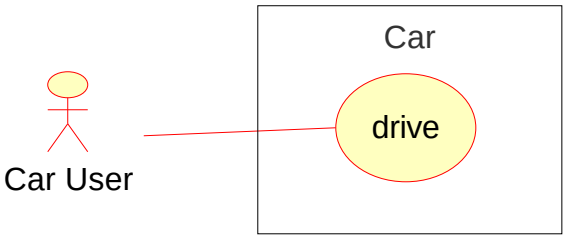

Figure 2.6: UML use case diagram

Figure 2.7: UML state diagram

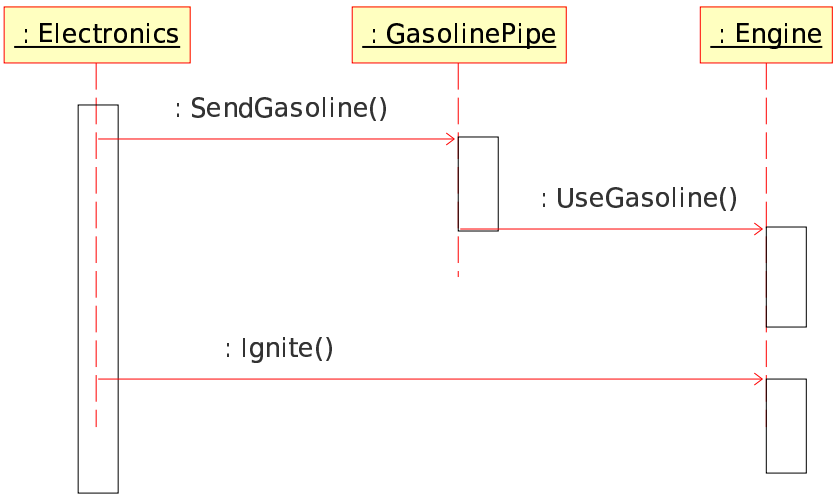

Figure 2.8: UML sequence diagram

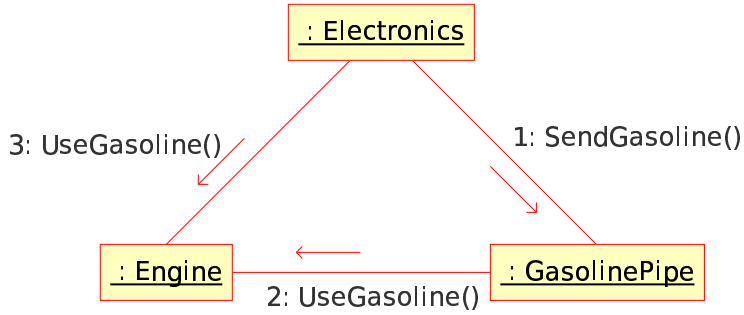

Figure 2.9: UML communication diagram

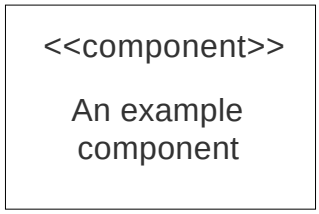

Figure 2.10: UML component diagram

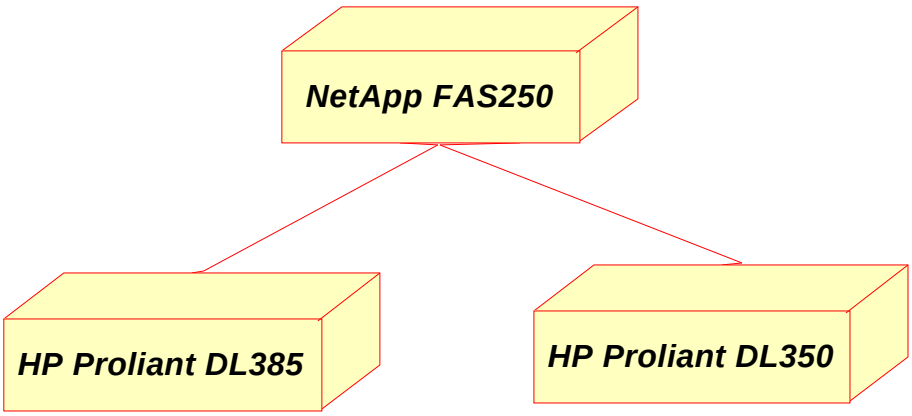

Figure 2.11: UML deployment diagram

2.2.3 Definition of Frameworks

Today frameworks try to provide developers with an object oriented design that is aimed at reuse. Although that gives an idea about frameworks, it scarcely describes them. So the main purpose of frameworks is best characterised by the definition of Mohamed Fayad et. al [10]: *"A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact."*

Of course that definition does not provide a complete insight into all tasks that are needed to create a framework. Nonetheless this gives a good hint about the idea that is driving the creation of frameworks. Reuse, abstraction, interaction and the design are however not the only topics that a good framework should be concerned about.

As the definition states, frameworks have a static aspect of object interaction which is predefined by the designer of the framework. So to some extent a framework determines the overall structure and flow of any program that utilises it.

An area that is most often covered with frameworks is Graphical User Interface (GUI) programming. Those kind of frameworks like to use the Model/View/Controller, also called MVC, concept. It was first introduced by Trygve Reenskaug in the user interface of Smalltalk. Actually MVC is not a real design pattern, when definitions are applied precisely, since it's not generically applyable, but rather a specialisation for graphical user interfaces.

MVC splits up the software system into three entities: Model (for storing the data), View (for presentation of the data) and Control (for the program logic). Its goal is a flexible design of the program allowing to change or extend the functionality and to reuse single components. In addition it reduces the complexity and therefore introduces an organisation and overview into the overall design, that can be used to distribute roles for the different parts of the model accordingly:

- Control - the software design is implemented by a programmer
- View - is created by a GUI designer
- Model - database experts take care of design and proper data storage.

Frameworks can ease and simplify development greatly when they provide design patterns like the Model/View/Control example shows. But frameworks do not only offer benefits. For example one drawback is their time consumption when creating a framework, since more time has to be spent on building an easily understandable and generic design and it's documentation. Therefore it also needs more knowledge when designing a good framework compared to function libraries which are quite straight forward in development. Another disadvantage is the time that is needed to learn using the framework, since it normally takes more time compared to function libraries. One reason is the need of more time and knowledge to created frameworks. An other reason is learning to use a framework consumes more time than learning to use a method library with a similar functionality. When taking this into account a framework designer has to make a trade off between a higher degree of flexibility or a higher easyness to learn utilising it.

Creating a framework consists of more than just inventing a good design. After the design is created, the following step is to validate it. Then the development has to take place which not only includes programming, but also writing of the documentation. At the end a real application can use the framework and test if the framework is properly working. That information can influence an other development cycle, stepping through the phases of framework development again in order to refine the framework.

Several areas are of high importance, when the design is done. These are usability, patterns, convenience, documentation and freedom as outlined in the following subsections.

2.2.4 Validation

Validation is a topic that is rather difficult to tackle while designing a framework. This stems from the fact that it is harder to validate generic components in the abstract than to validate a part of a program.

The lack of standards regarding how to design, implement and document frameworks is making validation a difficult task. In addition patterns of collaboration are not clearly shown in today's programming languages. Therefore they must be documented. So validating those patterns of collaboration links the analysis of the source code to the analysis of the documentation.

2.2.5 Development

Usually development starts out by conducting a domain analysis, which needs a number of examples to begin with. First those examples have be collected. Then each of them is analysed and the extracted information is used to define the domain of the framework.

A fundamental technique used in framework design is the so called white box reuse and it's counterpart the black box reuse. Whitebox reuse is concerned with providing a basic structure for the user where the developer has to fill in missing code. Most often the user has to create quite a lot on his own in order to finish the functionality. Never the less the design structure reduces the amount of work spent on designing. In contrast blackbox reuse is focused on providing components that present the user an interface and an explanation of the interface. How those components look like internally or what they really do, is hidden from the user. Therefore, they are called blackbox cause they act like a black box, getting some input from the user, doing something with it and giving back the expected result.

The first version of the framework mostly consists of white box reuse components. By implementing examples and by using the framework to build an application, weak points of the framework can be found. The gathered experience then leads to improvements that can change pieces of the framework from whitebox reuse design to blackbox reuse one.

Actually according to Fayad et. al [10] the only way to find out what is wrong about a framework is to use it. Another important statement about the development of frameworks by Fayad et. al [10] is that the development should be conducted by a research group that is composed of different people than the ones that are utilising the framework.

2.2.6 Design Patterns

According to Gamma et. al [13] Patterns in general give an architectural abstraction that has the following 4 features:

- Structure - interface of a set of basic elements as well as the static and/or dynamic structures that relate to them in composition.
- Functionality - provides useful functionality
- Abstraction - identifies and names a composition of elements of a certain internal structure and certain functionality.
- can be applied multiple times

One prominent design pattern for frameworks is the so called "hot spot". It stands for a flexible part of the framework that can be implemented by the user in order to adapt the framework for a specific application. According to Fayad et. al [10] a good framework documentation should outline the "hot spots" in detail.

According to Fayad et. al [10] three different types of patterns are of use for framework design:

- *Design Patterns* focus on technical design and implementation.
- *Modelling Patterns* focus on modelling.
- *Meta Patterns* focus on structural interrelationship.

Design Patterns are Alexandrian Patterns - they express solutions in an abstract way. Whereas Modelling Patterns express solutions in non abstract, but generally described classes.

Fayad et. al [10] explains that Framework design itself provides features at two levels:

- domain features - domain relevant features useful for applications
- structural characteristics - features facilitating the adoption and evolution of the framework.

The domain features researched help in creating an abstract domain model for the framework. That model is important for understanding the framework and needs to be defined in the frameworks documentation.

Design patterns are used to define designs. They can be split into three categories:

- patterns dealing with flexible object creation
- patterns dealing with structural aspects of object systems
- patterns dealing with behavioural aspects of object systems

Generalising a solution leads to a design pattern. Describing the pattern is tougher than finding it. Structural aspects are easier to describe than the problem that is solved and the context in which it can be applied.

2.2.7 Meta Patterns

Meta Patterns are defined according to a perspective on class methods. They describe the structural interrelationship between the methods. In order to create accurate Meta Patterns class methods need to be described accordingly. Class methods in turn can be characterised according to two methods:

- hook method is a method that does nothing other than returning to its caller, but is designed to be overridden in subclasses.
- template method provides either behaviour which can vary for subclasses or avoids duplication of code or controls at which points subclassing is allowed.

Depending on the content and context of the method, a method can be either hook or template method or both.

Meta Patterns are applied in order to obtain abstract decoupling in a design and they are developed in an analytical manner. They can be used to describe Design Patterns.

2.2.8 Modelling Patterns

A Modelling Pattern is an abstraction of a small group of classes that is likely to be helpful again and again according to Fayad et. al [10]. Some of them can be used both when analysing a problem domain and when designing and implementing software. Modelling Patterns are found by trial and error and by observation.

2.2.9 Design Issues

In the early days of object-oriented class libraries the design was created for programmers who wrote every line of application code themselves. Today visual builders and wizards, present the programmer a GUI Interfaces and generate much of the code. The generated classes are tied together by interfaces with a minimum of handwritten code.

Therefore, today's good frameworks must address visual programmers and programmers who write all application code themselves. To satisfy both sets of needs, the so called "part" has been designed. That "part" is a class with special characteristics to support visual programming and code generation. The part interface is composed into:

- attributes - provides access to selected properties of a part
- actions - provides access to selected behaviours of a part
- events - a means to notify other parts that something has changed within a part.

One major difference between parts and classes is the capability to notify other parts that something has changed within a part. Implementing that feature in classes reduces that distinction. Another most widely felt difference between ready to wear parts and tailor made classes is related to their different usage. When using classes, the programmer uses subclassing to utilise their feature.

In contrast, Visual Builders, which are Graphical User Interfaces used to create source code from modelling patterns, provide features by the means of configuration options.

When it comes to code generation the advantages and disadvantages of visual builders can be seen easily. This can be done by comparing following three main factors of framework design:

1. ease of coding
2. performance and size
3. robustness

Visual builders have the drawback that they impose some standards on how classes have to look like. This may cause problems e.g. the type of a variable in the definition of the interface can be incompatible with the needs of the programmer. That problem could be solved by overloading the method in question.

On the other hand, visual builders can show their strength when it comes to complex refactoring like renaming class features.

Anyway if visual builders are utilised or not, the ultimate goal for a class designer is reduction of complexity in the framework by reducing the number of classes and the number of member functions that must be overloaded or called to accomplish a task.

2.3 State of the art Meta Heuristic frameworks

Today there are several frameworks available that provide a design for modern heuristic algorithms. Only a subset of those frameworks have been taken into account since not all of them are capable of handling Tabu Search. Some of those frameworks provide a basic design for Tabu Search and therefore ease the implementation of Tabu Search algorithms. Others focus not on a single Meta heuristic technique, but on topics like providing a design for a set of algorithms and a hybridisation model for a set of several methods. The following section focuses on those frameworks and will give an overview of them and present an in depth analysis regarding their support for Tabu Search.

2.3.1 OpenTS

Harder [18] designed a Tabu Search framework called OpenTS that has been implemented in Java. It uses object orientation and permits the implement of a specific Tabu Search algorithm by deriving new classes from the framework one's. The key features of OpenTS are:

- Flexibility is provided by having the user provide her/his own data representation for a solution and therefore avoiding to limit the user to predefined data types.
- Black box reuse is offered for the central TabuSearch class and the tabu list components. Though it's still possible to derive user defined classes with extended functionality.

- The neighbourhood generation has to be defined by the user.
- Additional aspiration criteria can be created by the user.
- Integrated goal programming.
- Multidimensional fitness values can be utilised by the objective function component.

In figure 2.12 you can see an UML diagram showing the structure of OpenTS. As pointed out by Harder [19] the core classes that must be implemented by the user, are the solution representation, the objective function, the move and a move manager. The user can supply several different move types by creating more than one class that derives from Move. But the class derived from MoveManager must be capable of handling all of those moves, since the MoveManager class doesn't provide that capability. So this is left over for the user to implement when all moves have to be taken into account in the same Tabu Search algorithm. The class derived from Solution has to store a single solution representation and must provide a clone() method that duplicates the solution representation completely including the duplication of any referenced data. The class derived from the ObjectiveFunction has to provide an evaluation method that is capable of computing a fitness value for a given solution - move pair.

The framework provides a well defined interface for the interaction between the classes. This is an advantage for understanding the interactions, but it limits the user to that interface since there is no way to easily enhance the interactions. Hence modifying the way the classes interact by extending or changing that interface, is a very difficult task. For simple Tabu Search algorithms that may not be needed, but in more complex scenarios this can become a substantial drawback of the framework according to Hoong et. al [25].

Furthermore there is no utility for implementing a frequency based memory dimension of Tabu Search as described by Glover and Laguna [14]. Additionally breaking down moves into move attributes by utilising the ComplexMove class restricts the composite attribute type to integer.

The framework concentrates on providing a very basic design for Tabu Search algorithms around the most common features of Tabu Search.

2.3.2 EasyLocal++

Gaspero and Schaerf [6] present a framework named EasyLocal++. That framework is not restricted to Tabu Search algorithms, but it is designed to be usable for local search techniques in common. It's key features can be described as:

- Predefined designs for several common metaheuristics like Simulated Annealing, Hill Climbing, Tabu Search are offered.
- Allows to utilise hybrid methods to solve problems.
- Blackbox reuse of the central core of the framework.
- Whitebox reuse of the predefined algorithms.
- New Algorithms can be defined and integrated easily in already existing algorithms and programs.

As Gaspero and Schaerf [7] explain, local search is a paradigm for optimisation which is based on the idea of navigating the search space by iteratively stepping from one state to one of it's "neighbours". The Neighbourhood for a state is composed by the states which are obtained by applying a simple change to the current one. As OpenTS EasyLocal++ is also an object oriented framework. So its core is composed of a set of cooperating classes. Each of them is responsible for a different part of the search process. The developer, utilising EasyLocal++, has to derive his own classes from a subset of those classes. As Gaspero and Schaerf [6] mention, the classes in the framework can be split into five categories:

- *Data classes* which store the state of the search space, the moves and the input/output data.
- *Helpers* which perform actions related to some specific part of the search. For example, the Neighbourhood Explorer is responsible for everything concerning the neighbourhood.
- *Runners* are responsible for performing a run of a local search technique, starting at an initial solution and leading to a final one.
- *Solvers* control the search by generating the initial solution and deciding how and in which sequence, runners have to be activated.
- *Testers* represent a simple predefined interface of the user program and shall be a help for debugging and testing the produced code.

Figure 2.13 provided by Gaspero and Schaerf [8] shows a structural overview of architecture of the framework EasyLocal++.

The big advantage of the framework are the hybridisation models which allow the utilisation of different Local Search techniques while solving a problem. However the hybridisation capability is not perfect according to Breitschopf et. al [4], because it lacks an easy possibility to create and integrate new schemes.

The provided design for Tabu Search is a basic design that doesn't provide any specialisation capabilities out of the box. The frequency memory dimension is equally ignored as in OpenTS, since only short term memory is provided for storing tabu status. Long term memory and an interface for its evaluation regarding frequency information is not defined in the framework.

2.3.3 HOTFRAME

Fink and Voß[12] proposed a (iterated) Local Search, Simulated Annealing and Tabu Search framework known as the Heuristic OpTimization FRAMEwork (HOTFRAME). It is primarily based on the use of templates in C++ to obtain a generic and flexible approach.

The authors suggest some potential advantages of their design such as runtime efficiency and enhanced decoupling of components that would lead to a black box reuse. Therefore the framework presents following key features:

- blackbox reuse of tabu search, simulated annealing, and other local search techniques.
- runtime efficiency

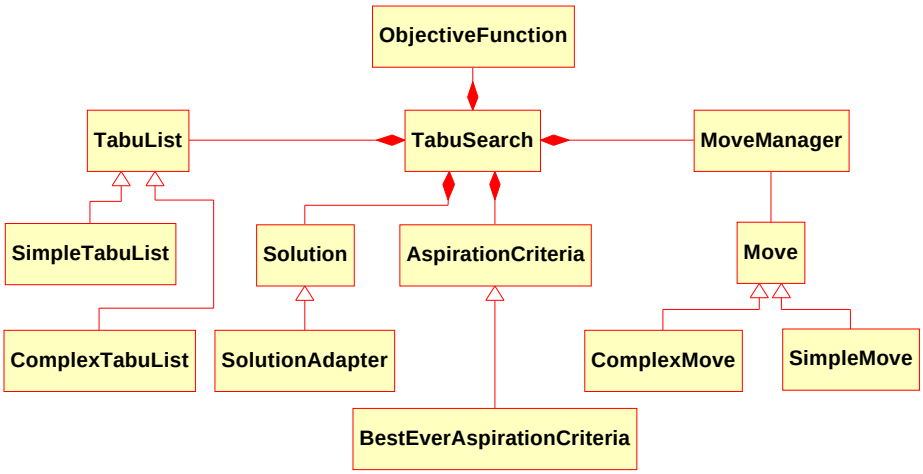

Figure 2.12: class diagram of OpenTS

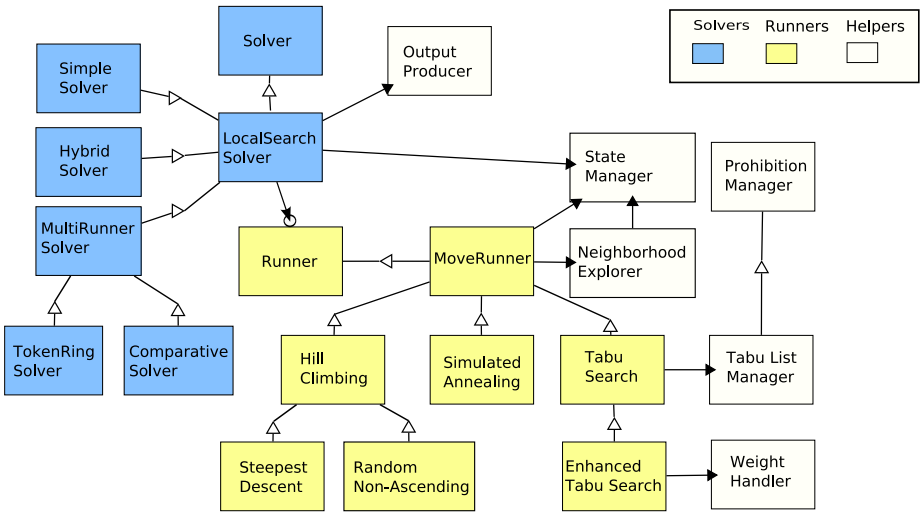

Figure 2.13: class diagram of EasyLocal++ by Gaspero and Schaerf [6]

- extensible with new metaheuristics components

Fink and Voß[12] propose that the features common to metaheuristics are captured in metaheuristic algorithms. These algorithms operate on problem specific structures, in particular the solution space and the neighbourhood.

Basically HOTFRAME can be divided into three core templates, the *problem data*, the *iterator* and the *neighbourhood*. The *problem data* and the *neighbourhood* has to be defined by the user. Whereas the *iterator* is hidden within the chosen algorithm and may be defined by the user when implementing his own *iterator* template.

Although HOTFRAME can be used in most problems, it lacks the capability that would allow developers to adaptively guide the search process according to Hoong et. al [25] The provided Tabu Search iterator template presents a basic Tabu Search algorithm that neither stores nor computes frequency information of the search path.

2.3.4 Templar

Jones published the Templar Framework in [21]. Like EasyLocal++ it's a metaheuristic framework for solving NP hard problems. The Templar Framework is object oriented and written in C++. It's key features are:

- Handling of strategies by a single control mechanism
- Whitebox reuse of a basic Tabu Search design

The Templar frameworks three most fundamental classes are TrEngine, TrProblem and TrRepresentation as shown by figure 2.14. TrEngine is the baseclass for implementing a given optimisation technique. So the user would derive his own class from TrEngine for programming an algorithm like simulated annealing. TrProblem contains deferred methods that have to be implemented for a specific problem by deriving a class from it. TrRepresentation is a base class for representing a solution in computer memory. The user has to derive his own class from TrRepresentation for his/her domain.

Interaction between different TrEngine objects is possible via so called TrConduits as presented by figure 2.15. These TrConduits have a wide range of messages that can be sent forth and back. To even increase the flexibility the messages can be filtered by TrConduit objects.

The Templar Framework is even thread safe and provides the option to execute different TrEngines in different threads. The framework provides features and functions for synchronisation and serialisation. It is even supports to distribute execution between computers, but that is dependent on its implementation. The author introduced that drawback, since utilising CORBA for distribution would have created a more complex solution and would not have allowed an easy way of switching from non distributed source to distributed source implementations.

The functionality of the Templar Framework can be summarised as follows: It is a metaheuristic framework for solving NP hard problems, that can be used to create local search techniques.

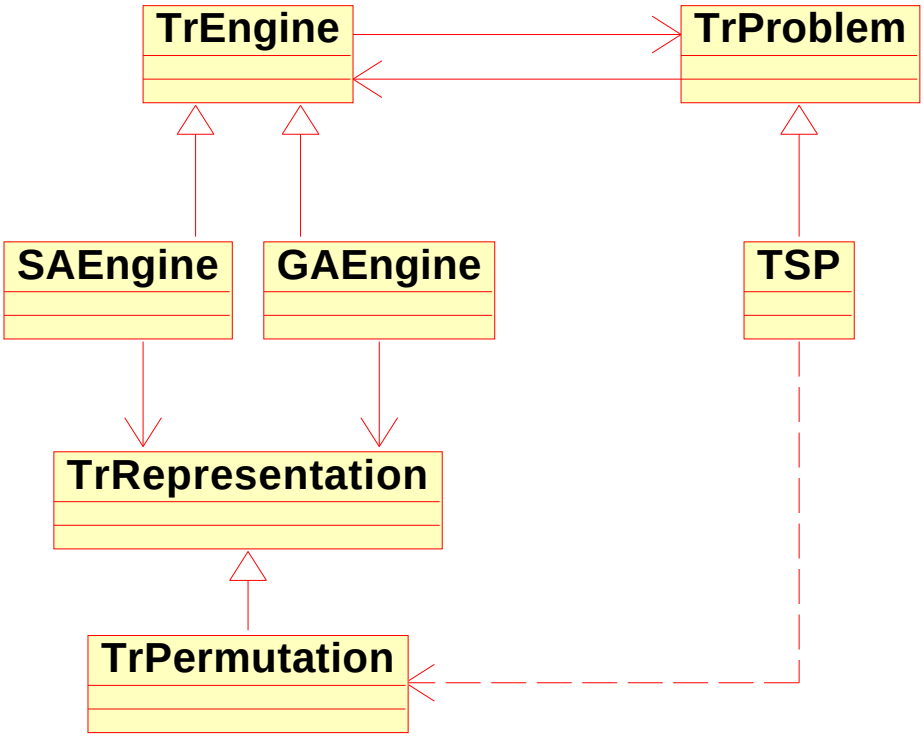

Figure 2.14: class diagram of Templar by Jones[21]

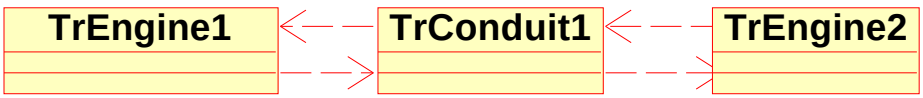

Figure 2.15: class interaction diagram of Templar by Jones[21]

2.3.5 TSF

A recently proposed framework called TSF, is described by Hoong et. al [25] as: It is composed of 4 categories of components: *interfaces, control mechanism, search engine* and *strategies software libraries* as shown in figure 2.16.

The interfaces define the basic components used by Tabu Search in an object oriented architecture. The control mechanism is for advanced users to define rules that guide the search adaptively in response to events encountered. TSF eliminates the tedious routine task of programming the search engine. The TSF search engine interacts with the interfaces and collects information which is passed dynamically to the control mechanism to readjust future search trajectory. Finally, TSF includes a strategy software library consisting of a set of software components to support various user-defined search strategies.

The key features of this framework are:

- Flexibility by utilising the provided interfaces for defining a problem
- Well defined interaction scheme between the frameworks classes
- External strategy software library for advanced search strategies
- Event triggered control mechanism

According to Hoong et. al [25] their framework performs better compared to others, because the user can influence the search process by utilising the event triggered control mechanism and the external software library.

The overall design is similar to OpenTS and therefore also lacks an interface for the acknowledgement of frequency memory as defined in Glover and Laguna [14].

2.3.6 OptLets

Breitschopf et. al [4] introduce the HeuristicLab Framework in the article "A Generic and Extensible Optimization Environment". Their framework provides a design structure that is fit for implementing Evolutionary Algorithms, Local Search or Swarm Algorithms.

OptLets has the following key features:

- High level of genericity by supporting a vast set of Meta Heuristics
- Hybridisation of Meta Heuristic techniques
- fast kick off by easy installation and introduction to the framework
- GUI front end offering basic visual builder capabilities

The big advantage of the OptLets Framework compared to others is the possibility to add new algorithms easily. Although this helps when the user needs a specific algorithm, it leaves a lot of the design to the user. Of course this is the trade off made in order to gain the high degree of flexibility that OptLets offers. Therefore it lacks in depth design for the Meta Heuristic methods, which must be provided by the user. That limits the out of the box usability, since the knowledge and the implementation of the algorithms is left to the user.

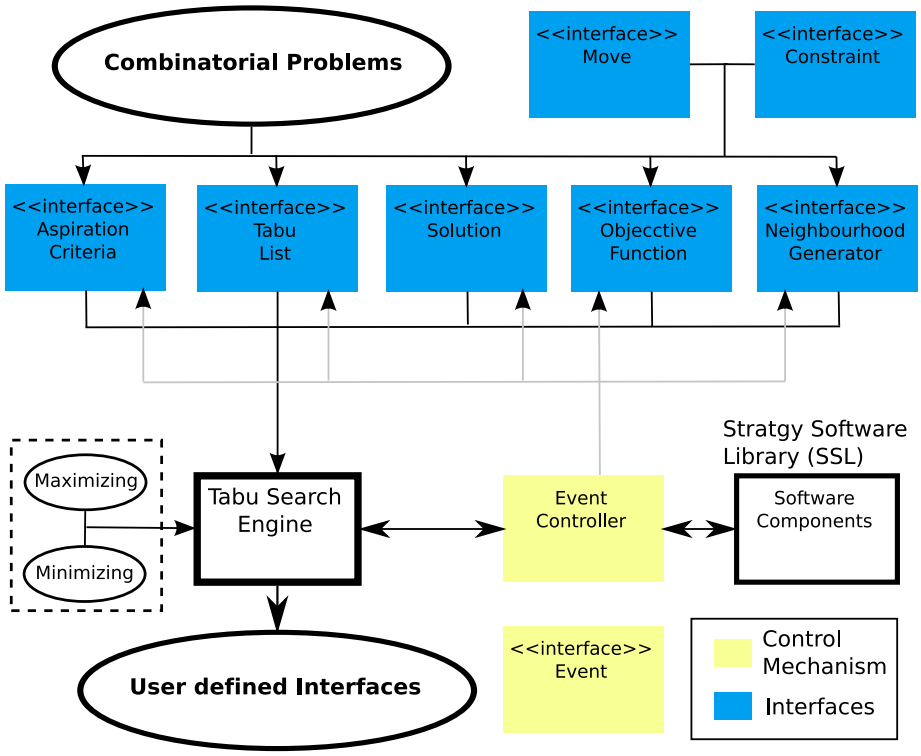

Figure 2.16: framework architecture of TSF by Hoong et. al [25]

Another drawback is the incapability to interweave the defined algorithms by hybridisation when solving a problem.

Summarised OptLets is yet another framework that strives to be applicable for a big set of Meta Heuristic techniques. It's most prominent feature not found in other frameworks is the GUI that helps with the installation and basic configuration options.

2.3.7 HSF

The Heuristic Search Framework, also called HSF, was created by Dorne and Voudouris [9] as a Java object-oriented framework. It supports the easy implementation of a single solution algorithms like Local Search, population-based algorithms such as Genetic Algorithms, and hybrid methods being a combination of the two. Its key features are:

- Reuse of components between different search techniques
- Flexible and generic design for metaheuristics

According to the authors the main idea in HSF is to break down any of these heuristic algorithms into a plurality of constituent parts. Thereafter, a user can use the library of parts to build existing or new algorithms. The major motivation of HSF is to provide a "well-designed" framework dedicated to heuristic methods in order to offer a representation of existing methods and to retain flexibility to build new ones. In addition, the use of the infrastructure of the framework avoids the need to re-implement parts that have already been incorporated in HSF and reduces the code necessary to extend existing components. Although it is a framework for more than one Meta Heuristic technique it doesn't offer hybridisation of algorithms.

2.3.8 Localizer++

Michel and Hentenryck [22] report Localizer++ as an extensible object-oriented library for local search. It supports both, declarative abstractions to describe the neighbourhood and high-level search constructs to specify local moves and Meta Heuristics. A variety of features typically found only in modelling languages are supported and its extensibility permits an easy integration of new, user defined abstractions.

Of particular interest is the conciseness and readability of Localizer++ statements and the efficiency of the Localizer++ implementation.

2.3.9 INCOP

INCOP is presented by Neveu and Trombettoni [31]. They created a library written in C++, which provides incomplete algorithms for optimising combinatorial problems. Neveu and Trombettoni [31] refer to their implementation as a library, though it has to be regarded as a framework. INCOP offers local search methods such as Simulated Annealing, Tabu Search as well as a population based method, Go With the Winners. Several problems have been encoded, including constraint satisfaction problems, graph colouring, frequency assignment. The user can easily add new algorithms and encode new problems.

The neighbourhood management has been carefully studied. First, an original parameterised move selection allows to easily implement most of the existing Meta Heuristics. Second, different levels of incrementality can be specified for the configuration cost computation, which highly improves efficiency.

2.3.10 GAILS

Varrentrapp presents his Guided Adaptive Iterated Local Search - Method and Framework in [35]. It incorporates machine learning into local search techniques by utilising reenforcement learning techniques. So called Local Search Agents are used for realisation of the reenforcement learning. The key features of the framework are:

- Support for Local Search techniques
- Reenforcement learning support

Tabu Search is discussed in the documentation of the framework though it doesn't provide a specific design for it. Frequency memory is not referred to in the documentation.

An advantage offered by OptLets is it's blackbox design for hybridisation. Since the framework itself doesn't provide an extensive set of predefined Meta Heuristics, their implementation is left to the user providing a high level of flexibility. On the other hand this needs a programmer that is proficient in the area of Meta Heuristics and it consumes time.

Chapter 3

Design

As a result to the previously presented literature research, the intention of this diploma thesis framework is to take the design of the state of the art frameworks and enhance it. This chapter presents the proposed Tabu Search framework along with an outline of it's differences to the state of the art frameworks.

3.1 The design requirements

Before a design is drafted, it's design requirements are put together in order to create some goal that should be reached.

Since those are written down first when a design is created, they are the best place to start explaining the accomplished work. Their purpose is to predefine goals, that should be reached within the design.

- The Framework should provide a design of the main features of the Tabu Search algorithm.
- The design should be problem independent for NP hard problems.
- The problem definition should define how a solution looks like in computational terms. (The data structure of the solution)
- There should be a way to specify a neighbourhood step function. According to Tabu Search terms this involves the definition of move operators. Those are utilised to construct a neighbourhood for a given solution. The construction of the neighbourhood should be conducted by applying the move operators to the given solution creating new solutions.
- A move and an algorithm should be able to generate a neighbourhood.
- The algorithm should be able to do a search when given a problem definition, a neighbourhood step function and a way to evaluate a solution.
- More than one neighbourhood should be possible.
- A tabu list should be maintained automatically.
- Additional configuration via aspiration criteria should be possible.

- It should stop the search when reaching a termination condition:
  - time limit exceeded
  - number of neighbourhood steps reached (That are the times the neighbourhood function got applied)
  - number of iterations
  - number of evaluations

  When the chosen and defined limit or limits get exceeded, the Tabu Search algorithm should terminate and remember the last best solution.
- Statistical data has to be maintained about the search:
  - number of evaluations
  - size of neighbourhood
  - size of the tabu list
  - change of fitness during iterations
  - number of moves chosen. That's the number of moves that lead directly from the initial solution to the solution found by Tabu Search.
  - time - How much time was spent in evaluations
- Short term and long term tabu mechanisms should be provided
- information about the search history should be kept
  - short term - should store the moves that have been visited recently
  - long term - should store the frequency of usage of the applied moves
- Selection of the solution for the next iteration should be able by:
  - picking the best non tabu solution
  - picking the best solution (tabu and nontabu)
  - utilising aspiration criteria
  - considering short term memory
  - considering long term memory

3.2 Creating the Design

The previously introduced state of the art frameworks provide a variety of interesting and good design patters for Tabu Search. Therefore the framework of this diploma thesis should not create a completely new design, but rather takes some of the already existing ideas into account and adds a new idea to it. So the new idea should not have been included by any other framework yet.

The analysis of the design of the modern heuristics frameworks presented above, shows that all of them provide a very simple design for the tabu memory. For example Harder [18] provides 2 classes for storing tabu moves called SimpleTabuList and ComplexTabuList. Both of them provide an interface to query recency information that provides information about current tabu moves.

But there is no way to acquire frequency information that would show us how often a specific move has been used in the search path.

Gaspero and Schaerf [6] do not provide a tabu list directly, but encapsulate it's functionality in their Prohibition Manager, which is a class used to store tabu moves. The interface to implement by the user is the computation of the inverse of a move. Two more functions are provided as an interface, but they do not need to be implemented by the user. They can be replaced when their functionality is not sufficient for the needs of the user. These two functions are move insertion and a test to check weather a certain move is tabu or not. The move insertion called InsertMove is defined to insert the move into the list and to assign a tenure period and it also discards all moves who's tenure period is expired. The test to check weather a move is tabu or not can also be defined by user. But there is no way to query for frequency information or the content of the tabu list that is stored in the class called Prohibition Manager. Therefore you can only access recency based information and frequency information is not available.

That makes the interface rather generic. It can be used to implement frequency based Tabu Search decisions, but it doesn't provide a basic design for doing so. Hence that framework does not provide a design for the frequency dimension of tabu memory.

The intention of this diploma thesis is not to create a completely new framework for Tabu Search, but to create a framework that provides something that the other's don't. Therefore the analysis of the state of the art was essential for detecting uncovered areas. As the section on state of the art framework shows, it is possible to discover that the state of the art frameworks lack an in depth design of the tabu memory compared to the definition of tabu memory provided by Glover [15]. That was a rather sound area to start with, since the tabu memory is a core element of Tabu Search and it would be an interesting part to experiment with. So the idea for the new part of the diploma's framework was to create something that covers a broader part of the four dimensions of Tabu Search memory structures than any of the other frameworks.

The next step was to make a decision about how far the design in respect of recency, frequency, intensification and diversification would be taken. Since recency has been covered by all present frameworks, it has to be covered as well. But frequency, intensification and diversification cannot be found in the design of any state of art framework.

So the exploratory focus is aimed on those three dimensions. Frequency information can be stored easily, since storing how often a specific move has been made is straight forward. But not all of them can be so easily incorporated into a framework. Putting intensification and diversification into a design structure is problematic, since they are tightly linked to the given problem. Therefore it's hard if not hardly possible to extrapolate a structure that can be used for all kinds of Tabu Search problems.

Therefore creating a design that gives the user the tools for recency and frequency based Tabu Search memory, seemes reasonable. Both of them can be extracted from the path of the search, as long as it contains solutions and moves. For a better understanding the algorithms for computing recency and frequency information for a move, are described below.

3.2.1 Recency information

Recency based information is acquired by searching the recently added moves of the search path. The tabu tenure defines how many of the last recently seen moves of the search path are tabu. So it gives the number of moves that have to be compared with a new move in order to determine the tabu status of the new move. If the tenure is x, then the x last moves of the search path have to be compared. When any move of the search path is found to be the inverse of the new move then the new move is tabu. Otherwise the new move is non-tabu.

3.2.2 Frequency information

On the other hand frequency information is gathered by stepping through the whole search path and counting the occurrences of the moves found within.

The last step in the process of creating the design was to make design patterns for recency and frequency memory. Before starting to explaining them in detail, an overview of the overall design of the diploma thesis framework will be given in the next section.

3.3 Design Tools

Before starting with the explanation of the design itself, a list of the design patterns and tools, that have been used in it's creation, is given.

Accessors are provided by utilising get/set functions.

All class variables are prefixed with an m_ in order to show that they are member variables of a class.

Variables in general use the Hungarian notation when their type is predefined by the programming language. For other variable types the Hungarian notation was not used, because that would have had the need of defining custom prefixes and those might mislead others, since they might have other preferences regarding creating prefixes for custom types. For better legibility the variables, that are defined by using user defined types, have names that give a good hint about their content and usage. In order to not further increase complexity pointers are an exception to that rule, because they are always prefixed with a p to mark explicitly, that they point to some sort of memory.

3.4 The design of the framework

This section starts with an overview of the general structure of the framework by describing the relationships between the different classes. Along with that description a basic understanding of the interaction of those classes will be given. Afterwards all the details about the interfaces, generics and classes composing the framework will be explained.

3.4.1 A first view on the framework

Since diagrams can give a good overview, figure 3.1 shows the class structure of the framework. The diagram shows us that the most important class is called TabuSearch. It has got an aggregation relationship with several other classes

in order to conduct the actual search. It is the central core of the framework that guides the search process. Some classes don't have an aggregation relationship with the TabuSearch class, though those are used by classes that have an aggregation relationship with the the TabuSearch class.

The TabuSearch class uses the abstract Solution class as a way to access the user defined solution representation. This is necessary due to the fact that the actual implementation of such a Solution class is problem specific. Therefore the framework provides an abstract Solution class that the user has to subclass. The problem specific solution representation has to be put into the derived class. Tightly related to the *Solution* class is the *ObjectiveFunction* class, since it has to evaluate Solution objects regarding their fitness. Since that is problem specific too, the user has to derive his own class from it that should provide the needed functionality.

The memory container classes called SearchHistory and TabuList are essential for the search, since the former stores a list of moves visited in the past and the later can tell if some move is tabu or not. The SearchHistory is coupled to the EliteSolutionList class that is a container for so called elite solutions. The decision if a solution is elite or not, is encapsulated in that class by a method. So if the default, which is set to best 5 percent of solutions, is not appropriate, the user has to derive his own class and overload the method with an other algorithm.

An other class that is directly accessed by the TabuSearch class is the MoveManger which is a container for MoveIterator objects. The MoveIterator class in turn is an iterator for the neighbourhood of a solution that is created by a specific move type. That move's definition is stored by the Move or the AttributeMove class. The idea of the Move class is to store one set of parameters for a move type, which defines the changes to make to a Solution object and to store the algorithm that is capable of applying the move to a Solution.

The last class that is directly accessed by the TabuSearch class is the Aspiration Manager. It is a container for AspirationCriteria objects, who have to be derived from the AspirationCriteria class by the user.

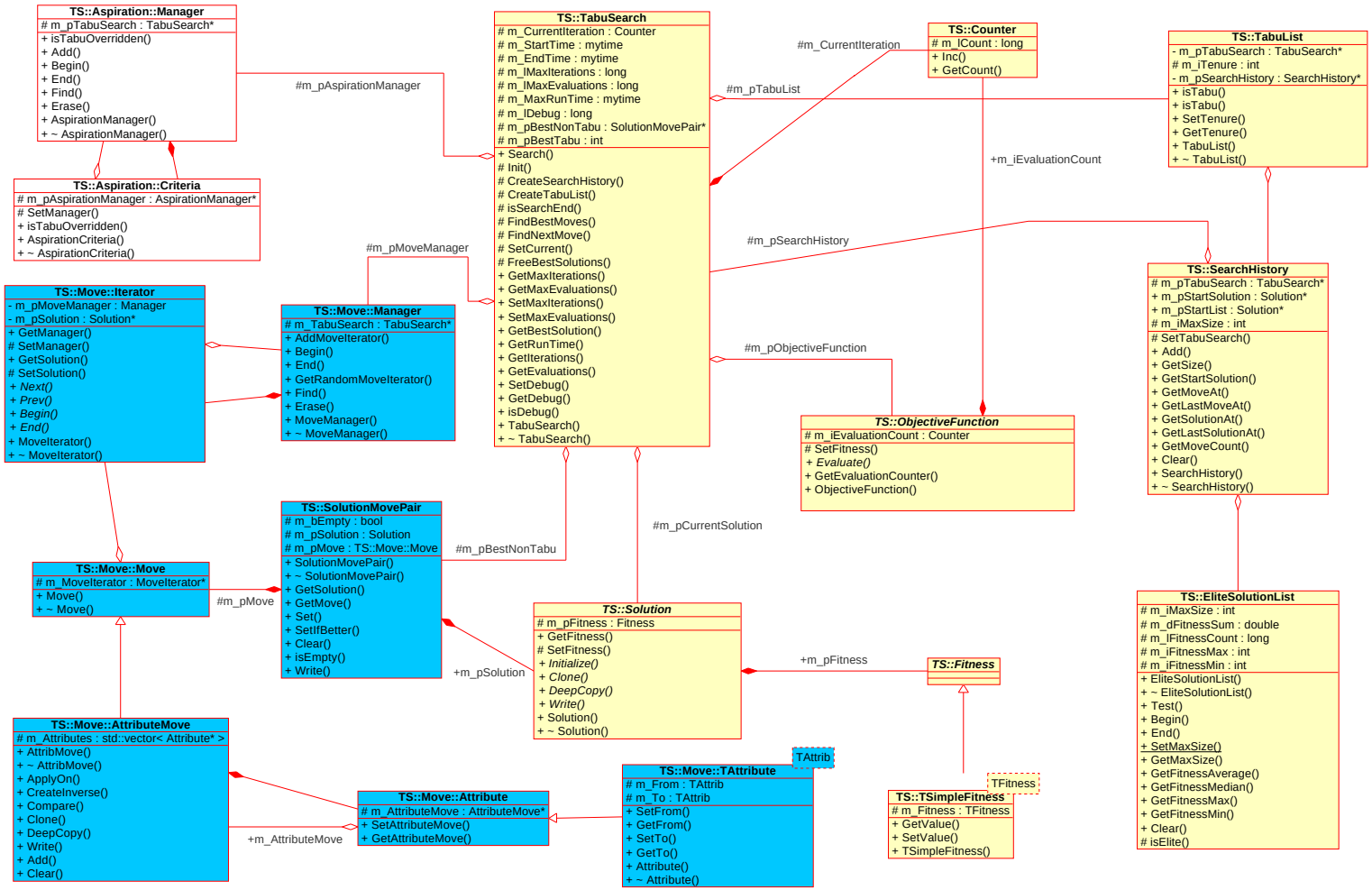

Figure 3.1: Overview UML class diagram

3.4.2 Details of the design

The following description gives an insight into the different interfaces, generics and classes of the framework. Their purposes are given along with a discussion of their details.

Interfaces

As already in discussed in 2.2.2 the main purpose of interfaces is to provide a generic access to specific functionality. Several classes of the framework need a similar functionality, but they do not share a domain in a way that a parent class can be devised. Instead a few interfaces were created and then those classes implement one or more interfaces. To create a remarkable distinction to other parts of the framework the name of any interface always starts with a capitalised "I" to signal that it is an interface. Some of the interfaces are generic enough to be discussed outside the context of the framework while others cannot. The following paragraphs discusses those generic ones. The rest of the frameworks interfaces will be introduced in detail in the appropriate context. Though their names will be listed here to give a complete list of all the interfaces of the framework.

The commonly applicable interfaces are called ICompareable, ICloneable and ISerializeable and are presented in figure 3.2 in a UML diagram. IApplyable is an interface that is limited to a specific area of application the so called Solution class. Therefore it will be explained in greater detail when it's area of application is discussed in conduction with the Move class.

ICloneable

It represents an interface for objects that can be cloned. Creating a clone of an object is the process of making a complete copy of the object. Normally that involves doing a so called "deep copy" of the data stored within the original object as presented by Champlain and Brian [5]. During the "deep copy" variable content is simply copied from the original object into the clone. But referential data types are not so simply handled, since a copy would mean that the clone references the same data as the original object. Therefore the referenced data has to be cloned too. That in turn leads to an other "deep copy". This repeating copy mechanism is actually summarised by the name "deep copy", since the "deep copy" is invoked for each referential data type found within the original object's data.

In order to provide a generic interface for cloning 2 abstract methods Clone() and DeepCopy() have been designed. Both of methods need a way to have a common referential type to act on, because the data type is normally not known in advance and might change because new classes are introduced. That leads to the need of a parent class that must be derived from when utilising this interface. In the case of this framework that parent class is called *Object*. It will be described in detail later on since it's only purpose at the moment is to provide a commonly usable type for passing forth and back references to objects. The diagram in figure 3.2 presents the complete interface.

The method *Object* Clone()* should return a new object containing the "deep copied" data. Any allocations involved when creating a new object should be done in the method.

The method void *DeepCopy(Object* pObj)* should create a "deep copy" of the data of the supplied object by copying it into the object that the method was invoked on. The parameter pObj object references an object by using a pointer. That object's data should be deep copied into the invoking object's one.

ICompareable

It is an interface for comparing 2 objects to each other. Since it just provides the methods and doesn't define the way the comparison should be done, a definition of the comparison is given as follows: The comparison taking place should be about the internally stored data. Other comparisons may be implemented by this interface, but they're not intended and not needed within this framework. Similar to the cloning the comparison should not stop at referential data types, but continue by comparing the data the referential data types refer to.

As figure 3.2 shows the interface provides 2 methods, one called Compare, the other named Equals. The method *int Compare(Object* pObj)* compares the internally stored data to the supplied object's one and it returns an integer value representing the result. When the internal data is less, matches or is greater than the supplied object's one, an integer smaller, equal or greater than 0 will be returned respectively. The method *bool Equals(Object* pObj)* is a test if the supplied object's data is equal to the invoking one or not. When it is then it returns true, otherwise false is returned.

ISerializeable

ISerializeable is an interface for writing the data of an object to some means of output. Therefore it provides a way to store the contents of an object during run time. So it provides a limited sort of debug functionality for the framework. That implies that it is not necessary for solving Tabu Search problems. Nevertheless it is a handy tool for searching for bugs. It is also interesting for fine tuning parameters of the Tabu Search and evaluating their quality.

In the case of C++ the means of output is a so called output stream. Normally the data of the invoking object is written to the supplied stream.

The method *Write(ostream &stream)* writes the object's contents to the supplied stream. Whether the data is formatted or not depends on the user's choice. The parameter stream is the output stream to write the data to.

IInvertible

It gives an interface that makes certain assumptions about the data stored in a class. Therefore it is not as commonly applicable as the previously defined interfaces. The interface IInvertible assumes that the class utilising it, stores invert able data. So there must exist a way to compute the inverse of one object's data. IInvertible provides an interface to those objects.

IInvertible is an interface for objects that contain invertible data that needs to be accessed from outside the class. The method *Object* CreateInverse()* should create a new object containing the inverse data. Any necessary allocations should be done too. This method uses a referential data type to the master class *Object* again. Otherwise the definition of the interface wouldn't be generic

enough. Anyway the returned pointer is referring to a newly allocated object, having the same type as invoking object, but containing the inverse data.

IApplyable

It defines an interface for objects that are applicable on Solution objects. It's only purpose is to provide access and force the implementation of the ApplyOn() operation. The ApplyOn method applies the object's contents on the supplied *Solution* object. That implies that the function changes the data of the supplied Solution object. In the framework this sort of functionality makes only sense for move operation related classes. Though no default approach can be given for applying a *Move* object on a *Solution* object. Therefore this interface has been created in order to provide a common interface for all objects that can be applied on a *Solution* object. The necessity of this interface will be shown below when the *Move* class and it's related classes will be explained. One method is defined by the interface. That is function is *ApplyOn(Solution* pSolution)* and it should apply the changes stored in the object to the supplied *Solution* object.

Object

Is a special class since it is the class that many of the other classes derive from. It's purpose is to provide a class type that the other classes can derive from. This is rather important for the way of type casting in object oriented languages work, since they normally provide a high type safety. Therefore it is possible to type cast a derived class to the *Object* class type. This enables the definition of methods for interfaces since future class definitions done by the user cannot be known. So a generic and well known class type needs to be used in the definition of interfaces because they need to work with any class type. Otherwise interface definition would be rather limited and not really useful in object oriented design. Therefore it is necessary to have a base class. That is the *Object* class in the case of this framework and it is a base class for other framework classes and eases their handling quite a lot. This idea of base class has already been utilised earlier by other other frameworks. A perhaps very prominent example is the Microsoft Foundaction Class framework. So a lot of other frameworks provide a class with similar design qualities. Modern Computer languages even go step further by defining a parent class that is always the master parent, even when no parent was defined. A good example for that case is C# providing a class called Object that is such a master parent as explained by Liberty [26]. When implementing the framework in C#, the framework's *Object* class would be rendered obsolete. But since that is an implementation specific issue, it is discussed in Chapter 3.5.

Figure 3.3 shows the definition of the Object class and the following paragraph gives an insight about it's methods.

The only function is *bool is(std::type_info pType)* which returns true when the supplied type matches the type of the object. Otherwise it returns false. The constructor of the *Object* class is empty.

Counter

The *Counter* class is a rather important class throughout the framework since it is used to count different types of things. For example the central TabuSearch class utilizes it in order to count the number of iterations.

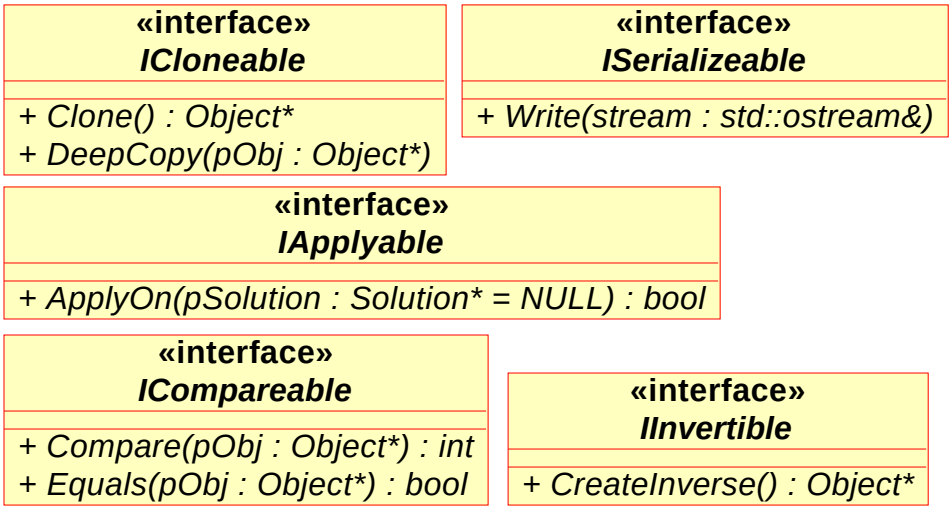

Figure 3.2: UML class diagram of the interfaces

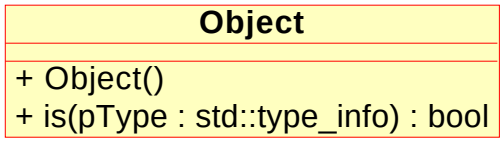

Figure 3.3: UML class diagram of the Object class

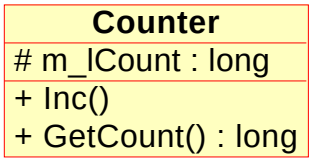

Figure 3.4: Counter UML class diagram

Figure 3.4 shows the UML class diagram of *Counter* and presents it's attributes and methods. The following list describes their functionality:
m_lCount is an integer value that stores the counter value. It is a protected values since there is no need to manipulate it outside the *Counter* class.
The method *Inc()* is just for increasing the counter value by 1.
The function *GetCount()* retrieves the current counter value.
The constructor *Counter()* simply initialises the counter value to 0.

TabuSearch

All state of the art frameworks present us a central class that contains at least major parts of the Tabu Search algorithm. Therefore this framework is also constructed around an important class that follows the same motivation. That is to contain the Tabu Search algorithm and to utilise other parts of the framework during the search process.

In figure 3.5 the complete *TabuSearch* class is shown in order to give an overview of the class for the following explanation. The actual Tabu Search algorithm is split up and distributed over several methods of the class. The purpose for the division of the algorithm is the ease of modification. That stems from the possibility to replace small parts of the algorithm by simply overloading a single function in a derived class. So there is no need to completely rewrite the whole algorithm when you want to change a small part of it. It will only be a matter of overloading and hence rewriting the functionality of a small method. The next paragraphs give an insight on the hotspots of the *TabuSearch* class.

Must define HotSpot
Init() can be used to set up variables with meaningful values before the search in the *Search()* method starts. This function should be overridden with a method that contains at least the creation of the initial *Solution* object.

May define HostSpots
The central method for the Tabu Search algorithm is *Search(pSolution : Solution*) : void*. First it initialises the object's variables by calling *Init()*. Then the method loops until *isSearchEnd()* reports true. The function *bool isSearchEnd()* returns true when the break condition of the search process is met. The *TabuSearch* class provides the following three break conditions: maximum number of evaluations is reached, maximum number of iterations is reached or a run time limit is reached. Other break conditions may be defined by overloading *isSearchEnd()*;

The actual search logic is not found within the *Search()* function, but in *FindBestMoves()* and *FindNextMove()*. Former searches the neighbourhood for the best non tabu and tabu solutions. Later decides if the non tabu or the tabu solution will be the next solution for the loop as a starting point.

The following two methods can be overridden when the developer wants to utilize her or his own classes instead of the predefined one's of the framework. The method *CreateSearchHistory()* creates a *SearchHistory* object when m_pSearchHistory has not been initialised with a *SearchHistory* object by the constructor of the class. *CreateTabuList()* works similar by analysing m_pTabuList and initialising it with a *TabuList* object when it has not been initialised by the constructor.

static Methods
*void SetCurrent (SolutionMovePair *pPair)* deallocates the current *Solution*

object, then stores the new current Solution. Additionally it stores the solution - move pair in the history. The supplied solution and move from the SolutionMovePair object get cloned (newly allocated). Therefore they must be destroyed. Releasing of the memory of the solution object is done here and TabuSearch(), but the cloned move must be released by the SearchHistory.

void FreeBestSolutions() deallocate the tabu and non tabu *SolutionMovePair* objects called m_pBestTabu and m_pBestNonTabu. The deallocation takes only place when possible the pointers are not null and the pointers get set to NULL afterwards.

mytime GetRunTime() retrieves the current runtime by using m_StartTime and the current time in order to calculate the time the search has already consumed.

long GetEvaluationCount() is a special method used to query the supplied *ObjectiveFunction* object for the numbers of evaluations done.

The constructor *TabuSearch()* has got several parameters. Those take all major objects of the framework in order to provide the capability to customise parts of the algorithm of the search. All parameters are pointers to objects. They following objects are essential and therefore must be given to the constructor: A MoveManager object, an *ObjectiveFunction* object and an AspirationManager object. The SearchHistory and the TabuList objects are optional. When they are not supplied, the TabuSearch class will create it's own by utilizing the SearchHistory class or the TabuList class respectively. That behaviour can be changed when CreateSearchHistory or CreateTabuList is overridden as describe above.

For utilising the provided Tabu Search algorithm for a simple case, the user only needs to create an instance of the TabuSearch class and call the Search function with an Solution object that is an instance of a class derived from Solution.

To give a better understanding of the internal functionality of the class the following list contains the variables of the TabuSearch class along with a description of their access methods:

m_CurrentIteration stores the number of the current iteration in the form of an object of type Counter. It's counter value can be queried by invoking *GetIterationCount()*. Manipulation of the counter itself is limited to the TabuSearch class, since it counts the number of iterations of the search process which is conducted by the TabuSearch class.

m_pCurrentSolution is a pointer to the a SolutionMove pair object containing the current solution object and the move that lead to it. It's content can be retrieved by calling *GetCurrent()*. It can be set by the method *SetCurrent(pPair:SolutionMovePair)*.

m_StartTime stores the start time of the search. It can be queried by *GetStartTime()*. Changing the value is limited to the TabuSearch class.

m_EndTime stores the end time of the search (when search terminates) and retrieval is possible by calling *GetEndTime()*. Setting the value is again limited to the TabuSearch class.

m_lMaxIterations contains a limit for the number of iterations. Can be accessed with *SetMaxIterations()* and *GetMaxIterations()*. When it is set to 0 it is ignored and doesn't impose a termination clause.

m_lMaxEvaluations stores the number of maximum evaluations. If it is 0, then it will not be used to terminate the search process. *SetMaxEvaluations()*

and *GetMaxEvaluations()* can be used to access it's value.
m_MaxRunTime stores the maximum run time of the search process.
m_pBestNonTabu is equal to m_pCurrentSolution, but stores the best non tabu solution and move. This variable has not got any access methods, since it's use is completely limited to the TabuSearch class.
m_pBestTabu is similar to m_pBestNonTabu, but contains best tabu solution and move. It also has not got any access methods, since it is limited to TabuSearch's use too.
m_pMoveManager references a *MoveManager* object that is a container for *MoveIterator* objects that create a neighbourhood for a solution. The method *GetMoveManager()* provides access to that variables value.
m_pObjectiveFunction stores a *ObjectiveFunction* object for evaluating *Solution* object regarding their fitness. Access is possible by the method *GetObjectiveFunction*.
m_pAspirationManager saves an *AspirationManager* object that can override the tabu state of a specific move by querying it's *AspirationCriteria* for overrides. It can be queried by *GetAspirationManager()*. **m_pSearchHistory** contains an object for saving the search's history with the type *SearchHistory*. The method *GetSearchHistory* returns it's value.
m_pTabuList stores the tenure value and determines the tabu state of a move. *GetTabuList* can be used to access the value of m_pTabuList.

Solution

The purpose of this class is to store a solution representation. Therefore the *Solution* class is defined to be abstract and just provides an interface and a predefined storage for a fitness value. That abstraction approach to tackle the solution definition is equal to other Tabu Search frameworks like Harder's [18] or Gaspero and Schaerf [7] or Hong et. al [25]. Both of them leave the implementation and data structure definition to the user of the framework, since it's hardly possible to define a generic data structure that could possibly contain any problem's data.

That implies that no objects of type *Solution* can be created directly, but a class has to be derived from *Solution* that implements the abstract interface. The design and structure of the data for storing the solution representation is completely left over to the user, since a generic definition is going to have unwanted drawbacks. Most likely that would include performance penalties, excessive use of memory and the major drawback that the user must adapt his problem model to the solution storage of the framework. To avoid those limitation the *Solution* class just provides an interface so that the rest of the framework may have a way to access it's essential data. That interface is presented in figure 3.6 Other important classes in this context are the *ObjectiveFunction*, the Fitness and the Move class, since they depend on the data structure of the solution representation. Those classes also provide only an abstract interface like the solution class. A discussion in depth of their structure is given below. So the user needs to implement a class that derives from *Solution*, which must contain a data structure capable of storing a solution representation for the user's problem.

The only variable, that this class provides, is called **m_pFitness.** It is for

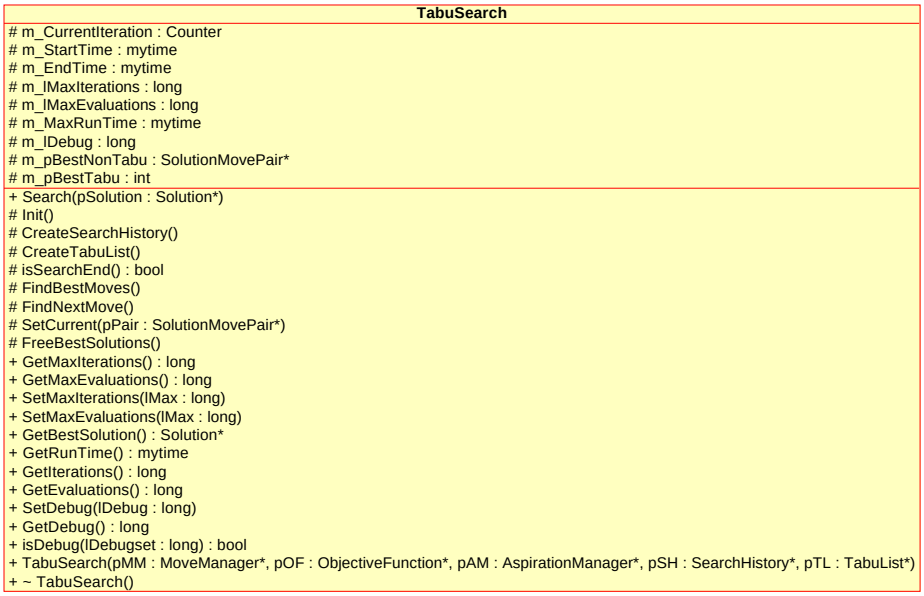

Figure 3.5: TabuSearch UML class diagram

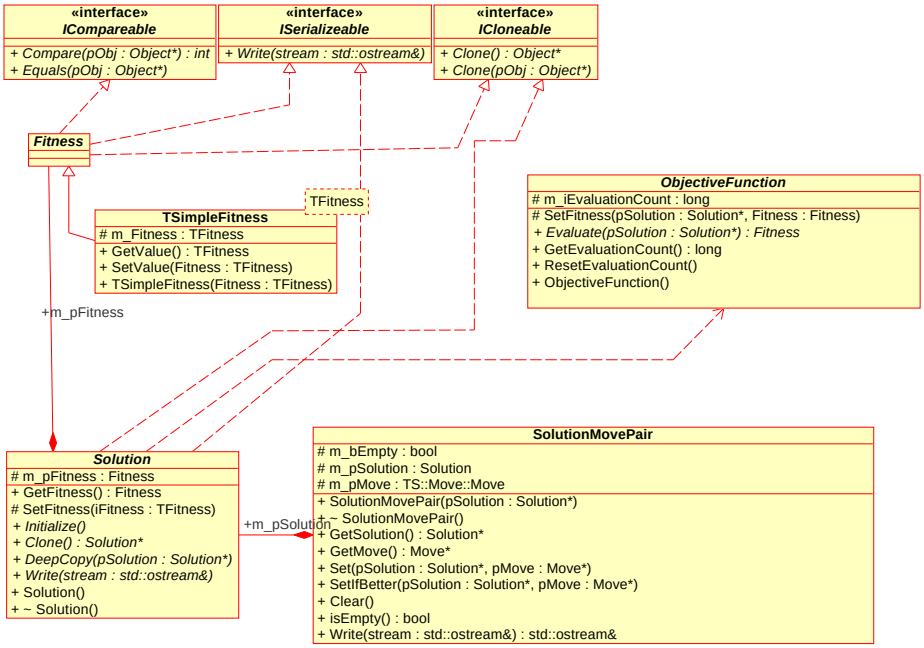

Figure 3.6: Solution and related classes UML class diagram

storing a value representing the fitness of a solution. That variable reduces the number of evaluations by caching the fitness value that is normally calculated by a class derived from *ObjectiveFunction*. The accessor function for m_Fitness, called *GetFitness()*, is public and returns the value of the protected variable. In contrary the mutator *SetFitness()* is protected and it can only be used by the *ObjectiveFunction* class for setting the fitness value. Since there is no need for any other class beside the *ObjectiveFunction* class to change the fitness value.

Must define HotSpots
Initialize() is an abstract method that must be overloaded with code that sets up the additional variables introduced for storing the data that represent the problem. There are another two functions that must be overloaded. Both are part of the ICloneable interface that the *Solution* class derives from. So *Clone()* and *DeepCopy()* have to be defined like the description of ICloneable interface suggests. When the framework's debugging capabilities are enabled, then the *Solution* class derives a second interface called ISerializeable. Therefore a further method has to be defined. It's called *Write()* and it's purpose has already been outlined by the ISerializeable interface.

Static Methods
The methods *SetFitness()* and *GetFitness()* manipulate and give access to the m_pFitness attribute as explained above. The constructor of the class simply initialises m_pFitness with a null value in order to store whether the *Solution* object has been evaluated or not. So when a *Solution* object doesn't have a corresponding *Fitness* object, then it hasn't been evaluated yet.

ObjectiveFunction

The *ObjectiveFunction* class encapsulates the objective function of the Tabu Search algorithm as seen in the OpenTS framework by Harder [18] or Hong et. al [25]. The UML diagram 3.6 presents the abstract *ObjectiveFunction* class.

The class contains only one variable named **m_EvaluationCounter** for counting the number of evaluations during a Tabu Search run. It is an object of the *Counter* class. It can be retrieved by a public *GetEvaluationCounter()* method. In contrary the manipulation of the counter is limited to the *ObjectiveFunction* class, since it's the only place where evaluations take place. Therefore there is no need to change the number of evaluations done so far anywhere else.

Must define HotSpot
Since the evaluation of a solution is problem specific as the data structure of the solution itself, the developer has to derive a class from the *ObjectiveFunction* and to overload one function. The one to overload is: *virtual TFitness Evaluate(pSolution).* It should compute a fitness value for the solution given by the parameter pSolution.

Static Methods
The computed fitness should be stored in the supplied *Solution* object utilising the function *SetFitness(Solution *pSolution, TFitness Fitness)* of the *ObjectiveFunction* class. That is the only way to manipulate the fitness variable of the *Solution* object since it's mutator method is protected and only the *ObjectiveFunction* class can access it.

An other static method is the *GetEvaluationCounter()* which returns the EvaluationCounter object as already mentioned above.

Fitness

Fitness values are also problem specific data. Once it needs an integer value to represent the fitness, an other time it's a float. Even multidimensional fitness values may be necessary. Therefore predefining the fitness value type is most likely restricting the freedom of the developer. That leads to only one solution, which is the abstract *Fitness* class. State of the art frameworks provide different approaches to this topic. For example Harder [18] doesn't provide a class but predefines the fitness value as an array of floating point values. The array may have be one, or higher dimensional. Though that might lead to a problem in case of multi-dimensional fitness values since there is no direct way to attach a comparison to an array of floating point values. That is avoided when the fitness value is encapsulated in a class.

So the *Fitness* class provides an interface for fitness values computed by the *ObjectiveFunction* class. So the user has to derive a class from the *Fitness* class and implement the abstract methods. The diagram 3.6 shows the UML definition of the *Fitness* class as a collection of the interfaces ICompareable, ISerializeable and ICloneable. The implementation of *ISerializeable* is only necessary when the framework's debugging capabilities are enabled. *ICompareable* is necessary since two fitness values must be compared by the TabuSearch class in order to judge if a specific move leads to a better solution. The interface *ICloneable* is needed, because the user has to create a derived class and the rest of the framework needs a way to create new objects of that user defined type. So as already explained above the *ICloneable* interface provides the necessary methods.

TSimpleFitness

The TSimpleFitness class is a template class that implements the interface of the abstract Fitness class as presented in figure 3.6. In addition to the interface of the Fitness class it defines a template based variable called **m_Fitness** that stores the generic fitness value. The access method is called *GetValue()* and retrieves the template's fitness value. Manipulation of the template's fitness value can be conducted via *SetValue(Fitness)*. The initialisation is conducted by the constructor *TSimpleFitness(TFitness value)*. Because the class is a template class any type can be used as a fitness value as long as the *ObjectiveFunction* can compute and store the fitness value. So this template class provides an easily utilised interface implementing the abstract *Fitness* class's. Therefore it is a part of the framework solely aimed at convenience.

SearchHistory

The *SearchHistory* class is a storage container for the path of the Tabu Search and presented in figure 3.7. It saves the starting solution of the search process. In addition the moves of the Tabu Search and their order are stored too. This is done by assigning the number of the iterations done so far to the move and storing the combination. For a better understanding the correlation of moves and the iteration count is explained 2 paragraphs below. Anyway providing a history to search in the manner the SearchHistory class does, is a new concept compared to the current state of the art frameworks. All of them provide

a way to store and recognise tabu solutions. But to store and retrieve information about how often a specific solution has been seen, as provided by the SearchHistory class, has never been shown in any other framework mentioned in 2.3.

May define HotSpot

The iteration count is essential when storing moves in the SearchHistory class. The starting solution has an iteration count of zero and therefore the first move stored starts also at zero. The next move would get a one assigned, the next a two, and so on. This assignment is done automatically by the function void *Add (Solution * fromSolution, Move * move)*. You can see it takes two parameters - a solution and a move object. Former is not necessary, but it is provided in the case that a developer wants to extend the basic functionality of this class by deriving his/her own. Later is important, since the move is stored with the iteration count as it's key.

Removing single elements of the search history is not possible, but to clear the complete history it is possible to use *Clear()*. That method is responsible to free any previously allocated memory and therefore it is also utilised in the destructor for deallocation.

By knowing the iteration count, it is possible to fetch the move applied at that specific iteration by calling *Move * GetMoveAt (int position)*. Using the same knowledge about the iteration count, it enables the acquisition of the solution at that specific iteration by the method *Solution * GetSolutionAt (int position)*. The methods *Move * GetLastMoveAt (int position)* and *Solution * GetLastSolutionAt (int position)* provide a simlar functionality like *GetMoveAt()* and *GetSolutionAt()*. They just don't start their count at the start of the search, but at the last iteration. So their indexing starts with 0 at the latest iteration and ends at the start of the search with the current iteration count.

Static Methods

Since memory consumption can become an issue with larger problems, the number of moves that can be stored is limited and can be queried by *GetSize()*. That approach sets a boundary for the length of the history. If that limit is reached and a new move added, then the oldest move gets discarded. The maximum number of moves that can be stored, can be set by the constructor of the SearchHistory class. Though that number itself as a threshold imposed by stl::vector which is the storage container for the moves. That maximum number can always be queried by int *GetSize()*.

An other method called *int GetSize (Move * move)* returns the number of moves stored so far. To reset the SearchHistory the method clear() can be used to empty the storage by deallocating the list of moves and the initial solution.

The constructor *SearchHistory(int iMaximumSize)* initialises the maximum number of history elements that will be stored as explained below at the m_iMaxSize attribute.

Several private attributes of the SearchHistory class influence it's behaviour and store different aspects of the search path as listed below:

m_pTabuSearch is a pointer to the related TabuSearch object and provides flexibility for user derived classes.

m_pStartSolution contains the initial Solution object, where the Tabu Search started at. It can be queried by using *GetStartSolution()*

m_pStartList stores the solution where the search history starts at. In order

to access it the previously explained *GetSolutionAt()* or *GetLastSolutionAt()* can be used.
m_MoveList is a standard template library container for the moves. Direct access to it is only possible within the SearchHistory class. The public useable access method is *GetMoveAt()* or *GetLastMoveAt()*, which have been described earlier.
m_MoveCounts contains the frequency counts for moves. By using *GetMoveCount(Move::Move* pMove)* the supplied frequency count of pMove is looked up in m_MoveCounts and returned. When the move is not found, then -1 is returned.
m_pEliteSolutionList is a pointer to an EliteSolutionList class. The pointer might be null in case that no EliteSolutionList is to be used. When it is not null then the referenced object will be utilized to store elite solutions of the search path. The exact operation of the EliteSolutionList class is described further below.
m_iMaxSize limits the maximum number of solutions stored within the search history when it doesn't exceed the maximum number of elements store able in stl::vector. It can be set with *SetMaxSize()* and it can be retrieved by *GetMaxSize()*.

TabuList

This class stores the tenure and can determine in conjunction with a *SearchHistory* object if a move is tabu or not. It's UML diagram presents that relationship in figure 3.7. The concept of a tabu list is widely used in the state of the art frameworks which are given in 2.3. Though it most often just stores the tenure and the tabu moves. The *TabuList* class of this framework uses the same basic idea as the state of the art. That is to store the tabu tenure and to decide whether a move is tabu or not. Still the *TabuList* class is different, because it doesn't store any tabu moves or solutions. Though it has to decide the tabu status of a move. In order to determine it, the *TabuList* class queries the SearchHistory object for the last recently added moves. The tenure is an integer value limiting the depth of the query by specifying how many iterations a chosen move has to stay tabu as explained in chapter 2.1.3.

The class stores three variables that are explained in the following list:
m_pTabuSearch is a pointer to the related *TabuSearch* object. The only way to change it's value is in the constructor of the *TabuList* class. The method GetTabuSearch() can be used to retrieve it.
m_pSearchHistory is a pointer to the related SearchHistory object. It can only be manipulated by the constructor of the *TabuList* class. The method GetSearchHistory might be used to query it.
m_iTenure is the number of iterations that a move is tabu when it has been used in the search process. The manipulation method for changing the tenure is void setTenure(int Tenure). In order to retrieve it's value int getTenure() can be utilised.

Static Methods
The class *TabuList* provides two methods for queries about tabu status of moves and solution objects. One method is *bool isTabu(Move * pMove)* that returns the tabu status of the supplied move. When that move is found within the last

n, where n = m_iTenure, moves of the SearchHistory object, then the move is declared tabu. Otherwise it is non tabu. The other method is *bool isTabu(Solution * pSolution)* which returns tabu when a solution is found within the last n, where n = m_iTenure, solutions of the SearchHistory object. Non tabu is returned when it's not found. The constructor of the class *TabuList(SearchHistory * pSearchHistory, int iTenure)* takes a pointer to a *SearchHistory* object and the desired tabu tenure.

EliteSolutionList

The *EliteSolutionList* class is an optional part in the framework. It's purpose is to store a number of solutions with the best fitness so far encountered. The motivation of elite solutions has been outlined in 2.1. Although the aspects of elite solutions can improve the search, the state of the art framework presented in 2.3 do not provide a design that is capable of providing elite solutions out of the box. Though some of them mention elite solutions approaches like Harder [18].

The following list describes the attributes of the *EliteSolutionList* class. **m_iMaxSize** contains the threshold value determining how many elite solutions might be stored. It can be manipulated with *SetMaxSize(int MaxSize)* and queried by *GetMaxSize()*. **m_pFitnessSum** is a pointer to a *Fitness* object containing the fitness sum of all elite solutions. It is for class internal use only and therefore has not got any access or manipulating methods. **m_FitnessCount** is a counter that counts the number of elite solutions. As the previously introduced attribute, this value is also for class internal use only. **m_pFitnessMax** contains the best elite solution found so far and can be queried by *GetFitnessMax()*. **m_pFitnessMin** stores the elite solution with the worst fitness. It can be accesses by *GetFitnessMin()*.

Must Define Methods

GetFitnessAverage() returns the computed average of all fitness values.
GetFitnessMedian() calculates the median for all fitness values.
bool isElite(Solution pSolution)* tests if the supplied solution is an elite solution or not and returns true or false respectively.

May Define Methods

Test(Solution pSolution)* is the method used for testing a solution for elite characteristics and registering the fitness value of the solution regardless of it's elite status.
Clear() is responsible for clearing and deallocating all elite solutions stored.

Static Methods

Begin() returns an stl constant iterator to the list of elite solutions.
The method *End()* returns similar to Begin() an stl constant iterator, but it points at the end of the list.

The constructor *EliteSolutionList()* doesn't need any additional parameters since the class is used by *SearchHistory* when the *SearchHistory* object is explicitly told so.

Move

The class called *Move* is a collection of interfaces and therefore an abstract class. It's intention is to store a single move which is an algorithm that applies

a small change to a solution that in turn creates a new solution. This new solution is part of the neighbourhood of the original solution. Therefore the *Move* class is designed to represent a so called "move" operation of the Tabu Search terminology which can be used to create a new solution from a given one. That new solution is then part of the neighbourhood of the given solution.

Most often such a move has got one or more parameters. Since a set of values for those parameters define a specific move, they have to be stored in the class. The already introduced coupling of the *Move* with the Solution class, makes it necessary for the *Move* class to know the data structure of the *Solution* class. Otherwise it wouldn't be possible for any *Move* objects to change the Solutions objects and hence generate new Solutions. So according to the abstract nature of the *Solution* class, the *Move* class is an other class that the user must derive a class from in order to utilise the framework. In order to create a structure that is easier to understand, the framework consists of a few packages that contain parts of the framework. The *Move* class is part of the Move package, which contains all the class that are directly related to the *Move* class.

m_pMoveIterator is the only variable stored by the abstract *Move* class and is a pointer to the related MoveIterator object that manages the *Move* object.

The rest of the interface of the *Move* class is defined by deriving from the ICompareable, ICloneable, IApplyable, ISerializeable and IInvertible interfaces as seen in figure 3.8. That UML diagram presents not only the *Move* class, but the other classes of the Move package and several interfaces of the framework.

Attribute

Similar to the *Move* class the *Attribute* class is abstract and derives it's methods from interfaces. Those are: ICompareable, ICloneable, IApplyable, ISerializeable and IInvertible. The *Attribute* class is designed to store an algorithm doing an atomic change to a solution. Therefore it represents the "move attribute" definition of Tabu Search. As mentioned before moves might be broken down into so called "move attributes" in Tabu Search that . In the case this is necessary for a specific Tabu Search implementation the abstract *Attribute* class provides an interface for those "move attributes". Since the *Attribute* class is abstract the user needs to provide a derived class implementing the interface in order to utilise the framework's capabilities for "move attributes". In order to ease the utilisation of that feature a template class has been defined that might help and avoid the need for implementing a derived class. It is called TAttribut and is explained right after the *Attribut* class.

The UML diagram in figure 3.8 presents the class and it's relationship with the other classes of the Move package. This diagram shows that the *AttributeMove* class utilises a list of *Attribute* classes in order to compose a single move.

The only private variable of *Attribute* is **m_AttributeMove**, which stores a pointer to the corresponding *AttributeMove* object. It can be manipulated with *SetAttributeMove(AttributeMove* pAttributeMove)* and queried by *GetAttributeMove()*.

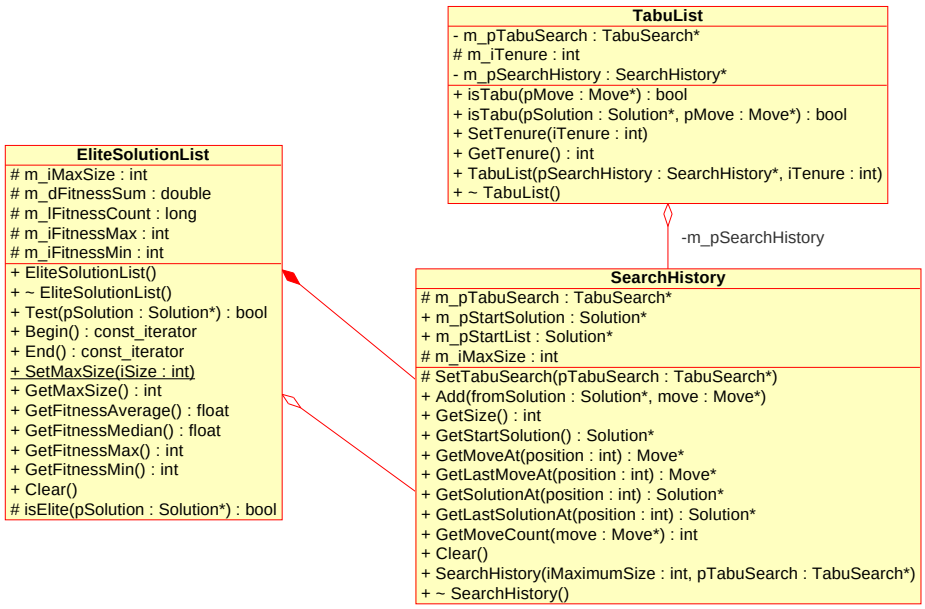

Figure 3.7: UML class diagram of memory related classes

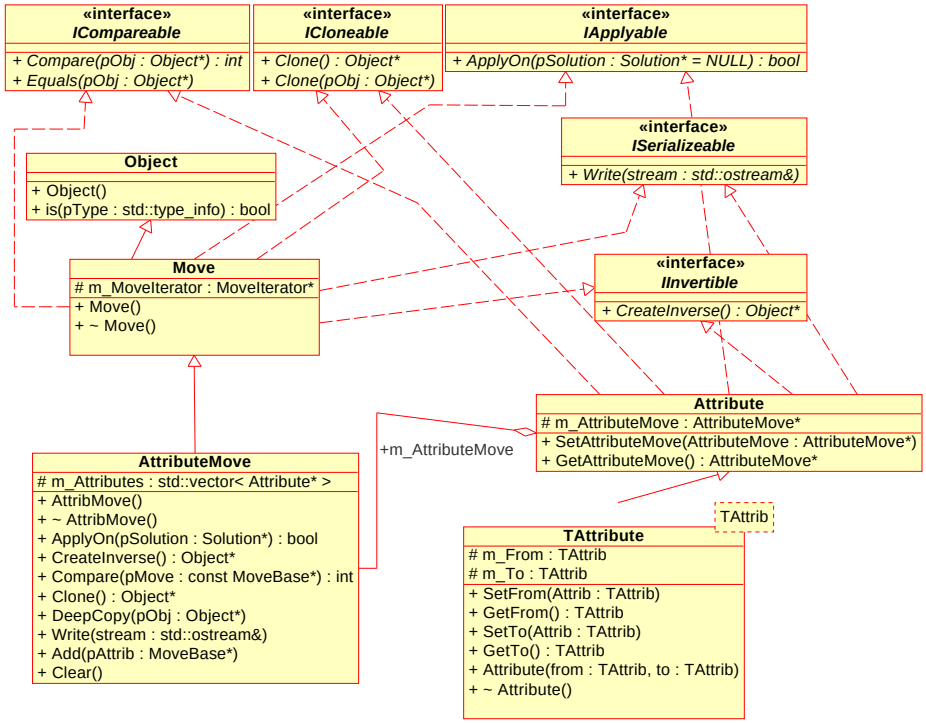

Figure 3.8: Move package UML class diagram

TAttribute

The *TAttribute* class is a template class derived and implementing the interface of *Attribute*. Additionally it provides storage for two template based values.

One called **m_From** that should store the from value of the move attribute. Its access method is called *GetFrom()* and it's manipulation method is named *SetFrom()*. An other called **m_To** which has to contain the to value of the move attribute. Similar to the **m_From** variable **m_To** has also got access an access method called *GetTo()* and a manipulation method named *SetTo()*.

The template's constructor *TAttribute(TAttrib from, TAttrib to)* just initialises the internally stored **m_From** and **m_To** with the supplied values.

This template class is used best for creating specific "move attributes" and for storing them in the container of the *AttributeMove* class.

AttributeMove

The intention of this class is to provide the possibility of breaking tabu search moves down into so called "move attribute". The flexible class structure tries to avoid limiting the developer to a predefined data structure for "move attributes". Other frameworks like OpenTS created by Harder [18] try to provide a limited approach to move attributes. Harder [18] for example represents "move attributes" by an array of integer values as defined in the ComplexMove class of his framework. The class *AttributeMove* stores a list of pointers to *Attribute* objects. It's parent class is *Move* as shown in 3.8. Therefore the AttributeMove class has to implement the interface of the Move class in order to define a non abstract class. Each of the methods *ApplyOn()*, *CreateInverse()*, *Compare()*, *Clone()*, *DeepCopy()* and *Write()* steps through the internal list of *Attribute* objects and utilize the corresponding methods of the *Attribute* object. To provide storage for *Attribute* objects it contains a list named **m_Attributes.** That list is only accessible from inside the *AttributeMove* class and derived ones, since there is no need to manipulate it's contents once the according *Attributes* have been stored inside. Never the less the list can be emptied by calling the method *Clear()*. In contrast the *Add(Attribute* pAttrib)* method adds *Attribute* objects to the list.

MoveIterator

The class *MoveIterator* simplifies the neighbourhood generation for a solution by providing an iterator interface that steps through the moves of that neighbourhood. This approach to create the neighbourhood step by step, is similar to the neighbourhood generation of Gaspero and Schaerf [6]. Therefore the *MoveIterator* object must know how to handle a *Move* object and it's parameters. This depends on the implementation of the *Move* object, which is done by the developer utilising the framework. Therefore the developer has to create one *MoveIterator* derived class per move type, that knows how to utilise the corresponding *Move* object and build a neighbourhood with a set of those objects.

The only protected variable is **m_pSolution** which points to a *Solution* object. So the generation of the neighbourhood is done for that stored *Solution* object. The method *GetSolution()* gives access to the stored object and *SetSolution(Solution* pSolution)* can be used to set it to the supplied *Solution* object.

m_pMoveManager is a pointer to a container class managing move iterators. The method *GetManager()* can be used to retrieve that *M*oveManager object and *SetManager(MoveManger* pMoveManager)* is to be used for manipulating it.

Must define HotSpots
The iterator interface provides 4 methods *Begin()*, *End()*, *Next()*, *Prev()* that must be defined by the developer. The functions *Begin(Solution* pSolution)* and *End(Solution* pSolution)* initialise the iterator by providing a *Solution* object. When the pointer to the *Solution* object is null, then the previously used solution is kept and the neighbourhood generation is reset. The methods *Next()* and *Prev()*, advances or backtracks the iterator in the neighbourhood. Both methods return an initialised *Move* object that can be used to create a solution of the neighbourhood of the stored solution.

Static Methods The static methods of the class have already been introduced, since they are the access and manipulation functions for the attributes of the class. The constructor *MoveIterator()* simply initialises internally stored variables with zero values.

MoveManager

This class is a container class for sets of *MoveIterator* objects. It manages the supplied *MoveIterator* objects by saving them in a list and providing access to it. That list then can be used to step through the neighbourhoods and to choose a best tabu move and non tabu one.

The *MoveManager* object doesn't need to be derived. It's main purpose is to be a container of the possible moves in order to structure the neighbourhood generation. So it's possible to generate several neighbourhoods and choosing the best solution. For example TSF by Hoong et. al [] shows a similar approach by providing a Neighbourhood Generator Interface, but it provides a higher flexibility by even taking constraints into account.

The variables stored with the *Movemanager* class are given by the following list: **m_pTabuSearch** is a pointer to the related *TabuSearch* object, that utilises the *MoveManager* object for stepping through the list of *MoveIterator* objects.
m_MoveIteratorList contains a list of *MoveIterator* objects. It can be accessed by *Begin()*, *End()*, *Find()* and *GetRandom()* as described below. Manipulation is possible with *Add()* and *Erase()* as detailed by the static methods description.

Static Methods
*Add(MoveIterator *pMoveIterator)* adds the supplied *MoveIterator* object to the internal list.
Begin() returns the start of the internal list of *MoveIterator* objects as a stl constant iterator.
End() returns the end of the internal list of *MoveIterator* objects as a stl constant iterator.
*Find(const MoveIterator *pMoveIter)* searches the internal list for the specified *MoveIterator* objects.
*Erase(const MoveIterator *pMoveIter)* removes the the specified *MoveIterator* objects from the internal list.
GetRandom() returns a *MoveIterator* objects at random. The constructor

MoveManager() simply initializes the internal list to be empty.

AspirationCriteria

The AspirationCriteria class is an abstract base class for aspiration criteria. So this class would be a parent class for the developer defined aspiration criteria.

A similar design can be found in HSF by Dorne and Voudouris [9] in the form of an interface or in OpenTS by Harder [19] as a single class that might contain more than one aspiration criteria.

m_pAspirationManager is the only attribute of the class. It is defined as a pointer to the container of this class and it can be accessed by *GetManger()*. Manipulation is possible with *SetManager(AspirationManager* pAspriationManager)*, but only by the class *AspiritionManager*. The variable needs only to be changed when the *AspirationCriteria* object is registered with it's container *AspiritionManager*. So there is no need to manipulate it any place else.

Must define HotSpot
*isTabuOverridden(const Solution * pSolution, const Move::Move * pMove)* is a method capable of overriding the tabu status of a move, when no better non tabu move is found. The parameter pSolution is a pointer to a solution object and pMove is a pointer to a Move object. Both may be needed in order to determine whether the tabu status is overridden or not.
The constructor *AspiritionCriteria()* doesn't take any arguments and is just the default one.

AspirationManager

Is a storage class for sets of *AspirationCriteria* objects, which have to be checked after each iteration when the non tabu move is not creating a better solution. The supplied *AspirationCriteria* objects are utilised in a first come, first serve basis. Therefore the objects registered first are used first in order to determine the aspiration. Adaption of the way the stored aspiration criteria are evaluated can be created by deriving this class and replacing it's isTabuOverridden() method.

OpenTS by Harder [19] only stores one aspiration criteria, and doesn't provide a way to choose between different aspiration criteria.

m_pTabuSearch is a pointer to the corresponding tabusearch object and it can be manipulated with *SetTabuSearch()* only by the *TabuSearch*. It's value can be queried by *GetTabuSearch()*.
m_CriteriaList is a STL list type containing AspirationCriteria objects. It can be accessed by *Begin()*, *End()*, *Find()* and *GetRandom()* as described below. Manipulation is possible with *Add()* and *Erase()* as detailed by the static methods description.

Static Methods
*isTabuOverridden(const Solution *pSolution, const Move::Move *pMove)* utilises the stored list of *AspirationCriteria* objects in order to determine the tabu state of the supplied *Move* and *Solution* objects. The actual computation is delegated to the *isTabuOverridden()* of the *AspirationCriteria* objects. Once such an object returns true for the tabu override, then this result is taken and returned equally.

*Add(AspirationCriteria *pAspCrit)* adds the supplied *AspirationCriteria* object to the internal list.
Begin() returns the start of the internal list of *AspirationCriteria* objects as a stl constant iterator.
End() returns the end of the internal list of *AspirationCriteria* objects as a stl constant iterator.
*Find(const AspirationCriteria *pAspCrit)* searches the internal list for the specified *AspirationCriteria* objects.
*Erase(const AspirationCriteria *pAspCrit)* removes the specified *AspirationCriteria* objects from the internal list.

The constructor *AspiritionManager()* initialises the internal list as empty.

SolutionMovePair

The *SolutionMovePair* class is a container for a solution - move pair. It is used by the *TabuSearch* in order to store solution - move pairs for the best tabu and non tabu move.

Since this a rather special helper class for storing the relationship between *Solution* and *Move* objects, no equivalent class could be found in the state of the art frameworks.

The following list explains the various attributes of the class: **m_bEmtpy** is a boolean value that stores whether the *SolutionMovePair* object is empty or not. The method *isEmpty()* returns it's value.
m_pSolution is used to store a pointer to a *Solution* object and can be queried by *GetSolution()*.
m_pMove finally contains a *Move* object that can be retrieved with *GetMove()*.

May define HotSpot
*SetIfBetter(Solution *pSolution, Move::Move *pMove)* stores the supplied solution - move pair only when the fitness of the internally stored one is worse than the fitness of the supplied solution - move pair.

Static Methods
Set() is a method for simply storing the supplied solution - move pair.
Clear() deallocates the internally solution - move pair and sets m_bEmpty to true.
Write(std::ostream& stream) just invokes the Write() methods of the stored *Solution* and the *Move* objects. The constructor *SolutionMovePair(Solution *pSolution, Move* pMove)* initialises the internal variables to represent an empty state, only when pSolution and pMove point contain null values.

3.5 Implementation

One part in framework development is the creation of the design Another is to actually use the framework in order to see if it's applicable or not and to find problematic parts. Afterwards those areas can be eliminate by changing the design. This shows that framework development is an ongoing process that cycles again and again as outlined in [10].

Depending on the programming language chosen there might be the need to adopt the design in order to utilise the features of the language. Therefore a short primer about the computer language of the framework of this thesis

and some of the advantages and disadvantages of that language is given. Additionally special features and enhancements for the framework introduced by the chosen language are outlined afterwards.

3.6 C++

Most of the framework parts are actually independent from the choice of object oriented computer language. Nevertheless some are restricted to C++, because the framework was first designed with the notion of implementing it in C++ later, though they might be implemented with small changes in other languages offering similar capabilities like C++. Anyway the chosen computer language used for implementing the framework in the first place was C++. The choice made was based on several facts:

- It is a well known object oriented language, since it has a long history of use and can therefore be judged to be stable.
- It is very portable, since there is a well known standard definition known as ANSI C++. There are compilers for Windows and Unix operating systems supporting that standard.
- It offers a high level of performance when compared to other object oriented languages.

Besides those benefits there are some other benefical tools that C++ provides. For example genericity, so called *templates* that can be used to define methods and functions that are type independent.

An other benefit are so called *inline* functions which can be used to define a method whose source code is inserted at the place where it is invoked. This can improve performance improvements for a *inlined* function having a small code foot print.

A further benefit is the *const* operator, which used on a variable, makes it read-only. In case of an object being declared *const*, modifications of its contents are forbidden. In functions that return some part of the contents of the *const* object, have to be declared *const* too, otherwise access to them is forbidden by C++.

One major disadvantage of C++ is related to the goto command and labels, since those statements can create source code that is very hard to read. Therefore this thesis avoids their use. Of course there lots of other problems like the goto command, revolving around the origin of C++'s origin in C. They have been avoided too.

Casting objects to a specific parent or child class can be problematic if the type is not known at the time of writing the program, but when it will be determined at run time. To remove that problem C++ provides the dynamic_cast<>() template function, which does the cast and throws an exception when the specified type cast cannot be conducted.

Another major benefit of C++ is the Standard Template Library, which is a library that contains several algorithms and data storage structures. These are quite easy to use and the programmer doesn't need to go through the error prone development process to write simple data storage structures on his own.

3.7 Implementation Specifics

As already outlined before, a special class has been defined called Object. Its only purpose is to provide an absolute parent for other classes, so that there is a global parent class. This makes the definition of the previously introduced interfaces of chapter 2.2.2 possible. Other object oriented languages might provide such a master parent class type, like C# does. Therefore this is a definition specifically introduced to C++.

The file structure in the file system is also important for understanding the source code. The framework uses three different file endings: .h, .hpp, .cpp Therefore one class usually has a .h and a .cpp file. The .h file is the so called header file containing the class and type definitions whereas the .cpp file contains the functions and their code for that class. The .hpp file is quite special since it contains inline functions and template definitions and is always included by the corresponding .h file.

The UML template TSimpleFitness class is defined as a C++ template class with a template parameter named TFitness. So fitness values can be of any type the user might chose.

Similar the UML template TAttribute class is defined as a C++ template class with a template parameter called TAttrib.

The classes utilising Standard Template Library features provide type definitions to shorten certain type declaration like: typedef std::vector<Attribute*> AttribVec; in order to enhance the readability of the source code.

Chapter 4

Application of the framework - A case study

As already discussed in chapter 2.2.4 a framework alone is not enough to testify it's usability, but a program utilising the framework can be used to do so. Therefore this chapter is concerned with an example application utilising the diploma thesis framework. A problem from literature has been chosen and is described in detail at the start of this chapter. Then an application is presented, which solves it by utilising the framework of this diploma thesis. Afterwards the gathered computation results are presented and compared to literature.

4.1 An example problem

The best way to test the design of the framework is by writing one or more applications utilising it. During the application development and implementation, problem spots within the tested framework can be found. Then a new development cycle of the framework can take place, where the gathered results of the test application are used to improve the design in order to remove the problem spots.

In the case of the Tabu Search framework of this diploma thesis, a NP hard problem from literature has been chosen. Because a test is best conducted with an example that has been proven to be solvable by Tabu Search, a well known problem had to be taken. The rotating workforce scheduling problem presented by Musliu et. al [30] fulfils that criteria. As its name suggests, rotating workforce scheduling is concerned with creating schedules for a number of employees where monotonic repetition of shifts is avoided by rotating shift assignments between employees. That might sound a bit complicated, but it is worth the effort, since workforce schedules affect both the health and the social life of the employees. Badly planned schedules can also increase the risk of work-related accidents. Therefore it is of high practical relevance to find workforce schedules, that on one hand fulfil the ergonomic criteria for the employees, and on the other reduce costs for the organisation.

4.1.1 Rotating workforce scheduling

Before discussing how the framework was utilised to solve the problem, the problem definition as found in Musliu et. al [30] is presented:
Instance:

- Number of employees: n.
- Set A of m shifts (activities) : $a_1, a_2, \ldots, a_m$, where a_m represents the special day-off "shift".
- w: length of schedule. The total length of a planning period is $n \times w$ because of the cyclic characteristics of the schedules.
- A cyclic schedule is represented by an $n \times w$ matrix $S \in A^{nw}$. Each element $s_{i,j}$ of matrix S corresponds to one shift. Element $s_{i,j}$ shows which shift employee i works during day j or whether the employee has time off. In a cyclic schedule, the schedule for one employee consists of a sequence of all rows of the matrix S. The last element of a row is adjacent to the first element of the next row, and the last element of the matrix is adjacent to its first element.
- Temporal requirements: $(m-1) \times w$ matrix R, where each element $r_{i,j}$ of matrix R shows the required number of employees for shift i during day j.
- Constraints:
  - Sequences of shifts permitted to be assigned to employees (the complement of inadmissible sequences): Shift change $m \times m \times m$ matrix $C \in A^{(m^3)}$. If element $c_{i,j,k}$ of matrix C is 1, the sequence of shifts (a_i, a_j, a_k) is permitted, otherwise it is not.
  - Maximum and minimum length of periods of consecutive shifts: Vectors $MAXS_m$, $MINS_m$, where each element shows the maximum respectively minimum permitted length of periods of consecutive shifts.
  - Maximum and minimum length of blocks of workdays: $MAXW$, $MINW$.

In order to create a better understanding of the area of rotating workforce scheduling an example schedule found in [29], [27] is discussed here:

A workforce schedule represents the assignments of the employees to the defined shifts for a period of time. In Table 4.1 a typical representation of workforce schedules is presented. This schedule describes explicitly the working schedule of 9 employees during one week. The first employee works from Monday until Friday in a day shift (D) and during Saturday and Sunday has days-off. The second employee has a day-off on Monday and works in a day shift during the rest of the week. The forth employee has got day-off starting with Monday and ending with Thursday and works in afternoon shifts (A) on Friday, Saturday and Sunday. Further, the last employee works from Monday until Wednesday in night shifts (N), on Thursday and Friday has days-off, and on Saturday and Sunday works in the day shift.

Table 4.1: One typical week schedule for 9 employees by Mörz and Musliu [27]

Employee/day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	D	D	D	D	D	-	-
2	-	D	D	D	D	D	D
3	D	-	-	N	N	N	N
4	-	-	-	-	A	A	A
5	A	A	A	A	-	-	-
6	N	N	N	N	N	-	-
7	-	-	A	A	A	A	A
8	A	A	-	-	-	N	N
9	N	N	N	-	-	D	D

Each row of this table represents the weekly schedule of one employee.

According to Musliu et. al [30] there are two main variants of workforce schedules: rotating (or cyclic) workforce schedules and non-cyclic workforce schedules. In a rotating workforce schedule all employees have the same basic schedule but start with different offsets. Therefore, while the individual preferences of the employees cannot be taken into account, the aim is to find a schedule that is optimal for all employees. In non-cyclic workforce schedules the individual preferences of the employees can be taken into consideration and the aim is to achieve schedules that fulfil the preferences of most employees. In both variations of workforce schedules other constraints such as the minimum number of employees required for each shift have to be met. In this paper we will consider the problem of rotating workforce scheduling. This problem is a NP-complete problem.

4.1.2 Solving rotating workforce scheduling

This section is concerned with the creation of an algorithm solving the presented rotating workforce scheduling problem by utilizing the framework. According to the design presentation in chapter 3 following classes must be subclassed: Solution, ObjectiveFunction, Move and MoveIterator. The source code of each function presented here, is given in appendix A along with an in depth discussion of its source code.

Two additional data types have been defined for easing the implementation: _eDay and CRequirements. First both are discussed, since the understanding of them is important for understanding the rest of the source code.

For storing schedules, an enumeration data type called _eDay has been defined, enumerating the days of a week.

Most important, for beeing able to solve a number of rotating workforce problems, is the flexibility regarding constraints. To satisfy this need, the class CRequirements has been defined and is used to save and load problems defined in text files. The class itself stores the number of shifts and employees, the number of shifts per day, the prohibited sequences of shifts, the maximum length of work- or day off - blocks and the maximum and minimum number of consecutive shifts of a specific shift type.

Solution

Solution is derived by CWeek which represents one week schedule. Since it has to represent a single week schedule, it has to store that information. This is done with a memory block of integer values, where each integer represents a shift of a day for an employee. In order to know the number of shifts and employees, the class CWeek needs access to the information of the CRequirements class. That access is provided by supplying the constructur with a CRequirements object. All necessary functions required by the Solution class are implemented like: DeepCopy(), Clone(), Write(), Compare() and Equals(). They fulfil the specification as presented with the coresponding interface in chapter 3.4.2 handling the internally stored week schedule. Appendix A.2 shows the implementation of those methods. Initialisation of the solution is done with Initialize() which sets up a solution that fulfils the number of shifts per day constraints. That in turn reduces the complexity of the problem, when the used move operators do not violate those constraints. In addition, the functions getShift(int iEmployee, _eDay day) and setShift (int iEmployee, _eDay day, int iShiftValue) have been defined as an easy way to access the week schedule. In order to calculate the fitness, a method is needed to step forth and back from one shift to another. A way to define such a stepping is to step from one day to the next in the shifts of one employee. When Sunday is reached, a wrap around to the next employee's Monday is done. The functions nextShift (int *iEmployee, _eDay *day) calculates the next shift of that stepping, in contrary prevShift (int *iEmployee, _eDay *day) steps the other way round.

ObjectiveFunction

ObjectiveFunction is derived by CObjectiveFunction which creates an integer fitness value stored with the template class TSimpleFitness<int> for a specific CWeek object.

The evaluation of solutions is the most time consuming task compared to the rest of the algorithm, because each solution has to be checked for many constraints. So it is especially vital for a successful implementation to pick a good and fast evaluation. Such an evaluation is presented in Musliu [29]: For each violation of a constraint a determined number of points (penalty) is given, based on the constraint and the degree of the constraint violation. The fitness of a population member is calculated as the sum of those points for the population member. So the fitness represents the sum of all penalties caused by the violation of constraints. Since the problem to be solved has only hard constraints, the solution will be found when the fitness of the solution reaches the value 0. The fitness is calculated like in Musliu [29]:

$$
\begin{aligned}
Fitness = & \sum_{i=1}^{NW} P1 \times Distance(WB_i, WorkBRange) + \\
& \sum_{i=1}^{ND} P2 \times Distance(DOB_i, DayOffBRange) + \\
& \sum_{j=1}^{NumOfShifts} (\sum_{i=1}^{NS_j} P3 \times Distance(SB_{ij}, ShiftSeqRange_j)) +
\end{aligned}
$$

$$P4 \times NumOfNotAllowedShiftSeq$$

Where - according to Musliu [29] - NW, ND, represent respectively, the number of work -, and days off - blocks, whereas NS_j, represents number of shift sequences blocks of shift j. WB_i, DOB_i, represent, a work block i, and days-off block i. SB_{ij}, represents the i-th shifts block of shift j. The function $Distance(XBlock, range)$ returns 0 if the length of the block $XBlock$ is inside the range of two numbers ($range$), otherwise returns the distance of length of $XBlock$ from the range. For example, if the legal range of work blocks is $4-7$ and length of work block $XBlock$ is 3 or 8 then this function will return value 1.

In the example program the following penalty weights are used: $P1 = 1, P2 = 1, P3 = 3, P4 = 2$

Move

Move is derived by CSwapShiftMove, which defines a swap of two shift blocks as defined in Musliu [29]. The major advantage of this operation is to leave the shift requirements of the days intact. The Move class provides a set of interfaces that must be implemented. Accordingly the required methods: CreateInverse(), ApplyOn(), Compare(), Equals(), Clone(), DeepCopy() and Write() have been implemented as can be seen in Appendix A.4. CreateInverse() just passes back a copy of the CSwapShiftMove object, because a CSwapShiftMove is also the inverse of itself. ApplyOn() does the actual swap on a CWeek object by swapping the shift block as specified by the internal variables when the contents of the shift block are not equal. If the contents are equal, the move will be futile and therefore not done. In that case ApplyOn returns false, otherwise it returns true. Compare() conducts a comparison of the internal variables of the CSwapShiftMove object with the supplied CSwapShiftMove object. It returns -1, 0 or 1 if the internal values are found, to be less than, to match, or be greater than the supplied CSwapShiftMove ones. Clone() creates a new object containing the same internal values. DeepCopy() just assignes the supplied CSwapShiftMove object's data to the internal variables. For each variable of m_eDay, m_iShift1, m_iShift2 and m_iLength an access method and a manipulation method have been defined. Since experiments have shown that a single swap is sometimes not solving the problem, a number of consecutive shifts can be swapped. Therefore the move stores, the day, two employees - who's shifts get swapped, and a length which defines the number of shifts to swap.

MoveIterator

MoveIterator is subclassed by CSwapShiftMoveIterator defining a MoveIterator that handles the CSwapShiftMove. The interface of MoveIterator is implemented by defining four functions: prev(), next(), begin() and end(). begin() and end(), which simply set the internal variables to the first or the last move and return a move representing the corresponding move. next() and prev() step through the moves by changing the internal variables storing the current position. The internal variables are the same as the ones of the CSwapShiftMove. They are used to construct a CSwapShiftMove object whenever getMove() is called. To limit the maximum length of a block getting swapped the variable m_iMaxLength is defined and can be set with the constructor.

Aspiration::Criteria is subclassed by CBestAspiration and overrides the tabu state of a move when it leads to a solution that is better than the current solution. Therefore it implements the isTabuOverridden() method which queries the Aspiration::Manager for the TabuSearch object in order to gain access to the fitness of the current solution. Then the fitness of the current solution is compared to the solution's fitness generated by the tabu move. If it's better, then true is returned which overrides the tabu state of the move. Otherwise false is returned.

The main program glues everything together. In order to provide a little bit of flexibility for testing purposes, command line arguments have been added, allowing a higher verbosity or starting with a specific random seed so that the initial solution is a specific one. As the source code in A.1 shows a set of objects needs to be created. The ones originating from the framework are: a TabuSearch, a MoveManager, an Aspiration::Manager, a SearchHistory and a TabuList object. The TabuSearch object provides the central Search class used to start the search and to retrieve the result. The SearchHistory and the TabuList provide the tabu memory for the search. The Manager classes manage the MoveIterator objects and the Aspiration::Criteria objects respectively. The MoveManager object needs at least one MoveIterator object and therefore it gets a new CSwapShiftMoveIterator object added by a call to Add(). The Aspiration::Manager doesn't need an Aspiration::Criteria object. Still it gets an object of type CBestAspiration added, because this is needed in order to make the Tabu Search working for this problem.

The only remaining necessary step is to initiate the search process by calling Search() of the TabuSearch object, passing it an instance of a Solution object, which is in our case a CWeek object. The rest of the search process is now done by the framework with the help of the supplied objects: CWeek, CObjectiveFunction, CSwapShiftMove, CSwapShiftMoveIterator and CBestAspiration.

When the search terminates, the results are printed and the program terminates.

4.1.3 Results

Depending on the number of employees, the shifts, and the different constraints, the algorithm might work better or worse. Therefore four different example problems in the area of rotating workforce scheduling have been chosen from literature as found in Mörz and Musliu [27] and solved. They are called Laporte, Heller and 27 Groups problem. The following sections define the constraints of those problems and include a solution that was found by the example program. Basically each problem has to fulfil the following constraints:

There exist three non overlapping shifts D, A, and N. A week schedule has to be constructed that fulfills: (1) Rest periods should be at least two days off, (2) Work periods must be between 4 and 7 days long. (3) minimum of consecutive D shift is 2, maximum is 7 (4) minimum of consecutive A shift is 2, maximum is 6 (5) minimum of consecutive N shift is 2, maximum is 4.

Problem 1

In addition to the basic constraints following constraints must be fulfilled: 4 employees, and requirements are 1 employees in each shift and every day.

Table 4.2: solution schedule for Problem 1

Employee/day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	-	D	D	A	A	N	N
2	N	-	-	D	D	D	D
3	D	A	A	-	-	A	A
4	A	N	N	N	N	-	-

Table 4.3: solution schedule for Laporte

Employee/day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	-	-	-	D	D	D	A
2	A	N	N	-	-	A	A
3	N	N	-	-	-	D	D
4	D	D	D	-	-	-	D
5	D	D	D	D	D	-	-
6	A	A	A	A	A	-	-
7	-	A	A	A	A	A	N
8	N	-	-	N	N	N	N
9	-	-	N	N	N	N	-

The solution found by the example program is given in table 4.2.

Problem 2 Laporte

Actuall this is not a specification that would fulfill the contraints for the problem presented by Laporte [24], but it's rather a subset of constraints of that problem.

In addition to the basic constraints following constraints must be fulfilled: 9 employees, and requirements are 2 employees in each shift and every day.

The solution found by the example program is presented in table 4.3.

Problem 3 Hellerplan

Heller [20] has shown a problem that is more difficult to solve than the presented problem by Laporte [24]. In addition to the basic constraints following additional ones must be fulfilled: 17 employees, and requirements are 4 employees in each shift and every day except for following day - shifts combinations. D shift and A shift on monday have to have 5 employees. D shift on sunday has to be 3. N shift on Tuesday, Wednesday and Thursday have to be 3.

The solution found by the example program is shown in table 4.4.

Problem 4 27-Groups

In addition to the basic constraints following additional ones must be fulfilled: 27 employees, and 7 employees in each shift on Monday, Tuesday, Wednesday, Thursday and Friday and 4 employees in each shift on Saturday and Sunday.

Table 4.4: solution schedule for Hellerplan

Employee/day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Emp0	D	D	N	N	N	-	-
Emp1	D	D	D	D	-	-	-
Emp2	A	A	A	A	A	-	-
Emp3	D	D	D	D	N	N	-
Emp4	-	A	A	A	A	A	-
Emp5	-	-	A	A	A	A	A
Emp6	A	-	-	N	N	N	N
Emp7	-	-	D	D	D	A	A
Emp8	A	-	-	-	D	D	D
Emp9	D	-	-	D	D	D	D
Emp10	A	A	A	-	-	-	D
Emp11	D	D	D	-	-	D	D
Emp12	N	N	N	-	-	N	N
Emp13	N	N	-	-	D	D	N
Emp14	N	-	-	A	A	N	N
Emp15	N	N	-	-	-	A	A
Emp16	A	A	N	N	N	-	-

The solution found by the example program is given in table 4.5.

Average results for 100 runs

The table 4.6 gives the average results acquired by executing 100 runs per problem. It shows the number of evaluations, the number of iterations and the time it took to find a solution with fitness zero. The measurements have been taken on an AMD Athlon 3200+ with 1GB RAM and Debian Gnu Linux (testing) with a kernel version 2.6.8 as an operating system. All unnecessary programs have been shut down for the measurements, so only the following daemons have been running aside the rotating workforce scheduling problem solver:

```
/sbin/portmap -i 127.0.0.1
/sbin/syslogd
/sbin/klogd
/usr/sbin/cupsd
/usr/bin/dbus-daemon --system
/usr/sbin/exim4 -bd -q30m
/usr/sbin/inetd
/usr/sbin/sshd
/usr/sbin/famd -T 0
/usr/sbin/atd
/usr/sbin/cron
```

The presented results show that the framework was used successful to solve the rotating workforce scheduling problem. The implementation gives similar results like the Tabu Search approach in Musliu [29].

Table 4.5: solution schedule for 27-Groups

Employee/day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Emp0	-	A	A	A	A	-	-
Emp1	-	-	D	D	D	A	A
Emp2	N	N	-	-	A	A	A
Emp3	A	-	-	D	D	D	N
Emp4	N	-	-	-	-	A	A
Emp5	A	N	N	N	-	-	D
Emp6	D	A	A	-	-	-	N
Emp7	N	N	N	-	-	N	N
Emp8	N	N	-	-	D	D	D
Emp9	N	N	N	N	-	-	-
Emp10	A	A	N	N	N	-	-
Emp11	N	N	N	N	N	-	-
Emp12	A	A	A	A	A	-	-
Emp13	A	A	A	N	N	N	-
Emp14	-	D	D	D	D	D	D
Emp15	-	-	D	D	N	N	-
Emp16	-	-	A	A	N	N	N
Emp17	-	-	-	D	D	A	A
Emp18	A	A	-	-	-	D	D
Emp19	D	D	D	D	D	-	-
Emp20	D	D	D	A	A	-	-
Emp21	N	N	N	N	N	-	-
Emp22	D	D	D	D	D	-	-
Emp23	D	D	A	A	A	-	-
Emp24	D	D	N	N	N	-	-
Emp25	D	D	D	A	A	-	-
Emp26	A	A	A	A	A	-	-

Table 4.6: examples result statistics

Problem	evaluations	iterations	time
1	54286	317,29	0.62 seconds
2	66673,9	92,76	0.52 seconds
3	1894687,3	924,54	4 Minutes 35.45 seconds
4	81573502,6	28945,23	26 Minutes 45.34 seconds

Chapter 5

Conclusion, Questions and Perspectives

This diploma thesis presented a framework for Tabu Search, which is intended to alleviate the time consuming task to reimplement the basic structure and algorithm of Tabu Search. Therefore it should lead to a reduction in time and resources needed for creating programs solving problems with Tabu Search. The state of the art frameworks already provide solutions, for the reimplementation of the basic structure and algorithm of Tabu Search. Still none of those frameworks acknowledges the frequency memory by providing a design for it. By extending the basic structure and algorithm of Tabu Search by a facility to gather frequency information the framework of this thesis provides a feature which is not yet to be found in state of the art.

The presentation of the design was created with the help of UML, which is known to be a modern tool for describing object oriented designs. The design of the framework itself provides a basic structure and algorithm for Tabu Search. The developer must derive 4 classes of the framework in order to provide the framework with the necessary information about the problem. The super classes that need to be subclassed are: Solution, ObjectiveFunction, Move and MoveIterator. The framework works like a blackbox, taking those 4 classes as an input and providing a solution to the given problem. Nevertheless the developer can modify the algorithms of that black box by deriving further classes, which in turn replace functionality in the framework.

A successful application of the proposed framework has been shown for a NP-hard problem. As the results of this application suggest, the work needed for creating Tabu Search algorithms can be reduced by utilising the framework.

The designs of the state of the art frameworks show that there are a lot of possible design extension that could be added to the framework proposed in this diploma thesis. Among those, the following design idea is most likely one of the more important ones. A future extension could be the design of a messaging system for the framework in a similar fashion like the one found in OpenTS. Such a messaging system generates messages when certain events occur like finding a new non-tabu solution, etc. Those messages can be intercepted by user defined call back functions, which have registered in the messaging system. This allows the developer to influence the search process at certain places without the need

to derive and implement major parts of the framework. Therefore this offers a convenient and easier way to extend and adopt the frameworks functionality to the users need. Additionally it would add the possibility to influence the search for the end user, when an user interface has been created for that task.

Appendix A

Example Problem Source Code

This section presents and discusses the source code to the program that solves the rotating workforce scheduling problem as shown in chapter 4.1.1. First the main program is given in order to present a high-level overview before looking into the details of the used objects.

A.1 The main part

The following lines of code show the creation of the Tabu Search objects, their feeding into the framework's TabuSearch, followed by the starting of the search and ending by displaying the results.

For a better understanding the class names of problem specific objects have been prefixed with a "C" , so that you can easily identify classes that are not part of the framework.

Listing A.1: Main program main.cpp

```
1  int main(int argc, char **argv) {
2  try {
3      CSettings Settings(15,45,DEFAULTPROBLEMFILE);
4      CRequirements Req;
5
6      Settings.ParseCmdlineArgs(argc,argv);
7      Settings.SeedInitialize();
8
9      Req.Load(Settings.GetFilename().c_str());
10
11     MoveManager *pMM = new MoveManager;
12     CSwapShiftMoveIterator *pSMI = new CSwapShiftMoveIterator(&Req,4);
13     pMM->Add((MoveIterator *)pSMI);
14
15     Aspiration::Manager *pAM = new Aspiration::Manager();
16     CBestAspiration *pCBA = new CBestAspiration();
17     pAM->Add(pCBA);
18
19     CObjectiveFunction *pObjFunc = new CObjectiveFunction(&Req);
20     SearchHistory *pSH = new SearchHistory (Settings.GetHistory());
21     TabuList *pTL = new TabuList (pSH, Settings.GetTenure());
22     TabuSearch *Tb = new TabuSearch(pMM, pObjFunc, pAM, pSH, pTL );
23
24     CWeek Week (&Req);
25     Week.Initialize();
26
27     Tb->SetMaxIterations( 24000 );
28
29     if (Settings.isVerbose())
30         Tb->SetDebug( DEBUG_TS_BEFORE_LOOP
31             | DEBUG_TS_AFTER_LOOP
32             | DEBUG_TS_MOVE_APPLIED
33             | DEBUG_TS_TABU_STATUS
34             | DEBUG_TS_SAVESOLUTION_BEFORE
35             | DEBUG_TS_SAVESOLUTION_AFTER
36             );
37
38
39     Tb->Search (&Week);
40     CWeek *pW = (CWeek *) Tb->GetBestSolution();
41     cout << "Best Solution found:" << endl << *pW << endl
42       << "Runtime: " << Tb->GetRunTime() << endl
43       << "Iterations: " << Tb->GetIterations()->GetCount()
44       << endl
```

```
45        << "Evaluations: " << Tb->GetEvaluations()->GetCount()
46        << endl;
47 } catch (const char *str) {
48     cout << "caught: " << str << endl;
49     return 1;
50 }
51 delete Tb;
52
53 return 0;
54 }
```

CSettings is a container object for the settings of the example program like which problem has to be read, which random seed is to be utilised, etc.

CRequirements is a problem specific object containing the constraints that have to be satisfied. It loads the constraints from a file who's filename was determined above.

Line 6 parses the command line parameters.
Line 7 initializes the random seed of the random number generator.
Line 9 loads the requirements from a problem file.
Line 11 creates a MoveManger object which is part of the framework. It's purpose is to be container for objects that create neighbourhood sets for a specific solution.
Line 12 creates a neighbourhood generating object that is problem specific and therefore is not part of the framework.
Line 13 adds the problem specific neighbourhood generation to the container's content.
Line 15 creates a container object for aspiration criteria.
Line 16 creates an aspiration object that removes tabu status when it is the best solution for the complete neighbourhood.
Line 19 creates a problem specific objective function for evaluating solutions regarding their fitness.
Line 20 creates a framework object that will contain the last 40 moves and their fitness values.
Line 21 creates a tabu list object and sets it's tenure to 20 and tells it to utilise the previously created history object for verifying if something is tabu or not.
Line 22 creates the TabuSearch object which is the framework's main object that contains the tabu search algorithm.
Line 24 and 25 create a problem specific solution representing object and initialise it.
Line 29 turns off verbosity according to command line parameters.
Line 39 starts the actual search.
Line 40 to 46 show the best solution found.

A.2 The solution representation

The framework provides an abstract class called Solution for representing solutions. Their purpose is to store a solution of the search space regardless it's fitness.

The class CWeek is derived from the Solution class and therefore it has to implement the abstract clone() functions. If the debugging facility of the

framework is used, then the write() function must also be defined. Since the debugging is turned on as default, it must be provided.

Listing A.2: class CWeek

```
1  CWeek::CWeek (CRequirements *pRequirements) : Solution()
2  {
3      this->m_pRequirements = pRequirements;
4      this->m_piWeek = new int[m_pRequirements->GetEmployeeCount () *
5                               m_pRequirements->GetLengthOfSchedule ()];
6      SetFitness(new TSimpleFitness<int> (100000));
7  }
8
9  CWeek::CWeek (const CWeek & Week) : Solution()
10 {
11     int iAllocSize = 0;
12
13     this->m_pRequirements = Week.m_pRequirements;
14     iAllocSize = m_pRequirements->GetEmployeeCount () *
15     m_pRequirements->GetLengthOfSchedule ();
16     this->m_piWeek = new int[iAllocSize];
17     memcpy (this->m_piWeek, Week.m_piWeek, sizeof (int) * iAllocSize);
18 }
19
20 CWeek::~CWeek ()
21 {
22     if (this->m_piWeek)
23     delete[] m_piWeek;
24 }
25
26 void
27 CWeek::Initialize (void)
28 {
29     int *map = new int[this->m_pRequirements->GetShiftCount ()];
30     int *day_R = new int[this->m_pRequirements->GetShiftCount ()];
31     int iShift;
32     int iMaxShift;
33     int iEmployee;
34     int iHelpShift;
35     int iRandomEmployee;
36     int y;
37
38     m_bChanged = true;
39
40     for (int iDay = 0; iDay < 7; iDay++) {
41     this->m_pRequirements->GetRequirements (day_R, (_eDay) iDay);
42
43     y = 0;
44     for (int x = 0; x < this->m_pRequirements->GetShiftCount (); x++) {
45         if (day_R[x] > 0)
46         map[y++] = x;
47     }
48     iMaxShift = y;
49     for (iEmployee = 0;
50          iEmployee < m_pRequirements->GetEmployeeCount ();
```

```
51          iEmployee++) {
52          if (iMaxShift == 0) {
53          break;
54          }
55          iShift = random () % iMaxShift;
56          setShift (iEmployee, (_eDay) iDay, (map[iShift] + 1));
57
58          day_R[map[iShift]]--;
59          if (day_R[map[iShift]] == 0) {
60          memmove (&map[iShift], &map[iShift + 1],
61               sizeof (int) * (iMaxShift - (iShift + 1)));
62          iMaxShift--;
63          }
64      }
65
66      if (iEmployee != 0) {
67          for (iEmployee;
68           iEmployee < m_pRequirements->GetEmployeeCount ();
69           iEmployee++) {
70          iRandomEmployee = random () % iEmployee;
71          iHelpShift = getShift (iRandomEmployee, (_eDay) iDay);
72          setShift (iRandomEmployee, (_eDay) iDay,
73                m_pRequirements->GetShiftFree ());
74          setShift (iEmployee, (_eDay) iDay, iHelpShift);
75          }
76      } else {
77          for (iEmployee;
78           iEmployee < m_pRequirements->GetEmployeeCount ();
79           iEmployee++) {
80          setShift (iEmployee, (_eDay) iDay,
81                m_pRequirements->GetShiftFree ());
82          }
83      }
84
85      }
86
87      delete map;
88      delete day_R;
89  }
90
91  void CWeek::DeepCopy (const Object *pSolution){
92      int iAllocSize = 0;
93      const CWeek *pWeek = dynamic_cast<const CWeek *> ( pSolution );
94
95      assert(this->m_pRequirements == pWeek->m_pRequirements);
96      iAllocSize = m_pRequirements->GetEmployeeCount () *
97      m_pRequirements->GetLengthOfSchedule ();
98      memcpy (this->m_piWeek, pWeek->m_piWeek, sizeof (int) * iAllocSize);
99
100     SetFitness( dynamic_cast<Fitness *> ( pWeek->GetFitness()->Clone() ) );
101 }
102
103 Object *CWeek::Clone (void) const
104 {
```

```
105     CWeek *pNewWeek = new CWeek(*this);
106     pNewWeek->SetFitness( dynamic_cast<Fitness *> ( GetFitness()->Clone() ) );
107     return pNewWeek;
108 }
109
110 void CWeek::Write(std::ostream &stream) const
111 {
112     stream << *this << endl;
113 }
114
115 int CWeek::Compare(const Object* pObj) const
116 {
117     const CWeek *pWeek = dynamic_cast<const CWeek *> (pObj);
118
119     for (int iEmployee=0;
120             iEmployee < m_pRequirements->GetEmployeeCount();
121             iEmployee++) {
122         for (int iDay = 0; iDay < 7; iDay++) {
123             if ( getShift (iEmployee, (_eDay) iDay) <
124                     pWeek->getShift (iEmployee, (_eDay) iDay) ) {
125                 return -1;
126             }
127             if ( getShift (iEmployee, (_eDay) iDay) >
128                     pWeek->getShift (iEmployee, (_eDay) iDay) ) {
129                 return -1;
130             }
131         }
132     }
133     return 0;
134 }
135
136 bool CWeek::Equals(const Object* pWeek) const
137 {
138     return Compare(pWeek) ? true : false;
139 }
```

Line 1-7 show the constructor, which initializes and allocates internal variables for the solution representation.
Line 9-14 shows a copy constructor for the class.
Line 20-24 shows the destructor deallocating memory resources.
Line 26 the initialization Function which creates a schedule that fullfills the requirement matrix as outlined in Chapter 4.
Line 40 starts the outer loop over the 7 days of a week.
Line 41 copies the shift requirements containing how many shifts of a specific type the day must contain into a temporary vector.
Line 43 to 47 initialises an array that keeps track of the shifts whose day requirements haven't been met yet.
Line 49 starts a loop over the employees of a day that's either terminated when no more employees are available or all shift requirements are met (Line 31-33).
Line 55,56 set the shift value of an employee to a random shift.
Line 58 to 62 removes the shift previously used from the requirements.
Line 66 to 83 fill the rest of the day with day off shifts when there are employees still left.

Line 91 shows the DeepCopy(const Object *pSolution) function that should copy the contents of the pSolution object to the invoking one. A very important issue to notice here is that the cloning should not be a copying of pointers, but of the memory they point to. If necessary allocation of new memory takes place here. In the case of the rotating workforce scheduling example there is no extra allocation necessary since all allocations can take place in the constructor of CWeek, because the memory size needed is known on object creation.
Line 96 calculates the size of the array of shifts for all employees for a week.
Line 98 copies the memory contents from the pWeek object to this*.
Line 100 copies the fitness value too.
Line 103 defines the Clone() function that creates a new object and returns it.
Line 105 allocates the new object and copies the memory contents by using a copy constructor.
Line 110 defines the Write() function which is only necessary for debugging.
Line 115 defines the Compare() function which compares two week schedules shift by shift. When a difference is found the function terminates with the appropriate return value.
Line 136 defines the Equals function that simply utilises the Compare() function for testing of equity between two week schedules.

A.3 The objective function

The framework's class representing the objective function is called Objective-Function. The problem specific implementation, deriving from it, is named CObjectiveFunction. The user only needs to implement one function called evaluate which returns the fitness of the supplied solution object.

Listing A.3: class CObjectiveFunction

```
1  CObjectiveFunction::CObjectiveFunction(
2          CRequirements *pRequirements)
3   : ObjectiveFunction ()
4  {
5      m_pRequirements = pRequirements;
6  }
7
8  Fitness* CObjectiveFunction::Evaluate (Solution * pSolution)
9  {
10     ObjectiveFunction::Evaluate(pSolution);
11
12     INTLIST::const_iterator IntIter;
13     _FAILURES failures;
14     TSimpleFitness<int> *pValue;
15     int iValue = 0;
16
17     DetectFailures ((CWeek*) pSolution, failures);
18
19     IntIter = failures.SequenceLength.begin ();
20     while (IntIter != failures.SequenceLength.end ()) {
21         iValue += (*IntIter) * 3;
22         IntIter++;
23         if (gl_debug)
```

```
24              cout << "Sequence length error" << endl;
25      }
26
27      IntIter = failures.WorkBlocks.begin ();
28      while (IntIter != failures.WorkBlocks.end ()) {
29      iValue += *IntIter;
30      IntIter++;
31      if (gl_debug)
32          cout << "workblock length error" << endl;
33      }
34
35      IntIter = failures.FreeBlocks.begin ();
36      while (IntIter != failures.FreeBlocks.end ()) {
37      iValue += *IntIter;
38      IntIter++;
39      if (gl_debug)
40          cout << "freeblock length error" << endl;
41      }
42
43      IntIter = failures.NoSequences.begin ();
44      while (IntIter != failures.NoSequences.end ()) {
45      iValue += (*IntIter) * 2;
46      IntIter++;
47      if (gl_debug)
48          cout << "NoSequence error" << endl;
49      }
50
51      pValue = new TSimpleFitness<int>(iValue);
52      SetFitness (pSolution, pValue);
53
54      return pValue;
55  }
56
57  bool CObjectiveFunction::DetectFailuresStartSearch (CWeek * pW,
58      int *piEmpl, int *piDay)
59  {
60      int iHelp1 = 0;
61      int iHelp2 = 0;
62      bool bIsFree = false;
63
64      *piEmpl = 0;
65      *piDay = eMon;
66
67      if (iHelp1 == this->m_pRequirements->GetShiftFree ()
68      && iHelp2 == this->m_pRequirements->GetShiftFree ()) {
69      bIsFree = true;
70      } else if (iHelp1 != this->m_pRequirements->GetShiftFree ()
71              && iHelp2 != this->m_pRequirements->GetShiftFree ()) {
72      bIsFree = false;
73
74      for (*piEmpl = 0; *piEmpl < this->m_pRequirements->GetEmployeeCount ();
75       (*piEmpl)++) {
76      for (*piDay = eMon; *piDay <= eSun; (*piDay)++) {
77          iHelp1 = pW->getShift (*piEmpl, (_eDay) * piDay);
```

```
78          if (bIsFree
79          && iHelp1 != this->m_pRequirements->GetShiftFree ()) {
80          return true;
81          }
82          if (!bIsFree
83          && iHelp1 == this->m_pRequirements->GetShiftFree ()) {
84          return true;
85          }
86      }
87      }
88
89  }
90
91  void CObjectiveFunction::DetectFailures (CWeek * pW, _FAILURES & failures)
92  {
93      int iWorkBlockMin = this->m_pRequirements->GetMinWorkBlocks ();
94      int iWorkBlockMax = this->m_pRequirements->GetMaxWorkBlocks ();
95      int iFreeBlockMin = this->m_pRequirements->GetMinFreeBlocks ();
96      int iFreeBlockMax = this->m_pRequirements->GetMaxFreeBlocks ();
97      const __SHIFT *pShift;
98      _eDay sDay = eMon;
99      int iEmpl = 0;
100     int iStartEmpl = 0;
101     int iStartDay = 0;
102     bool bNextRoundEnd;
103     bool bEnd;
104
105     failures.iFreeBlockCount = 0;
106     failures.iNoSeqCount = 0;
107     failures.iSeqLenCount = 0;
108     failures.iWorkBlockCount = 0;
109     failures.FreeBlocks.clear ();
110     failures.NoSequences.clear ();
111     failures.SequenceLength.clear ();
112     failures.WorkBlocks.clear ();
113
114     DetectFailuresStartSearch (pW, &iStartEmpl, &iStartDay);
115     iShift = pW->getShift (iStartEmpl, (_eDay) iStartDay);
116
117     iWorkcount = 0;
118     iFreecount = 0;
119     iShiftCount = 0;
120
121     iEmpl = iStartEmpl;
122     sDay = (_eDay) iStartDay;
123     pW->prevShift (&iEmpl, &sDay);
124     iLastShift = pW->getShift (iEmpl, sDay);
125     iEmpl = iStartEmpl;
126     sDay = (_eDay) iStartDay;
127
128     bNextRoundEnd = false;
129     bEnd = false;
130     do {
131     if (m_pRequirements->NoSequenceFind (iLastShift, iShift)) {
```

```
132         failures.iNoSeqCount++;
133         failures.NoSequences.insert (failures.NoSequences.end (), 1);
134     }
135         if (iLastShift != m_pRequirements->GetShiftFree ()) {
136         pShift = m_pRequirements->ShiftFind (iLastShift);
137         if (iShiftCount == 0) {
138         } else if (iShiftCount > pShift->SequenceLength.iMax) {
139             failures.iSeqLenCount++;
140             failures.SequenceLength.insert (failures.
141                             SequenceLength.end (),
142                             iShiftCount -
143                             pShift->SequenceLength.
144                             iMax);
145         } else if (iShiftCount < pShift->SequenceLength.iMin) {
146             failures.iSeqLenCount++;
147             failures.SequenceLength.insert (failures.
148                             SequenceLength.end (),
149                             pShift->SequenceLength.
150                             iMin - iShiftCount);
151         }
152         }
153         iShiftCount = 0;
154     }
155     if (iShift != m_pRequirements->GetShiftFree ()) {
156             failures.iFreeBlockCount++;
157             failures.FreeBlocks.insert (failures.FreeBlocks.end (),
158                         iFreeBlockMin -
159                         iFreecount);
160             failures.iFreeBlockCount++;
161             failures.FreeBlocks.insert (failures.FreeBlocks.end (),
162                         iFreecount -
163                         iFreeBlockMax);
164         }
165         }
166         iWorkcount++;
167     } else {
168             failures.iWorkBlockCount++;
169             failures.WorkBlocks.insert (failures.WorkBlocks.end (),
170                         iWorkBlockMin -
171                         iWorkcount);
172             failures.iWorkBlockCount++;
173             failures.WorkBlocks.insert (failures.WorkBlocks.end (),
174                         iWorkcount -
175                         iWorkBlockMax);
176         }
177         }
178         iFreecount++;
179     }
180     iLastShift = iShift;
181     iShiftCount++;
182     pW->nextShift (&iEmpl, &sDay);
183     iShift = pW->getShift (iEmpl, sDay);
184     if (bNextRoundEnd)
185         bEnd = true;
```

```
186     if (iEmpl == iStartEmpl && iStartDay == sDay)
187         bNextRoundEnd = true;
188     }
189     while (!bEnd);
190 }
```

Line 1-6 defines the constructor which stores a pointer to the requirements object.
Line 8 defines the evaluation function which differs between the 4 different types of violations. Each violation is stored accordingly with its weight value. The complete fitness is calculated by summing up all violations multiplied by the according weigth.
Line 10 calls the evaluate of the parent class.
Line 17 does the actual failure detection (which will be explained below).
Line 19-49 creates a sum from the different failure types and counts with predefined weights for each failure type.
Line 51-52 sets the new fitness value for the solution.
Line 57-89 contains the function DetectFailuresStartSearch which is a helper function that searches for the first occurance of a change from shift to day off shift or day off shift to shift. This information is needed by the DetectFailures() function below.
Line 92-112 initialize helper variables.
Line 114 retrieves the starting point for the evaluation in the schedule matrix.
Line 117-119 initialises another set of variables
Line 121-126 preload the previous shift for the starting shift, since this information is needed insinde the following loop
Line 130-189 steps through the complete schedule and detects any violation.
Line 137-152 recognizes minimum and maximum shift block violations
Line 155-166 recognizes free shift block violations
Line 167-177 recognizes workblock length violations
Line 180-188 are responsible for loading the next shift and deciding about the termination of the loop.

A.3.1 Neighbourhood generation

For generating neighbourhoods the framework basically provides 3 classes. Actually there are 5, but for writing a simple Tabu Search algorithm they aren't necessary.

The basic part is the Move class. It's purpose is to store a single move operation and to apply it to a solution. Since it must know how a solution looks like in order to have the ability to modify it, it has to be derived by a user's class.

The second part is the MoveIterator class. It creates a neighbourhood given a solution as a start point. Then it can be used to step through the neighbourhood. Again this class is coupled tightly with the representation of a solution and must be derived by a user's class.

The third and last one is the MoveManager which is a container class for MoveIterator objects. This class normally provides everything that it needs to, so there is no need to create a class that derives from it.

A.4 Move

The Move class has 4 abstract functions named createInverse, applyOn, compare and clone. Therefore those have to be implemented by deriving classes.

In the case of rotating workforce scheduling a simple move to define is the swapping of the shifts of two employees within one day. This sounds quite complex, but in truth it's quite easy. Just think of picking one day and then swapping the shifts of two employees. The benefit of this move definition is that it doesn't violate the shift requirements of a day.

Listing A.4: class CSwapShiftMove

```
1  CSwapShiftMove::CSwapShiftMove (
2      _eDay Day, int iShift1, int iShift2, int iLength)
3      : Move()
4  {
5      m_eDay = Day;
6      m_iShift1 = iShift1;
7      m_iShift2 = iShift2;
8      m_iLength = iLength;
9  }
10
11 Object *
12 CSwapShiftMove::CreateInverse () const
13 {
14     return (Move*)(new CSwapShiftMove(m_eDay,
15                          m_iShift2, m_iShift1, m_iLength));
16 }
17
18 bool CSwapShiftMove::ApplyOn (Solution *solution) const
19 {
20     _eDay HelpDay;
21     CWeek *pWeek = (CWeek*) solution;
22
23     HelpDay = m_eDay;
24     for (iLen=0; iLen < m_iLength; iLen++) {
25         iShift1val = pWeek->getShift(m_iShift1, HelpDay);
26         iShift2val = pWeek->getShift(m_iShift2, HelpDay);
27
28         pWeek->setShift(m_iShift1, HelpDay, iShift2val);
29         pWeek->setShift(m_iShift2, HelpDay, iShift1val);
30         pWeek->nextShift(&iUseless,&HelpDay);
31     }
32
33     return bResult;
34 }
35
36 int CSwapShiftMove::Compare (const Object *move) const
37 {
38     const CSwapShiftMove *castmove;
39
40     castmove = dynamic_cast<const CSwapShiftMove *> (move);
41     if (m_eDay < castmove->m_eDay)
42     return -1;
```

```
43      else if (m_eDay > castmove->m_eDay)
44      return 1;
45
46      if (m_iShift1 == castmove->m_iShift2 &&
47              m_iShift2 == castmove->m_iShift1)
48          return 0;
49
50      if (m_iShift1 < castmove->m_iShift1)
51      return -1;
52      else if (m_iShift1 > castmove->m_iShift1)
53      return 1;
54
55      if (m_iShift2 < castmove->m_iShift2)
56      return -1;
57      else if (m_iShift2 > castmove->m_iShift2)
58      return 1;
59
60      if (m_iLength < castmove->m_iLength)
61      return -1;
62      else if (m_iLength > castmove->m_iLength)
63      return 1;
64
65      return 0;
66  }
67
68  bool CSwapShiftMove::Equals(const Object *pObj) const {
69      return Compare(pObj) ? true : false;
70  }
71
72  Object* CSwapShiftMove::Clone () const
73  {
74      return new CSwapShiftMove (m_eDay, m_iShift1,
75                                 m_iShift2, m_iLength);
76  }
77
78  void CSwapShiftMove::DeepCopy (const Object* pObj) {
79      const CSwapShiftMove *pMove =
80          dynamic_cast<const CSwapShiftMove *> (pObj);
81
82      m_eDay = pMove->m_eDay;
83      m_iShift1 = pMove->m_iShift1;
84      m_iShift2 = pMove->m_iShift2;
85      m_iLength = pMove->m_iLength;
86  }
87
88  ostream& operator << (ostream &stream, CSwapShiftMove move)
89  {
90      cout << "Day: " << move.m_eDay << " Shift1: "
91          << move.m_iShift1 << " Shift2: " << move.m_iShift2
92          << " Length: " << move.m_iLength;
93
94      return (stream);
95  }
96
```

```
97 void CSwapShiftMove::Write(std::ostream &stream) const
98 {
99     stream << *this << endl;
100 }
```

Line 1-9 show the constructor which initializes internal variables with the given parameters. Line 11-16 defines the inverse of the move which is simply the swapping of the shifts. Actually in this case the move's inverse is a special case, since you do not necessarily need it, because the move can reverse itself. This is caused by the move's definition. Since it swaps the shifts of two employees, it can be undone by swapping the same shifts again. It doesn't matter in which order the two employees are stored and therefore the actual inverse is not really necessary.
Line 18-34 define the function for modifying solution objects. It swaps the shifts and returns true, when those shifts aren't the same. When both shift types are identical it returns false and does nothing, since this sort of swap doesn't change anything (e.g. swapping a night shift with a night shift).
Line 36-66 defines the comparison of two Moves, that return -1, 0 or 1 if the move object compared to is bigger, equal or smaller.
Line 68-70 define a function testing for equity between two Move objects. When they are equal, true is returned, otherwise false
Line 72-76 defines a clone function which allocates a new object of type Move.
Line 78-86 defines DeepCopy() by loading the data from the given Move object.
Line 88-100 contain debug output functions.

A.5 CSwapShiftMoveIterator

The CSwapShiftMoveIterator creates the neighbourhood of a solution by allocating new CSwapShiftMove objects. In order to know how to initialize those CSwapShiftMove object, it remembers the day and the two shifts beeing swapped by the last CSwapShiftMove object.

Listing A.5: class CWeek

```
1  CSwapShiftMoveIterator::CSwapShiftMoveIterator(
2          CRequirements *pRequirements, int iMaxLength)
3   : MoveIterator ()
4  {
5      m_pRequirements = pRequirements;
6      m_pMove = new CSwapShiftMove(eMon, 0, 0, 1);
7      begin(NULL);
8      m_iMaxLength = iMaxLength;
9  }
10
11 CSwapShiftMoveIterator::~CSwapShiftMoveIterator ()
12 {
13 }
14
15 TS::Move::Move *
16 CSwapShiftMoveIterator::begin (Solution *pSolution)
17 {
```

```
18      m_pSolution = pSolution;
19      m_eDay = eMon;
20      m_iShift1 = 0;
21      m_iShift2 = 1;
22      m_iLength = 1;
23
24      return getMove();
25  }
26
27  TS::Move::Move *
28  CSwapShiftMoveIterator::end (Solution *pSolution)
29  {
30      m_pSolution = pSolution;
31      m_eDay = eSun;
32      m_iShift1 = m_pRequirements->GetEmployeeCount () - 2;
33      m_iShift2 = m_pRequirements->GetEmployeeCount () - 1;
34      m_iLength = m_iMaxLength;
35
36      return getMove();
37  }
38
39  inline bool CSwapShiftMoveIterator::setNext ()
40  {
41      m_iShift2++;
42      if (m_iShift2 >= m_pRequirements->GetEmployeeCount ()) {
43          m_iShift1++;
44          m_iShift2 = m_iShift1 + 1;
45          if (m_iShift1 >= m_pRequirements->GetEmployeeCount () - 1) {
46                  if (m_iLength > m_iMaxLength)
47                      return false;
48                  m_iLength++;
49                  m_eDay = eMon;
50              } else {
51                  m_eDay = (_eDay)((int)m_eDay + 1);
52              }
53              m_iShift1 = 0;
54              m_iShift2 = 1;
55          }
56      }
57      return true;
58  }
59
60  TS::Move::Move *CSwapShiftMoveIterator::next ()
61  {
62      if (setNext())
63          return getMove();
64      return NULL;
65  }
66
67  inline bool CSwapShiftMoveIterator::setPrev ()
```

```
68 {
69         m_iShift1 = m_iShift2 - 1;
70                     return true;
71                 m_iLength--;
72                 m_eDay = eSun;
73             } else {
74                 m_eDay = (_eDay)((int)m_eDay - 1);
75             }
76             m_iShift1 = m_pRequirements->GetEmployeeCount () - 2;
77             m_iShift2 = m_pRequirements->GetEmployeeCount () - 1;
78         }
79     }
80     return false;
81 }
82
83 TS::Move::Move *CSwapShiftMoveIterator::prev ()
84 {
85     if (setPrev())
86         return getMove();
87     return NULL;
88 }
89
90 inline
91 TS::Move::Move* CSwapShiftMoveIterator::getMove ()
92 {
93     m_pMove->setDay (m_eDay);
94     m_pMove->setShift1 (m_iShift1);
95     m_pMove->setShift2 (m_iShift2);
96     m_pMove->setLength (m_iLength);
97
98     return m_pMove;
99 }
```

Line 1-9 define the constructor initialising the iterator with Mondays first two shifts.

Line 18-26 define the function that sets the iterator to the beginning of the neighbourhood. Since the iterator is for the CSwapShiftMove, it starts at Monday with the first two shifts.

Line 27-37 show the function for setting the iterator to the end of the neighbourhood. Since the beginning is the start of the week, the end of the neighbourhood is Sunday's last two shifts which is actually the end of the week's schedule.

Line 39-58 define a helper function for advancing to the next move. Since the move stores a day and two employees who's shift get swapped, there are three values to take care of. The second employee is set to the next employee until there aren't any more employees. Then the first employee is set to it's next employee and the second employee is the new first employee's next one. If the first employee is set to the last employee the second employee would be set to a non existing one. Therefore the whole loop is started over with on the next day. When the current day is Sunday and the first employee is the last employee on that day, then there are no further moves and false is returned.

Line 60-65 show the actual accessible next() function that first tries to advance to the next move. If it's successful, then the next move is returned, otherwise no next move exists (since the iterator is already at the end of the neighbourhood) and therefore NULL is returned.
Line 67-81 calculates the previous CSwapShiftMove object by a similair technique like the setNext() function.
Line 83-88 show the prev() function which steps back to the previous move operator.
Line 90-99 define getMove which returns the current CSwapShiftMove object.

Appendix B

Installation

The framework was developed using Linux and is currently limited to the application on Linux.

The source code is to be found on the supplied CD. In the root directory a file named framework.tgz is stored. Following commands are necessary to extract the source code from the compressed archive:

```
tar -xzf framework.tgz
```

That will create a directory name fw containing the source code of the framework and the source code of the presented example.

In order to build the framework and the example program automake, autoconf and libtool have to be installed on the unix system. Additionally a c++ compiler is needed. Then following command can be issued in the directory fw:

```
./bootstrap --make
```

That will create a library for the framework and an executeable for the example program.

The source code of the framework can be found in the subdirectory src. Dokumentation to the source code of the framework can be found in the subdirectory doc. Any example programs can be found in the subdirectory examples. Each example program has got its own directory. The problem definition files utilised by the example programs are to be found in the subdirectory data. An other directory is m4, which is only utilised by automake and not of high interest.

Appendix C

Execution

After installation and compilation, the example program can be execute by typing the following command in the fw directory:

```
examples/bsp2/example -r
```

Pseudo random sets of numbers can be created on linux utilizing libc functions. The pseudo number generator can be initialized with a so called seed that allows it to reproduce a set of pseudo random numbers. This feature is quite useful for recreating initial workforce schedules. Therefore the program has an interactive capability where you can set the initial seed. The following execution enables that feature with a command line argument:

```
examples/bsp2/example -i
```

Then the program asks whether it should take the last known seed. Pressing any key and the return key takes the last known seed, while pressing 'n' and the return key leads to the next question. That one asks whether it should use the presented random seed or not. Again pressing any key and enter accepts the random seed, otherwise an input prompt for a seed is shown where one can enter a seed value. After entering an integer value and pressing enter, the value is used as a new seed. The following lines present such an interactive input.

```
examples/bsp2/example -i
Oldseed : 134554050 [Y/n]: n
Random Seed: 6720 [Y/n]: n
Seed: 12345
```

Whenever the program terminates, it prints its last solution and some data to the screen like:

```
Seed used: 12345
Best Solution found:
Employee | Mon | Tue | Wed | Thu | Fri | Sat | Sun Fitness: 0
Emp0     | -   | A   | A   | N   | -   | A   | A
Emp1     | D   | D   | N   | -   | D   | D   | D
Emp2     | N   | -   | D   | D   | N   | -   | A
Emp3     | A   | N   | -   | A   | A   | N   | -
Runtime: Seconds: 0 Millisecs: 32388
Iterations: 48
```

```
Evaluations: 6503
SHistSize: 45
SHistCount: 36
```

First the seed is shown, then the best solution found and it's fitness value. Afterwards the runtime is printed and the number of iterations and evaluations. The last two values concern the search history, which is 45 entries long and accommodates 36 entries.

Bibliography

[1] G. Ausiello, P. Crescenzi, G. Gambosi, A. Marchetti-Spaccamela V. Kann, and M. Protasi. *Complexity and Approximation Combinatorial optimization problems and their approximability properties*. Springer Verlag, 1999.

[2] R. Battiti and G. Tecchiolli. The reactive tabu search. *OSRA Journal on Computing*, 6(2):126–140, 1994.

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993.

[4] Th. Scheidl Ch. Breitschopf, G. Blaschek. Optlets: A generic framework for solving arbitrary optimization problems. *WSEAS Transactions on Information Science and Applications*, 2(5), 2005.

[5] Michel de Champlain and Brian G. Patrick. *C#: Practical Guide for Programmers*. Morgan Kaufmann Publishers, 2005.

[6] Lucia di Gaspero and Andrea Schaerf. Easylocal++: An object-oriented framework for flexible design of local search algorithms. Technical Report UDMI/13/2000/RR, Università di Udine, Udine, Italy, 2000.

[7] Lucia di Gaspero and Andrea Schaerf. Easylocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software: Practice and Experience*, 36(10):733–765, 2003.

[8] Lucia di Gaspero and Andrea Schaerf. Easylocal++: An object-oriented framework for flexible design of local search algorithms and metaheuristics. In *In Proc. of the 4th Metaheuristic International Conference (MIC2001)*, pages 287 – 292, Porto, Portugal, July 16-20, 2001.

[9] Raphael Dorne and Christos Voudouris. *HSF: the iOpt's framework to easily design metaheuristic methods*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[10] Mohamed E.Fayad, Douglas C.Schmidt, and Ralph E. Johnson. *Building Application Frameworks*. Wiley Computer Publishing, 1999.

[11] Ralph Johson Erich Gamma, Richard Helm and John Vlissides. *Desirgn Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] A. Fink and S. Voß. Hotframe - heuristische lösung diskreter planungsprobleme mittels wiederverwendbarer software-komponenten. *OR News*, 4:18–24, 1998.

[13] Ralph Johnson Gamma Erich, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[14] F. Glover and M. Laguna. *Tabu Search. Readings*. Kluwer Academic Publishers, 1997.

[15] Fred Glover. Tabu search–part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[16] Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[17] Fred Glover and Eric Taillard. A user's guide to tabu search. 41(1):1–28, 1993.

[18] Robert Harder. Opents. Internet, 2001. http://www.coin-or.org/Presentations/OpenTS.ppt.

[19] Robert Harder. Opents. Internet, 2001. http://www.coin-or.org/OpenTS/.

[20] N. Heller, J. McEwen, and W. Stenzel. Computerized scheduling of police manpower. *St. Louis Police Department, St. Louis, MO*, 1973.

[21] Martin Stuaert Jones. An object-oriented framework for the implementation of search techniques. Master's thesis, University of East Anglia, Norwich, UK, 2000.

[22] P. Van Hentenryck L. Michel. Localizer++: An open library for local search. Reseach Report CS02-01, Brown University, 2001.

[23] PA Laplante. *Dictionary of Computer Science: engineering, and technology*. CRC Press Boca Raton, 2001.

[24] G. Laporte. The art and science of designing rotating schedules. *Journal of the Operational Research Society*, 50:1011–1017, 1999.

[25] Hoong Chuin Lau, Wee Chong Wan, and Xiaomin Jia. A generic object-oriented tabu search framework. In *Proceedings of MIC'2003 - 5th Metaheuristics International Conference*, 2003.

[26] Jesse Liberty. *Programming C#, 3rd Edition*. O'Reilly, 2003.

[27] Nysret Musliu Michael Mörz. Genetic algorithm for rotating workforce scheduling. In *In Proc. of the 2nd IEEE Internation Conference on Computational Cybernetics*, pages 121 – 126, Vienna, Austria, 2004.

[28] Z. Michalewicz and B. F. Fogel. *How to solve it: modern heuristics*. Springer-Verlag, 2000.

[29] Nysret Musliu. Heuristic methods for automatic rotating workforce scheduling. *International Journal of Computational Intelligence Research*, to appear, 2006.

[30] Nysret Musliu, Johannes Gärtner, and Wolfgang Slany. Efficient generation of rotating workforce schedules. *Discrete Applied Mathematics*, 118(1-2):85–98, 2002.

[31] Bertrand Neveu and Gilles Trombettoni. Incop: An open library for incomplete combinatorial optimization. In *Dans Proc. International Conference on Principles Constraint Programming, CP'03*, volume 2833 of LNCS, pages 909–913, Kinsale,Ireland, 2003.

[32] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. Halsted Pr., 1993.

[33] Joseph Schmuller. *SAMS Teach Yourself UML in 24 Hours*. Sams Publishing, 800 East 96th Street, Indianapolis, Indiana, 46240 USA, 2004.

[34] E. Taillard. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–445, 1991.

[35] K. Varrentrapp. Gails: Guided adaptive iterated local search - method and framework. Technical Report AIDA-04-05, FG Intellektik, FB Informatik, TU Darmstadt, 2004.

[36] Vijay Vazirani. *Approximation Algorithms*. Berlin ; New York : Springer, 2001.