

Model-Driven Software Development with SuperCollider and the UML

Jens GULDEN

Formal Models, Logic and Programming (FLP),
Technical University Berlin
jgulden@cs.tu-berlin.de

Abstract

The SuperCollider programming language is widely perceived as a script-like interface to the corresponding SuperCollider audio server. However, the language has grown to a serious object-oriented programming language by now, and hence allows applying visual modeling techniques for model-driven development of object-oriented software systems. This article shows how diagram-based visual modeling with the Unified Modeling Language (UML) can be used for creating SuperCollider software.

Keywords

Modeling, Model-Driven, Object-Oriented, SuperCollider, Unified Modeling Language (UML)

1 Introduction

While the SuperCollider programming language ([1]) is most commonly used just as a scripting language for passing commands to a SuperCollider audio server, it also provides all basic language constructs for developing (small or medium-sized) object-oriented software systems. Such language constructs are e. g. the declaration of *classes*, *instantiation* of objects, *polymorphism* (method-overwriting), *class-* and *instance-variables*, *object-references* etc., which are all available in SuperCollider.

Because of its support for object-orientation, visual modeling techniques can be used together with the SuperCollider language, such as *class-diagrams* of the Unified Modeling Language (UML) which provide a visual description of the

declarations that make up an object-oriented software system.

The present article describes how the UML can be applied for model-driven development of SuperCollider software. The upcoming chapter two introduces UML class-diagrams and how they are used for modeling object-oriented software systems. The third chapter shows how code in the SuperCollider language is integrated to implement a model's declarations, and in chapter four runnable code is generated from a model. Chapter five then demonstrates how a specific UML-tool is configured to generate executable code in the SuperCollider language. The sixth chapter summarizes the relevant mappings between visual class-diagram elements of the UML on the one hand, and language constructs of the SuperCollider language on the other hand. In chapter seven, an example is presented of how model-driven development with the UML has been applied to create a SuperCollider implementation of the XML Document Object Model (DOM). Chapter eight finally gives information on how to download the presented work, chapter nine sketches possible future plans, and the closing tenth chapter contains a short conclusion.

2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML, [2]) is not a programming language. The word “language“ denotes a set of visual elements used in diagrams that describe the architecture of object-oriented software systems.

UML *class-diagrams* express which classes are part of the software system, which operations and variable-members they have, and how the classes are related to each other. Class-diagrams thus provide an overview on the declarative structure of

an object-oriented software system. Figure 1 shows a small example class-diagram.

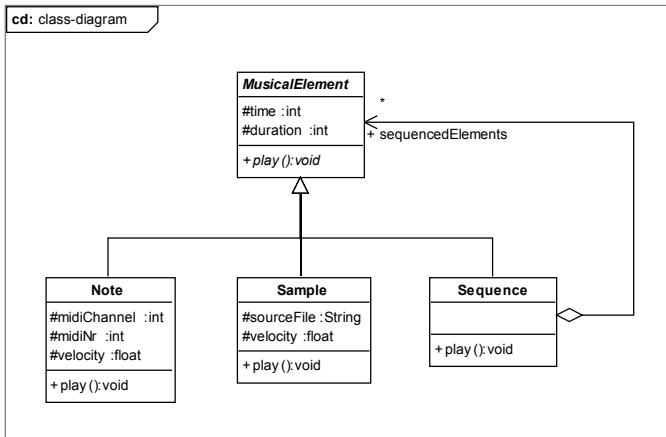


Fig. 1: Class-diagram example

Class-diagrams describe all declarations in an object-oriented software system, but they do not contain any programming language code (editing code is discussed in the following section).

When developing software, the UML does not replace a programming language like the SuperCollider language. Instead, it embeds the overall structure of the language into a visual diagram.

Consequently, any UML class-diagram can be transformed into program code of an object-oriented programming language. The above example diagram could e. g. result in program code as shown in example 1.

```

/*
 * SuperCollider3 source file "MusicalElement.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class MusicalElement ---
//
MusicalElement {
    // --- attributes
    var time; // type int
    var duration; // type int

    // --- play() : void ---
    //
    play {
        "ABSTRACT".die; // simulate abstract method
    }
} // end MusicalElement

/*
 * SuperCollider3 source file "Note.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Note ---
//
Note : MusicalElement {

```

```

// --- attributes
var midiChannel; // type int
var midiNr; // type int
var velocity; // type float

// --- play() : void ---
//
play {
    // ... do something to play the midi note ...
} // end play
} // end Note

/*
 * SuperCollider3 source file "Sample.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Sample ---
//
Sample : MusicalElement {
    // --- attributes
    var sourceFile; // type String
    var velocity; // type float

    // --- play() : void ---
    //
    play {
        // ... do something to play the sample ...
    } // end play
} // end Sample

/*
 * SuperCollider3 source file "Sequence.sc"
 * Licensed under the GNU General Public License (GPL).
 */

// --- class Sequence ---
//
Sequence : MusicalElement {
    // --- relationships
    var <>sequencedElements; // 0..*-relation to type MusicalElement

    // --- play() : void ---
    //
    play {
        sequencedElements.do( { // play elements in sequence
            arg element;
            waitUntilTime(element.time); // ...pseudo-code...
            element.play();
        } );
    } // end play
} // end Sequence

```

Example 1: Generated code from the example diagram

UML class-diagrams are also called *models* of the software. Graphical models can help to express the core ideas behind a software system on a conceptually higher level than it is possible with raw program code. This is why this kind of visual software development is called *model-driven development*.

There are several advantages which motivate the use of model-driven development. An individual programmer can benefit from better navigable code and more efficient access to elements of the software system, compared to plain files. Also,

tasks that otherwise would require stereotype typing work can be automated using visual editing, e. g. the creation of new classes or the declaration of relationships among classes.

When communicating about software in a group of people, visual diagrams help to enrich the semantics of the formalized software system and allow to express complex domains of discourse using geometric means and spatial constellations. UML class-diagrams are thus especially helpful for sharing ideas about the architecture of a software system among several people.

3 SuperCollider method bodies in the UML-model

As the UML is not a programming language itself, but provides graphical elements for declaring the structure of object-oriented software, the actual implementation of methods is still done by conventional programming using a traditional programming language.¹ Thus, the method bodies declared in the visual diagram are to be 'filled out' by textual programming. Some UML-tools provide their own source-code editor that allows editing a method-body inside the modeling application.²

Fig. 2 shows a screenshot of how this can look like within a specific UML-tool. The source-code editor at the bottom contains the code of the method highlighted in the diagram above.

4 Generating executable SuperCollider code

In order to get executable software from the model, a complete set of source-files is generated from it. These files contain traditional source-code in the *target programming language* and can then be compiled or interpreted to finally make up an executable program. It depends on the UML-tool

¹There are approaches which also allow to derive dynamic behaviour of software systems, thus the code of method-bodies, from graphical diagram modeling. This is, however, beyond the scope of this article.

²An alternative approach is using external code-editors (usually within integrated development environments) for code-editing, and letting the modeling-tool take care for synchronizing between the model and the code. This approach is called "round-trip-engineering", while the process described in this article (editing code inside the modelig-tool and then generating source-files) is called "forward-engineering".

which target programming languages are available to generate code for. In an ideal case, a UML-tool can be configured freely to generate code for any object-oriented language. This is the case with the tool used for demonstration in this article, *Poseidon for UML* ([4]). This tool uses editable code-generation templates for configuring the target programming language, which allows generating code not only for well-known standard object-oriented languages such as Java or C++, but also for SuperCollider.

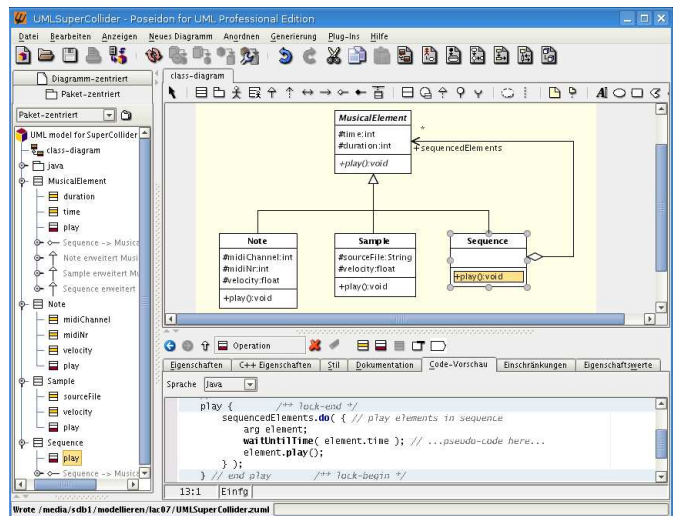


Fig. 2: Editing the source code of a method-body

5 Configuring a UML-tool to generate SuperCollider code

There is no standard on how UML-tools are to be configured for a new target programming language. As a consequence, this section is restricted to the use of the UML-tool Poseidon for UML ([4]), which is used for demonstration throughout this article.

In Poseidon, the code generation process is controlled by file templates into which variable code fragments from the model are inserted while evaluating the template. By evaluating properties from the model via variables and simple case-distinctions (*if*-branches, etc.), the desired SuperCollider code is generated. Example 2 shows a sample of the code-generation template for the SuperCollider language, as it has been used to configure the tool Poseidon.

```

## --- Render a single attribute. ---
#macro (renderOneAttribute $preparedAttr)
...
#set ($SCvisibility = "")
#if ($static.indexOf("static")!=-1)
#set ($SCvar = "classvar")
#else
#set ($SCvar = "var")
#end
#set ($visibility.indexOf("public")!=-1)
#set ($SCvisibility = "<>")
#end
  ${SCvar} ${SCvisibility}${name}${initialValueExpr};
  // type #stripPkg($preparedAttr.getTypeAsString())
#end

## --- Render attributes from associations/relationships. ---
#macro (renderAttributesForAssociationEnds $prepAssocEnds)

#foreach ($preparedAssocEnd in $prepAssocEnds)
...
#set ($SCvisibility = "")
#if ($visibility.indexOf("public")!=-1)
#set ($SCvisibility = "<>")
#end
  ${SCvar} ${SCvisibility}${name}${SCinitialValueExpr};
  // ${typeCommentMulti}relation to type #stripPkg(
$preparedAssocEnd.getTypeAsString() )
#end
#end
#end

```

Example 2: Part of the code-generation template for SuperCollider, as used with *Poseidon for UML*

Although the use of templates of this kind is specific to Poseidon for UML, other UML-tools can also be configured for non-standard target programming languages. The principles demonstrated in this article are portable to other modeling-tools.

6 Mapping between UML and SuperCollider

This section summarizes the mappings between UML model elements and the corresponding SuperCollider language constructs. These mappings have already been implicitly encoded in the configuration templates discussed in the previous section.

Table 1 lists the mappings between the UML model elements and the SuperCollider language constructs as they are used in the code-generation process. This list can be used as a basis for configuring further UML-tools to use SuperCollider as the target programming language.

UML	SuperCollider
Class	Class
Package	None, UML-packages are ignored for SuperCollider code-generation. They can however be useful inside the UML model to organize classes. ³
Attribute	(Instance-) Variable
Static Attribute (<u>underlinedIdentifier</u>)	Class-Variable
Method (<i>'operation'</i>)	Method (<i>'message'/'function'</i>)
Static Method (<u>underlinedIdentifier</u>)	Class-Method
Abstract Method (<i>italicIdentifier</i>)	Method which always throws a runtime error (thus must be overwritten to be used) ⁴
0..1/1..1-Relationships	Variable, reference to single instance
0..*/1..*-Relationships	Variable, list containing references to multiple instances
Types of attributes, function-arguments and return-values.	None, but comments. The SuperCollider language is untyped, but including type declarations in the UML model will generate comments in the source code which increases readability of the code.
Public (+) visibility of attributes	Variable, with getter and setter declaration (" <i><></i> ").

³This implies that class names must be chosen to be unique all over the whole set of classes available for SuperCollider. A possible convention is to use a package-like prefix, e.g. DOMNode, DOMELEMENT, DOMText etc.

⁴This is currently implemented by calling "ABSTRACT".die but might more elegantly be done using this.subclassResponsibility().

Package (~), Protected (#), Private (-) visibility of attributes	None, normal variable is used. Besides getter/setter access, visibility is not reflected in the SuperCollider language.
Public (+), Package (~), Protected (#), Private (-) visibility of methods	None. Method visibility is not reflected in the SuperCollider language.

Table 1: Mapping between UML model elements and SuperCollider language constructs

Distinguishing between different levels of visibility is not reflected explicitly in the SuperCollider language, however, it may be useful to use visibility-modifiers in the model in order to express additional semantics about the declared elements (e. g., use 'private' visibility (-) for members that are only used internally by the same class, and 'protected' visibility (#) for methods that are intended to be used by subclasses only).

7 Example: XML-DOM implementation for SuperCollider

The Document Object Model (DOM, [5]) is a standardized set of interfaces for representing XML documents in an object-oriented structure, making them handable for processing with an object-oriented programming language. [6] is an implementation of the DOM API version 1.0 for SuperCollider. The library has been completely developed using a model-driven development environment as described above, and thus serves as a real-world example for model-driven, UML-based software development with SuperCollider. The library has already proven its usefulness in music-related software-projects such as the emulation of the signal-noise of the ENIAC computer (ENIAC NOMOI, [7]), or creating a prototype implementation for an XML-format describing parts, positions and time in audio waveforms ([8]).

The DOM API version 1.0 is an early revision of the DOM standard, however already useful for most common XML-related tasks. It defines a minimum set of 10 interfaces which have been modeled using the class-diagram shown in Fig. 3.

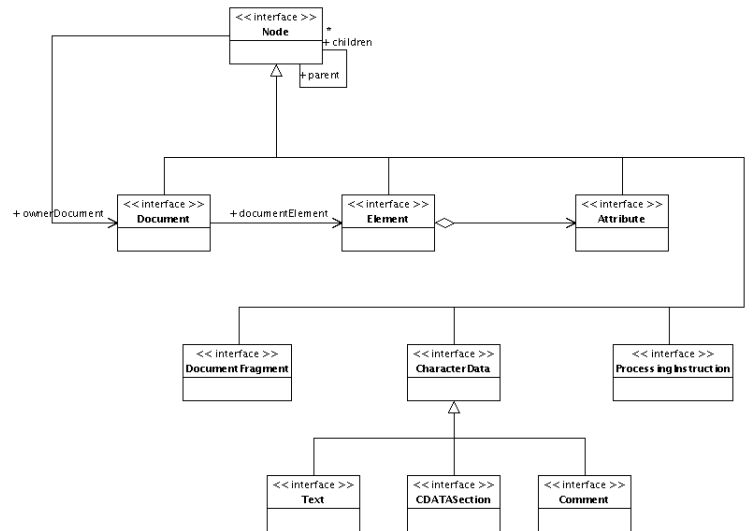


Fig. 3: Class-diagram of the XML DOM API implementation

The corresponding classes for implementing these interfaces in the SuperCollider language have also been developed entirely using a UML class-diagram, see appendix A.

The source code shown in the following example 3 gives an insight into the code as generated from the XML DOM API model.

```

// --- class DOMNode ---
//
// Attributes of the node are stored via Dictionary-entries.
//
DOMNode {
    // --- attributes
    classvar <node_ELEMENT = 1; // type int
    classvar <node_ATTRIBUTE = 2; // type int
    classvar <node_TEXT = 3; // type int
    classvar <node_CDATA_SECTION = 4; // type int
    (...)
    var nodeValue; // type String
    var nodeName; // type String
    var nodeType; // type int
    (...)
    var attributes = nil; // type Dictionary

    // --- relationships
    var ownerDocument; // 0..1-relation to type DOMDocument
    var parent; // 0..1-relation to type DOMNode
    var children; // 0..*-relation to type DOMNode

    // --- getNodeName() : String ---
    //
    getNodeName {
        ^nodeName;
    } // end getNodeName

    // --- setNodeName(name) : void ---
    //
    setNodeName { arg name; // type String
        nodeName = name;
    } // end setNodeName
    (...)
}
  
```

Example 3: SuperCollider code generated from the XML DOM API model, class DOMNode.sc

An additional example directly related to musical applications is shown in appendix B. The diagram contains an early sketch of a synthesizer class library, as it could be developed with SuperCollider.

8 Downloads

All technical details about the presented approach can be found at <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/751>.

The UML integration is done on top of the UML modeling tool *Poseidon for UML* ([4]). The code-generation templates for SuperCollider can be downloaded at <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/uploads/751/sc-templates.zip>. They are freely available and licensed under the GPL ([9]).

Poseidon is a commercial product. A free-of-charge “Community Edition” of *Poseidon* was available for non-commercial use until version 4.x. Since version 5.x, only an evaluation version limited to use in time is available free of charge any longer.

The XML DOM API example is available both as model-file for *Poseidon* and as generated SuperCollider source code at <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/747>. The XML DOM API implementation is also free and licensed under the GPL.

9 Future Plans

As the product *Poseidon for UML* is no longer available as a free-of-charge Community Edition, it has become necessary to migrate to a free UML modeling tool. One possible candidate to migrate to might be *Fujaba* ([10]).

10 Conclusion

Applying visual modeling techniques is not restricted to wide-spread mainstream programming languages. Since the SuperCollider programming language is equipped with all necessary constructs for object-oriented software development, it becomes possible to adopt existing UML-tools to SuperCollider as their target programming language. The article has shown that visual modeling of software is no longer an exotic issue of theory and science only, but has reached a degree of practical usefulness that allows its

application also in non-mainstream development contexts.

The SuperCollider language, on the other hand, has proven to be mature enough for being involved in a state-of-the-art model-driven development process.

References

- [1] Gulden, J., *Developing with [SuperCollider and] the Unified Modeling Language (UML)*, software, <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/751>
- [2] McCartney, J. et al, *SuperCollider3 - A real time audio synthesis programming language*, software, <http://supercollider.sf.net/>
- [3] Booch, G., Jacobson, I., Rumbaugh, J., *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading (Mass.), 1999
- [4] Gentleware AG, *Poseidon for UML*, software, <http://www.gentleware.com/products.html>
- [5] World Wide Web Consortium (W3C), *Document Object Model (DOM)*, <http://www.w3.org/DOM/>
- [6] Gulden, J., *XML parsing and formatting [for SuperCollider]*, software, <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/747>
- [7] Carlé, M. et al, *ENIAC NOMOI*, media project, http://www.medienwissenschaft.hu-berlin.de/~mc/ENIAC_NOMOI_eng.php
- [8] Gulden, J., Rutz, H., *Proposal for an XML format representing Time, Positions and Parts of Audio Waveforms*, this conference
- [9] GNU Software Foundation, *GNU General Public License (GPL)*, legal license, <http://www.gnu.org/licenses/gpl.txt>
- [10] University of Paderborn, Software Engineering Group, *Fujaba Tool Suite*, software, <http://www.fujaba.de/>

