Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Håvard Roalsø Eiring

Presenting The Marble Library: Separating the Concerns of Tightly Coupled Abstractions

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

NTNU
Norwegian University of
Science and Technology

Håvard Roalsø Eiring

Presenting The Marble Library: Separating the Concerns of Tightly Coupled Abstractions

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

1 Preface

This thesis is submitted as a requirement for graduating with a Master's degree in Cybernetics and Robotics from the Norwegian University of Science and Technology (NTNU), Department of Engineering Cybernetics. Associate Professor Sverre Hendseth has been the supervisor of the work presented in this thesis. Central ideas motivating the development of software presented in this thesis stems from the thesis supervisor's work on the *Marble modelling language*. This work is presented in [3] and documented in [2]. These documents are included in the zip folder which is submitted along with the thesis (see Appendix A). The contents of these documents has also inspired some of the features of the software library developed in this thesis. The source material is cited where said features are presented.

Finally, it should be mentioned that the work presented in this thesis builds on parts of the work presented in [1], a specialization project where some of the fundamental functionality for the software developed in this thesis was developed. This work isn't published. Therefore, the specialization project report is included in the files submitted along with this thesis. The features of the software developed in the specialization project is described in Section 5.4. The same section also elaborates on the extent to which the software developed in the specialization project was reused in the software developed in this thesis.

2 Abstract

Separation of concerns is a highly effective software design principle which when applied, results in flexible and tidy software with high degrees of freedom. Unfortunately, there are contexts where this design principle cannot be applied. Recursive, hierarchical structures, implemented as object trees, represent an intuitive approach which one can use in many contexts. Their generative power make them very useful in e.g. the field of procedural generation. However, their power is severely restricted by the fact that several trees addressing separate concerns, are sooner or later forced to be integrated together. In other words, the separation of concerns principle cannot be applied. Therefore, software developers are forced to cram the trees together in some unsatisfactory ad hoc solution, resulting in unmaintainable software of poor quality.

In an effort to resolve this issue, the purpose of this thesis was to develop the *Marble* library. This software library was to support creation of object trees and facilitate the necessary tight interactions between them while the trees are kept separated from each other, effectively applying the separations of concerns design principle. Two library applications were also to be developed, in order to demonstrate the features of the library.

Equipped with an extensive set of features capable of creating a wide variety of object trees, the resulting library showed great versatility and ability to adapt to a wide range of contexts. In addition, the applications that were developed (the *Marbleviz* and the *Random Lyrics Generating Melody* applications), demonstrated how the Marble library can be used to let recursive, hierarchical structures interact tightly while still being kept separate from each other, effectively applying the separations of concern design principle. Utilizing the library's ability to keep sections addressing separate concerns independent of each other, the latter application also demonstrated how the library could be used to develop procedural generation methods.

3 Sammendrag

Separasjon av ansvarsområder er et meget effektivt design-prinsipp innen programvareutvikling. Bruken av dette prinsipper fører til fleksibel og ryddig programvare av stor frihetsgrad. Rekursive, hierarkiske strukturer, implementert som objekttrær, representerer en intuitiv framgangsmåte man kan bruke i mange sammenhenger. Deres generative egenskaper gjør at de egner seg meget godt innen for eksempel fagfeltet om prosedyrisk generering. Uheldigvis begrenses potensialet deres kraftig av at trær med hver sine distinkte ansvarsområder før eller siden tvinges til å bli integrert sammen. Prinsippet om separasjon av ansvarsområder kan dermed ikke anvendes. Derfor blir programvareutviklere tvunget til å klemme de distinke trærne sammen i form av en utilfredsstillende ad hoc løsning. Resultatet er dermed dårlig strukturert programvare som er meget krevende å vedlikeholde.

For å løse dette problemet var målet med denne masteroppgvaen å utvikle *Marble*-biblioteket. Dette programvarebiblioteket skulle støtte generering av trær og fasilitere tette interaksjoner mellom dem mens de holdes separert fra hverandre. Dermed anvendes nemlig prinsippet om separasjon av ansvarsområder. To applikasjoner av biblioteket skulle også utvikles for å demonstrere bibliotekets evner og mulige bruksområder.

Utrustet med at bredt spekter av verktøy i stand til å generere trær i mange ulike arter, endte biblioteket opp med å vise stor grad av allsidighet og evne til å tilpasse seg mange sammenhenger. Applikasjonene som ble utviklet (*Marbleviz-* og *Tilfeldig Tekst Som Genererer Melodi*-applikasjonen), demonstrerte Marble-bibliotekets evne til å la rekursive, hierarkiske strukturer interagere tett med hverandre mens de fortsatt holdes separert fra hverandre. Dermed ble prinsippet om separasjon av ansvarsområder anvendt på et vellykket vis. Ved å utnytte bibliotekets evne til å holde seksjoner som omhandler hver sine ansvarsområder separert, demonstrerte også den sistnevnte applikasjonen hvordan biblioteket kan brukes til å utvikle metoder innen prosedyrisk generering.

Table of Contents

List of Figures

Listings

4 Introduction

4.1 Thesis Description

The purpose of this thesis is to develop the *Marble* library, a software library written in the C++ programming language that aims to enable the highly valuable separation of concerns design principle to a specific set of software development contexts where it currently is impossible. These contexts involve *recursive, hierarchical structures*, or object trees, that are so tightly coupled that applying the separation of concerns principle is impossible.

Two applications of the library is also to be developed. The purpose of these applications is to present the library's capabilities and how they can be used in a larger scale. The first one will enable trees to be drawn by Graphviz (graph visualization software). The second one will generate a random lyrics tree which will itself generate a melody tree. Related to the purpose of the library, the fact that the trees comprising the applications are kept separate will be emphasized.

4.2 Motivation and Context

4.2.1 Separation of Concerns

Separation of concerns is software design principle where the sections which comprise a computer program are kept separate from each other. The reasons for this is that these sections usually address their own distinct concerns, so keeping them separated from each other is a natural approach. However, the most significant benefit this design principle provides is that the software becomes tidier, more flexible and more easily maintainable. This is caused by the fact that the sections can more easily be modified and extended when they're independent of the other sections.

Unfortunately, there are cases where the separation of concerns principle cannot be applied. The purpose of this thesis is therefore to develop a software library which enables the application of this design principle. However, the goal of the library is to enable the application of this principle only for a subset of the cases where it cannot be done. In this subset of software development contexts, the goal is to develop a program or an application which consists of several tightly interacting sections. The sections are addressing their own distinct concerns, which is partly why it would be beneficial to keep them separated. In this subset of contexts, the sections are also represented as *recursive, hierarchical structures* (representations which are elaborated on in the following section), concretely implemented as object trees.

The reason that the separation of concerns principle can't be applied in this context, is that these trees are so severely dependant on each other, that keeping them separated is impossible. That is, the objects in the trees are so tightly coupled to objects in other trees that one typically has no other choice than to cram the trees together in some unsatisfactory ad hoc solution (the structures of the trees might

be so different that cramming them together won't even be possible) [3] (p. 7). The result is a single tree comprised of a cluster of rigid relationships between objects. In other words, the software becomes messy and unmaintainable.

A concrete example might shed some light on the issue at hand. Imagine that a mathematician and a software developer decide to develop a simulator together. The mathematician is responsible for providing the differential equations, while the software developer is responsible for optimizing the equations set before the code is executed. In this case, the mathematics and the optimization are the distinct sections in cooperation. The executable simulator code is generated as a result of the mathematician and the software developer writing code together. That is, the software developer optimizes the code by making changes to the equations written by the mathematician. This means that whenever the mathematician makes any changes, the software developer has to find out if they need to make any additional changes themselves and make sure everything still works. Because of these tight dependencies between the two sections, this practice has the potential to become a cumbersome process (this example is inspired by [3](p. 10)).

This motivates the development of a software library which allows the mathematician to write their equations and the software developer to write their optimization specifications separately, while still keeping loose coupling between them such that the simulator code can be generated. This way, the mathematician could make changes to their code while the library ensures that the optimization specifications are applied to the code automatically. In other words, the software developer won't have examine every change made by the mathematician to see if everything is still optimized sufficiently. The result is software of significantly higher quality, avoiding the need for some unsatisfactory ad hoc solution where the sections are blended together.

4.2.2 Recursive, Hierarchical Structures

As previously mentioned, the goal for the library developed in this thesis is to enable separation of concerns in applications comprised of several *recursive, hierarchical structures*. These structures are common in software design because the idea of forming bigger components by joining smaller ones is an approach that makes sense in many contexts. Such compositions of structures also imply that they have a hierarchical property. In other words, the importance or rank of a set of subcomponents are lower than the one of the supercomponent they form. Furthermore, this thesis focuses on hierarchical structures that have an additional recursive property. That is to say, the structures expand by creating subcomponents for themselves. In other words, the top component creates its subcomponents, which all create their own subcomponents, and so on.

Now, why are such structures so common in software design? As already mentioned, the idea of recursive expansion and subcomponents forming supercomponents is an intuitive and simple solution to many problems. The call stack of a computer program is a convenient example. The stack expands recursively as a function calls another, satisfying the recursive property. In addition, a hierarchical relationship is

established between a function and the function it calls, since the calling function is responsible for the subfunction. However, recursive, hierarchical structures are common in many other domains. The generation of melody is a recurring context in this thesis. A melody often consists of lines, consisting of bars, which consist of notes, forming a hierarchical structure. In addition, when generating a melody, it's convenient to make the lines generate their bars and the bars generate their notes, recursive expansion in other words (this example is inspired by [2](p. 7-8)). Note that in the software presented in this thesis, these recursive, hierarchical structures will be implemented as object trees. Mentions of recursive, hierarchical structures and trees will therefore be interchangeable in this thesis.

4.2.3 Procedural Generation: Lyrics Generating Melody

The generative power of recursive, hierarchical structures has great potential. However, their power is severely constrained by the software design related issues discussed earlier in this section. Unfortunately, they are prone to suffer from clusters of unmaintainable dependencies that occur when one is forced to integrate separate structures that don't fit each other well [3](p. 7). To elaborate upon these difficulties, consider the applicability of recursive, hierarchical structures in procedural generation (algorithmic generation of data, see Section 5.3). Recursive, hierarchical structures have great potential in this field. The generative power their recursive and hierarchical properties facilitate are well suited for generating game world content like landscapes, creatures and music [3](p. 6). Nevertheless, one will quickly encounter the previously discussed design related constraints.

A concrete example might shed some light on the difficulties one encounters when integrating trees that don't fit each other well. Imagine the case where one is tasked with using procedural generation to generate music for a game world. The music should include both lyrics and melody. To make sure the lyrics and melody fits each other, a wise approach would be to generate the lyrics first, then let the lyrics generate the melody. However, to generate coherent music, there are a significant amount of considerations to make. For instance, the amount of lyric lines should match the amount of melody lines and the amount of syllables in a lyric line should match the amount of notes in a melody line. One might also want to ensure that interesting rhythms are generated, which is one of many other considerations to make when generating a melody [2](p. 20-21). Keeping a lyrics tree and a melody tree separate would naturally be a tidier and more easily maintainable solution. However, since the generation of the melody is so closely tied to the lyrics tree, the only remaining option is to somehow build the melody tree into the lyrics tree, even though their hierarchical structures are likely not to fit each other well. In order to keep the trees separate, there is need for a software library which enables the lyrics tree to somehow gather all the specifications it generates for the melody tree, before these specifications are used to generate the melody tree separately from the lyrics tree.

As Section 5.3 discusses, enabling *Multi-Level, Multi-Content PCG* is one of the central challenges within procedural generation. However, if one would've been able to keep the all the necessary specifications for content generation as separate trees,

it could possibly ease these difficulties. As we've just seen, one could use a lyrics tree (and possibly other trees) as the specification for melody generation. If a story should be generated, rules for what the story should be about could be kept as a separate specification tree, along with e.g. a specification tree which determines story length, structure, language, etc.

5 Background Study

5.1 C++

As C++ is chosen as the programming language used to develop the Marble library, a study of selected C++ features has been made.

5.1.1 Constructors

Instances of user created classes are given their first values (initialized) by constructors. That being said, there are different types of constructors. A default constructor is a constructor which either takes no arguments or has default arguments. A copy constructor is a constructor which initializes a new object with the values belonging to another already existing object. A copy assignment operator on the other hand, assigns the values of one existing object to another already existing object of the same class. Listing 1 demonstrates how the different constructors for a class is declared and how they are used regarding initializing of and assignment to objects. The copy constructor has special importance because it's called when an object is passed by value [4](p. 4-6).

```
1  class Marble {
2      public:
3          Marble(); // default constructor
4          Marble(const Marble& rhs); // copy constructor
5          Marble& operator=(const Marble& rhs); // copy assignment
               operator
6  };
7
8  Marble m1; // the default constructor is called
9
10 Marble m2(m1); // the copy constructor is called
11
12 m1 = m2; // the copy assignment operator is called.
```

Listing 1: Different types of constructors in C++

5.1.2 Always Initialize Objects Before They're Used

When declaring an object in C++ (that is, without initializing it), the object's default constructor will in most cases be called. The object is then initialized to the some deafult value, e.g. an empty string for a string object from STL. However, this is not always the case. That is, a default constructor won't always be called following the declaration of an object. The rules for whether a default constructor should be called or not are quite comprehensive and could be considered too complicated for a programmer to memorize. Some of the built-in types from the C-part of C++ like arrays for instance, are not guaranteed to be automatically initialized when declared. For this reason, always initialing objects before they're used is considered a good policy to follow. Failing to follow this policy could lead to situations where

one tries to read an uninitialized object. This results in undefined behaviour, which one for obvious reasons should avoid at all costs [4](p. 26-33).

For user-created types, the constructor is responsible for initializing the member variables. Therefore, when writing a class' constructor, one should be aware of the difference between assignment and initialization of member variables. One might think that the member variables of the Marble class is being initialized in lines 10-11 in Listing 2. However, the member variables have already been initialized by their default constructors (or at least the ones which are guaranteed to be initialized) before the body of the constructor is executed. After the initialization, they are then assigned new values in lines 10-11. Therefore, the calls to the default constructors are wasted. In stead, one should use the member initialization list, as demonstrated in Listing 3. Here, the member variables are initialized using their copy constructors [4](p. 26-33).

```
1  class Marble {
2  private:
3      Marble* pParent;
4      std::string typeID;
5  public:
6      Marble(Marble* pParent, std::string typeID);
7  }
8
9  Marble::Marble(Marble* pParent, std::string typeID) {
10     this->pParent = pParent;
11     this->typeID = typeID;
12 }
```

Listing 2: Assignment of member variables in the constructor body

```
1  Marble::Marble(Marble* pParent, std::string typeID)
2      : pParent(pParent), typeID(typeID) {}
```

Listing 3: Demonstration of the member initialization list

5.1.3 Explicit Keyword

The *Explicit* specifier makes sure the constructor of a class can't perform any implicit type conversions, only explicit ones. An implicit type conversion may occur when e.g. a function which expects an input of some specific type is passed an input object of some other type. As a consequence, the input object may be, if possible, converted to the expected type. Declaring a class' constructor as explicit makes sure unexpected and unwanted behaviour caused by implicit type conversions are avoided. Therefore, declaring constructors as explicit is considered by many as a policy which is wise to follow, unless of course implicit type conversions are wanted [4](p. 5).

5.1.4 Compilers Generating Constructors and Destructors Implicitly

If one doesn't declare a copy constructor, a copy assignment operator and a destructor for a class, compilers will generate them implicitly. A default constructor will also be generated if no other constructors are declared. These constructors and the destructor will only be generated if there's a need for them, though. E.g. if the copy assignment operator is never used, the compiler won't generate the copy assignment operator. Also worth mentioning is that the compilers will generate the potential constructors and destructors as public and inline [4](p. 34-37).

Now, what exactly does the compiler generated constructors and destructors do? The default constructors and destructors simply invoke the constructors and destructors of non-static member variables and of potential classes said class inherits from. Regarding the copy constructor and the copy assignment operator, every non-static member variable is simply copied from the original object to the other object [4](p. 34-37).

However, compilers only generate copy assignment operators when the resulting code is legal. Consider the following example (inspired by [4](p. 36):

```
1  class Marble {
2  private:
3      std::string& typeID;
4  public:
5      Marble(std::string& tID) : typeID(tID) {}
6  };
7
8  std::string system = "System";
9  std::string branch = "Branch";
10
11 Marble m1(system);
12 Marble m2(branch);
13
14 m1 = m2;
```

Listing 4: An example of when the compiler can't generate a copy assignment operator

Since no copy assignment operator is declared for the class, and there is a need for one since the "="-operator is used in the last line, the compiler would have to generate one. However, the resulting code would be illegal. This is caused by the fact that the *typeID* member variable is a reference to a string, which means that the value of the *m2* object's typeID member variable can't be assigned to m1's typeID member variable. In cases like these, the code won't compile and one would have to declare and define a copy assignment operator and find a way to avoid such illegal pieces of code [4](p. 34-37).

5.1.5 Using Pass-By-Reference-To-Const Instead of Pass-By-Value

In C++, by default, objects are passed to functions by value. The cost of passing by value in terms of effectiveness, is related to the amount of constructors and destructors which are called. Initially, the copy constructor of the object is called to initialize the function parameter (which has been passed by value). As we know, this also entails calls to the constructors of potential member variables or classes which the object inherits from. In addition, a destructor is called for every constructor which has been called when the function returns. The cost associated with all these calls is of course something one would like to avoid, if possible. Passing by reference mitigates this problem, since no new object is created by the function. However, the const part of passing by reference to const is important to remember. When passing by value, one can be certain that the original object isn't modified in any way. Passing by reference to const achieves the same result [4](p. 86-90).

The "slicing problem" is also an unfortunate effect of passing by value. If your function accepts a parameter of a base class and an object of a derived classed is passed by value, everything which separates the derived class from the base class is "sliced" off. This is caused by the fact that the copy constructor of the base class is called to initialize the parameter. Therefore, when calling virtual member functions of the parameter object, the functions defined in the base class will be called, not the ones which potentially have been overridden by the derived class. Passing by reference to const also makes sure this problem is avoided [4](p. 86-90).

5.2 An Introduction to Make

Make is a software development tool which keeps track of dependencies between the files in a software development project in order to build an executable out of them. Make does this by generating commands to be executed by a UNIX shell. It also performs other useful tasks in the software building process like removing temporary files [5](p. ix).

To build an executable, source files must be compiled to object files, which are then linked to other object files and libraries. This means that if some of the source files have been modified, they have to be recompiled and relinked to the necessary files. Therefore, executing the necessary building commands becomes difficult in large projects with complex dependency hierarchies. Fortunately, after one has specified the dependencies between the files, make generates these commands automatically. All one has to do is to execute the command: [5](p. ix)

```
$ make
```

This way, make saves you the time and struggle in finding the correct commands to execute while also being a useful tool for keeping track of dependencies between files in potentially large projects where many developers are involved [5](p. x).

If an executable is what you want to build using make, this executable is called a *target*. The files which are used to build the target are called prerequisites or

dependents. These files may also have prerequisites. This means that the source files are prerequisites of the object files, which are prerequisites of the executable [5](p. 1).

The dependencies between files which must be specified for make to do it's job are specified in the *description* file. The name of this file is *Makefile*. Some dependencies must be specified in this file, but many of them are automatically detected by make [5](p. 2).

Listing 5 demonstrates an example of a simple Makefile (inspired by [5] p. 3). The first line in each entry is the *dependency line*. The name of the target which the entry represents is specified on the left hand side left of the colon, while the prerequisites of the target are listed on the right hand side. The second line in the entries are the *command lines*, which contains the command which builds the target [5](p. 3).

```
1 program : main.o marble.o
2     g++ -o program main.o marble.o
3
4 main.o : main.cpp
5     g++ -c main.cpp
6
7 marble.o : marble.cpp
8     g++ -c marble.cpp
```

Listing 5: An example of a simple Makefile

When executing the command,

```
$ make program
```

make first checks whether the program target exists. If not, the target's command has to be executed. But if it already exists, the command is necessary to execute only if its prerequisites have been modified after the last build of the target. In both cases, make will check if the target's prerequisites has been modified to find out what commands are necessary to execute. But the prerequisites' prerequisites also have to be checked, and so on. Therefore, make tracks back to the bottom of the prerequisite dependency hierarchy. Then, the necessary commands are executed on the way back up the hierarchy [5](p. 3-4).

5.3 Procedural Generation

Procedural Generation (or PCG for short) can be defined as software capable of algorithmically generating game content with minimal user input. Put differently, PCG allow games to generate game worlds containing landscapes, creatues, music, stories, etc., with limited input from game designers, who generally are responsible for all content creation [6](p. 1).

The decreased need for game designers is one of the benefits of using PCG methods. Humans are expensive and slow. Replacing them with computer software as much as possible would be more effective in that sense. In addition, one won't have to depend of the creativity of a potentially small group of designers in order to create a

game world rich with content. Humans also have a tendency to imitate each other, limiting their creative output. Well designed PCG methods could therefore have the potential to generate content even the most skilled and creative designers couldn't [6](p. 3-4).

In the last three decades, many games using PCG methods have been published, notable examples being commercial successes like *Minecraft* and *Spore*. Unfortunately, many PCG methods are quite limited at the moment. However, recent years has seen an increase in research activity on the matter, the goal of these efforts being to find new PCG methods and to improve existing ones. There are several central challenges in the field of PCG where finding a solution would be a significant achievement. For instance, making *Multi-Level, Multi-Content PCG* possible is yet to be accomplished. Here, only a game engine and a set of game rules would be predetermined, the rest of the game content being generated by PCG methods. This means that all of the generated content would have to be of high quality and fit perfectly with each other to produce a coherent gaming experience. It should be noted that making the content fit together perfectly is an especially demanding challenge when generating large and complex game worlds [6](p. 5).

5.4 Preliminary Work on the Marble Library

This thesis builds upon the work presented in [1], a specialization project where some of the fundamental functionality for the Marble library was developed. The essence of the roles that the classes play and how they interact with each other are mostly unchanged. However, while the purposes of the classes are largely the same, their implementations have been modified significantly, improving old implementations and enabling additional features. Put more concretely, most of the essential member variables and method signatures have been kept, but the method definitions have changed significantly. The rest of this section presents the most important features of the software developed in the specialization project. Again, some of these features are present in the work presented in this thesis, but almost all of their implementations have been reworked and expanded.

Objects of the *Marble* class serve as the building blocks with which the user builds trees. The most important member variables of this class are as follows:

1. A collection of children (Marble objects).
2. A collection of *Spec* objects (discussed in the next paragraph), which should be executed by this particular Marble object.
3. A collection of Spec objects, which should be executed by descendant Marble objects.
4. A collection of properties. These properties are objects of the *Property* class, which was created to manage the properties polymorphically. The *Integer* class, derived from the Property class, was created. Naturally, it contains a member variable of type *int*. There is also *StringProperty* class, which have a string value associated with them.

The most important methods of the Marble class are:

1. *AddProperty*, which adds a pointer to an object of type Property to the collection of properties.
2. *AddSpec*, which adds a pointer to an object of type Spec to the collection of Spec objects.
3. *Traverse*, the algorithm which executes Spec objects and makes sure descendant bound Spec objects reach their targets, before recursively calling itself on its children.

The *Spec* class is also one of the classes which has been kept. The purpose of Spec objects is to make modifications to a tree. They represent the specifications of a tree, which explains the name. The most important member variable class is the pointer to its owner, which the Spec objects uses to perform operations on the owner. Meanwhile, the *Execute* method is the central method of the class. This is a purely abstract method, where derived classes provide a definition depending on what modifications the object should apply to the owner. Two derived class have been created, *AddProperty* (now renamed to *PropertySpec*) and *Expansion*. The former adds a property to the target Marble object, as the name implies. The latter creates Marble objects and adds them to the target's collection of children. The type of children to create must be specified. This is specified by creating a class deriving from Expansion, where the definition of the *NewChild* method (now renamed to *CreateChild*) determines the type of children to create. For instance, a *MarbleExpansion* object will create children of type Marble.

Now, what could be done with this set of features? As pages 23-27 of [1] presents, the first step is to create the root object of a tree to be generated. The next step is to create Spec objects to determine the structure of the tree, as well as the properties of objects in the tree. Expansion Spec objects would determine the amount of children for each object in the tree, as well as their type (either Marble, or some custom class derived from Marble). Spec objects of type AddProperty would be used to add properties to the objects in the tree, either integer properties or string properties. When all these Spec objects have been created and added to the root object, the final step is to call the Traverse method. This method generates the tree by executing Spec objects and by making sure all Spec objects reach their targets. Figure 1 presents a tree which was created in the specialization project [1](p. 24). Here, *BarExpansion*, *LineExpansion* and *NoteExpansion* Spec objects have been used to generate the tree. Meanwhile, an AddProperty object has been used to add an integer property to all objects of type *Note*. To sum it all up, the parts of the software developed in the specialization project which have been the most valuable for the development of the Marble library, is how the Traverse method executes Spec objects to generate trees consisting of Marble objects.

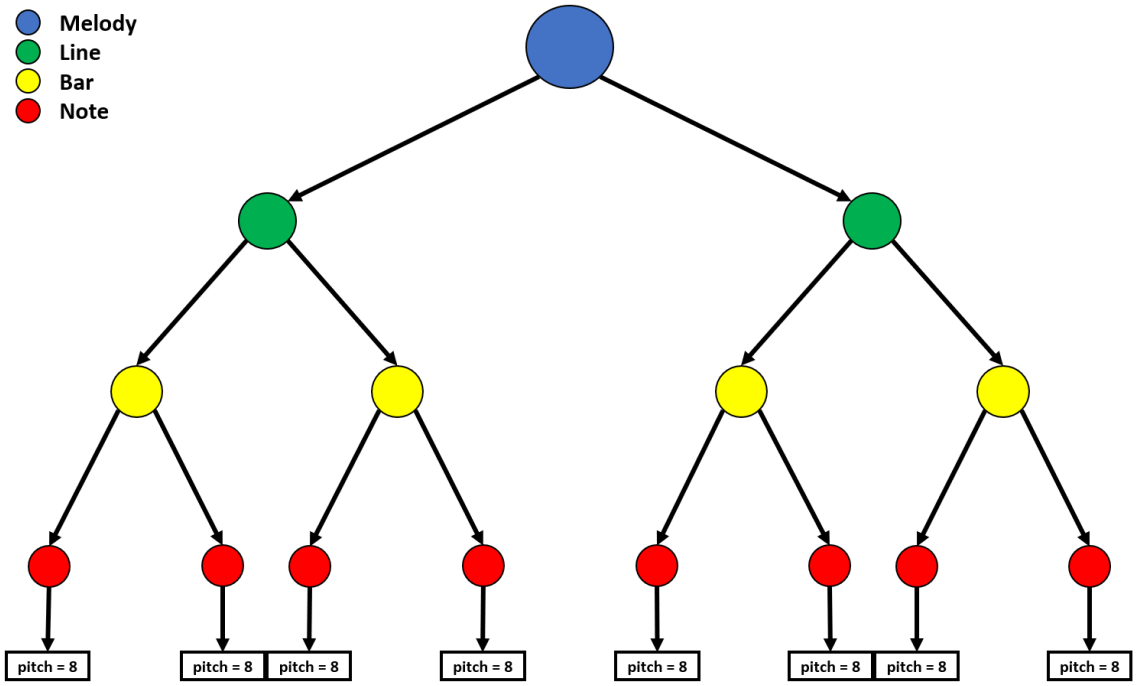

Figure 1: A tree which was created in the specialization project. The figure is collected from [1](p. 24)

6 Specifications for The Marble Library

6.1 Building Trees

6.1.1 The Marble Class and The Spec Class

Quickly restated, the purpose of the Marble library is to let a user build separate trees which through loose coupling all can cooperate in the generation of central tree. Naturally, the first specification for the library should therefore be that it can be used to build trees. Now, what is necessary when building a tree? Establishing what kind of building blocks (nodes) the tree should consist of is a natural place to start. This leads to the library's first primary specification:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.

Note that the specifications presented in this section also include those which were satisfied by the code in the specialization project [1]. These pre-satisified specifications are tagged with **(SP)**. However, features that were present in the specialization project software which have been significantly modified or redone won't have this tag. Even though they're tagged with **(SP)**, the implementations of these specifications will still be presented in Section 7. This is caused by the fact that their implementations are relevant to present as well as the fact that some of the implementations have been slightly modified from their specialization project [1] implementations.

When building a tree, there's also need for a building plan. That is, information about the structure of the tree (e.g. how many children each Marble object should have) must be provided. In other words, the specifications of the tree must be determined. This leads to the library's second primary specification:

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.

For a Spec object to able to make modifications to a tree, information must be provided such that it knows where to make the modification. Therefore, all Spec objects must have at least one Marble object as their target. In the specialization project [1], all Spec objects where given an object of a class named *Target*, which specified what Marble object(s) they should be executed by. The increased complexity caused by this additional class is unnecessary however, and it should be removed. In stead, the Spec objects should find their targets by evaluating every Marble object in the tree, checking if some condition is satisfied. This should be something simple like searching for a Marble object with a specific name. However, it should also be possible to designate the targets based on more advanced conditions. For instance, it might be helpful to let the Spec objects join a Marble object's Spec collection if e.g. the Marble object has some specific property.

The modification a Spec object makes must be defined. Therefore, all Spec objects must have a function associated with them that defines this modification. A wide variety of different Spec objects for different purposes might become necessary. However, the first essential Spec object is one that expands trees, i.e. creating Marble objects and adding them to the target's collection of children. The type of Marble object to expand with will vary however. There should therefore be an *Expansion* class from which other classes can inherit. These derived classes will create Marble objects of a specific type. Since building trees of Marble objects (the base class) is the standard, a *MarbleExpansion* class should be created. Finally, in the specialization project [1], the Spec objects were "cloned" (meaning that a new identical object was created) when they joined a Marble object's collection. This is an unnecessarily complex and memory consuming approach. In stead, a single Spec object should be able to be executed by several Marble objects. For now, these are the specifications for the Spec class:

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.
   (a) All Spec objects must have at least one Marble object as their target.
   (b) The Target class should be removed. In stead, the Spec objects should decide whether or not to join a Marble object's Spec object collection based on some simple default evaluation, e.g. checking the "name" of the Marble object.
   (c) Optionally, the Spec objects should also be able to base the decision of joining a Marble object's Spec collection on more advanced conditions.
   (d) All Spec objects must have a function associated with them that defines the modification they make. **(SP)**
   (e) An *Expansion* class, deriving from the Spec class, should be created. Objects of this class should create Marble objects before adding them to the target's collection of children.
      i. A MarbleExpansion class should be created. Objects of this class will expand trees with Marble objects.
   (f) In stead of cloning Spec objects, a single Spec object should be able to by executed by several Marble objects.

Returning to the Marble class, to compose trees out of Marble objects, each Marble object needs to have a collection containing its children. Spec objects must also be able to make their modifications. Therefore, each Marble object must have a collection of Spec objects waiting to make their modifications on their owner. In addition, the Marble class must have a method that executes the Spec objects waiting, thereby generating the tree. As previously mentioned, the software developed in the specialization project [1] included a Target class. However, there was also a *Filter* class. Every Marble object would hold a Filter object, which would evaluate Target objects to determine if the corresponding Spec object should be added to the collection of Spec objects to execute. Using a class to support this functionality is

unnecessarily complex, and it should be removed. In stead, Marble objects should simply let the Spec object check a simple condition, e.g. if the Marble object has the correct "name". Marble objects should also allow Spec objects to evaluate more advanced conditions, like accessing the Marble object's properties.

The only tree node which will be directly accessible to the programmer is the root. Therefore, all Spec objects must be added to this object. This includes the Spec objects which aren't meant to be executed by the root. In other words, there's need for functionality which makes sure all Spec objects are distributed throughout the tree, such that they can reach their targets. There's another important motivating factor behind the practice of adding all Spec objects to the root before distributing them to their targets. **This practice ensures loose coupling between interacting trees**. The reason for this is that trees that depend on each other can add Spec objects to each others roots. This way, all dependencies between trees are routed to the roots, avoiding the cluster of node-to-node dependencies between trees which would be the only alternative (inspired by [3] (p. 26)).

To complete the generation of a tree, a Marble object now has three responsibilities. Firstly, its Spec objects must be executed. Secondly, the Spec objects in the Spec collection must be added to the Marble object's children, such that all Spec objects can reach their targets. This implies that Marble objects will collect Spec objects that it should execute, as well as Spec objects that are meant to reach a descendant. In addition to the collection of Spec objects to execute, the Marble class should therefore have a collection of Spec objects that are meant to reach one or several descendants. Finally, the same steps must be followed by the children. Therefore, these three steps should be strung together in one Marble class method. This way, the method can be called recursively on the Marble object's children. Stringing all tree generation steps together in a method also results in a simpler user interface. When a root object and all necessary Spec objects have been created, the user only has to call this method to generate the tree. This method will from now on be referred to as the *Traverse* method, since it traverses the tree while its being generated. As it stands, the list of specifications for the Marble class are as follows:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.
   (a) The Marble class must have a collection of Marble objects representing the object's children. **(SP)**
   (b) The Marble class must have a collection of Spec objects waiting to be executed. **(SP)**
   (c) The Marble class must have a collection of Spec objects which are meant to reach one or several descendants. **(SP)**
   (d) The Marble class must have a method which can be used to add Spec objects to its collection of Spec objects.
   (e) The Filter class, which was created in the specialization project [1] should be removed. In stead, Marble objects should let Spec object evaluate them using some simple default condition (e.g. checking its "name").

(f) As an option, the Spec objects should also be able to evaluate the Marble objects using more advanced custom conditions.

(g) The Marble class should have the *Traverse* method, responsible for:

i. executing its Spec objects (letting the Spec objects make their modifications on it). **(SP)**

ii. distributing the Spec objects in its collection throughout the tree, such that the Spec objects can reach their targets. This is done by adding all Spec objects to all children. **(SP)**

iii. recursively calling the Traverse method of its children.

6.1.2 Generating Random Trees: Seeds

There are cases where one would want to generate trees where randomness plays a part, e.g. in procedural generation (see section 5.3). Therefore, the Marble class should have a random method that can be used e.g. to create a random amount of children or to add an integer property with a random value. The use of random functions implies that the Marble class should have a seed which could be passed as an argument to a Marble object's constructor. However, when a Marble object is expanded with children objects, these children can't inherit their parent's seeds. If they did, their random functions would return the same values. This means that if e.g. the children should create a random amount of children, all of them would create the same amount. As a result, the tree wouldn't have been as random as it should've been. In other words, all Marble object must have a unique seed.

The Marble class should however have two random functions at its disposal. This would be a useful in many contexts where a random tree is to be generated. For instance, one could use one of the random functions to determine the amount of children for each Marble object, while the other random function is used to create properties for the objects. This way, one could create several trees of the same structure, but with differing properties (inspired by [3] p. 16). The additions to the list of specifications for the Marble class are as follows:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.

(h) Each Marble object should have two unique seeds.

(i) The Marble class should have two random functions.

6.2 Giving the Trees Purposes: Triggers and Properties

There is little purpose behind building a tree if the tree can't "do" anything. Therefore, a tree should be able to hold certain properties and execute specific tasks such that a programmer can use them in applications.

6.2.1 Marble Objects Holding Properties

Enabling Marble objects to hold properties would be useful in many contexts. This way, they can store values which the tree needs to accomplish some task. For instance, Note objects in a tree representing a melody might need an integer property for the note's pitch. Integer properties aren't the only type of properties that would be useful however. One should also be able to add properties holding string values to Marble objects, assigning a Marble object some name for instance. Therefore, the Marble class should have a collection of properties. Naturally, it should also have a method which adds properties to the collection of properties.

6.2.2 Dynamic Properties

There are many contexts where a Marble object would need to retrieve and store a property by searching for it among its ancestor's properties. This thesis will refer to such properties as *dynamic properties* (similar to the dynamic scope described in [2] p. 6). These properties are useful e.g. in cases where a Marble object has one or several properties which somehow should govern or influence the properties or behaviour of its descendants. Consider a tree where all Marble objects in a tree already has a string property with the identifier "name". If one wants all Marble objects to have an additional "parentName" property, a dynamic property could be used where they would search their ancestor's properties for a "name" property. When such a property is found, it could be stored in the Marble object with e.g. the identifier "parentName". It's assumed that the search stops as soon as a property with the correct identifier is found.

6.2.3 Calculating Properties

The library should provide the use of properties where the specifics of the property will depend on the Marble object it's assigned to. For instance, if one wants to assign a unique string property to all Marble objects, the string would have to be calculated based on each Marble object. One could for instance base the calculation of the names on each Marble object's path in the tree (inspired by [2](p. 13)). Put more concretely, a *calculating property* should assign a Marble object a property based on some calculation.

6.2.4 Adding Properties Using Spec Objects

Since only the root object is directly accessible to the programmer, adding properties to other objects in the tree is only possible with the help of a Spec object. The Spec object would have the property attached to it such that it could carry it all the way to the target Marble object. As it stands, the additions to the the list of specifications are as follows:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.
   (j) The Marble class should have a collection of properties. **(SP)**
   (k) The Marble class should have a method which adds a property to the collection of properties.

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.
   (g) There should be Spec object which can carry a property to its target Marble object. **(SP)**

3. **Properties which can be assigned to Marble objects should include:**
   (a) Integer properties **(SP)**
   (b) String properties
   (c) Dynamic properties: properties which are retrieved by searching among the properties of the Marble object's ancestors.
   (d) Calculating properties: properties where their specifics are based on some calculation (the calculation possibly being applied to the Marble object itself).

6.2.5 Trees Executing Functions: Triggers

As previously mentioned, the trees should have the ability to execute specific tasks or functions. This gives the trees greater functional power which means that they'll be applicable in a wider range of contexts. One should be able to assign specific functions to any Marble object in a tree, since the objects will have different roles and responsibilities. From now on, these functions will be referred to as *triggers*. Executing the triggers of the Marble objects in a tree should happen in succession, as applications of the library is likely to depend of a combination of several triggers. To make sure all triggers are executed, the Traverse method should include execution of any triggers present at a Marble object. There is one final piece necessary to make the triggers work. As with the properties, there is need for a Spec object which can carry a trigger to a Marble object in a tree, since the root object is the only one which is directly available to the programmer. The following specifications are added:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.

   (l) Marble objects should be able to hold triggers. These functions should be versatile enough to be applicable in a wide range of contexts.

   (m) Triggers should be executed by the Traverse method.

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.

   (h) There should be a Spec object that can carry a trigger to a Marble object in a tree.

6.2.6 Building Trees of Custom Marble Objects: The Output Object

To increase the library's flexibility in regards to the range of applications it would be useful to, the Marble class should be implemented such that one could create derived classes with specific purposes. This way, one could build trees consisting of objects of derived classes. These objects could then have predefined properties and triggers or even additional member variables, depending on the needs of the library user.

The *Output* class, derived from the Marble class, should be provided by the library (inspired by [2](p. 4)). Objects of this class should be responsible for printing strings to the console, something which should be done using a trigger. This would be a useful feature because these objects can be used to provide the library user with transparency and an overview of how their tree application is working or progressing. They could also be helpful in debugging a tree application. For an output object to be useful, they must be able to print strings which contain information about their parent. For instance, if one wants to print the "names" of all Marble objects in a tree, it should be possible to add Output objects to all the Marble objects such that the Output objects prints strings containing their parent's "name"-property. The additions to the list of specifications are as follows:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.

   (n) The *Output* class should be created (derived from the Marble class). Output objects should be equipped with a trigger that prints strings to the console. Output objects should have access to its parent's properties.

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.

   (e) An *Expansion* class, deriving from the Spec class, should be created. Objects of this class should create Marble objects before adding them to the target's collection of children.

      ii. An *OutputExpansion* object should be created. Objects of this class will expand trees with Output objects.

6.3 Interaction Between Trees

As previously mentioned, the purpose of the Marble library is to let a user build separate trees which through loose coupling can cooperate in the generation of central tree. In other words, the library must allow Marble objects to create Spec objects and add them to a separate tree. Triggers can be used by Marble objects to create Spec objects. However, the roots of separate trees must be accessible to any Marble object for this to work:

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.

   (o) All Marble objects must have access to the roots of external trees such that Spec objects can be added to them.

6.4 Full Specification List

1. **The *Marble* Class.** Objects of this class (and derived classes) will serve as the nodes of a tree.
   (a) The Marble class must have a collection of Marble objects representing the object's children. **(SP)**
   (b) The Marble class must have a collection of Spec objects waiting to be executed. **(SP)**
   (c) The Marble class must have a collection of Spec objects which are meant to reach one or several descendants. **(SP)**
   (d) The Marble class must have a method which can be used to add Spec objects to its collection of Spec objects.
   (e) The Filter class, which was created in the specialization project [1] should be removed. In stead, Marble objects should let Spec object evaluate them using some simple default condition (e.g. checking its "name").
   (f) As an option, the Spec objects should also be able to evaluate the Marble objects using more advanced custom conditions.
   (g) The Marble class should have the *Traverse* method, responsible for:
      i. executing its Spec objects (letting the Spec objects make their modifications on it). **(SP)**
      ii. distributing the Spec objects in its collection throughout the tree, such that the Spec objects can reach their targets. **(SP)**
      iii. executing triggers.
      iv. recursively calling the Traverse method of its children.
   (h) Each Marble object should have two unique seeds.
   (i) The Marble class should have two random functions.
   (j) The Marble class should have a collection of properties. **(SP)**
   (k) The Marble class should have a method which adds a property to the collection of properties.
   (l) Marble objects should be able to hold triggers. These functions should be versatile enough to be applicable in a wide range of contexts.
   (m) Triggers should be executed by the Traverse method.
   (n) The *Output* class should be created. Output objects should be equipped with a trigger that prints strings to the console. Output objects should have access to its parent's properties.
   (o) All Marble objects must have access to the roots of external trees such that Spec objects can be added to them.

2. **The *Spec* Class.** Objects of classes derived from this one will modify the tree to meet the specification it represents.

   (a) All Spec objects must have at least one Marble object as their target.

   (b) The Target class should be removed. In stead, the Spec objects should decide whether or not to join a Marble object's Spec object collection based on some simple evaluation, e.g. checking the "name" of the Marble object.

   (c) Optionally, the Spec objects should also be able to base the decision of joining a Marble object's Spec collection on more advanced conditions.

   (d) All Spec objects must have a function associated with them that defines the modification they make. **(SP)**

   (e) An *Expansion* class, deriving from the Spec class, should be created. Objects of this class should create Marble objects before adding them to the target's collection of children.

      i. A MarbleExpansion class should be created. Objects of this class will expand trees with Marble objects.

      ii. An *OutputExpansion* object should be created. Objects of this class will expand trees with Output objects.

   (f) In stead of cloning Spec objects, a single Spec object should be able to by executed by several Marble objects.

   (g) There should be Spec object which can carry a property to its target Marble object. **(SP)**

   (h) There should be a Spec object that can carry a trigger to a Marble object in a tree.

3. **Properties which can be assigned to Marble objects should include:**

   (a) Integer properties **(SP)**

   (b) String properties

   (c) Dynamic properties: properties which are retrieved by searching among the properties of the Marble object's ancestors.

   (d) Calculating properties: properties where their specifics are based on some calculation (the calculation possibly being applied to the Marble object itself).

7 Library Implementation

This section describes how the library features are implemented according to the list of specifications in Section 6.4. How one should use said features is presented in the library tutorial (Section 8).

7.1 The Property Class

Since parts of the Marble class implementation are related to properties, the implementation of the Property class will be presented firstly. Satisfying Library Specification 1j, the Marble class contains a vector of pointers to dynamically allocated objects of the Property class. Now, how does the Property class work? Its definition is presented in Listing 6 (helper methods which are irrelevant in this context have been omitted).

```
1  // file "lib/property.h"
2  class Property {
3  private:
4    Marble* prototypeOwner;
5  protected:
6    std::string identifier;
7  public:
8    explicit Property(std::string identifier);
9    virtual ~Property() = 0;
10   virtual Property* Clone() const = 0;
11 };
```

Listing 6: The definition of the Property class

All Property objects have an *identifier*, which is initialized by the constructor. The *Clone* method also has an important responsibility. When one wants to add a Property object to a Marble object, the Property object must first be created using dynamic allocation. This object will from now on be referred to as the *prototype* of the property. Since a Marble object should be able to own a property which itself can make modifications to, a *clone* of the prototype must be created. Therefore, anytime a property is added to a Marble object, a cloned version of the prototype will be added. The method is declared pure virtual since there are several classes deriving from the Property class. That is, every derived class must define the Clone method such that the correct Property type is created. For instance, the *IntegerProperty* class defines it like this:

```
1  // file "lib/property.cpp"
2  Property* IntegerProperty::Clone() const {
3    return new IntegerProperty(*this);
4  }
```

The reason why the Clone method can't be defined polymorphically in the Property base class, is that the type of object to allocate dynamically with the *new* operator must be declared explicitly.

Naturally, the Marble destructor will deallocate all the Property objects it holds (see section 7.2.6). However, something must be held responsible for the deallocation of the Property prototypes. Therefore, the Property class contains the *prototypeOwner* member variable. This is the pointer to the Marble object responsible for deallocating it. This pointer will be set to the first Marble object which clones the property.

7.1.1 Integer and String Properties

As Library Specifications 3a and 3b state, a Marble object should be able to hold integer properties and string properties. Therefore, the *IntegerProperty* and the *StringProperty* classes were created, deriving from the Property class. Their definitions are presented in Listing 7 (irrelevant helper functions have been omitted). Member variables representing the values of the properties have been added, which are initialized by values passed to their constructors. Their Clone methods are also defined according to the explanation given previously in this section.

```
1  // file "lib/property.h"
2  class IntegerProperty : public Property {
3  private:
4    int value;
5  public:
6    explicit IntegerProperty(std::string identifier, int value);
7    Property* Clone() const override;
8  };
9
10 class StringProperty : public Property {
11 private:
12   std::string value;
13 public:
14   explicit StringProperty(const std::string identifier,
15       std::string value);
16   Property* Clone() const override;
17 };
```

Listing 7: The defintions of the IntegerProperty and StringProperty class

7.1.2 Dynamic properties

Library Specification 3c states that a Marble object should be able to hold *dynamic* properties. These properties will search the Marble object's ancestors for a specific property, clone said property and store it at the Marble object. In other words, when adding a dynamic property with the search identifier "color", the first property found among the ancestors with the same identifier will be stored at the Marble object.

Now, how exactly does this work? When a dynamic property is added to a Marble object using the *AddProperty* method, the *AncestorPropSearch* function is called. This function, along with the definition of the *DynamicProperty* class, is presented in Listing 8 (helper functions in the DynamicProperty class have been omitted). Notice that the DynamicProperty class has a *searchIdentifier* member variable. Starting at the Marble object's parent, the AncestorPropSearch function checks if a Property

object matching this search identifier is present in its property collection. If such a Property object exists, the pointer to this Property object is returned. If not, the function calls itself recursively on the parent's parent, and so on. If the AncestorPropSearch function returns something other than *nullptr*, the AddProperty method clones this property and pushes it back to its vector of Property pointers.

```
1  // file "lib/property.h"
2  class DynamicProperty : public Property {
3  private:
4    const std::string searchIdentifier;
5  public:
6    explicit DynamicProperty(std::string id, const std::string sid);
7    Property* Clone() const;
8  };
9
10 // file "lib/marble.cpp"
11 Property* Marble::AncestorPropSearch(std::string identifier) const{
12   static bool firstCall = true;
13
14   Property* pProp = nullptr;
15
16   if (!firstCall) {
17     try {
18       Property* pProp = properties.at(identifier);
19       return pProp;
20     }
21     catch (std::out_of_range& oor) {}
22   }
23
24   if (pProp == nullptr) {
25     if (pParent == nullptr) {
26       return nullptr;
27     }
28     else {
29       firstCall = false;
30       pProp = pParent->AncestorPropSearch(identifier);
31     }
32   }
33
34   firstCall = true;
35   return pProp;
36 }
```

Listing 8: The defintions of the AncestorPropSearch function and the DynamicProperty class

7.1.3 Calculating Properties

According to Library Specification 3d, the Marble library should support the use of *calculating* properties. When such a property is added to a Marble object, based on some calculation, another Property should be created and added (e.g. a SringProperty). This is useful e.g. if one wants to assign a unique name to multiple Marble objects. The *CalculateProperty* class was created to satisfy this specification. The defintion of this class is presented in Listing 9. Notice that the class contains a func-

tion pointer called *Calculate*. The function this pointer points to is the definition of the calculation to perform. The function must return a pointer to a Property object, since the function should create a new Property object using dynamic allocation. The function must also accept a pointer to a Marble object as an argument. This way, calculations can be made using the owner's properties and member variables.

```
1 // file "lib/property.h"
2 class CalculateProperty : public Property {
3 public:
4   explicit CalculateProperty(std::string identifier,
5       Property* (*FuncPtr)(Marble*));
6   Property* Clone() const;
7   Property* (*Calculate)(Marble*);
8 };
```

Listing 9: The definition of the CalculateProperty class

7.2 The Marble Class

Listing 10 presents the definition of the Marble class. Note that helper functions which are irrelevant in this context have been omitted. To satisfy Library Specification 1a, the Marble class contains a map (from the STL library) which stores the Marble object's children. The map, declared at line 18, has keys which are integers starting at zero (their *pathID*), while the corresponding values are pointers to the children. Satisfying Library Specifications 1b and 1c, the Marble class also contains two vectors which store pointers to executable Spec objects and descendant bound Spec objects, respectively (line 6 and 7).

In addition, the Marble class has a map storing pointers to *Property* objects (objects of class *Property*, these will be referred to as the properties of a Marble object), thereby satisfying Library Specification 1j. As line 20 suggests, the keys to the map are string values representing the properties' identifiers while the map values are pointers to the heap allocated Property objects. Finally, satisfying Library Specification 1l, the Marble class contains a vector of function pointers storing the triggers. As declared in line 10, these functions return *void* and takes a pointer to a Marble object as an argument. This way, the triggers can access their owner's properties and member variables, extending the use cases for the triggers.

```
// file "lib/marble.h"
class Marble {
private:
  Marble* pParent;

  std::vector<Spec*> executionSpecs;
  std::vector<Spec*> descendantSpecs;
  std::vector<Spec*> specsOwned;
  std::vector<Property*> propPrototypes;
  std::vector<void (*)(Marble*)> triggers;
  std::vector<void (*)(Marble*)> finalTriggers;
  std::pair<std::size_t, std::size_t> seeds;
  static std::deque<Marble*> discovered;

protected:
  int pathID;
  std::vector<int> path;
  std::map<int, Marble*> children;
  const std::string typeID;
  std::map<std::string, Property*> properties;

public:
  explicit Marble(Marble* pParent, const std::string typeID,
      std::pair<std::size_t, std::size_t> seeds = {0,0});
  virtual ~Marble();

  void Traverse();
  void AddSpec(Spec* pSpec);
  void AddExternalTreeSpec(std::string rootName, Spec* pSpec);
  void AddProperty(Property* pProp);
  void AddTrigger(void (*pTrigger)(Marble*));
  void AddFinalTrigger(void (*pTrigger)(Marble*));

  bool ExecuteHere(Spec* pSpec);

  Property* AncestorPropSearch(std::string identifier) const;
  std::string AncestorStringSearch(std::string identifier) const;
  std::string FindScopeOperators(std::string inputStr) const;

  int RandomInteger1(int min, int max);
  int RandomInteger2(int min, int max);

};
```

Listing 10: The Marble class definition (irrelevant methods have been omitted)

7.2.1 The AddSpec Method

The *AddSpec* method satisfies Library Specification 1d. Presented in listing 11, this method is responsible for deciding if the incoming Spec object should be added to the *executionsSpecs* vector or the *descendantSpecs* vector. To accomplish this, the *ExecuteHere* method is used to decide whether or not the incoming Spec object should be executed at the Marble object. If this method returns false, the Spec object is added to the *descendantsSpecs* vector such that it's later passed on to descendants.

Now, how does the ExecuteHere method make its decision? Library Specification 1e stated that the Spec objects should by default find their targets by evaluating all Marble objects based on some simple condition. Therefore, the Marble class has a member variable of type string (from the STL library) called *typeID*. This variable is set when it's passed to the Marble constructor. A Marble object representing e.g. a Note might have a "Note" typeID. Correspondingly, the Spec object constructor accepts a *targetTypeIDs* vector, containing string values. So if a Spec object should be bound for Note objects, it should be created by passing a vector containing "Note" to its constructor. Line 28-30 of listing 11 shows how the ExecuteHere method searches for the Marble object's typeID in the incoming Spec object's vector of targetTypeIDs. Notice that right beforehand, the ExecuteHere method checks if the incoming Spec object has a function pointer called *AdvancedDesignation* other than *nullptr*. Satisfying Library Specification 1f, if present, this function may decide whether or not to join the Marble objects executable Spec objects based on more advanced custom conditions.

Notice that the AddSpec method accepts a pointer to a Spec object as an argument. This way, the function can polymorphically accept any object derived from the Spec class. All Spec object are also allocated on the heap, working with pointers is therefore natural. Since the Spec objects are heap allocated, something must be responsible for deallocating them when the program ends. This responsibility falls on the first Marble objects which receives the Spec object. In other words, this Marble object becomes the Spec object's *owner*, as lines 5-7 in listing 11 suggest.

```
// file "lib/marble.cpp"
void Marble::AddSpec(Spec* pSpec) {

  Marble* pSpecOwner = pSpec->GetOwner();
  if (pSpecOwner == nullptr) {
    pSpec->SetOwner(this);
    this->specsOwned.push_back(pSpec);
  }

  if (ExecuteHere(pSpec)) {
    executionSpecs.push_back(pSpec);
  }
  else {
    descendantSpecs.push_back(pSpec);
  }
}

bool Marble::ExecuteHere(Spec* pSpec) {
  std::vector<std::string> targetTypeIDs;
  targetTypeIDs = pSpec->GetTargetTypeIDs();

  if (pSpec->AdvancedDesignation != nullptr) {
      return pSpec->AdvancedDesignation(this, pSpec);
  }

    // return true if the Spec object's targetTypeIDs
    // include this->typeID
  return std::find(
          targetTypeIDs.begin(), targetTypeIDs.end(), typeID
      ) != targetTypeIDs.end();

}
```

Listing 11: The definitions of the AddSpec method and the ExecuteHere method

7.2.2 The Marble Constructor

The Marble constructor (presented in Listing 12) is responsible for satisfying a couple of the Library Specification items. First of all, the string value for the typeID member variable must be passed to the constructor. As previously stated, this member variable makes sure Spec objects can find their targets, satisfying Library Specification 1e. The constructor also accepts the pointer to the Marble object which is creating it, its parent in other words. As lines 12-16 in Listing 12 suggest, this pointer is used to retrieve the parent's path in order to set the new Marble object's path. At line 19, the new Marble object is also added to its parent's map of children.

The Marble library is designed such that only a single Marble object should be created in the main function of the program. From now on, this object will be referred to as the *system* object. Consequently, the roots of the trees to be created will be children of the system object. Library Specification 1o states that all Marble objects in a tree should have access to the roots of other trees. Lines 21-24 facilitates this functionality. Here, the if statement checks if this Marble object is a child of the system object. If so, a *RootProperty* is added to the parent (the system object). This property contains a pointer to the Marble object which is being constructed (which is the root of a new tree). The RootProperty is now owned by the system object, which is an ancestor shared by all Marble objects in any tree. This means that any Marble object in any tree can use a dynamic property to retrieve this root pointer, effectively satisfying specification 1o.

The next responsibility for the constructor is setting the seeds. Library Specifications 1h states that every Marble object should have two unique seeds. To accomplish this, lines 33-35 creates the *name* string variable, which is the concatenation of its typeID and its path. Since this variable is unique to all Marble objects, it can be used to set a unique seed. However, to increase the "randomness" of the tree generation, the seeds should be influenced by an additional factor. Therefore, the seeds of the parent should be incorporated. Consequently, the seeds are defined by averaging a hash of the name variable and a hash of the parent's seeds, as lines 45-46 suggest.

```cpp
// file "lib/marble.cpp"
Marble::Marble(Marble* pParent, const std::string typeID,
    std::pair<std::size_t, std::size_t> sds)
  : pParent(pParent), seeds(sds), typeID(typeID) {

  if (pParent == nullptr) {
    this->path = { 0 };
    this->pathID = 0;
  }
  else {
    // setting path and pathID
    std::vector<int> parentPath = pParent->GetPath();
    std::map<int, Marble*>& parentChildren=pParent->GetChildren();
    this->pathID = parentChildren.size();
    parentPath.push_back(this->pathID);
    this->path = parentPath;

    // adding the new Marble instance to its parent's collection of
        children
    parentChildren.insert(std::pair<int, Marble*>(pathID, this));

    if (pParent->GetParentPointer() == nullptr) {
      // this object is a root
      Property* rp = new RootProperty(this->typeID + "Root", this);
      pParent->AddProperty(rp);
    }
  }

  // seeds
  std::pair<int,int> parentSeeds(0,0);
  if (pParent != nullptr && sds.first == 0 && sds.second == 0) {
    parentSeeds = pParent->GetSeeds();

    std::string name = this->typeID;
    for (int i : this->path) {
      name += "_" + std::to_string(i);
    }

    std::hash<std::string> str_hash;
    std::size_t name_hash = str_hash(name);

    std::hash<int> int_hash;
    std::size_t parent_seed1_hash = int_hash(parentSeeds.first);
    std::size_t parent_seed2_hash = int_hash(parentSeeds.second);

    this->seeds.first = (name_hash + parent_seed1_hash) / 2;
    this->seeds.second = (name_hash + parent_seed2_hash) / 2;
  }

}
```

Listing 12: The definition of the Marble constructor

7.2.3 Adding Spec Objects to External Trees

Section 7.2.2 described how pointers to all root objects were stored in RootProperties at the system object. The *AddExternalTreeSpec* method (presented in Listing 13) uses these RootProperties to add Spec objects to trees external to the Marble objects calling the function. Therefore, Library Specification 1o is satisfied.

The first objective of the AddExternalTreeSpec function is to retrieve the correct RootProperty from the system object. Therefore, a dynamic property is created at line 8. Recall that when the Marble constructor creates RootProperties, their identifiers are set by adding "Root" to the root object's typeID (see line 23 in Listing 12). Therefore, the identifier the dynamic property should search for is the *rootName* parameter value appended with "Root", as line 9 suggests. Note that the purpose of the rootName parameter is that one can choose what root object one wants to add a Spec object to. E.g. if an object with typeID "Melody" is the root of a tree, "Melody" should be passed as the parameter to the AddExternalTreeSpec function.

When using the AddProperty method with the dynamic property (line 13), the AncestorPropSearch function is used to find the correct RootProperty. The RootProperty is then cloned and added to the Marble object's properties. Therefore, the RootProperty can then be looked up in the Marble objects properties using the *GetProperty* method (line 17). Next, to retrieve the root object pointer which the RootProperty holds, the Property pointer must be downcasted to a RootProperty pointer using dynamic casting. Then, the Spec pointer, which is the second method parameter, can be added to the external root object using the AddSpec method (line 28). To make sure this newly added Spec object reaches its target however, the external root object must be traversed. Therefore, at lines 33-35, the root object is added to the static *discovered* deque, unless it's already there.

```
// file "lib/marble.cpp"
void Marble::AddExternalTreeSpec(std::string rootName,
    Spec* pSpec) {
  // objective: create a dynamic property which will retrieve a
  // RootProperty and store it at this object. Then add the
  // Spec object to the root.

  Property* rootProp = new DynamicProperty(rootName + "Root",
      rootName + "Root");

  // Adding the dynamic property, which then will retreive
  // store the RootProperty
  AddProperty(rootProp);

  // Looking up the Property again, which now contains
  // the RootProperty
  Property* pProp = GetProperty(rootName + "Root");

  // Downacasting the Property pointer to a RootProperty
  // pointer
  if (pProp != nullptr) {
    RootProperty* pRP = dynamic_cast<RootProperty*>(pProp);
    if (pRP != nullptr) {
      Marble* pRoot = pRP->GetValue();

      // Adding the Spec object to the root
      // of the external tree
      pRoot->AddSpec(pSpec);

      // If this root has already been
      // traversed, it must be traversed
      // again
      if (std::find(discovered.begin(), discovered.end(),
          pRoot) == discovered.end()) {
        discovered.push_back(pRoot);
      }
    }
  }
}
```

Listing 13: The defintion of the AddExternalTreeSpec method

7.2.4 The Random Functions

As Library Specification 1i states, the Marble class should have two random methods accompanying the seeds. These methods return random integers and are presented in Listing 14. They use the *std::mt19937* pseudo-random number generator, which uses the Mersenne twister algorithm. The *RandomInteger1* method uses the Marble object's first seed and the *RandomInteger2* method uses the second. Since a new instance of the random number generator is created every time the method is called, the Marble object's seeds must be updated for each call, preventing the generation of the same integers for every call. As lines 6 and 14 suggest, the seeds are redefined by another integer generated by the mt19937 object. In addition, a *std::uniform_int_distribution* object is used when generating the random integers such that a lower and upper bound for the integers can be set.

```
1  // file "lib/marble.cpp"
2  int Marble::RandomInteger1(int min, int max) {
3    std::mt19937 mt(seeds.first);
4    std::uniform_int_distribution<int> uni(min,max);
5    int rand = uni(mt);
6    seeds.first = mt();
7    return rand;
8  }
9
10 int Marble::RandomInteger2(int min, int max) {
11   std::mt19937 mt(seeds.second);
12   std::uniform_int_distribution<int> uni(min,max);
13   int rand = uni(mt);
14   seeds.second = mt();
15   return rand;
16 }
```

Listing 14: The definitions of the RandomInteger1 and RandomInteger2 methods

7.2.5 The Traverse Method

The Traverse method (presented in Listing 15 and as a flowchart in Figure 2) is the most important piece of code in the Marble library. It orchestrates the tree generation and makes sure triggers are executed. Firstly, all Spec objects in the object's executionSpecs vector are executed, satisfying Library Specification 1(g)i. Note that some variables have been renamed in Listing 15 to improve readability. For instance, the executionSpecs vector is renamed *exSpecs*. To prevent a Spec object from being executed several times by the same Marble object however, all Spec objects has a vector of pointers to Marble object where it has been executed. Therefore, before executing a Spec object, the Marble object checks if it already has executed the Spec object.

The next step is to pass all Spec objects to the Marble object's children, satisfying Library Specification 1(g)ii. First up is the descendant bound Spec objects. Here, the AddSpec method is called for every child and for every Spec object. As previously explained, this method will decide whether the incoming Spec object will be

added to the child's vector of either executable or descendant bound Spec objects. Next, the executable Spec objects are passed, similarly to the descendant bound Spec objects. However, there is a guard added at line 32 which prevents infinite expansion which would be caused in some cases. Consider the case where a Marble object with typeID "Marble" is targeted by a *MarbleExpansion* Spec object. This Spec object would create Marble objects which also would have typeID "Marble". The Marble object would then execute the Spec object, before the same Spec object would be added to the new children, since they also have the "Marble" typeID, thereby causing infinite expansion.

When all Spec objects have been dealt with, it's time to execute the triggers (Library Specification 1(g)iii). The Marble class stores triggers in a vector of function pointers. These pointers point to functions which return *void* and takes a pointer to a Marble object as an argument. As line 47 suggests, the pointer to the Marble object executing the triggers are passed as this argument, such that the triggers can access and or modify the Marble object's properties and member variables.

The next step is to recursively call the Traverse method on the Marble object's children, satisfying Library Specification 1(g)iv. This is done in a breadth first manner. Therefore, pointers to all of the Marble object's children are pushed back to *discovered*, a static *deque* data structure (from the STL library). Then, the Traverse method is called on the Marble object which the pointer in the front of the deque points to.

When the discovered deque is empty and the Marble object currently executing the Traverse method has no children, the tree has been completely traversed. Then, the final step is to execute potential *final triggers*. As the if statement in line 64 suggests, only the system object can execute a final trigger. Final triggers are useful in cases where a Spec object must be added after the tree has been fully generated. For instance, if one wants to use an advance designation function to add a Spec object to the lastly traversed Marble object in a tree, the tree must be fully generated first. Deciding whether or not a Marble object is the lastly traversed one is impossible when the tree is still under construction.

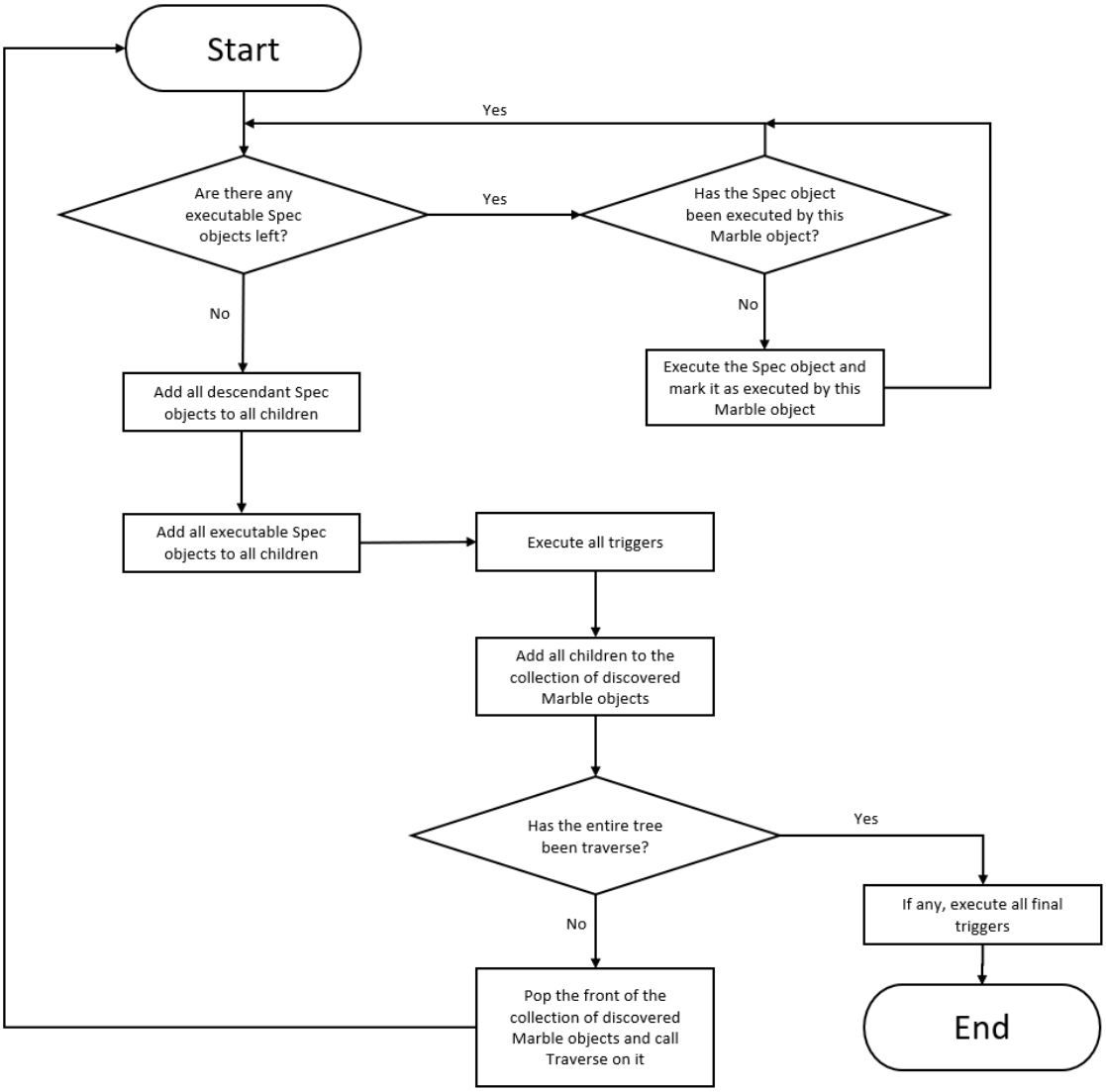

Figure 2: Flowchart describing the Traverse method, created using Microsoft Powerpoint

```
// file "lib/marble.cpp"
void Marble::Traverse()
{

  // Executing Spec objects
  for (auto it= exSpecs.begin(); it!=exSpecs.end(); it++)
  {
    std::vector<Marble*> exM = (*it)->GetExecutedMarbles();
    if (std::find(exM.begin(), exM.end(), this) == exM.end())
    {
      (*it)->Execute(this);
      (*it)->MarbleExecuted(this);
    }
  }

  // Passing descendant bound Spec objects
  for (auto dSpecIt=dSpecs.begin();dSpecIt!=dSpecs.end();dSpecIt++)
  {
    for (auto cIt = children.begin(); cIt != children.end(); cIt++)
    {
      cIt->second->AddSpec(*dSpecIt);
    }
  }

  // Passing executable Spec objects
  for (auto specIt=exSpecs.begin();specIt!=exSpecs.end();specIt++)
  {
    std::vector<std::string> ttIDs;
    ttIDs = (*specIt)->GetTargetTypeIDs();
    for (auto cIt=children.begin();cIt!=children.end();cIt++)
    {
      if (typeID==cIt->second->GetTypeID() &&
          std::find(ttIDs.begin(), ttIDs.end(), cIt->second->
              GetTypeID()) != ttIDs.end())
      {
          break; // prevent never ending expansion when an object
                 // is expanded with children with the same typeID
                 // as their parent
      }
      cIt->second->AddSpec(*specIt);
    }
  }

  // Executing triggers
  std::vector<void (*)(Marble*)>::iterator it;
  for (it = triggers.begin(); it != triggers.end(); it++)
  {
    (*it)(this);
  }
  triggers.clear();

  // Breadth first traversal
  for (auto it = children.begin(); it != children.end(); it++)
  {
    discovered.push_back(it->second);
  }
  if (discovered.size() != 0)
```

```
57    {
58      Marble* nextMarble = discovered.front();
59      discovered.pop_front();
60      nextMarble->Traverse();
61    }
62
63    // Executing final triggers
64    if (pParent == nullptr)
65    {
66      for (auto it = finTrigs.begin(); it != finTrigs.end(); it++)
67      {
68        (*it)(this);
69      }
70    }
71  }
```

Listing 15: The definition of the Traverse Method

7.2.6 The Marble Destructor

The Marble library is implemented using a significant amount of heap allocation. Marble objects, Property objects and Spec objects are all allocated on the heap. Since the Marble objects are responsible for deallocating the Property and Spec objects, the Marble destructor (presented in Listing 16) has great importance in preventing memory leaks. Firstly, the *delete* operator is applied to all children, making sure the tree is destructed in a depth first fashion. Next, any Spec objects the Marble object owns is deallocated. Note that these are not the Spec objects in the executionSpecs and descendantSpec vectors. Recall that a Spec object is *owned* by the first Marble object it's assigned to. Next, all properties the Marble objects hold are deallocated. The final step of the destructor is the deallocation of the all property prototypes. Note also that the Marble destructor is declared *virtual*. This ensures that when objects of types derived from the Marble class are deallocated through a pointer of type *Marble**, the derived class' destructor is called. Had it not been declared virtual, the Marble destructor would've been called in stead.

```
1  // file "lib/marble.cpp"
2  Marble::~Marble() {
3    for (auto it=children.begin();it!=children.end();it++){
4      delete it->second;
5    }
6
7    for (auto it=specsOwned.begin();it!=specsOwned.end();it++){
8      delete *it;
9    }
10
11   for (auto it=properties.begin();it!= properties.end();it++){
12     delete it->second;
13   }
14
15   for (auto it=propPrototypes.begin();it!=propPrototypes.end();it
         ++){
16     delete *it;
17   }
18 }
```

Listing 16: The definition of the Marble destructor

7.2.7 The Output Class

As Library Specification 1n states, the library should come with an *Output* class. This class should derive from the Marble class such that Output objects can be incorporated in trees. Furthermore, Output objects should print specific strings to the console. They should also be able to access their parent's properties, enabling them to print more useful information.

The class definition (presented at the top of listing 17) declares a vector of string values. The string values are the lines which the Output object will print. As line 7 suggests, the *lines* vector is defined by an parameter passed to the Output constructor. Now, how should the contents of the lines vector be printed to the console? Using a trigger is a suitable option in this case. Lines 12-22 contain the definition of the *OTrig* function which will be used as a trigger. Firstly, since the argument of a trigger must be a pointer of type *Marble**, *dynamic casting* is used to cast the Marble pointer to an Output pointer. This pointer is then used to retrieve the Output object's lines vector, before its contents is printed to the console, line by line. To make sure all Output objects are equipped with this trigger, the trigger is added in the Output constructor using the *AddTrigger* method (line 26).

An important part of specification 1n is that an Output object should be able to access and print properties of it's owner. Therefore, the Output constructor is defined such that if a word in the lines vector is prepended with "^", a search for a StringProperty with the succeeding word as the identifier is made among all of its ancestor's properties. As lines 28-35 of Listing 17 suggest, every line in the lines vector is passed to the *FindDynamicString* function. If the "^" operator is detected, the succeeding property identifier is replaced with the corresponding StringProperty's value.

```
1  // file "lib/marble.h"
2  class Output : public Marble {
3  protected:
4    std::vector<std::string> lines;
5  public:
6    explicit Output(Marble* pParent,
7        std::vector<std::string> lines = {});
8    std::vector<std::string> GetLines() const;
9  };
10
11 // file "lib/marble.cpp"
12 void OTrig(Marble* pMaster) {
13   try {
14     Output* pO = dynamic_cast<Output*>(pMaster);
15     for (std::string line : pO->GetLines()) {
16       std::cout << line;
17     }
18   }
19   catch (std::bad_cast& bc) {
20     std::cout << "OutputTrigger : bad_cast" << std::endl;
21   }
22 }
23
24 Output::Output(Marble* pParent, std::vector<std::string> lines)
25     : Marble(pParent, "Output") {
26   AddTrigger(OTrig);
27
28   std::vector<std::string>::iterator it;
29   for (it = lines.begin(); it != lines.end(); it++) {
30     std::string dynamicString = FindDynamicString(*it);
31     if (dynamicString != "") {
32       this->lines.push_back(dynamicString);
33     }
34   }
35 }
```

Listing 17: The Output class definition, constructor definition and the OTrig function definition

The defintion of the FindDynamicString function is presented in Listing 18. Firstly, the input string is searched for the "^" character. If one is found, the input string is then searched for a character ending the identifier, e.g. space, comma, forward slash, etc. Then, the extracted identifier is used in a search for StringProperties among the ancestors' properties using the *AncestorStringSearch* function (line 34). Unless the resulting string is empty, the identifier in the input string is replaced with the value of the retrieved StringProperty.

```
// file "lib/marble.cpp"
std::string Marble::FindDynamicString(std::string inputStr) const {
    // objective: extract all words/identifiers prepended with "^"
    // in inputStr. Make searches for StringProperties with
    // these identifiers among the ancestors' properties and
    // insert the value of the corresponding StringProperties
    // in inputStr

    // Finding the start and end index of potential words prepended
        with "^"
  std::size_t start_idx = inputStr.find("^");
  std::size_t end_idx = std::string::npos;
  std::vector<std::string> id_ends = { " ", ",", ".", "\\", "/",
      "[", "]", "{", "}", "(",")", "\"", "\n" };
  // while a "^" has been found
  while (start_idx != std::string::npos) {

    end_idx = std::string::npos;
    for (std::string id_end : id_ends) {
      std::size_t idx = inputStr.find(id_end, start_idx);
      if (idx < end_idx) {
        end_idx = inputStr.find(id_end, start_idx);
      }
    }

    if (end_idx == std::string::npos) {
      end_idx = inputStr.size();
    }
    // extracting an identifier prepended with "^"
    std::string identifier = inputStr.substr(start_idx + 1,
        end_idx - start_idx - 1);
    if (start_idx != 0) {
        // searching the ancestors' properties with the identifier
      std::string ancestorString =
          AncestorStringSearch(identifier);
      if (ancestorString != "") {
          // replacing the identifier with the corresponding
          // StringProperty's value
        inputStr = inputStr.substr(0, start_idx) +
            AncestorStringSearch(identifier) +
             inputStr.substr(end_idx, inputStr.size());
      }
      else {
        return "";
      }
    }
    start_idx = inputStr.find("^");
  }
  return inputStr;
}
```

Listing 18: The definition of the FindDynamicString function

The definition of the AncestorStringSearch function is presented in Listing 19. This function uses the AncestorPropSearch (see Listing 8) to retrieve the pointer to the StringProperty which matches the extracted identifier. Since the Ancestor-

PropSearch functions returns a pointer to a Property object, the purpose of the AncestorStringSearch function is to downcast the pointer to the property's derived type, namely StringProperty. This, way the StringProperty's value can be retrieved using the *GetValue* method, as presented in line 7.

```
1  // file "lib/marble.cpp"
2  std::string Marble::AncestorStringSearch(std::string identifier)
       const {
3    std::string ancestorString = "";
4    Property* pProp = AncestorPropSearch(identifier);
5    StringProperty* pSP = dynamic_cast<StringProperty*>(pProp);
6    if (pSP != nullptr) {
7      ancestorString = pSP->GetValue();
8    }
9    return ancestorString; // "" is returned if the property isn't
         found (nullptr)
10 }
```

Listing 19: The definition of the AncestorStringSearch function

7.3 The Spec Class

Objects of classes derived from the Spec class will modify trees to meet the specifications they represent. Library Specification 2a and 2b state that every Spec object should have at least one Marble object as its target and that the Spec objects should determine whether or not they should be executed by a Marble object based on checking some simple default condition. These specifications are satisfied by the *targetTypeIDs* member variable within the Spec class (presented in Listing 20, helper functions have been omitted). This vector contains string values which represent the typeIDs of the Marble objects where the Spec object should be executed. As the ExecuteHere method in Listing 11 demonstrates, if a Marble object's typeID is present in a Spec object's targetTypeIDs vector, the Spec object is added to the Marble object's collection of Spec objects to execute.

As Library Specification 2c states, a Spec object should also be able to determine whether it should be executed at a Marble object or not based on more advanced conditions. This specification is satisfied by the *AdvancedDesignation* function pointer, declared at line 19. A function this pointer points to must return a boolean value, indicating whether or not the Spec object should be executed by a specific Marble object. It must also accept a pointer to a Marble object as a parameter. This way, the function can make its decision based on properties and member variables of the Marble object in question. The ExecuteHere method (Listing 11) demonstrates how the AdvancedDesignation function is used. Before going through the Spec object's targetTypeIDs, if present, the ExecuteHere function will return the value the AdvancedDesignation function returns. In other words, advanced designation takes precedence over typeID designation.

The most important part of the Spec class however, is the *Execute* method. This method defines the modifications which the Spec object applies to its owner when executed, thereby satisfying 2d. Its parameter is a pointer to the Marble object on

which the modifications should be made. The method is also declared purely virtual since derived classes with different purposes should provide their own definitions.

Library Specification 2f states that only a single copy of a Spec object should exist. That is, if the Spec objects target multiple Marble objects, they should all execute the same Spec object. Therefore, the Spec class contains the *executedMarbles* vector, which keeps track of what Marble objects which have executed it. This prevents Marble objects from executing the Spec object multiple times. Since all Spec objects are dynamically allocated, a Marble object must be responsible for deallocating it. This responsibility falls on the first Marble object which the Spec object is added to, which the Spec object's *owner* pointer then will point to. Finally, it should be mentioned that the Spec destructor is declared virtual. This way, when Spec objects are deallocated through a pointer of type *Spec**, the destructor of the derived class will be called, in stead of the Spec destructor.

```
1  // file "lib/spec.h"
2  class Spec {
3  protected:
4    const std::vector<std::string> targetTypeIDs;
5    std::vector<Marble*> executedMarbles;
6    Marble* owner;
7
8  public:
9    explicit Spec(const std::vector<std::string> targetTypeIDs,
10       bool (*AdvDes)(Marble*, Spec*) = nullptr);
11   virtual ~Spec();
12
13   virtual void Execute(Marble* pMaster) = 0;
14   void MarbleExecuted(Marble* pM);
15
16   void SetOwner(Marble* pM);
17   Marble* GetOwner();
18
19   bool (*AdvancedDesignation)(Marble* pM, Spec* pS);
20 };
```

Listing 20: The definition of the Spec class

7.3.1 The Expansion Class

The purpose of the Expansion class (the class definition is presented in Listing 21), derived from the Spec class, is to create new Marble objects and add them to another Marble object's collection of children. This class contains the *numNewChildren* integer member variable which, naturally, represents the amount of children which should be created. The definition of its Execute method is presented in lines 14-18 in Listing 21. Here, the *CreateChild* method of the Expansion class is called as many times as the value of the numNewChildren value. Since the type of Marble objects to create (e.g. Marble, Output or other Marble derived classes) will vary, the create child method is declared pure virtual. This way, classes deriving from the Expansion class can define it to create a specific type of Marble objects.

Library Specifications 2(e)i and 2(e)ii state that an Expansion class which creates

Marble objects and an Expansion class that create Output objects should be created. Therefore, the *MarbleExpansion* and the *OutputExpansion* classes were created. Their definitions of the CreateChild method are presented at the bottom of Listing 21. The MarbleExpansion class' definition consists of returning a dynamically allocated Marble object. Note that the *pMaster* Marble pointer (which points to the Marble object currently executing the Expansion object) is passed to the Marble constructor since it's the new object's parent pointer. As explained in Section 7.2.2, the Marble constructor will make sure the new Marble object is inserted in its parent's collection of children. The *childTypeID* variable passed to the Marble constructor is a member variable of the MarbleExpansion class. This variable contains the typeID that the new Marble object should have. Regarding the OutputExpansion class' definition of the CreateChild method, it should be mentioned that the *lines* variable is a member variable of the OutputExpansion class. It contains the vector of string values which the new Output object should print when traversed.

```
1  // file "lib/spec.h"
2  class Expansion : public Spec {
3  protected:
4    const int numNewChildren;
5
6  public:
7    explicit Expansion(const std::vector<std::string> targetTypeIDs,
8        const int numNewChildren, bool (*AdvDes)(Marble*, Spec*) =
             nullptr);
9    void Execute(Marble* pMaster) override;
10   virtual Marble* CreateChild(Marble* pMaster) const = 0;
11 };
12
13 // file "lib/spec.cpp"
14 void Expansion::Execute(Marble* pMaster) {
15   for (int i = 0; i < numNewChildren; i++) {
16     CreateChild(pMaster);
17   }
18 }
19
20 Marble* MarbleExpansion::CreateChild(Marble* pMaster) const {
21   return new Marble(pMaster, childTypeID);
22 }
23
24 Marble* OutputExpansion::CreateChild(Marble* pMaster) const {
25   return new Output(pMaster, lines);
26 }
```

Listing 21: The definitions of the Expansion class, its Execute method and the CreateChild methods of the MarbleExpansion and OutputExpansion classes

7.3.2 The TriggerSpec Class

According to Library Specification 2h, a Spec object which can carry a trigger to a Marble object should be created. Therefore, the *TriggerSpec* class was created. The definitions of the class and its Execute method is presented in Listing 22. The TriggerSpec class has a member variable called *pTrigger*. This is a function pointer which will point to the trigger it's meant to carry. This pointer is initialized by a parameter passed to the constructor. As line 16 suggests, the TriggerSpec class' Execute method simply adds its trigger pointer to the Marble object which is executing the TriggerSpec object.

```
1  // file "lib/spec.h"
2  class TriggerSpec: public Spec {
3  private:
4    void (*pTrigger)(Marble*);
5  public:
6    explicit TriggerSpec(
7        const std::vector<std::string> targetTypeIDs,
8        void (*pTrigger)(Marble*),
9        bool (*AdvDes)(Marble*, Spec*) = nullptr
10       );
11   void Execute(Marble* pMaster) override;
12 };
13
14 // file "lib/spec.cpp"
15 void TriggerSpec::Execute(Marble* pMaster) {
16   pMaster->AddTrigger(this->pTrigger);
17 }
```

Listing 22: The definitions of the TriggerSpec class and its Execute method

7.3.3 The PropertySpec Class

Library Specification 2g states that the Marble library should provide a Spec object which can carry Property objects to Marble objects. Therefore, the *PropertySpec* class was created. The definitions of the PropertySpec class and its Execute method is presented in Listing 23. The class has a member variable called *properties*. This is a vector containing pointers to the Propery prototypes which should be cloned and added to the target Marble object(s). Consequently, its Execute method is defined such that the each Property pointer in the properties vector is added to the Marble object executing the PropertySpec object.

```
// file "lib/spec.h"
class PropertySpec : public Spec {
private:
  std::vector<Property*> properties;
public:
  explicit PropertySpec(
      const std::vector<std::string> targetTypeIDs,
      std::vector<Property*> properties,
      bool (*AdvDes)(Marble*, Spec*) = nullptr
      );

  void Execute(Marble* pMaster) override;
};

// file "lib/spec.cpp"
void PropertySpec::Execute(Marble* pMaster) {
  std::vector<Property*>::iterator it;
  for (it = properties.begin(); it != properties.end(); it++) {
    pMaster->AddProperty(*it);
  }
}
```

Listing 23: The definitions of the Property class and its Execute method

8 Library Features: The Tutorial

The purpose of this tutorial is to present how one uses the features of the Marble library. Each feature is accompanied by a snippet of code which utilizes said feature. These pieces of code are located in the *tutorial* folder of the delivered source code. The Makefile has the *tutorial* target, which will generate all executable files for the tutorial. Targets for each individual tutorial code snippet is also included in the Makefile.

8.1 Setting Up The Library

The Makefile can be used to compile the source code accompanying this thesis. As previously mentioned, all tutorial executable files can be generated using the following command:

```
$ make tutorial
```

Since the library is dynamically linked however, the *LD_LIBRARY_PATH* environment variable must be set and exported (at least when using Linux Ubuntu or Mint) before one can run the executable files. The environment variable must be set to the path where the library .so file is located (after it has been compiled), which is in the same directory as the Makefile. Exporting the environment variable can be done like this:

```
$ export LD_LIBRARY_PATH=/path/to/src/
```

The final step is to include the *libmarble.h* header file, located in the *lib* directory. Now, one can start writing C++ code accessing the library's features.

8.2 Creating the Marble System

When using the Marble library, the first step is to create a Marble object which Spec objects will be added to in order to generate trees. Throughout this tutorial, this object will be referred to as "the system object". The system object can be an object of the Marble class or potential derived classes. For simplicity, we will use the Marble class. Listing 24 shows how the Marble constructor is used to create the system object. The first constructor argument is the pointer to the parent marble. Since we're creating the system object, *nullptr* is passed as the argument. The second constructor argument is the system objects's *typeID*. The typeIDs of the marble objects are necessary for Spec objects to reach their targets, something which will be demonstrated later in this tutorial. Now that the system object has been created, the next step is to expand and configure the tree using Spec objects.

```
1 #include "lib/libmarble.h"
2
3 int main() {
4
5     Marble system(nullptr, "System");
6
7     return 0;
8 }
```

Listing 24: Creating the system object

8.3 Spec Objects: Structure, Properties and Behaviour of Trees

Before a tree is created, its specifications must be provided. These specifications decide the structure, properties and behaviour of the tree. The specifications are represented by Spec objects, and this part of the tutorial will present how to use them. Since the Spec class is purely abstract, we have to use objects of derived classes. In this example we'll use a *MarbleExpansion* object, which simply adds children of type Marble to the target object. Details of this expansion must be provided to the constructor of the MarbleExpansion object as demonstrated in line 7 of Listing 25. The first argument is a vector of strings, representing the typeIDs of the Marble objects where the Spec object should be executed. This argument is needed for all Spec objects. If one wants to designate the Spec object to several Marble objects with different typeIDs, one would create the Spec object as demonstrated in Listing 27. The next argument is an integer which indicates how many children which should be created. Furthermore, the last argument is a string which will be the typeID for the new children. Just like this MarbleExpansion object, all Spec object should be created using dynamic allocation. The next step is to add the Spec object to the system object. This is done using the *AddSpec* method, as demonstrated in line 8 of listing 25. Finally, the Traverse method is called to make sure the Spec object is executed.

To get an overview of a tree one has created, one can call either the *PrintTreeDF* or *PrintTreeBF* method. They will print the path, the typeID and the properties of the objects comprising the tree to stdout. The former will do so using depth first traversal of the tree, and the former using breadth first. In Listing 25, PrintTreeDF is called, resulting in the output presented in Listing 26. One may notice that the system object has a property called "ChildRoot", this property is explained in Section 8.15.

```cpp
// file "tutorial/spec_tut.cpp"
#include "../lib/libmarble.h"

int main() {

    Marble system(nullptr, "System");
    Spec* mexp = new MarbleExpansion({"System"}, 2, "Child");
    system.AddSpec(mexp);
    system.Traverse();

    system.PrintTreeDF();

    return 0;
}
```

Listing 25: Using a MarbleExpansion Spec object

```
0 0  Child
0 1  Child
0    System  Properties: ('ChildRoot', 0x5605850b1270)
```

Listing 26: The Output of the MarbleExpansion Example

```cpp
Spec* mexp = new MarbleExpansion({"Marble 1", "Marble 2",
    "Marble 3"}, 1, "Child");
```

Listing 27: Designating a Spec object for Marble objects with different typeIDs

8.4 Generating Trees Using Expansion Objects

In the previous section, we used MarbleExpansion Spec objects to build a tree consisting of a system object which had two child objects. However, MarbleExpansion objects can be used to create bigger trees, as big as you want them. Listing 28 presents code which creates a tree consisting of three layers, in addition to the system object. One layer for the "Trunk", one for the "Branches" and one for the "Leaves". The PrintTreeDF method can be used to get an overview of the resulting tree. However, the next section will present how this tree can be drawn using *Marbleviz*.

```
1  // file "tutorial/tree_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6      Marble system(nullptr, "System");
7
8      Spec* trunkexp = new MarbleExpansion({"System"}, 1, "Trunk");
9      system.AddSpec(trunkexp);
10
11     Spec* branchexp = new MarbleExpansion({"Trunk"}, 2, "Branch");
12     system.AddSpec(branchexp);
13
14     Spec* leafexp = new MarbleExpansion({"Branch"}, 4, "Leaf");
15     system.AddSpec(leafexp);
16
17     system.Traverse();
18
19     system.PrintTreeDF();
20
21     return 0;
22 }
```

Listing 28: Code which generates a three-layered tree

8.5 Drawing Trees Using Marbleviz

As an application of he Marble library, the *Marbleviz* application has been developed with which one can draw trees (this application is presented in Section 9.3). In order to do so, one has to call the *MarblevizDraw* method, as demonstrated at line 18 in Listing 29. Here, Spec objects are added such that when the Traverse method is called, DOT language code representing the tree is printed to stdout. When calling MarblevizDraw, one has to pass the system object as an argument, followed by a vector of the typeIDs of the Marble objects one wishes to include in the drawing. The last argument is optional and is the title of the drawing.

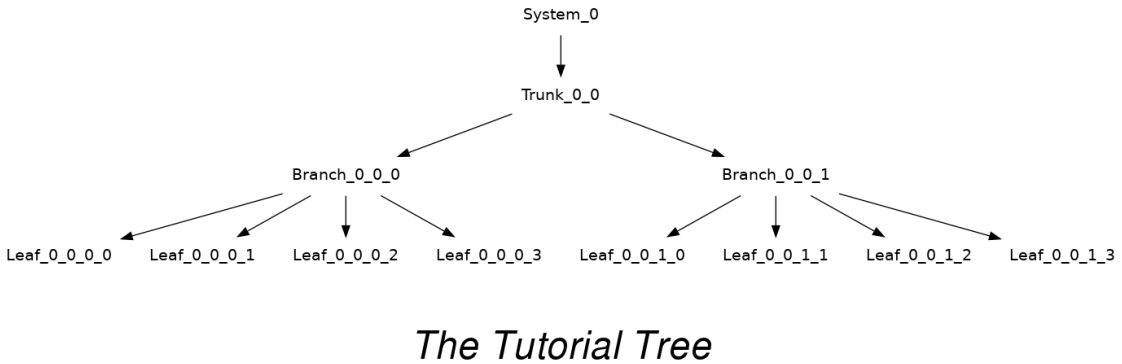

Figure 3: The Tutorial Tree, drawn using Graphviz

```
// file "tutorial/tree_mv_tut.cpp"
#include "../lib/libmarble.h"
#include "../marbleviz/marbleviz.h"

int main() {

  Marble system(nullptr, "System");

  Spec* trunkexp = new MarbleExpansion({"System"}, 1, "Trunk");
  system.AddSpec(trunkexp);

  Spec* branchexp = new MarbleExpansion({"Trunk"}, 2, "Branch");
  system.AddSpec(branchexp);

  Spec* leafexp = new MarbleExpansion({"Branch"}, 4, "Leaf");
  system.AddSpec(leafexp);

  MarblevizDraw(system, {"System", "Trunk", "Branch", "Leaf"},
      "The Tutorial Tree");

  system.Traverse();

  return 0;
}
```

Listing 29: Drawing a tree using Marbleviz

The DOT code printed to stdout can be piped to a GV file which Graphviz (graph visiualization software) can use to create a PNG file depicting the tree. E.g. using the following terminal command (ubuntu):

```
$ make tree_mv_tut
$ ./tree_mv_tut > tree.gv && dot -Tpng tree.gv -o tree.png
```

Listing 30: The commands which generate the PNG file depicting the tree

Remember that the *LD_LIBRARY_PATH* environment variable must be set and exported before running the executable (see Section 8.1). Graphviz must also be installed. The resulting tree drawing is presented in figure 3.

8.6 Generating Asymmetrical Trees

So far, we've only generated symmetrical tress. However, the Marble library can also be used to create asymmetrical trees. Listing 31 demonstrated how this could be done. Here, the system object is expanded with Marble objects with typeIDs "Marbletown" and "Marbleburg" using two separate MarbleExpansion Spec objects. Three additional MarbleExpansion Spec objects are then used to further expand the tree asymmetrically. The resulting tree is presented in Figure 4.

```
// file "tutorial/asym_tut.cpp"
#include "../lib/libmarble.h"
#include "../marbleviz/marbleviz.h"

int main() {

  Marble system(nullptr, "System");

  Spec* mt = new MarbleExpansion({"System"}, 1, "Marbletown");
  system.AddSpec(mt);

  Spec* nm = new MarbleExpansion({"System"}, 1, "Marbleburg");
  system.AddSpec(nm);

  Spec* ms = new MarbleExpansion({"Marbletown"}, 1,
      "Marble_Square");
  system.AddSpec(ms);

  Spec* mc = new MarbleExpansion({"Marbletown"}, 1,
      "Marble_Castle");
  system.AddSpec(mc);

  Spec* mst = new MarbleExpansion({"Marble_Square"}, 1,
      "Marble_Street");
  system.AddSpec(mst);

  MarblevizDraw(system, {"System", "Marbletown", "Marbleburg",
      "Marble_Square", "Marble_Castle", "Marble_Street"},
      "The Marble Republic");

  system.Traverse();

  return 0;
}
```

Listing 31: Code generating an asymmetric tree of Marble Objects

8.7 How to Use Advanced Spec Object Designation

Until now, we've designated Spec objects by means of the typeIDs of the Marble objects where the Spec object should be executed. However, there may be cases where one wants to base Spec object designations on other properties or values. Therefore, the Spec object constructor has an optional function pointer parameter. Such a function returns a boolean value and takes two pointers as parameters. One

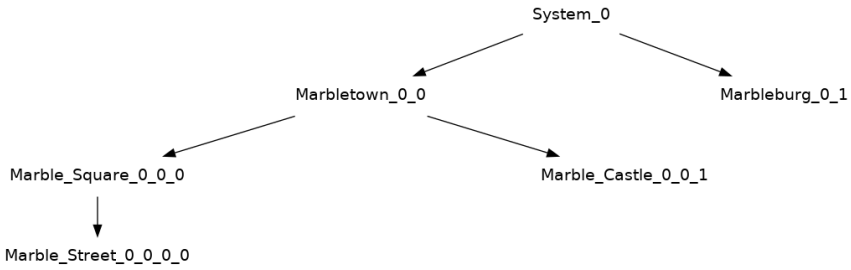

Figure 4: An asymmetric tree of Marble objects, drawn using Graphviz

pointer to a Marble object and one to the Spec object in question. The returned boolean indicates whether or not the Spec object should be executed at the Marble object which the Marble pointer points to (true if it should be executed).

Listing 32 presents how advanced designation should be used. The *apple* and *car* expansion Spec objects have the *BuyApple* and *BuyCar* functions as advanced designations functions, respectively. This means that the apple object will only be executed at *system* if system's *dollars* property is more than 1. The car object however, will only be executed if the dollars property is above 10000. An IntegerProperty named "dollars" with value 20 is added to the system object, meaning only the apple object should be executed. This is apparent in Figure 5, which is the resulting tree drawn in Graphviz.

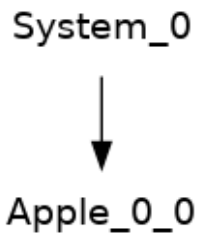

Figure 5: Tree created using advanced designation, drawn using Graphviz

```
// file "tutorial/advdes_tut.cpp"
#include "../lib/libmarble.h"
#include "../marbleviz/marbleviz.h"

bool BuyApple(Marble* pMaster, Spec* pSpec) {
  if (pMaster->GetTypeID() != "System") { return false; }
  int dollars = pMaster->GetPropValue<int, IntegerProperty>(
      "dollars");

  return dollars > 1;
}

bool BuyCar(Marble* pMaster, Spec* pSpec) {
  if (pMaster->GetTypeID() != "System") { return false; }
  int dollars = pMaster->GetPropValue<int, IntegerProperty>(
      "dollars");

  return dollars > 10000;
}

int main() {

  Marble system(nullptr, "System");

  Property* dollars = new IntegerProperty("dollars", 20);
  system.AddProperty(dollars);

  Spec* apple = new MarbleExpansion({}, 1, "Apple", BuyApple);
  system.AddSpec(apple);

  Spec* car = new MarbleExpansion({}, 1, "Car", BuyCar);
  system.AddSpec(car);

  MarblevizDraw(system, {"System", "Apple", "Car"});

  system.Traverse();

  return 0;
}
```

Listing 32: Designating Spec objects using advanced designation

8.8 Adding Properties to a Marble Object

Just like Spec objects, Property objects have to be allocated dynamically, using the *new* operator. Constructor arguments which must be provided vary between Property derived classes, but they all need the *identifier* parameter, which is the first in the parameter list. Line 7 in Listing 33 presents the creation of a *StringProperty*, one of the Property types provided by the library. The value of the StringProperty must be provided to the constructor, which is "Bob" in this case. Next, the property is added to the system object using the *AddProperty* method. Since no Spec objects have been added, there is no need to call the Traverse method. Note that this method can only be used to add properties to the system object, since one doesn't have explicit access to other potential objects in the tree. The next section describes how properties are added to any tree object.

```
1  // file "tutorial/prop_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6      Marble system(nullptr, "System");
7      Property* name = new StringProperty("name", "Bob");
8      system.AddProperty(name);
9
10     system.PrintTreeDF();
11
12     return 0;
13 }
```

Listing 33: Adding a StringProperty object to the system object

8.9 Adding Properties to Tree Objects

The AddProperty method can only be used to add properties to Marble objects when their pointers are explicitly available. Since there isn't explicit access to other potential objects in the tree, a Spec object is necessary. The Spec derived class to use is the *PropertySpec* class. How this class should be used is presented in Listing 34. In addition to the vector of typeIDs for its targets, the PropertySpec constructor takes a second parameter which is a vector of pointers to properties the target Marble objects should receive. In this case, the Property object which the PropertySpec object should carry is the *childstr* StringProperty, as demonstrated in line 12 of 34. After the PropertySpec object is created, it must be added to the system object using the AddSpec method, followed by a call to the Traverse method. As listing 35 presents (the output of the PrintTreeDF call), the Marble object with typeID "Child" has successfully received the property.

```
1  // file "tutorial/propspec_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6      Marble system(nullptr, "System");
7      Spec* mexp = new MarbleExpansion({"System"}, 1, "Child");
8      system.AddSpec(mexp);
9
10     Property* childstr = new StringProperty("childstr",
11         "I am a child.");
12     Spec* propspec = new PropertySpec({"Child"}, {childstr});
13     system.AddSpec(propspec);
14
15     system.Traverse();
16
17     system.PrintTreeDF();
18
19     return 0;
20 }
```

Listing 34: Adding properties to tree objects using the PropertySpec class

```
1 0 0  Child   Properties: ('childstr', 'I am a child.')
2 0    System  Properties: ('ChildRoot', 0x559f5face380)
```

Listing 35: The Output of the PrintTreeDF call in Listing 34

8.10 How to Use Dynamic Properties

Dynamic properties allow Marble objects to access and store properties owned by their ancestors. In Listing 36, the system object is given the StringProperty with the value "Bob". In addition, it's expanded with two child objects. At line 14 however, a DynamicProperty is created, the first argument of the constructor naturally being the identifier of the property. The second argument on the other hand, is the identifier which should be searched for among the target Marble object's ancestors. In this example, the goal is to search for a property with the "name" identifier. The dynamic property is then attached to a PropertySpec object with Marble objects with the "Child" typeID as its target. Since the system object has a "name" property, a cloned version of this property will be stored at the target objects, now with the "parentName" identifier. Listing 37 presents the output of the PrintTreeDF call, where the "Child" objects now have the "parentName" property.

```
1  // file "tutorial/dynprops_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6      Marble system(nullptr, "System");
7
8      Property* name = new StringProperty("name", "Bob");
9      system.AddProperty(name);
10
11     Spec* mexp = new MarbleExpansion({"System"}, 2, "Child");
12     system.AddSpec(mexp);
13
14     Property* parentName = new DynamicProperty("parentName",
15         "name");
16     Spec* pnspec = new PropertySpec({"Child"}, {parentName});
17     system.AddSpec(pnspec);
18
19     system.Traverse();
20
21     system.PrintTreeDF();
22
23     return 0;
24 }
```

Listing 36: How to use dynamic properties

```
1 0 0   Child    Properties: ('parentName', Bob)
2 0 1   Child    Properties: ('parentName', Bob)
3 0     System   Properties: ('ChildRoot', 0x55e7d6ca5550), ('name',
    Bob)
```

Listing 37: The output of the PrintTreeDF call in listing 36

8.11 How to Use Calculating Properties

Calculating properties are properties which are based on some calculation applied on its owner. Calculating properties are created using the *CalculateProperty* class, as demonstrated at line 20 in Listing 38. The second argument of its constructor is a function pointer. The function the pointer points to is responsible for creating a Property object based on some calculation. The created Property object will be added to the Marble object. The function must also take a pointer to a Marble object as a parameter such that calculations can be applied to the Marble object's member variables and properties. As the *CalcName* function in Listing 38 suggests, a unique name for the Marble object is calculated, based on its typeID and path. The type of property which is returned could be any, in this case it's a StringProperty. Methods from the Marble class that are relevant for use when calculating properties are listed in Listing 43. The resulting output of the PrintTreeDF call is presented in Listing 39, where each Marble object has received a StringProperty, calculated by the CalcName function, representing a unique name.

```cpp
// file "tutorial/calcprops_tut.cpp"
#include "../lib/libmarble.h"

Property* CalcName(Marble* pMaster) {
    std::vector<int> path = pMaster->GetPath();
    std::string name = pMaster->GetTypeID();
    for (int i : path) {
        name += "_" + std::to_string(i);
    }
    return new StringProperty("name", name);
}

int main() {

    Marble system(nullptr, "System");

    Spec* mexp = new MarbleExpansion({"System"}, 2, "Child");
    system.AddSpec(mexp);

    Property* nameprop = new CalculateProperty("name", CalcName);
    Spec* namespec = new PropertySpec({"System", "Child"},
        {nameprop});
    system.AddSpec(namespec);

    system.Traverse();

    system.PrintTreeDF();

    return 0;
}
```

Listing 38: How to use calculating properties

```
0 0  Child   Properties: ('name', Child_0_0)
0 1  Child   Properties: ('name', Child_0_1)
0    System  Properties: ('ChildRoot', 0x564f4207e450), ('name',
   System_0)
```

Listing 39: The output of the PrintTreeDF call in listing 38

8.12 Triggers: Functions Executed by Marble Objects When Traversed

Triggers are functions which the Marble objects in a tree call when the Traverse method is called. A trigger is added to a Marble object using the *AddTrigger* method, as demonstrated in Listing 40. Trigger functions must return void and take a pointer to a Marble object as a parameter. This way, the trigger can access Marble objects' member variables and properties. Listing 40 presents how "Hello world!" can be printed using a trigger. After the trigger has been added to the Marble object, the Traverse method is called, in which the triggers of all Marble objects in the tree are executed.

```cpp
// file "tutorial/triggers_tut.cpp"
#include "../lib/libmarble.h"

void Trigger(Marble* pMaster) {
    std::cout << "Hello world!" << std::endl;
}

int main() {

    Marble system(nullptr, "System");
    system.AddTrigger(Trigger);
    system.Traverse();

    return 0;
}
```

Listing 40: How to add a trigger to a Marble object

8.13 Adding Triggers to Tree Objects

As with properties, a Spec object must be used to add triggers to Marble objects in a tree, since there isn't direct access to them. Therefore, the *TriggerSpec* class is used, as demonstrated in Listing 41. In this example, a tree is expanded with Marble objects like in Listing 25. Then, a TriggerSpec object is created, passing the pointer to the *Trigger* function to the constructor. Then, the TriggerSpec object is added to the system object and the Traverse method is called. Note that in this example, the trigger uses the *pMaster* pointer, which points to the owner of the trigger. A wide variety of useful triggers can be created by using this pointer. Methods from the Marble class which are relevant for triggers are listed in Listing 43. When the Traverse method is called, the tree is expanded, the triggers are added and subsequently called. The resulting output is presented in Listing 42.

```cpp
// file "tutorial/triggerspec_tut.cpp"
#include "../lib/libmarble.h"

void Trigger(Marble* pMaster) {
    std::cout << pMaster->GetTypeID() << " calling trigger" <<
        std::endl;
}

int main() {

    Marble system(nullptr, "System");

    Spec* child1 = new MarbleExpansion({"System"}, 1, "Child 1");
    system.AddSpec(child1);

    Spec* child2 = new MarbleExpansion({"System"}, 1, "Child 2");
    system.AddSpec(child2);

    Spec* trigspec = new TriggerSpec({"Child 1", "Child 2"},
        Trigger);
    system.AddSpec(trigspec);

    system.Traverse();

    return 0;
}
```

Listing 41: How to use the TriggerSpec class

```
Child 1 calling trigger
Child 2 calling trigger
```

Listing 42: The output of the Traverse method in listing 41

```cpp
std::map<int, Marble*>& GetChildren();
std::vector<int> GetPath();
const std::string GetTypeID();
std::map<std::string, Property*> GetProperties();
Property* GetProperty(std::string id);
Marble* GetParentPointer();
// T1 is the expected return value for the Property (e.g. int for
   IntegerProperty). T2 is the Property type (e.g. IntegerProperty)
template<typename T1, typename T2>
T1 GetPropValue(std::string id);
int RandomInteger1(int min, int max);
int RandomInteger2(int min, int max);
```

Listing 43: Methods from the Marble class which are relevant for triggers and calculating properties

8.14 Final Triggers

In some cases, certain configurations of a tree can't be done until the traversal is completed and the tree is fully generated. Consider the case where one wants to add a Spec object to the last object that is being traversed, i.e. the last object encountered in a breadth first traversal of the tree. Such a Spec object can't be designated the way others are during traversal because the tree is still being generated. In other words, the Spec object can't be added before the tree is fully generated. This is where final triggers enter the picture. They can only be added to and called by the system object, and they are called after the traversal is completed. This way, one can add a final trigger which adds a Spec object to the object which is last in traversal.

Listing 44 presents how this can be accomplished. Here, by means of a final trigger, the goal is to add a PropertySpec object to the last "Child" object which is traversed. The approach used in this example is to count the amount of objects with the typeID "Child" in the tree. Then, the PropertySpec object is added using an advanced designation function which counts how many "Child" objects it has encountered. As the PropertySpec object traverses the tree, the advanced designation function returns true when its counter has matched the amount of "Child" objects in the tree.

Firstly, the system object is expanded with two objects with typeID "Child". Then, *FinalTrigger* is added using the *AddFinalTrigger* method. The FinalTrigger function adds *CountingTrigger*, which counts the amount of "Child" objects in the tree, as well as a PropertySpec object to the Marble object that's traversed lastly. Since the final trigger calls Traverse, the calls must be guarded by conditions to prevent infinite recursion.

```cpp
// file "tutorial/final_triggers_tut.cpp"
#include "../lib/libmarble.h"

int childCounter = 0;

void CountingTrigger(Marble* pMaster) {
  childCounter++;
}

bool AdvDes(Marble* pMaster, Spec* pSpec) {
  if (pMaster->GetTypeID() != "Child") { return false; }
  static int childrenTraversed = 0;
  childrenTraversed++;
  return childrenTraversed == childCounter;
}

void FinalTrigger(Marble* pMaster) {
  // Counting "Child" objects
  static bool firstCall1 = true;
  if (firstCall1) {
    firstCall1 = false;
    Spec* triggerspec = new TriggerSpec({"Child"},
        CountingTrigger);
    pMaster->AddSpec(triggerspec);
    pMaster->Traverse();
  }

  // Adding a StringProperty to the last "Child" object being
     traversed
  Property* sp = new StringProperty("sp",
      "I am the last child being traversed");
  Spec* spspec = new PropertySpec({}, {sp}, AdvDes);
  static bool firstCall2 = true;
  if (firstCall2) {
    firstCall2 = false;
    pMaster->AddSpec(spspec);
    pMaster->Traverse();
  }
}

int main() {

  Marble system(nullptr, "System");

  Spec* mexp = new MarbleExpansion({"System"}, 2, "Child");
  system.AddSpec(mexp);

  system.AddFinalTrigger(FinalTrigger);

  system.Traverse();

  system.PrintTreeDF();

  return 0;

}
```

Listing 44: How to use final triggers

8.15 Trees Generating Trees: Adding Spec Objects From Within Another Tree

The Marble library enables the user to create trees which generates other trees. To understand how this works, how the library recognizes distinct trees within the system object will be explained. In Listing 45, the system object is expanded with two Marble objects which are meant to be the roots of separate trees. The *SecondTree* Marble object is also expanded with a child called "SecondTreeChild". Whenever the system object is expanded, it receives a *RootProperty*, which is a pointer to the root of a tree. In this case, the system object will get two RootProperties, one called "FirstTreeRoot" and one called "SecondTreeRoot", as the PrintTreeDF call shows in Listing 46. This way, dynamic properties lets any Marble object access all root objects in the system.

```
1  #include "../lib/libmarble.h"
2
3  int main() {
4
5    Marble system(nullptr, "System");
6
7    Spec* firsttree = new MarbleExpansion({"System"}, 1,
8        "FirstTree");
9    system.AddSpec(firsttree);
10
11   Spec* secondtree = new MarbleExpansion({"System"}, 1,
12       "SecondTree");
13   system.AddSpec(secondtree);
14
15   Spec* secondtreechild = new MarbleExpansion({"SecondTree"}, 1, "
       SecondTreeChild");
16   system.AddSpec(secondtreechild);
17
18   system.Traverse();
19
20   system.PrintTreeDF();
21
22   return 0;
23 }
```

Listing 45: Expanding the system object with two roots

```
0 0   FirstTree
0 1   SecondTree
0     System      Properties: ('FirstTreeRoot', 0x55fe9b1827c0), ('
    SecondTreeRoot', 0x55fe9b182ad0)
```

Listing 46: The output of the PrintTreeDF call in listing 45

Next, it will be presented how the first tree can generate the second one by extending the code in Listing 45 such that the "FirstTree" object adds a Spec object to the

"SecondTree" object, by means of a trigger. This code is presented in Listing 47. The trigger adds a Spec object to the second tree using the *AddExternalTreeSpec* method. The first parameter of this method is the name of the Marble object where the Spec object should be added, in this case "SecondTree". This Marble object needs to be a tree root, as the method will search for a property at the system object called "SecondTreeRoot". The resulting tree, drawn in Graphviz, is presented in Figure 6.

```
1  // file "tutorial/ext_tree_specs_tut.cpp"
2  #include "../lib/libmarble.h"
3  #include "../graphviz/graphviz.h"
4
5  void Trigger(Marble* pMaster) {
6    int anyNumber = 1;
7    Spec* exp = new MarbleExpansion({"SecondTreeChild"},
8        anyNumber, "ChildFromTrigger");
9    pMaster->AddExternalTreeSpec("SecondTree", exp);
10 }
11
12 int main() {
13
14   Marble system(nullptr, "System");
15
16   Spec* firsttree = new MarbleExpansion({"System"}, 1,
17       "FirstTree");
18   system.AddSpec(firsttree);
19
20   Spec* secondtree = new MarbleExpansion({"System"}, 1,
21       "SecondTree");
22   system.AddSpec(secondtree);
23
24   Spec* trigger = new TriggerSpec({"FirstTree"}, Trigger);
25   system.AddSpec(trigger);
26
27   Spec* secondtreechild = new MarbleExpansion({"SecondTree"},
28       1, "SecondTreeChild");
29   system.AddSpec(secondtreechild);
30
31   GraphvizDraw(system, {"System", "FirstTree", "SecondTree",
32       "SecondTreeChild", "ChildFromTrigger"});
33
34   system.Traverse();
35
36   return 0;
37 }
```

Listing 47: Adding a Spec object to an external tree

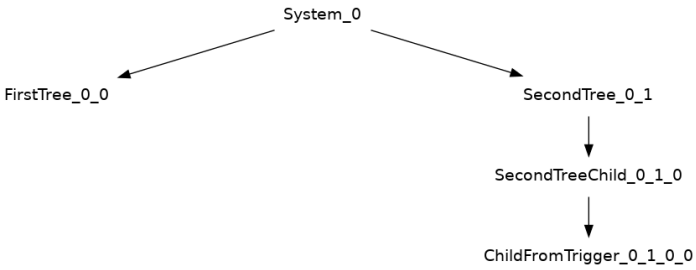

Figure 6: Tree created using the AddExternalTreeSpec method, drawn using Graphviz

8.16 The Output Object

So far, all trees have been built using only objects of the Marble class. But the library comes with another class derived from the Marble class, called the *Output* class. As the name implies, its purpose is to print strings to the console when the tree is being traversed. This feature is useful in many contexts, the Marbleviz application (presented in Section 9.3) uses Output objects to print the DOT code which Graphviz uses to draw trees. As an example, the code in Listing 48 builds a tree that, by means of an Output object, prints "Hello world!" when traversed. Note that a vector containing the strings which an Output object should print, is passed to the constructor of the *OutputExpansion* object which is responsible for creating the Output object, as demonstrated at line 8.

```
1  // file "tutorial/output_1_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6    Marble system(nullptr, "System");
7
8    Spec* oexp = new OutputExpansion({"System"}, 1,
9        {"Hello world!\n"});
10   system.AddSpec(oexp);
11
12   system.Traverse();
13
14   return 0;
15 }
```

Listing 48: A tree printing "Hello wolrd!" when traversed

In many cases however, it might be necessary to print properties which aren't explicitly accessible when creating the OutputExpansion object. Therefore, Output objects can be created to print dynamic properties. Listing 49 is Listing 48 extended with a StringProperty being added to the system object. Furthermore, line 12 of Listing 49 demonstrates how an Output object prints dynamic strings. The identifier of the string one wishes to print is prepended with "^" ("^systemstr" in this case) and when the Output object's constructor is called, the corresponding property is retrieved from the Output object's ancestors. Proving that an Output object can print dynamic strings, the following is printed to the console when the

Traverse method is called:

```
Output: This string belongs to system
```

```
1  // file "tutorial/output_2_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  int main() {
5
6    Marble system(nullptr, "System");
7
8    Property* systemstr = new StringProperty("systemstr",
9        "This string belongs to system");
10   system.AddProperty(systemstr);
11
12   Spec* oexp = new OutputExpansion({"System"}, 1,
13       {"Output: ^systemstr\n"});
14   system.AddSpec(oexp);
15
16   system.Traverse();
17
18   return 0;
19 }
```

Listing 49: An Output object printing a dynamic string when traversed

8.17 Building Trees Using Custom Objects

Naturally, it's possible to create classes deriving from the Marble class and build trees out of them. One of the benefits of this is that one can add necessary member variables and methods to the derived class. In addition, one could e.g. add properties and triggers in the derived class's constructor in stead of in the main function, resulting in tidier code.

When creating a class derived from the Marble class, its constructor must call the Marble constructor, as demonstrated at line 6 in Listing 50. A corresponding Expansion class for the newly created Marble derived class must also be created if one wishes to create objects of this class via a Spec object. The constructor of this class must call the constructor of the Expansion class as demonstrated in line 14 in Listing 50. In addition, the *CreateChild* method must be declared and defined. The definition should simply be to dynamically allocate an object of the newly created Marble derived class and returning the pointer to it, like demonstrated at line 16. The contents of the main function in Listing 50 simply shows how the new classes can be used.

```
1  // file "tutorial/custom_objs_tut.cpp"
2  #include "../lib/libmarble.h"
3
4  class Custom : public Marble {
5  public:
6    Custom(Marble* pParent) : Marble(pParent, "Custom") {}
7  };
8
9  class CustomExpansion : public Expansion {
10 public:
11   CustomExpansion(std::vector<std::string> targetTypeIDs,
12       const int numNewChildren,
13       bool (*AdvDes)(Marble*, Spec*) = nullptr)
14     : Expansion(targetTypeIDs, numNewChildren, AdvDes) {};
15   Marble* CreateChild(Marble* pMaster) const {
16       return new Custom(pMaster);
17   }
18 };
19
20 int main() {
21
22   Custom c(nullptr);
23
24   Spec* cexp = new CustomExpansion({"Custom"}, 1);
25   c.AddSpec(cexp);
26
27   c.Traverse();
28
29   c.PrintTreeDF();
30
31   return 0;
32
33 }
```

Listing 50: Creating derived classes of the Marble and Expansion classes

8.18 Generating Random Trees

The Marble class has two functions which return random integers. They are called *RandomInteger1* and *RandomInteger2*, and they are using different seeds. These functions can be used anywhere one sees fit. In the example in Listing 51, they are used in the constructor of the Custom class (from Listing 50) to create a random amount of children and a random IntegerProperty. When using the random function, it's important to set the seeds in the constructor of the system object, as demonstrated at line 33 in Listing 51. Listing 52 contains the output of the PrintTreeDF call from Listing 51. Here, one can see the IntegerProperty which was assigned a random integer value. Two child objects were also randomly created (the minimum and maximum bounds were 0 and 2, respectively).

```cpp
// file "tutorial/seeds_tut.cpp"
#include "../lib/libmarble.h"

class Custom : public Marble {
public:
  Custom(Marble* pParent);
};

class CustomExpansion : public Expansion {
public:
  CustomExpansion(std::vector<std::string> targetTypeIDs,
      const int numNewChildren,
      bool (*AdvDes)(Marble*, Spec*) = nullptr)
      : Expansion(targetTypeIDs, numNewChildren, AdvDes) {};
  Marble* CreateChild(Marble* pMaster) const {
      return new Custom(pMaster);
  }
};

Custom::Custom(Marble* pParent) : Marble(pParent, "Custom") {
  int numChildren = RandomInteger1(0,2);
  Spec* children = new MarbleExpansion({"Custom"}, numChildren,
      "Child");
  AddSpec(children);

  int randint = RandomInteger2(0, 100);
  Property* randintprop = new IntegerProperty("randint", randint);
  AddProperty(randintprop);
}

int main() {

  Marble system(nullptr, "System", {5555, 4444});

  Spec* cexp = new CustomExpansion({"System"}, 1);
  system.AddSpec(cexp);

  system.Traverse();

  system.PrintTreeDF();

  return 0;

}
```

Listing 51: Generating random trees using random functions

```
0 0 0   Child
0 0 1   Child
0 0     Custom   Properties: ('randint', 29)
0       System   Properties: ('CustomRoot', 0x558dc9c9b720)
```

Listing 52: The output of the PrintTreeDF call in Listing 51

9 *Marbleviz*: An Application of the Marble Library

The purpose of this thesis was not only to develop the Marble library. The intention was also to develop two applications of the library in order to present the library's capabilities and how they can be used in a larger scale. The first application should let the user draw trees using Graphviz (graph visualization software). This application will from now on be referred to as *Marbleviz*.

9.1 Specifications

The specifications for the Marbleviz application is inspired by [2](p. 13-14). Graphviz is open source graph visualization software which can be used to draw trees created in the Marble library. Graphviz can draw trees which are specified in GV files. These GV files contain code written in the DOT language. In other words, to draw trees created in the Marble library, they must be translated into DOT code which can be piped to a GV file. This GV file can then be opened by Graphviz such that a PNG file depicting the tree can be generated. Therefore, the Marbleviz application should output DOT code representing a tree to the console. This way, the DOT code can be piped to a GV file.

To generate valid DOT code which can be opened by Graphviz, the first lines of DOT code must represent configuration settings about the graph. For instance, fontname, fontsize, image size and ratio should be specified. In addition, it should be possible to label the tree drawing with a title. The next step is to output the DOT code representing the tree structure. This should be done by outputting code for each individual arrow between the nodes. As an example, a tree consisting of a "Parent" object which owns two "Children" should look like this:

```
"Parent_0" -> "Child_0_0" [color=black]
"Parent_0" -> "Child_0_1" [color=black]
```

For this to work, each Marble object needs a unique name, as the DOT code snippet above implies. The easiest way to do this is to assign each Marble object a name which is the concatenation of a string representing their typeID (e.g. "Note") and their path in the tree (inspired by [2] (p. 13)). Finally, the Marbleviz application should have a simple user interface. Ideally, one would just call a single function which would add all necessary Spec objects to the root of the tree one wants to draw. Calling the Traverse method would then make sure the DOT code would be printed.

1. **The Marbleviz Application** should be developed, supporting the following features:

   (a) DOT code representing the tree to be drawn should be output to the console. This code must complete in the sense that it could be directly piped to a GV file which Graphviz can use to generate a PNG file. The DOT code should include:

      i. configuration details like font name, font size, image size and ratio, etc.

      ii. code representing each arrow in the tree. This implies that each Marble object needs to have a unique name. The name should be a string concatenated by their typeID and their path in the tree.

      iii. a closing bracket, which should be printed lastly.

   (b) The user interface should be as simple as calling a function which adds all necessary Spec objects to the root of the tree one wants to draw.

9.2 Implementation

Marbleviz specification 1b states that the user interface of the Marbleviz application should be as simple as a function which adds all necessary Spec objects. The function which satisfies this specification is the *MarblevizDraw* function. Listing 53 presents its definition. Here, satisfying Marbleviz specification 1a, Spec objects are added to the system object of the tree to draw. This way, when the tree is traversed, DOT code which can be used to draw the tree in Graphviz is printed to the console.

At line 8, the *dmoexp* object of type OutputExpansion is created. It's job is to expand the system object with an Output object which will output the strings in the *dotModelLines* vector. The strings in this global vector (presented in Listing 54 and collected from [2] (p. 30)) specify configuration settings about the graph to draw, e.g. fontsize, colors, etc. Note that at line 7 of Listing 54, a dynamic string with identifier "title" is present. The Output object, which the dmoexp Spec object will create, will therefore search it's ancestors for a StringProperty with the same identifier. Since the system object is the object's only ancestor however, the StringProperty *title* must be added to the system object, as lines 5-6 suggest. The value of this StringProperty, which will be the title of the tree drawing, is defined by the *titleStr* function parameter.

Since the Output object printing the configuration settings is added to the system object's children, this will be the first Output object to be traversed. This means that its trigger will be the first Output trigger to be executed. In other words, the configuration settings is the first DOT code which will be printed to the console. Marbleviz specification 1(a)i is therefore satisfied.

```
// file "marbleviz/marbleviz.cpp"
void MarblevizDraw(Marble& system,
    std::vector<std::string> types, std::string titleStr) {

  StringProperty* title = new StringProperty("title", titleStr);
  system.AddProperty(title);

  OutputExpansion* dmoexp = new OutputExpansion(
      {system.GetTypeID()}, 1, dotModelLines);
  system.AddSpec(dmoexp);

  CalculateProperty* name = new CalculateProperty("name",
      NameFunc);
  PropertySpec* namespec = new PropertySpec(types, {name});
  system.AddSpec(namespec);

  DynamicProperty* parentname = new DynamicProperty("parentName",
      "name");
  PropertySpec* parentnamespec = new PropertySpec(types,
      {parentname});
  system.AddSpec(parentnamespec);

  DotArrowExpansion* daexp = new DotArrowExpansion(types, 1);
  system.AddSpec(daexp);

  system.AddFinalTrigger(FinalTrigger);

}
```

Listing 53: The MarblevizDraw function

```
// file "marbleviz/marbleviz.h"
const std::vector<std::string> dotModelLines = {
  "digraph \"unix\" {\n",
  " graph [ \n",
  " fontname = \"Helvetica-Oblique\",\n",
  " fontsize = 36,\n",
  " label = \"\n^title\",\n"
  " size = \"100,100\" \n",
  " ratio = 0.25,\n",
  " ];\n",
  " node [\n",
  " shape = polygon,\n",
  " sides = 4,\n",
  " distortion = \"0.0\",\n",
  " orientation = \"0.0\",\n"
  " skew = \"0.0\",\n",
  " color = white,\n",
  " style = filled,\n",
  " fontname = \"Helvetica-Outline\" \n",
  " ];\n",
};
```

Listing 54: The dotModelLines vector

The next step is to create and add Spec objects such that DOT code, representing the arrows in the tree drawing, is printed to the console. To accomplish this, every Marble object in the tree needs to have a unique name, as stated in Marbleviz specification 1(a)ii. This way, DOT code representing the arrows can be generated. To assign unique names to all Marble objects, a CalculatingProperty is used. This property, called *name*, is created at line 12 of Listing 53. As the constructor parameter suggests, the property will be calculated by the *NameFunc* function, which is presented in Listing 55. This function creates a string which is concatenated from the target Marble object's typeID and its path, converted to a string. This string is then returned as a StringProperty. At line 14 of Listing 53, the *namespec* PropertySpec object is created. This Spec object will carry the name property to all Marble objects which have any of the typeIDs contained within *types* function parameter. In other words, this vector contains the typeIDs of all the Marble objects which should be included in the Graphviz drawing. Finally, at line 15, the namespec Spec object is added.

```
1 // file "marbleviz/marbleviz.cpp"
2 Property* NameFunc(Marble* pMarble) {
3   std::vector<int> path = pMarble->GetPath();
4   std::string name = pMarble->GetTypeID();
5   for (int i : path) {
6     name += "_" + std::to_string(i);
7   }
8   return new StringProperty("name", name);
9 }
```

Listing 55: The NameFunc function, used to calculate the name property

Next, in order for the Marble objects to output DOT code representing the arrows in the tree, they need to hold a property representing their parent's name as well. Therefore, at line 17 of Listing 53, the *parentName* property is created. This is a dynamic property, where the target Marble object will search it's ancestors for a property with the identifer "name", which have already been added. Now, every Marble object to be included in the drawing has a unique "name" property, as well as a property representing the name of their parent.

The next step is to expand each Marble object to be included in the drawing with a *DotArrow* object (inspired by [2] p. 30-31). The DotArrow class is derived from the Marble class, and objects of this class are intended to represent the arrows between the objects in the Graphviz drawing. The constructor of the DotArrow class is listed in Listing 56. Here, the dynamic properties *to* and *from* are created and added. They will hold the same value as their owner's name and parentName properties. The *color* StringProperty is also created and added. Lastly, the DotArrow object is expanded with an Output object. Notice the string passed to its constructor. This string is the DOT code which will amount to the arrow which points from the Marble object's parent to the Marble object itself. Three dynamic strings are included, from, to and color. This way, the Output object prints the DOT code for the DotArrow which owns it. Consequently, Marbleviz specification 1(a)ii is satisfied. The DotArrow also has a static integer member variable, *instanceCounter*, which the constructor increments. This is explained in the last paragraph of this section.

```
// file "marbleviz/marbleviz.cpp"
int DotArrow::instanceCounter;

DotArrow::DotArrow(Marble* pParent) : Marble(pParent, "DotArrow") {
  this->instanceCounter++;

  Property* to = new DynamicProperty("to", "name");
  AddProperty(to);

  Property* from = new DynamicProperty("from", "parentName");
  AddProperty(from);

  Property* color = new StringProperty("color", "black");
  AddProperty(color);

  OutputExpansion* oexp = new OutputExpansion({"DotArrow"}, 1,
      {" \"^from\" -> \"^to\" [color=^color]\n"});
  // The Output ctor searches dynamically for properties, they must
      be created lastly
  AddSpec(oexp);
}
```

Listing 56: The DotArrow constructor

For the printed DOT code to be directly piped to a GV file which Graphviz can open successfully, there is one thing that's missing. A closing bracket. This means that there is need for an Output object somewhere that's responsible for printing the bracket. However, this Output object must be added to the Marble object which is traversed lastly. If, not the bracket won't appear at the bottom of the printed DOT code. It's impossible to designate an OutputExpansion object for the Marble object which is traversed lastly, because the tree is still being generated. Therefore, we need to use a final trigger, a function which the system object calls when traversal is completed and the entire tree is generated. This trigger is listed in Listing 57. Here, the OutputExpansion object for the bracket output is created and added, before the Traverse method is called. There is a condition guarding the Traverse call to prevent infinite recursive calls.

```
// file "marbleviz/marbleviz.cpp"
void FinalTrigger(Marble* pMaster) {
  OutputExpansion* bracketoutput = new OutputExpansion({}, 1,
      {"}\n"}, DotArrowAD);
  pMaster->AddSpec(bracketoutput);
  static bool firstCall = true;
  if (firstCall) {
    firstCall = false;
    pMaster->Traverse();
  }
}
```

Listing 57: The final trigger

The OutputExpansion object created in Listing 57 uses advanced designation. The advanced designation function, the *DotArrowAD* function, is presented in Listing 58. Since the static instanceCounter member variable of the DotArrow class is incremented every time the constructor is called, it's value is the amount of DotArrow objects in the tree. The *DotArrowAD* function counts how many DotArrow objects it encounters. The objects are encountered in a breadth first fashion (just like the traversal), so when it has encountered DotArrow objects the same amount of times as the value of instanceCounter, the function has reached the DotArrow object which is traversed lastly. This way, the bracketouput object reaches its correct designation and the closing bracket is printed lastly. Therefore, Marbleviz specification 1(a)iii is satisfied.

```
1  // file "marbleviz/marbleviz.cpp"
2  bool DotArrowAD(Marble* pM, Spec* pS) {
3    static int dotArrowCounter = 0;
4    DotArrow* pDA = dynamic_cast<DotArrow*>(pM);
5    if (pDA != nullptr) {
6      dotArrowCounter++;
7      int numDotArrows = pDA->GetInstanceCounter();
8      if (dotArrowCounter == numDotArrows) {
9        dotArrowCounter = 0;
10       return true;
11     }
12     else { return false; }
13
14   }
15   else {
16     return false;
17   }
18 }
```

Listing 58: The advanced designation function for the bracketoutput object listing 57

9.3 Presenting the Marbleviz Application

To review the features of Marbleviz, the code presented in the Marbleviz tutorial section (Section 8.5) should be revisited. This code is relisted in Listing 59. Here, lines 7-16 create and add Expansion Spec objects. The resulting tree will consist of Marble objects with typeIDs "Trunk", "Branch" and "Leaf". To generate a PNG file depicting the tree, one has to call the MarblevizDraw function, as demonstrated in line 18. Here, the system object is passed by reference to the function such that the Marbleviz Spec objects can be added. The second parameter is a vector containing the typeIDs of the Marble objects one wishes to include in the tree drawing. The last parameter is the title one wishes to label the drawing with.

```
1  // file "tutorial/tree_mv_tut.cpp"
2  #include "../lib/libmarble.h"
3  #include "../marbleviz/marbleviz.h"
4
5  int main() {
6
7    Marble system(nullptr, "System");
8
9    Spec* trunkexp = new MarbleExpansion({"System"}, 1, "Trunk");
10   system.AddSpec(trunkexp);
11
12   Spec* branchexp = new MarbleExpansion({"Trunk"}, 2, "Branch");
13   system.AddSpec(branchexp);
14
15   Spec* leafexp = new MarbleExpansion({"Branch"}, 4, "Leaf");
16   system.AddSpec(leafexp);
17
18   MarblevizDraw(system, {"System", "Trunk", "Branch", "Leaf"},
19       "The Tutorial Tree");
20
21   system.Traverse();
22
23   return 0;
24 }
```

Listing 59: Drawing a Tree Using Marbleviz

When the Traverse method is called, the triggers of the Output objects added by the Marbleviz function will print DOT code to the console. The output of the code in Listing 59 is presented in Listing 60. Lines 1-19 contain the DOT code for the configuration settings. It's important that the configuration settings DOT code is printed first. Luckily, the order of which triggers are executed can be controlled. Recall that the Output object printing the configuration settings was added to the system object, meaning that it would be the first Output object to be traversed. It's also worth mentioning that at line 6, the user specified title has successfully replaced the dynamic string ("^title").

Lines 20-30 contain the DOT code representing the arrows to be drawn. These have been printed by the Output objects which belong to the DotArrow objects. Notice how the *parentName* properties have been retrieved successfully using dynamic properties. For instance, one of the Marble object has received the "Trunk_0_0" *name*

property (using a calculating property) and the "System_0" parentName property (using a dynamic property). This Marble object has been expanded with a DotArrow object, which copies the values of the name and parentName properties to its *to* and *from* properties (again using dynamic properties). Furthermore, the DotArrow owns an Output object which prints the DOT code representing the arrow. Line 20 in Listing 60 is the result.

It should also be noted that the closing bracket has successfully been printed lastly, at line 31. Recall that the bracket printing Output object was created using an OutputExpansion Spec object that used an advanced designation function. Since the advanced designation function had to locate the Marble object last in traversal, the OutputExpansion object had to be added using a final trigger.

```
1  digraph "unix" {
2   graph [
3   fontname = "Helvetica-Oblique",
4   fontsize = 36,
5   label = "
6  The Tutorial Tree",
7   size = "100,100"
8   ratio = 0.25,
9   ];
10  node [
11  shape = polygon,
12  sides = 4,
13  distortion = "0.0",
14  orientation = "0.0",
15  skew = "0.0",
16  color = white,
17  style = filled,
18  fontname = "Helvetica-Outline"
19  ];
20  "System_0" -> "Trunk_0_0" [color=black]
21  "Trunk_0_0" -> "Branch_0_0_0" [color=black]
22  "Trunk_0_0" -> "Branch_0_0_1" [color=black]
23  "Branch_0_0_0" -> "Leaf_0_0_0_0" [color=black]
24  "Branch_0_0_0" -> "Leaf_0_0_0_1" [color=black]
25  "Branch_0_0_0" -> "Leaf_0_0_0_2" [color=black]
26  "Branch_0_0_0" -> "Leaf_0_0_0_3" [color=black]
27  "Branch_0_0_1" -> "Leaf_0_0_1_0" [color=black]
28  "Branch_0_0_1" -> "Leaf_0_0_1_1" [color=black]
29  "Branch_0_0_1" -> "Leaf_0_0_1_2" [color=black]
30  "Branch_0_0_1" -> "Leaf_0_0_1_3" [color=black]
31 }
```

Listing 60: The DOT code output by the code in listing 29

As explained in the tutorial section, the following commands can be used to generate the PNG file depicting the tree:

```
$ make tree_mv_tut
$ ./tree_mv_tut > tree.gv && dot -Tpng tree.gv -o tree.png
```

Listing 61: The commands which generates the PNG file depicting the tree

Remember that the *LD_LIBRARY_PATH* environment must be set and exported

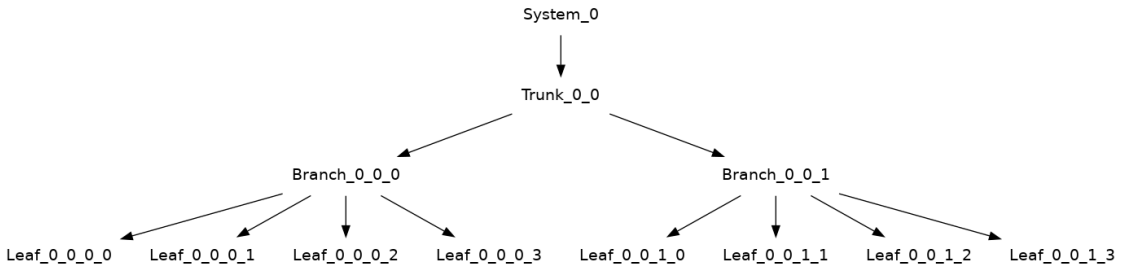

Figure 7: The Tutorial Tree, drawn using Graphviz

before running the executable (see Section 8.1). The resulting PNG file, named "tree.png", is presented in Figure 7.

10 *Random Lyrics Generating Melody*: An Application of the Marble Library

The second Marble library application developed in this thesis is in the context of musical procedural generation. Firstly, the purpose of this application is to generate a random tree of lyrics. The tree should be randomly generated to fit the context of procedural generation. Secondly, the lyrics tree should then generate a melody tree which "fits" the lyrics. The quotation marks are used because the intention is not to produce coherent music, but to prove that it could be accomplished by a more musically proficient programmer. A set of rules will be used to generate the melody, but the resulting melody won't be playable. The development of this application is especially important because it would prove the Marble library's ability to allow trees to generate the specifications for another tree while the trees are kept separate from each other. This way, tightly interacting trees won't have to be crammed together in some ad hoc solution because of the cluster of node-to-node dependencies between them. Their structures might also be so different that cramming them together wouldn't be possible.

The hierarchical properties of lyrics and melody means that they can effectively be represented as trees. A melody consists of lines which consist of bars which consist of notes. Lyrics consist of lines which consists of words which consist of syllables. If a melody should be generated to fit some lyrics, the melody will depend highly on the properties of the lyrics. The amount of lyric lines should match the amount of melody lines, the amount of syllables should match the amount of notes, bars should end on word endings, etc. In other words, the melody tree would be so tightly coupled to the lyrics tree that one would likely have no other choice than cramming them together in one tree. However, the structure of a lyrics tree might not fit very well with the structure of melody tree, so that might not even be possible at all.

In stead, using the Marble library, the trees can be kept separate while they're still allowed to interact, which would result in significantly more maintainable and tidier code. Just to recap, this is made possible by the fact that Spec objects are added to the roots of trees, before they are distributed throughout the tree. In other words, all dependencies from one tree node to a node in another tree is routed to the root. Namely, a tree could generate Spec objects and add them to another tree, using their access to the other tree's root.

Note that the concept of the melody "fitting" the lyrics is emphasized because it implies that the lyrics tree is in fact generating the Spec objects for the melody tree. If the lyrics changes, so will the melody.

10.1 Specifications

The lyrics tree should consist of a root object of type *Lyrics* (class derived from the Marble class). The Lyrics object should have children of type *LyricLine*, which should have children of type *Word* (inspired by [2] (p. 21-22)). When a Word object is created, a word must be picked from somewhere. Therefore, some pool of words must be established. Since the lyrics should be generated randomly, these words should, naturally, be picked at random. Additionally, the amount of LyricLine objects and Word objects to create should also be random.

Regarding the melody tree, the root object should be of type *Melody*. The children of the Melody object should be of type *Line*. The Line objects should have children of type *Bar*, which should have children of type *Note* (inspired by [2] (p. 7-8)). As previously mentioned, the purpose of this application isn't to generate coherent music. For simplicity's sake, three rules will therefore be followed when making the lyrics tree generate the melody tree (inspired by [2] (p. 24)):

1. The amount of Lines in the melody tree should equal the amount of LyricLines in the lyrics tree. This way, each Line in the melody tree corresponds to a LyricLine.
2. The number of Bars in a Line should be calculated based on the sum of the syllable weights in the corresponding LyricLine.
3. The amount of Notes in a Line should equal the amount of syllables in the corresponding LyricLine. They should be distributed to the Line's Bars randomly, for simplicity's sake.

The lyrics tree should generate the Spec objects necessary to satisfy these rules using triggers. As it stands, the full list of specifications are as follows:

2. **A Random Lyrics Tree Generating a Melody Tree**

   (a) The lyrics tree should consist of Marble objects of type Lyrics, LyricLine and Word.

   (b) A pool of words from which Word objects can pick their words must be established.

   (c) The words should be picked at random.

   (d) The amount of LyricLine and Word objects to create should be at random.

   (e) The melody tree should consist of Marble objects of type Melody, Line, Bar and Note.

   (f) These three rules should be followed when making the lyrics tree generate the melody tree:

      i. The amount of Lines in the melody tree should equal the amount of LyricLines in the lyrics tree. This way, each Line in the Melody tree corresponds to a LyricLine.

      ii. The number of Bars in a Line should be calculated based on the sum of the syllable weights in the corresponding LyricLine.

      iii. The amount of Notes in a Line should equal the amount of syllables in the corresponding LyricLine. They should be distributed to the Line's Bars randomly, for simplicity's sake.

   (g) The lyrics tree should generate the Spec objects necessary to satisfy these rules using triggers.

10.2 Implementation

10.2.1 Generating a Random Tree of Lyrics

First order of business is generating a random tree representing some lyrics to a song. To do so, a pool of words to draw from must be established (Specification 2b). The purpose of this example is not to construct overly complex lyrics. Therefore, for simplicity's sake, the lyrics are limited such that each sentence consists of an article, followed by a subject and a verb. E.g. "The Bear Eats.". In addition, each line of the lyrics can either consist of a single sentence, or two subsentences conjoined by a conjunction. E.g.: "The Bear Eats And The Snake Sleeps.". The pool of words consists of a few words in each word class, stored in global vectors. Listing 62 presents the vector containing the pool of subjects. The vector consists of pairs of a string and a vector. The string represents the word and the vector contains arbitrary weights of the word's syllables. Pairing the words with syllable weights is inspired by [2](p. 22), where a lyrics tree is also generated using such pairs. The syllable weights will have significance later when the melody tree is generated.

```
1 // file "lyrics/lyrics.h"
2 const std::vector<std::pair< std::string, std::vector<int>>>
     subjects = {
3   {"Bear", {2}},
4   {"Snake", {2}},
5   {"Scor--pi--on", {6,1,1}},
6   {"Tur--tle", {8,2}},
7   {"Spar--row", {5,1}},
8   {"Rab--bit", {3,1}},
9 };
```

Listing 62: The subjects which the lyrics tree draws from

As Specification 2a states, the lyrics tree should consist of objects of types Lyric, LyricLine and Word. To let the Word objects draw words from the pool of words, they need access to it. Therefore, the word vectors must be added to the tree as properties. This way, Word objects can access the word vectors using dynamic properties. Since the library contains no Property class which can hold vectors such as the one in Listing 62, the *WordsProperty* class (see Listing 63) was created, deriving from the Property class.

```
// file "lyrics/lyrics.cpp"
class WordsProperty : public Property {
private:
  const std::vector<std::pair<std::string, std::vector<int>>>
      words;
public:
  WordsProperty(std::string id,
      const std::vector<std::pair<std::string, std::vector<int>>>
         words);
  void PrintValue() const override;
  Property* Clone() const override;
  const std::vector<std::pair<std::string, std::vector<int>>>
      GetWords() const;
};
```

Listing 63: The WordsProperty class declaration

The lyrics tree to be generated will consist of the classes *Lyrics*, *LyricLine* and *Word*, which are all derived from the Marble class. An object of the Lyrics class will be the root of the tree. Listing 64 presents the Lyrics constructor. At lines 4-15, the vectors of words from all the word classes are added as WordsProperties. This way, Word objects can reach them using dynamic properties. As lines 18-20 suggest, the Lyrics constructor also creates and adds a *LyricLineExpansion* object, where the RandomInteger1 method is used such that a random amount of LyricLine objects are created, satisfying Specification 2d. To prevent the tree from reaching a size where drawing it with Marbleviz results in an overly large image, a maximum of three LyricLine objects are created

```
// file "lyrics/lyrics.cpp"
Lyrics::Lyrics(Marble* pParent) : Marble(pParent, "Lyrics") {
  // Adding the vectors of word classes as WordProperty objects.
  Property* articlesProp = new WordsProperty("articles", articles);
  AddProperty(articlesProp);

  Property* verbsProp = new WordsProperty("verbs", verbs);
  AddProperty(verbsProp);

  Property* subjectsProp = new WordsProperty("subjects", subjects);
  AddProperty(subjectsProp);

  Property* conjunctionsProp = new WordsProperty("conjunctions",
      conjunctions);
  AddProperty(conjunctionsProp);

  // Adding an Expansion object for a random amount of LyricLine
     children.
  int numLines = RandomInteger1(1,3);
  Spec* llexp = new LyricLineExpansion({"Lyrics"}, numLines);
  AddSpec(llexp);

}
```

Listing 64: The Lyrics constructor

Next, the LyricLine constructor (see Listing 65) uses the RandomInteger1 method to make sure that the number of Word objects per LyricLine is randomly set to either 3 or 7, constituting either a single sentence or two conjoined sub sentences.

```
1  // file "lyrics/lyrics.cpp"
2  LyricLine::LyricLine(Marble* pParent)
3      : Marble(pParent, "LyricLine") {
4    // Adding an Expansion object for either 3 or 7 Word children.
5    int numWords = RandomInteger1(0,1) ? 3 : 7;
6    Spec* wexp = new WordExpansion({"LyricLine"}, numWords);
7    AddSpec(wexp);
8  }
```

Listing 65: The LyricLine constructor

The Word constructor's (see Listing 66) first objective is to determine what word class it belongs to (articles, subjects, verbs or conjunctions). This is determined by the object's *pathID* member variable, i.e. the word's order in the sentence. Then, the object must retrieve the pool of words belonging to its word class. This is done using a dynamic property, as line 12 suggests. Since all the word pools are stored as WordsProperties at the Lyrics root object, adding a dynamic property using the word class as the search identifier, makes sure the same WordsProperty is stored at the current Word object. Onward from line 16, a word is randomly drawn from the object's pool of words, satisfying Specification 2c. The word which is drawn is stored in the Word objects *word* member variable, as line 20 suggests.

```
1  // file "lyrics/lyrics.cpp"
2  Word::Word(Marble* pParent) : Marble(pParent, "Word"), word() {
3
4    // Determining the word class for this Word object.
5    std::string wordClass = "";
6    if (pathID == 0 || pathID == 4) { wordClass = "articles"; }
7    else if (pathID == 1 || pathID == 5) { wordClass = "subjects"; }
8    else if (pathID == 2 || pathID == 6) { wordClass = "verbs"; }
9    else if (pathID == 3) { wordClass = "conjunctions"; }
10
11   // Retrieving the vector of this object's word class from the
        Lyrics root object.
12   Property* pProp = new DynamicProperty(wordClass, wordClass);
13   AddProperty(pProp);
14
15   // Picking a random word from the vector of words.
16   WordsProperty* pWordsProp = dynamic_cast<WordsProperty*>(
        properties[wordClass]);
17   if (pWordsProp != nullptr) {
18     const std::vector<std::pair<std::string, std::vector<int>>>
          words = pWordsProp->GetWords();
19     int wordIdx = RandomInteger2(0, words.size() - 1);
20     this->word = words.at(wordIdx);
21   }
22   else { std::cout << "Word::Word() property " << wordClass <<
23       " was not found" << std::endl; }
24
25 }
```

Listing 66: The Word constructor

10.2.2 The Lyrics Tree Generating a Melody Tree

Now that the functionality for generating a random lyrics tree is established, triggers generating the Spec objects for the melody tree can be added. As stated in Specification 2e, the melody tree will consist of objects of four classes. In hierarchical order, they are *Melody*, *Line*, *Bar*, and *Note*. Adhering to Specification 2f, when generating this tree, three rules will be followed:

1. The amount of Lines in the melody tree should equal the amount of LyricLines in the lyrics tree. This way, each Line in the Melody tree corresponds to a LyricLine.
2. The number of Bars in a Line should be calculated based on the sum of the syllable weights in the corresponding LyricLine.
3. The amount of Notes in a Line should equal the amount of syllables in the corresponding LyricLine. They should be distributed to the Line's Bars randomly, for simplicity's sake.

Adhering to Specification 2g, a trigger will be used to add Spec objects to the melody tree. To satisfy rule 1, a trigger designated for the Lyrics root object is used. This trigger is presented in Listing 67. It simply counts the amount of LyricLine objects among its children and adds a LineExpansion object to the Melody tree with this amount. Special attention should be paid to line 11, where the AddExternalTreeSpec method is used to add the LineExpansion Spec object to the Melody object. When the system object is expanded with a Melody object, the system object is assigned a RootProperty containing the pointer to the Melody object, as demonstrated in Section 8.15. Therefore, the AddExternalTreeSpec method can be used to add Spec objects to the Melody object, passing "Melody" as the first parameter. As explained in Section 7.2.3, this method will then use a dynamic property to retrieve the RootPropety with identifier "MelodyRoot" before adding the Spec object to it.

```
1  void LyricsTrigger(Marble* pMaster) {
2    // Counting the amount of LyricLines
3    int numLines = 0;
4    std::map<int, Marble*>& mChildren = pMaster->GetChildren();
5    for (auto it = mChildren.begin(); it != mChildren.end(); it++) {
6      LyricLine* pL = dynamic_cast<LyricLine*>(it->second);
7      if (pL != nullptr) { numLines++; }
8    }
9    // Creating and adding a LineExpansion object, using the amount
       of LyricLines created.
10   Spec* lexp = new LineExpansion({"Melody"}, numLines);
11   pMaster->AddExternalTreeSpec("Melody", lexp);
12 }
```

Listing 67: The LyricsTrigger, which adds LineExpansion objects to the melody tree

The *LyricLineTrigger* is used to satisfy rule 2 and rule 3. This trigger is presented in Listing 68. As the name implies, it's designated to the LyricLine objects. The purpose of this trigger is to create and add *BarExpansion* and *NoteExpansion* objects to the melody tree.

The BarExpansion objects are created based on the sum of the syllable weights of the words within the LyricLine. However, since the amount of LyricsLine objects matches the amount of Line objects in the melody tree, care must be taken such that the BarExpansion object reaches the Line object which corresponds to the LyricLine object where it was created. In other words, an advanced designation function must be used. If not, the BarExpansion object meant for one specific Line object, will be executed by all Line objects. As evident in line 18 of Listing 68, the advanced designation function *bexpAD* is used. This function makes sure the Line objects receive only a single BarExpansion object each, in breadth first order. Since the LyricLine objects also calls their triggers in breadth first order, the Line objects will successfully receive a single BarExpansion object from their corresponding LyricLine.

Satisfying rule 3, the LyricLineTrigger also creates creates one NoteExpansion object for each Bar object in the melody tree. As the number of Notes for each Line should equal the amount of syllables in the corresponding LyricLine, the amount of Notes to create are randomly distributed to the NoteExpansion objects. Like with the BarExpansion objects, an advanced designation function is used with the NoteExpansion objects. This function, *nexpAD*, serves the same purpose as the bexpAD function does. It makes sure the NoteExpansion objects reaches the correct Bar objects, in a breadth first fashion.

```
void LyricLineTrigger(Marble* pMaster) {
  // First, calculating the amount of Bars for the corresponding
      melody Line object
  // Finding the sum of the syllable weights in this LyricLine
  int numSyls = 0;
  int sylWeightSum = 0;
  std::map<int, Marble*>& mChildren = pMaster->GetChildren();
  for (auto it = mChildren.begin(); it != mChildren.end(); it++) {
    Word* pW = dynamic_cast<Word*>(it->second);
    if (pW != nullptr) {
      std::pair<std::string, std::vector<int>>
          wordsyl = pW->GetWord();
      for (int weight : wordsyl.second) { sylWeightSum += weight; }
      numSyls += wordsyl.second.size();
    }
  }
  // Creating and adding a BarExpansion (based on the syllable sum)
      for the corresponding melody Line object
  int numBars = sylWeightSum/8;
  Spec* bexp = new BarExpansion({"Line"}, numBars, bexpAD);
  pMaster->AddExternalTreeSpec("Melody", bexp);

  // Second, calculating the amount of Notes for Each Bar
  // The amount of notes equals the amount of syllables
  // A melody Line object's Notes are randomly distributed to its
      Bars
  int numNotes = numSyls;
  std::map<int,int> barNotes;
  if (numNotes >= numBars) {
    for (int i = 0; i < numBars; i++) {
      barNotes[i] = 1;
      numNotes--;
    }
    for (int i = 0; i < numNotes; i++) {
      int baridx = pMaster->RandomInteger1(0, numBars-1);
      barNotes[baridx]++;
    }
    for (auto it = barNotes.begin(); it != barNotes.end(); it++) {
      // Creating and adding the NoteExpansion objects
      Spec* nexp = new NoteExpansion({"Bar"}, it->second, nexpAD);
      pMaster->AddExternalTreeSpec("Melody", nexp);
    }
  }
  else {
    std::cout << "Not enough notes" << std::endl;
  }

}
```

Listing 68: The LyricLineTrigger, which adds BarExpansion and NoteExpansion objects to the melody tree

10.3 Presenting *Random Lyrics Generating Melody*

10.3.1 Generating a Random Lyrics Tree

Before generating the melody tree, first order of business is generating a random tree of lyrics. The code generating this tree is presented in Listing 69. Notice that seeds are passed to the constructor of the system object. This is necessary since we are using the RandomInteger methods. The RandomInteger1 method is used to determine the shape of the tree (the amount of LyricLine and Word objects), which means that changing the first seed will influence this. Meanwhile, RandomInteger2 is used to pick the words, which means that the words depend on the second seed. A *LyricsExpansion* object is added, as well as a TriggerSpec object which carries a pointer to the *PrintLyricLinesTrigger* function. This trigger is bound for all LyricLine objects and will print the line's words. The output of the code in Listing 69 is presented in Listing 70. Here, the triggers are printing the words of each line.

```
1  // file "lyrics/random_lyrics_main.cpp"
2  #include "lyrics.h"
3
4  int main() {
5
6    Marble system(nullptr, "System", {76998, 13416});
7
8    Spec* lexp = new LyricsExpansion({"System"}, 1);
9    system.AddSpec(lexp);
10
11   Spec* printtrigger = new TriggerSpec({"LyricLine"},
         PrintLyricLinesTrigger);
12   system.AddSpec(printtrigger);
13
14   system.Traverse();
15
16   return 0;
17 }
```

Listing 69: Generating a random lyrics tree and printing the words

```
A Tur--tle Sleeps
The Bear Cries But A Scor--pi--on Jumps
The Spar--row Jumps
```

Listing 70: The output of the code in Listing 69

The output presented in Listing 70 can be recreated using the following commands:

```
$ make random_lyrics
$ ./random_lyrics
```

Remember that the *LD_LIBRARY_PATH* environment variable must be set and exported before running the executable (see Section 8.1).

10.3.2 The Lyrics Tree Generating a Melody Tree

Listing 71 presents the main function code which generates the lyrics tree. When the Traverse method is called, the lyrics tree will add Spec objects to the Melody object such that the melody tree is also generated. The triggers generating the melody Spec objects are added using TriggerSpec objects (lines 13-18), and the system is expanded with the Melody root object such the triggers have an external tree root to add their Spec objects to (lines 20-21). To make sure the same lyrics tree as the one in Listing 69 is generated, the same seeds are passed to the system object constructor (line 8).

```
1  // file "lyrmel_main.cpp"
2  #include "lyrics.h"
3  #include "../melody/melody.h"
4  #include "../marbleviz/marbleviz.h"
5
6  int main() {
7
8    Marble system(nullptr, "System", {76998, 13416});
9
10   Spec* lexp = new LyricsExpansion({"System"}, 1);
11   system.AddSpec(lexp);
12
13   Spec* lyrtrig = new TriggerSpec({"Lyrics"}, LyricsTrigger);
14   system.AddSpec(lyrtrig);
15
16   Spec* lyrlinetrig = new TriggerSpec({"LyricLine"},
17       LyricLineTrigger);
18   system.AddSpec(lyrlinetrig);
19
20   Spec* mexp = new MelodyExpansion({"System"}, 1);
21   system.AddSpec(mexp);
22
23   MarblevizDraw(system, {"System", "Lyrics", "LyricLine", "Word",
24       "Melody", "Line", "Bar", "Note"},
25       "The Trees of Lyrics and Melody");
26
27   system.Traverse();
28
29   return 0;
30 }
```

Listing 71: The main function code which generates a lyrics tree, which generates a melody tree

The resulting lyrics tree is presented in Figure 8. Meanwhile, the melody tree it generates is presented in Figure 9. Notice that the trees satisfy the rules that were imposed. The melody tree has three Line objects, the same amount of LyricLine objects that the lyrics tree has. As Listing 70 presents, the words of the LyricLine objects amount to 4, 9 and 4 syllables, respectively. The Line objects in the melody tree consequently has 4, 9 and 4 Note objects, respectively. Notice also that the Note objects are distributed to the Bar objects in what convincingly looks like random distribution.

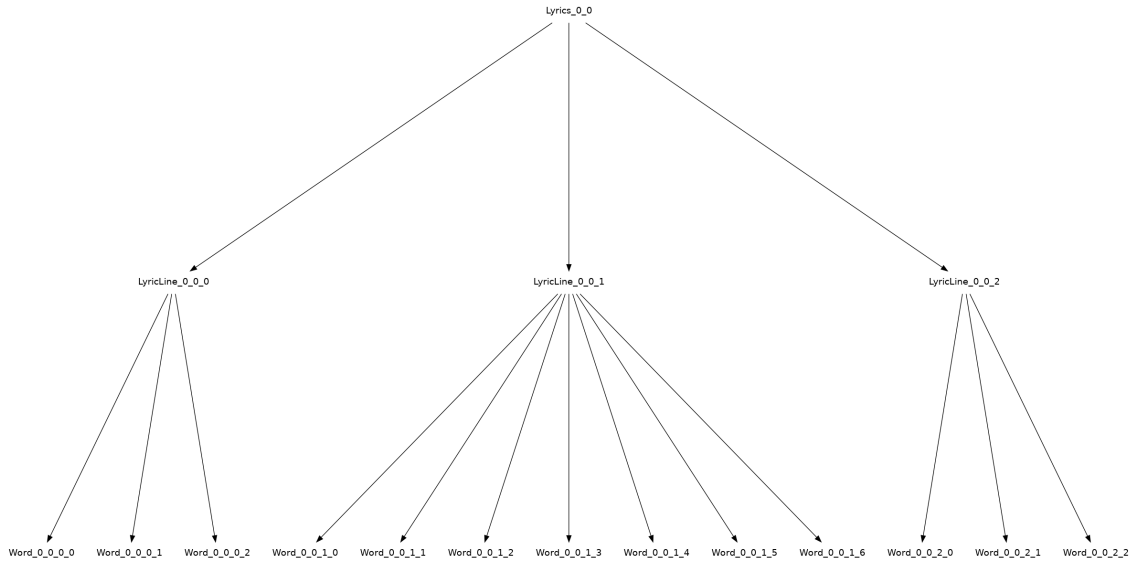

Figure 8: The lyrics tree generated in Listing 71, drawn using Graphviz

Figure 9: The melody tree generated by the lyrics tree in Listing 71, drawn using Graphviz

The trees in Figure 8 and 9 can be drawn in a PNG file using the following command (the name of the resulting file is "lm.png"):

```
$ make lyrmel
$ ./lyrmel > lm.gv && dot -Tpng lm.gv -o lm.png
```

Remember that the *LD_LIBRARY_PATH* environment variable must be set and exported before running the executable (see Section 8.1). Graphviz must also be installed.

11 Discussion

In this section, the results of the Marble library features, the *Marbleviz* application and the *Random Lyrics Generating Melody* application, will be discussed. Being important qualities of a software library, the versatility and the user interface of the Marble library will be discussed. Since the main motivational factor for the development of the Marble library was to enable the separation of concerns, the applications' utilization of said design principle will be discussed. Shedding light on the library's degree of versatility, how the applications have utilized the library's features to satisfy their specifications will also be discussed.

11.1 The Marble Library

11.1.1 The Versatility of the Marble Library

One of the most important goals of any software library is that is should be useful in as many contexts as possible. Therefore, its features should be designed such that they are applicable and adaptable to a wide range of use cases. The "function based" features of the Marble library are an important factor for facilitating the library's versatility. These features include triggers, calculating properties and advanced designation. Since the user defines the behaviour of these features by writing functions, they're applicable in a wide range of contexts. Triggers makes sure the Marble objects in a tree can perform practically any task the user wants them to. Calculating properties makes sure properties can be tailored to any Marble object based on all the possible calculations which can be contained in a function body. Advanced designation is also a powerful feature. Having explicit control over where a Spec object should be executed increases the amount of trees one can create significantly.

Dynamic properties is also an important feature which facilitates versatility. The ability to retrieve information from objects of higher rank and precedence is such an important concept is so many contexts that it makes the library applicable in a significantly wider range of use cases. Another feature which increases the library's versatility is the Output objects. Just like the dynamic properties, there are many cases where visualizing the contents of a tree structure is necessary. The fact that Output objects support dynamic strings is also a feature that increases the Output objects range of use cases.

The versatility of the library is further enhanced by the fact that the library is easy to expand. That is, creating new classes derived from the Marble and Spec classes is simple. New derived class can support functionality necessary in many contexts, enhancing the versatility of the library. New Marble classes can easily be created since the definition of a Marble derived class requires nothing other than, naturally, that the constructor of the new class must call the Marble class' constructor. Necessary member variables and constructor contents can then be added to suit the user's needs. In addition, creating new Spec derived classes is also simple. Again, the constructor of the new class must call the Spec constructor.

Other than that though, the user is required to do nothing else than to define the Execute method of the Spec derived class, defining the purpose of the Spec class.

11.1.2 User Interface

Providing a simple user interface is an important task for all software libraries. Unnecessary details and information should be stripped away such that the user is quickly able to understand how the library should be used. The Marble library has therefore been developed such that the user interface is as simple as possible. The steps required to generate a tree is simple. A system object must be created, Spec objects must be created and added, and finally, the Traverse method must be called. Setting up some of the Spec objects require only a few extra steps. A Property object must be created before creating a PropertySpec object. A function must be defined before creating a TriggerSpec or a CalculatingProperty. The workflow of the library couldn't be much more simpler.

In addition, efforts have been made such that the object constructors require as few parameters as possible. The necessary constructor parameters are also intuitive. The first parameter passed to the Spec constructor is the object's set of targets, which is only natural. The remaining parameters depend on the type of Spec objects. For instance, the amount of children an Expansion object should create must be specified. Meanwhile, the Property constructors require an identifier and the property value, which is only natural. The Marble constructor however, requires the pointer to the parent object as the first parameter. This is intuitive when defining the Execute method of Expansion classes. However, passing *nullptr* as the first parameter of the system object is not. This is therefore a weak spot of the user interface.

The tree building classes which the library provides the user with is also an important part of the user interface. Since the classes are divided into no more than three groups, i.e. Marble classes, Spec classes and Property classes, getting an overview of the library's features is easy. In addition, the roles of their classes are quite easy to understand. The fact that objects of one of the classes will serve as the building block of a tree is quite intuitive. Understanding the Spec objects provide the specifications for a tree should also be easy given the name. For the same reason, the purpose of the Property class is also clear. The only interactions between the classes which the user has to worry about is using the AddSpec method of the system object, as well as attaching Property objects to PropertySpec objects, which really aren't complex interactions to comprehend.

For the library to work as intended, all Spec and Property objects must be allocated dynamically. If not, they'll be deleted twice, something which results in undefined behaviour. The library instructions would contain this information, so one might claim that few users would fall in this pitfall. However, a better solution might've been to somehow check if the objects are allocated on the heap. If not, an error message could be output.

A useful feature of the library is creating classes derived from the Marble class and

building trees out of them. This is an easy feat, considering no methods need definitions. One only has to make sure the constructor of the new class calls the Marble constructor, passing the parent pointer and the new class' typeID as parameters. The addition of a Marble derived class involves the creation of an Expansion class creating said Marble type. This entails the definition of the CreateChild, which must return the pointer to a dynamically allocated object. This is a little more complicated than creating a new Marble derived class. However, if one follows the library instructions (tutorial), it shouldn't be difficult to write this single line of code correctly. On the other hand, creating a new Spec derived class is simple. Here, the only method one must define is the Execute method, naturally.

11.2 The Marbleviz Application

11.2.1 Applying The Separation of Concerns Principle

One of the central motivational factors behind the development of the Marble library was to enable separation of concerns. That is, distinct sections of a program which address their own concerns being kept more or less separate from each other, despite the fact that they must be integrated to form a complete program.

The Marbleviz application is an example of how the Marble library enables separation of concerns. How exactly does separations of concerns relate to the Marbleviz application? The generation of a tree can be considered as one distinct section of a program addressing its own concern. Meanwhile, a section tasked with translating a tree into Graphviz readable DOT code can be considered as a separate section. Even though they're separate sections addressing separate concerns, they still need to cooperate in forming a computer program which generates a tree and outputs it in DOT code.

Now, how would one develop such a program without the help of the Marble library? Imagine that two software developers were assigned to each section. Potentially every time the tree generation developer would write some code, the "Graphviz" developer would have to tack on their necessary code. Every time the tree generation developer would remove or rewrite code, the "Graphviz" developer would have to make sure their section still performed sufficiently. In other words, these sections are severely, tightly coupled.

Developing this computer program with the help of the Marble Library however, the sections are kept completely separate. Consider Listing 29 as an example. The code in this section is solely focused on generating a tree, no considerations about making the code compatible with DOT code generating software has been made. At line 18, the MarblevizDraw function is called. Here, all the Spec objects necessary to output the DOT code representing the tree are added. The tie between these section necessary to for the program to function cannot be any simpler than this.

The definition of the MarblevizDraw function is presented in Listing 53. Notice that no assumptions about the tree to be drawn are made. That is, this section only address one single concern, translating any tree into DOT code. Since these section

are completely separate, the tree generation code can be changed in any way one wants and the DOT code would still come out correctly. Similarly, the DOT code generating section could be modified without touching the tree generation code. This application is in other words an excellent demonstration of how the Marble library can facilitate separation of concerns such that software developers can keep their software simple, easily maintainable and with high degrees of freedom.

11.2.2 Library Versatility: Utilization of the Library's Features

The Marbleviz application also illustrates the versatility of the library and how it's adaptable to a wide range of contexts. Not only is the Marble usable for building trees with little purpose other than e.g. storing information. Using the features it provides, including triggers and different types of properties, software which outputs source code for graph visualization software to read can be developed. The Output class along with its dynamic string feature proved itself very useful. Thanks to this class, implementing the printing of the DOT code was quick and easy. Easily allowing all Marble objects to be assigned a unique name, the CalculatingProperty class also proved itself useful.

Thanks to the DynamicProperty class, all Marble objects were easily able to store a StringProperty containing their parent's name. In addition, the DotArrow objects could retrieve their owner's *name* and *parentName* properties in a simple and effective manner. Speaking of the DotArrow class, creating this class and defining its constructor to add necessary properties proved itself very simple and useful. Finally, adding an Output object to the Marble object which is last to be traversed, is not an easy challenge. However, since the library supports the use of final triggers and advanced Spec object designation, even this task could be accomplished. The fact that almost all of the library's features have been used to build an application with a purpose as specific as translating a tree into DOT code, is a testament to the Marble library's versatility.

11.3 The *Random Lyrics Generating Melody* Application

11.3.1 Applying The Separation of Concerns Principle

The library application developed in this thesis, where a randomly generated lyrics tree generates a melody tree, is another example of how the Marble library enables the separation of concerns design principle. As previously discussed, the arts of writing lyrics and composing melodies are independent arts that each demand special expertise. A lyricist must concern themselves with rhymes, verses, choruses, semantics and so on. A composer on the other hand, must concern themselves with rhythm, pitch, etc. Despite their differences however, these arts must be combined in order to produce music. Unfortunately, combining lyrics and melody such that coherent and interesting music is produced, is no easy task. As previously discussed, the consequence of combining the generation of lyrics and melody is that both practices impose heavy restrictions on each other. They must "fit" each other, in other

words. To name a few of these restrictions, it would be natural to enforce bars to end on word endings and melody lines to end on a heavily weighted syllable, etc.

Because of the heavy restrictions lyrics and melody generation impose on each other, realizing music generation in software becomes challenging. Since the section generating the lyrics addresses a concern independent from the one which the melody generating section does, keeping these sections separate would be beneficial. Namely, keeping independent sections separate improves the scalability, flexibility and maintainability of the software. Unfortunately, since the specifics of lyrics and melody generation are so closely tied together, one would likely have no other choice that to cram these sections together in one. For instance, allowing all the words in the lyrics to interact with all the bars in the melody would likely be impossible if the sections were kept separate. Making matters worse, the structures of the sections might be so different that cramming them together won't even be possible.

Luckily, as the *Random Lyrics Generating Melody* application demonstrates, the Marble library allows melody and lyrics to be generated such that they are kept separate. In stead of trying to integrate a lyrics tree and a melody tree together, the Marble objects in the lyrics tree take advantage of Spec objects. The application demonstrated how triggers could be used to easily add Spec objects to the root of the melody tree. In other words, the only action needed to reach Marble objects in a separate tree was to create a Spec object with the desired typeID or advanced designation function, followed by calling the AddExternalTreeSpec method. That is, no cluster of dependencies between objects in the lyrics tree and objects in the melody tree is needed. The trees are kept separated and they are only coupled through the call of a single method.

However, the fact that the trees are separately kept means nothing unless it can be proven that the trees are still cooperating. That is, it must be proven that the melody fits the lyrics no matter how the lyrics change. As the *Random Lyrics Generating Melody* application demonstrates in Section 10.3.2, the rules which were set to determine the generation of the melody tree were successfully satisfied. This means that no matter how the lyrics tree is randomly generated, the triggers which generate the melody Spec objects will, adhering to the set of rules, tailor the Spec objects such that the melody fits the lyrics. The bottom line is that, even though the trees are kept separate, they are able to cooperate in the generation of music. In other words, music is generated while the underlying sections addressing different concerns are separated. The result is tidy, maintainable software with high degrees of freedom.

11.3.2 Trees Generating Trees

The main factor motivating the development of the Marble library was to enable the separation of the concerns of tightly coupled abstractions. As previously explained, the library accomplishes this by letting trees generate Spec objects which are added to the roots of other objects. In other words, separation of concerns is made possible by the library's ability to let trees generate trees. Therefore, it's important to review this particular feature. The *Random Lyrics Generating Melody* application

utilizes said feature. After the generation of the lyrics tree was implemented, all the work which remained to implement the generation of the melody tree, was to add two triggers. Since the triggers had access to its Marble object's owners, creating the Spec objects necessary to correctly generate the melody object was a relatively simple task. Making trees generate trees is easy and effective in other words, effectively enabling separation of concerns.

11.3.3 Procedural Generation

Section 5.3 explains how *Multi-Level, Multi-Content PCG* still remains to be accomplished. Here, only the game engine and a set of game rules are predetermined. The rest of the game content is generated by PCG methods. As discussed in Section 4.2.3, recursive, hierarchical structures (trees) are well suited for game content generation. However, trees which address different concerns are forced to be crammed together, resulting in unmaintainable software of poor quality.

The results of the Random Lyrics Generating Melody application indicates that the library's features could be of help in procedural generation. The generation of the lyrics and melody tree is in fact a form of procedural generation. The fact that the library is able to let several trees cooperate in the generation of a tree means that procedural generation would be possible on an even bigger scale. As an example, imagine the case where one wants to generate a city for a game world. Here, separate trees addressing e.g. road construction, park construction, construction of buildings, etc., could generate Spec objects for a central tree representing the fully constructed city, ready to be rendered into a game world.

11.3.4 Library Versatility: Utilization of the Library's Features

To emphasize the wide range of contexts where the Marble library is applicable, how the library's features can be utilized to develop a musical procedural generation application (the *Random Lyrics Generating Melody* application) will be reviewed. First of all, creating the *WordsProperty* class such that the Marble objects could hold a fairly complex data structure containing words was quite simple. Only the constructor and the Clone method of the new derived class was necessary to define. This simple library extension was an important factor in facilitating the random generation of lyrics tree, speaking to the library's versatility.

The fact that new Marble derived classes can quickly and easily be created where one can make their constructors create and add Property and Spec objects, proved itself very useful. Additionally, the DynamicProperty class played an important part in the generation of the lyrics as it let the Word objects draw their words from the pools of words. The random functions of the Marble class also proved their usefulness, as they effectively facilitated the creation of a random amount of LyricLine objects and Word objects. The fact that the Marble class provides two random functions also let the words drawn by the word object be independent from the random amount of children to create. This way, one could change the words

which the lyrics consist of, without changing the structure of the lyrics, which is a useful feature.

Making a tree generate the Spec objects for another tree might seem like a daunting task at first. However, thanks to the powerful feature of triggers, this task could easily be accomplished. Adding Spec objects to another tree is also quick and effective, thanks to the AddExternalTreeSpec which effectively gains access to the roots of external trees using dynamic properties to retrieve the RootProperties at the system object. The fact that an application can utilize all these features in order to randomly generate a tree of lyrics which itself can generate a melody tree which fits the lyrics, proves how versatile the Marble library is.

11.4 Separating the Concerns of Applications in General

The Marbleviz and Random Lyrics Generating Melody applications have demonstrated how the Marble library allows them to keep their sections separated. The goal is however, that the library should allow any tree building application to keep their trees separated. Luckily, the library mechanisms that enable separation of concerns are not tailored to the applications developed in this thesis in any way. In other words, the library makes no assumptions about the trees which are kept separate. The separation of the concerns is made possible by the fact that Spec objects can be added to root objects, which are available to all Marble objects, before the Traverse method makes sure they hit their targets. In other words, any application consisting of tree representable sections which can address their concerns by creating Spec objects, will be able to keep its sections separated.

11.5 The Library's Versatility From a General Perspective

While the Marbleviz and Random Lyrics Generating Melody applications utilize many of the same library features, the purposes of the applications are fairly different. This indicates that the library is versatile enough to be useful in a wide range of contexts. Reviewing the applications' use of the features will also indicate which of the features which are the most versatile and, by extension, the most powerful.

The ability one has to create new Marble derived classes, adding necessary member variables and defining their constructors to add Spec and Property objects, was a key factor in allowing the applications to address such different contexts. The functionality of the applications are also heavily dependant on the use of triggers. The versatile power of this feature is apparent, since they can do everything which can be defined in a function body.

Finally, the last feature which should be emphasized is the use of dynamic properties. Their ability to enable quick and easy flow of data throughout a tree has proven them to be especially useful. Retrieving data from higher ranking Marble objects has, through the development of the library applications, proven itself to be a frequently recurring concept which dynamic properties resolve excellently.

11.6 Future Work

If one were to resume the development of the Marble library, the most obvious place to start would likely be to extend the library's features by creating new classes derived from the Marble, Spec and Property classes. Marble derived classes with predefined triggers covering basic tree related needs, much like the Output class, could be created. As it stands, the library only provides two Property classes that contain "classic" data types, the StringProperty and IntegerProperty classes. Therefore, it would be natural to make the library provide additional data types, e.g. containing floating-point values, arrays, maps, etc.

There is however, another issue which likely must be dealt with if the library development were to continue. In the work presented in this thesis, only a single tree has generated Spec objects for another. However, the intention is that several trees should be able to generate Spec objects for the same tree. Unfortunately, some of these Spec objects might represent specifications that don't agree with each other. For instance, there might be disagreements about how many children a specific Marble object should have or what the value of a specific property should be. Somehow, these disagreements must be resolved. A possible solution would be to assign a value to each Spec object which would indicate their level of importance. That is, how important it is that they are executed. Spec objects of higher importance would then "beat" Spec objects of lower importance. Pragmatically, choosing between Spec objects of equal importance might be left to chance. One would have to examine the consequences of highly important Spec objects being ignored to decide if this approach would work.

12 Conclusion

The Marble library has been proven to be successful in enabling the separation of concerns principle in contexts where it previously was impossible. Where once trees addressing separate concerns were forced to be integrated into one, creating them using the Marble library enables them to be kept separate while still being able to interact tightly the way the application they form demands them to.

The Marbleviz application demonstrated how the Marble library enables separations of concerns. While one section can be responsible solely for generating a tree, the Marbleviz application represents a section solely focused on translating trees into DOT code. Even though these sections are independent of each other, they are still able to effectively cooperate in forming a program which generates an object tree before printing the DOT code representing the tree to the console.

The Random Lyrics Generating Melody application also demonstrated how the Marble library enables separation of concerns. While lyrics and melody generated in software originally would involve the two sections being intertwined in an unmaintainable cluster of object-to-object dependencies, the application demonstrated how a melody tree could be generated by a lyrics tree while these trees are kept separate. The fact that this application randomly creates one tree which subsequently governs the generation of a second tree, proves how the Marble library could be used to let several trees cooperate in generating Spec objects for a central one. This indicates that the Marble library could be useful in the field of procedural generation.

Finally, the Marble library itself is equipped with an extensive set of features that makes it versatile enough to be applicable in a wide range of contexts. The use of triggers is a feature which is worth mentioning. This feature gives the programmer the ability to make trees execute any task which can be contained in a function body, significantly increasing the library's versatility. Dynamic properties, easily allowing flow of data throughout a tree, also proved itself to be useful in many contexts. In addition, the user interface of the library successfully enables the user to generate and manage trees by following simple and intuitive steps.

Bibliography

[1] Håvard Roalsø Eiring. 'Resolving Dependencies Between Tightly Coupled Recursive, Hierarchical Structures'. In: (2021).

[2] Sverre Hendseth. *Marble Documentation: A Sound Tutorial*.

[3] Sverre Hendseth. *Presenting the Marble Modelling Language*.

[4] Scott Meyers. *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 2005.

[5] Andrew Oram and Steve Talbott. *Managing Projects with make*. 2nd ed. O'REILLY, 1993.

[6] Julian Togelius, Noor Shaker and Mark J. Nelson. 'Introduction'. In: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Ed. by Noor Shaker, Julian Togelius and Mark J. Nelson. Springer, 2016, pp. 1–15.

Appendix

A Files Included in the Submitted Zip Folder

- TTK4550_specialization_project.pdf (the specialization project report [1])
- marbledoc_soundtutorial.pdf (Marble Documentation: A Sound Tutorial [2])
- marbledoc_slides.pdf (Presenting the Marble Modelling Language [3])
- source_code (directory containing the source code of the work presented in this thesis):
  - lib
    - libmarble.h
    - marble.cpp
    - marble.h
    - property.cpp
    - property.h
    - spec.cpp
    - spec.h
  - lyrics
    - lyrics.cpp
    - lyrics.h
    - lyrmel_main.cpp
    - random_lyrics_main.cpp
  - marbleviz
    - marbleviz.cpp
    - marbleviz.h
    - marbleviz_main.cpp
  - melody
    - melody.cpp
    - melody.h
    - melody_main.cpp
  - tutorial
    - advdes_tut.cpp
    - asym_tut.cpp
    - calcprops_tut.cpp
    - custom_objs_tut.cpp
    - dynprops_tut.cpp
    - ext_tree_specs_tut.cpp
    - final_triggers_tut.cpp

    - output_1_tut.cpp
    - output_2_tut.cpp
    - propspec_tut.cpp
    - prop_tut.cpp
    - seeds_tut.cpp
    - spec_tut.cpp
    - tree_mv_tut.cpp
    - tree_tut.cpp
    - triggerspec_tut.cpp
    - triggers_tut.cpp
- main.cpp
- Makefile

Håvard Roalsø Eiring

Presenting The Marble Library: Separating the Concerns of Tightly Coupled Abstractions

NTNU
Norwegian University of
Science and Technology