Modellbasierte Codegenerierung SoSe24

Codegenerator Aufgaben

Für diese Aufgaben wird die „Codegenerator IDE“ aus dem Moodle Kurs benötigt. Die Installation funktioniert hierbei genauso wie bei MerapiUML: einfach das Zip-Archiv entpacken, sicherstellen, dass eine Java Runtime verfügbar ist, und eclipse.exe starten.
Weitere Projekte, die für diese Aufgaben gebraucht werden finden Sie im Moodle als Labor/Aufgaben/ als Zip Archive zum Download.

Im Verlauf von vier Aufgaben soll ein C-Codegenerator für UML Klassen entwickelt werden. Eine besonders tiefe Kenntnis der Sprachen C und UML ist hierbei nicht erforderlich, denn der erwartete C-Code für verschiedene UML Modelle ist durch Unit-Tests vorgegeben.
Des Weiteren sind bereits einige Templates vorgegeben, um die Sie sich nicht mehr kümmern brauchen. Diese können Sie auch zur Inspiration nehmen um die fehlenden Templates zu implementieren.
Es existieren bereits alle Dateien, die zur Lösung der Aufgaben notwendig sind, es steht Ihnen aber natürlich frei, noch weitere Dateien zu erstellen. Denken Sie daran, wenn Sie weitere Templates erstellen, dass diese in der Uml2C Klasse registriert werden müssen.
Noch ein Tipp: um die merkwürdigen Klammern in den Templates zu schreiben («») kann im Eclipse Editor die Tastenkombination “Strg + <” und “Strg + Shift + <” verwendet werden.
Bei Fragen würden wir es bevorzugen, wenn Sie diese im Forum des Moodle stellen. Für direkte Diskussionen können Sie auch gerne während der Vorlesungstermine Fragen stellen, die wir dann im Nachhinein besprechen.

Damit Sie nicht den gesamten UML Standard wälzen müssen, werden für die Aufgaben die relevanten UML Elemente und ihre Attribute und Beziehungen jeweils in Diagrammform angegeben. Auf der nächsten Seite folgen zwei Beispiele.

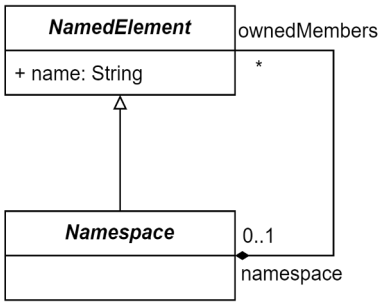

Abbildung 1. *NamedElement* und *Namespace*

Es werden die zwei UML Elemente *NamedElement* und *Namespace* beschrieben. Ein *NamedElement* hat, überraschend, das Attribut *name* vom Typ *String*. Ein *NamedElement* kennt optional einen *Namespace*. Jeder *Namespace* ist selber auch ein *NamedElement* und hat daher auch das Attribut *name*. Die Beziehung zwischen *NamedElement* und *Namespace* ist bidirektional und ein *Namespace* besitzt alle *NamedElements* die sich in ihm befinden.
Beide UML Elemente sind abstrakt, weshalb ihr Name kursiv geschrieben ist. Das bedeutet, dass Sie niemals ein NamedElement an sich antreffen werden, sondern nur bestimmte Ableitungen davon, z.B. *Parameter*, *Class*, *Statemachine*, ...

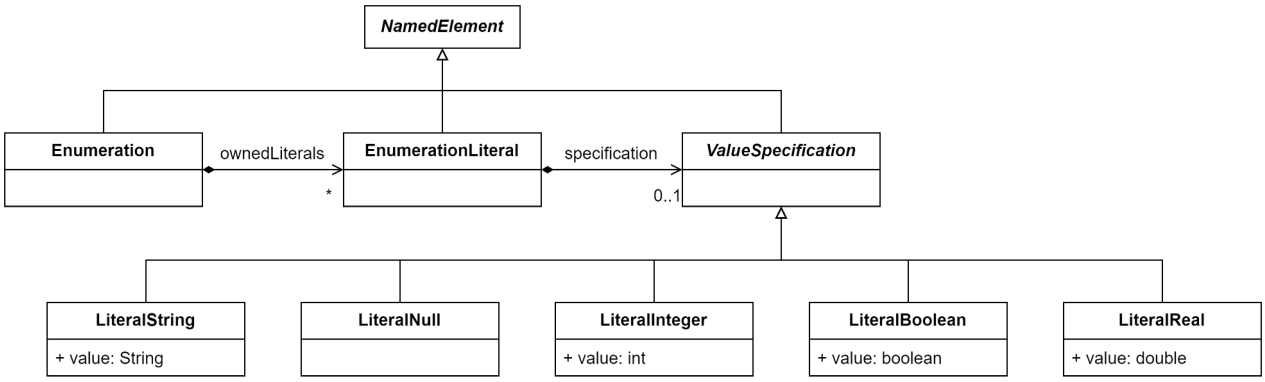

Abbildung 2. Enumeration

Die drei UML Elemente *Enumeration*, *EnumerationLiteral* und *ValueSpecification* sind jeweils von *NamedElement* abgeleitet und haben daher auch das Attribut *name* und eine optionale Beziehung zu einem *Namespace*.
Das UML Element *Enumeration* besitzt eine beliebige Anzahl von *EnumerationLitetrals*, welche ihrerseits optional eine *ValueSpecification* besitzen können.
ValueSpecification ist ein abstraktes UML Element. Einige der Ableitungen sind hier aufgelistet: *LiteralString*, *LitertalNull*, *LiteralInteger*, *LiteralBoolean*, *LiteralReal*.

Vergleichen Sie die Diagramme mit den mitgelieferten Templates des Codegenerators um zu ergründen, wie ein Template mit optionalen Beziehungen, 1:n Beziehungen und Generalisierungen umgehen kann.

Aufgabe 1: Parameter & Property

In dieser Aufgabe sollen Sie sich mit dem Arbeiten am Codegenerator vertraut machen. Ziel ist es, die Tests im Projekt *CodegenTest1* zu erfüllen. Um diese zu starten, öffnen Sie das Projekt, machen Sie einen Rechtsklick auf die Datei “CodegenAufgabe1.launch” und wählen Sie “Run As -> CodegenAufgabe 1”. Nach einem Augenblick sollte sich ein JUnit Fenster öffnen und die ausgeführten Tests anzeigen.
Um den Codegenerator um die fehlenden Funktionen zu erweitern, bearbeiten Sie im Projekt *Codegenerator* die Dateien *src/codegenerator/templates/ParameterTemplate.xtend* und *PropertyTemplate.xtend*.
Um herauszufinden, wie das getestete UML Modell jeweils aussieht, können Sie die Tests im Projekt CodegenTest unter *src/codegenerator.test.exercise1* anschauen. Sie können auch auf einen Test im JUnit Fenster doppelklicken um direkt zu diesem Test zu springen.
Wenn Sie in dem JUnit Fenster einen fehlgeschlagenen Test markieren, sollte darunter der *Failure Trace* angezeigt werden. Der oberste Eintrag zeigt für gewöhnlich an, welche Ausgabe des Templates erwartet und welche tatsächlich ausgegeben wurde. Falls der Code über mehrere Zeilen geht, können Sie per Doppelklick eine bessere Ansicht öffnen.
Um sich mit Xtend und dem Codegenerator vertraut zu machen, empfehlen wir, das Template EnumLiteralTemplate.xtend und die dazugehörigen Tests zu betrachten. Das Template ist bereits vollständig implementiert und von sehr geringem Umfang, zeigt aber das Aufrufen weiterer Templates, sowie den Umgang mit den '"Template Strings"'.

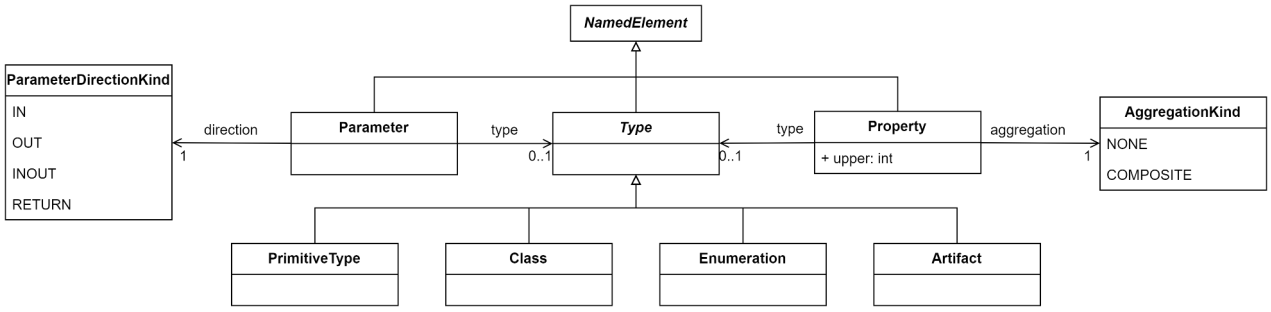

Abbildung 3. Parameter und Property

Ein paar Dinge sind zu diesem Diagramm anzumerken: die UML Elemente wie *PrimitiveType* sind über *Type* auch von *NamedElement* abgeleitet und haben daher *name* und *namespace*.
Die beiden Typen links und rechts sind Enumerationen. Das heißt, dass das Attribut *direction* von *Parameter* nur einen der vier aufgelisteten Werte annehmen kann. *AggregationKind* hat eigentlich noch einen dritten Wert, diesen brauchen wir jedoch nicht. Um diese Enumeration Werte im Code zu nutzen, muss jeweils noch „_LITERAL“ angehängt werden. Also z.B. `ParameterDirectionKind.IN_LITERAL`.

Aufgabe 2: StructuredClassifier & Operation

In dieser Aufgabe sollen die Tests in “CodegenAufgabe 2.launch” erfüllt werden. Die Templates hierfür sind *StructuredClassifierTemplate.xtend* und *OperationTemplate.xtend*, sowie nochmal *ParameterTemplate.xtend* und *PropertyTemplate.xtend*. Diese sollen C-Structs und Funktionen generieren, die unter anderem Parameter und Properties enthalten, die Sie bereits in Aufgabe 1 implementiert haben. Zudem gibt es noch eine Erweiterung an *ParameterTemplate* und *PropertyTemplate* vorzunehmen um Arrays generieren zu können und Enumerationen als *type* zu unterstützen.
Der Case “typedefinition” des Templates *EnumTemplate.xtend* hat einige Ähnlichkeiten zu dem, was im StructuredClassifierTemplate erwartet wird, hier können Sie wieder nach Inspiration schauen. Beachten Sie auch das Keyword SEPARATOR, welches vielleicht bei der Parameterliste hilfreich werden kann.
Machen Sie Gebrauch von den Templates, welche Sie in Aufgabe 1 implementiert haben, indem Sie diese über die `generate(…)` Funktion aufrufen.

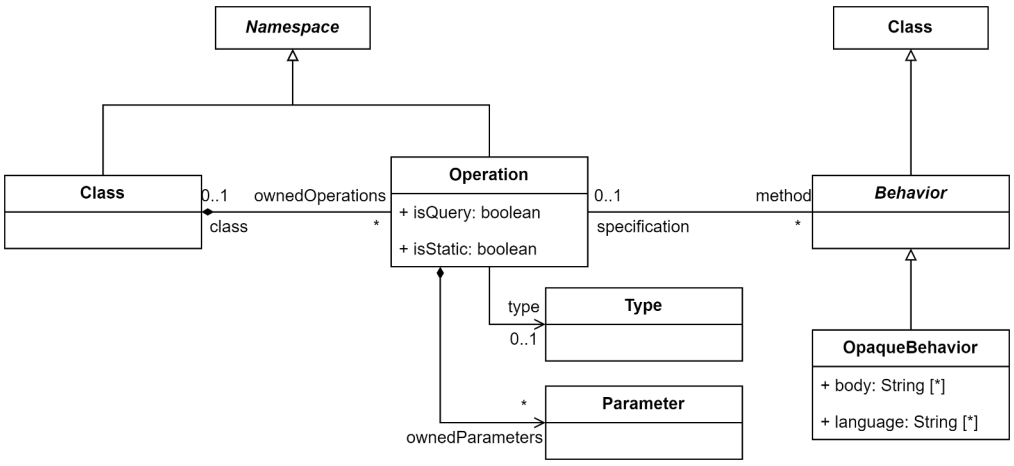

Abbildung 4. *Operation* und *Behavior*

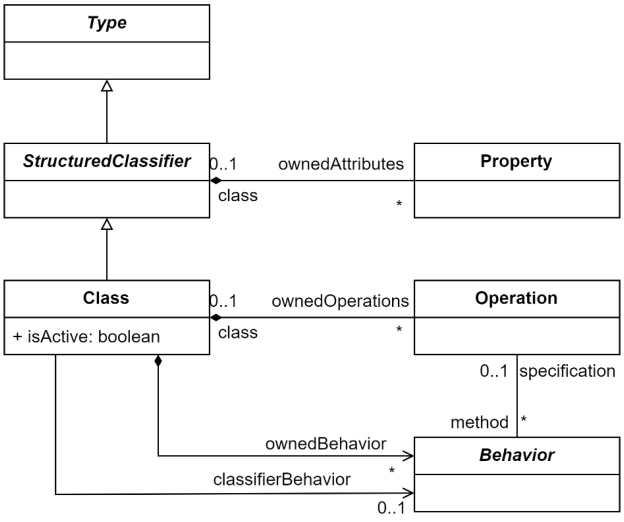

Abbildung 5. *StructuredClassifier* und *Class*

Aufgabe 3: Class

Nun gilt es, die beiden Pfade - Funktion und Daten - die Sie bisher separat implementiert haben, in klassischer OOP-Manier zusammenzuführen. Die Tests hierfür werden über "CodegenAufgabe3.launch" gestartet. Das Template, welches es zu implementieren gilt, ist das *ClassTemplate.xtend*. Beachten Sie, dass es sowohl Code für die .h, als auch für die .c Datei generieren sollen, Kontext ist also wichtig (siehe auch hier wieder *EnumTemplate.xtend*). Viele wichtige Beziehungen von *Class* finden Sie in Abbildung 5 auf der vorherigen Seite.
Beachten Sie die bereits vorhandene *generateIncludes* Methode, welche Ihnen einige Arbeit abnehmen kann, aber noch erweitert werden muss.
Um alle Tests zu bestehen ist es nötig, noch ein paar weitere Anpassungen am vorhandenen Code vorzunehmen: das *TypeTemplate* sowie *generateIncludes* müssen um *Enumeration*en erweitert werden, außerdem soll *generateIncludes* die Include-Pfade in alphabetischer Reihenfolge erzeugen.

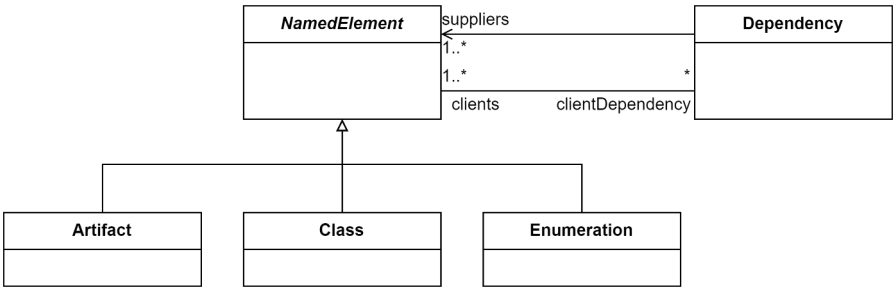

Abbildung 6. *Dependencies*

Hier noch ein Tipp:

```
val names = new HashSet<String>()

names += "A"
names += "A"
names += "C"
names += "B"

for (n : names.sort) {
    println(n)
}
// output:
// A
// B
// C
```

Listing 1: sortiert über ein Set iterieren

Aufgabe 4: Objekte

Um lauffähigen Code zu generieren, fehlen noch zwei Dinge: Instanzen und eine main-Funktion. Instanzen werden in den Dateien ihrer Klasse angelegt, Tests hierfür können über die "CodegenAufgabe4.launch" ausgeführt werden. Des Weiteren ist das *ValueSpecTemplate* zu erweitern, um *InstanceValues* korrekt zu generieren. Hierfür müssen Sie neben dem Template auch die *Uml2C* Klasse anpassen, um das Template für *InstanceValues* zu registrieren.
Für die Main-Funktion muss ein neues Template *MainTemplate.xtend* erstellt werden, welches für den UML Type *Model* eine Datei namens „main.c" erzeugt. Ein Beispiel für den Inhalt dieser ist in Listing 3 zu finden. Implementieren Sie für die Main-Funktion auch selber mindestens einen Test für ihr Template. Die Main-Funktion soll alle Instanzen von Klassen finden, welche ein `classifierBehavior` besitzen und dieses ausführen. Hierfür kann Abbildung 5 hilfreich sein.

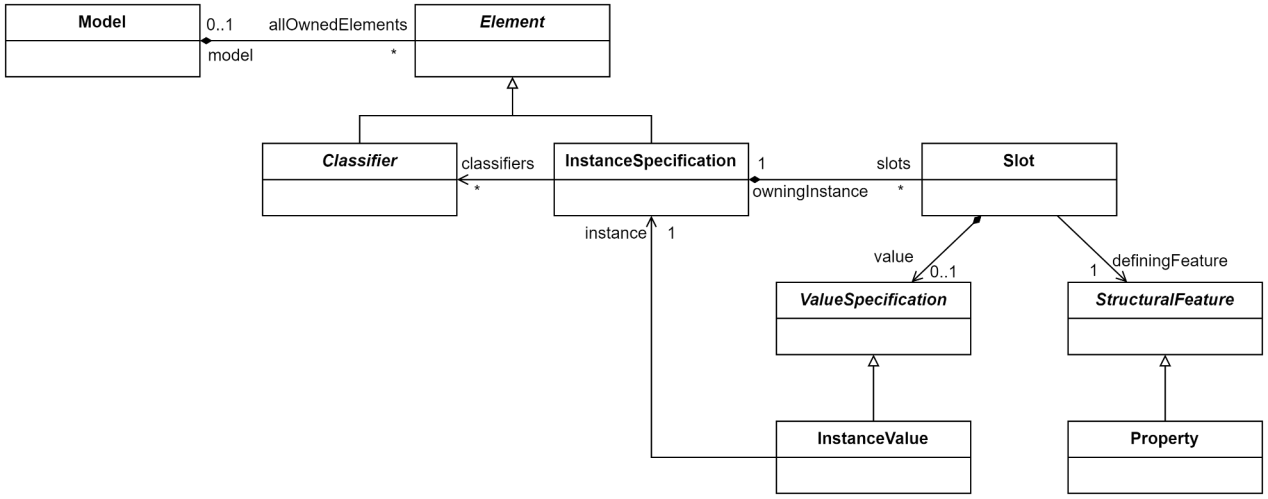

Abbildung 7. *InstanceSpecification*

```
val instances = umlModel.allOwnedElements
    .filter(InstanceSpecification)
    .filter[ i |
        i.classifiers.size == 1 &&
        i.classifiers.head instanceof Class
    ]
```

Listing 2: alle Instanzen in einem Modell finden

```
#include "Duhm/Game.h"

int main() {
      Duhm_Game_run(&Duhm_game);
}
```

Listing 3: Beispiel für die main.c Datei

Das Duhm Modell benutzt Artifacts, um vorhandene C Dateien einzubinden. Dafür muss ein neues Template angelegt und im Codegenerator registriert werden.

Um den fertigen Codegenerator mit einem echten Modell zu testen, stehen zwei weitere Projekte bereit:

Das Projekt *Duhm* enthält das zu testende Modell, sowie ein vorbereitetes Keil Microvision Projekt um die generierten Dateien zu kompilieren. Sie können dies in MerapiUML importieren oder einfach so entpacken.

CodegeneratorApp ist in die IDE zu importieren, um ihren Codegenerator auch außerhalb der Tests nutzen zu können. Machen Sie einen Rechtsklick auf das Projekt und wählen Sie *Run as → Java Application*. Es sollte in der Konsole ausgegeben werden, dass Argumente fehlen. Drücken Sie nun die Tasten *Strg + 3* und suchen Sie nach *Run Configurations...* . Wählen Sie links in dem Fenster CodegenIO aus und wechseln Sie anschließend auf den Tab *Arguments* und hängen folgendes an das Feld *Program Arguments* heran:

```
-loadModel "C:\...\Duhm\Duhm.uml" -outputCode "C:\...\Duhm\src-gen"
```

Wobei die Pfade in das von ihnen entpackte *Duhm* Projekt zeigen sollen.
Wenn ihr Codegenerator vollständig funktioniert, sollte der generierte Code mit Keil Microvision kompiliert und auf die Hardware geflasht werden können.