Praxis der Softwareentwicklung: Analyse formaler Eigenschaften von Wahlverfahren

Entwurfsdokument

Hanselmann Hecht Klein Schnell Stapelbroek Wohnig

Karlsruher Institut für Technologie

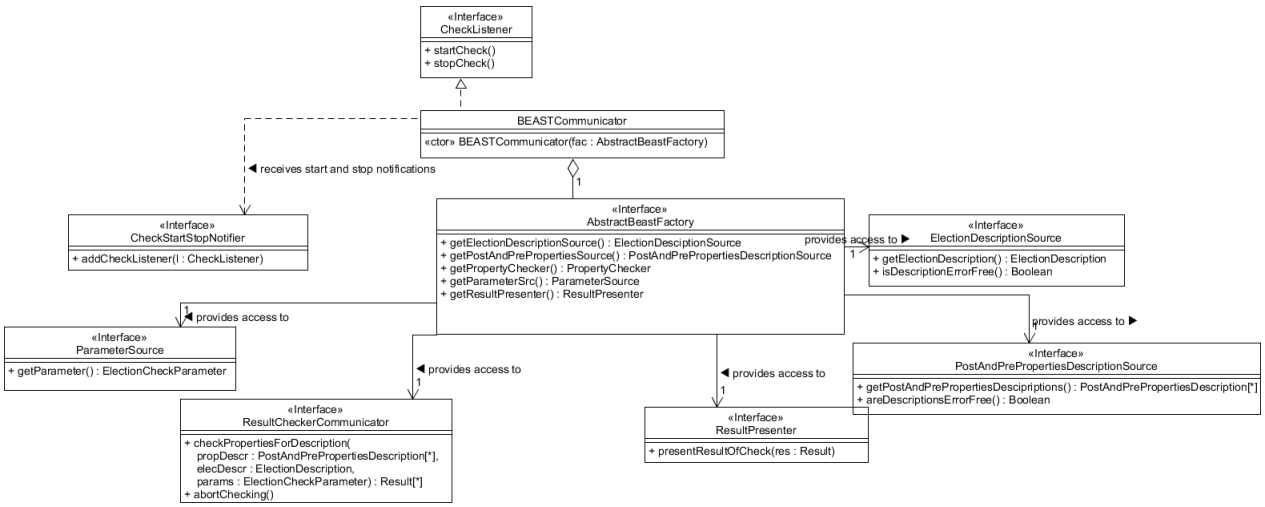
WS 2016/17

Inhaltsverzeichnis

1 Einleitung

1.1 Einleitung

Aus dem Pflichtenheft:
“Unser Programm ist im Wesentlichen eine sehr umfangreiche Schnittstelle, um mit CBMC zu kommunizieren. Es bietet dem Benutzer über eine GUI die Möglichkeit, formale Eigenschaften für Wahlverfahren sowie diese Wahlverfahren selbst anzugeben und zu editieren. Weiterhin liefert es Möglichkeiten, die Interaktion mit CBMC zu gestalten: Für wie viele Wähler etc. die Eigenschaft überprüft werden soll. Nach erfolgreicher Überprüfung durch CBMC bekommt der Benutzer schließlich eine Antwort des Programms, in der er bei Nichterfüllung der Eigenschaft ein Gegenbeispiel angezeigt bekommt. Wird kein Gegenbeispiel gefunden, so wird eine Erfolgsmeldung ausgegeben. All dies wird graphisch über die GUI aufbereitet."
Der vorliegende Entwurf setzt alle im Pflichtenheft gestellten Anforderungen um. Die High-Level Aufteilung ist folgende:



- Eine “Quelle“ für eine Beschreibung eines Wahlverfahrens - ElectionDescriptionSource
- Eine “Quelle“ für ein Beschreibungen der formalen Eigenschaften, welche diese Wahlverfahren erfüllen soll - PostAndPrePropertiesDescriptionSource
- Eine “Quelle“ für Eingabeparameter, für welche das Wahlverfahren die formalen Eigenschaften erfüllen soll - ParameterSource
- Eine Überprüfungsinstanz welche Wahlverfahren und Eigenschaften entgegennimmt das Ergebnis der Überprüfung zurückgibt - PropertyChecker

- Eine Komponente welche diese Ergebnisse darstellen kann - ResultPresenter
- Ein Kommunikator, welcher die Koordination zwischen obigen Komponenten übernimmt - BEASTCommunicator
- Eine Komponente welche dem Benutzer die Möglichkeit gibt diese Kontrolle zu starten und zu stoppen - CheckStartStopNotifier

Wir werden die genannten Komponenten so realisieren, dass sie die im Pflichtenheft gegebenen Anforderungen erfüllen. Um die Möglichkeit zu bieten, dass Komponenten über die genaue Klasse der anderen Komponenten Bescheid wissen ohne dass der Kommunikator dieses Wissen benötigt wird eine abstrakte Fabrik verwendet. Dies ist wichtig da wie bereits erwähnt die Komponenten nicht komplett orthogonal sind - der PropertyChecker muss über die interne Darstellung der Wahl- und Eigenschaftenbeschreibung informiert sein.

Wir implementieren die hier Beschriebenen Komponenten folgendermaßen:

ElectionDescriptionSource	C-Editor
PostAndPrePropertiesDescriptionSource	Eigenschaftenliste
ResultPresenter	Eigenschaftenliste
ParameterSource	Parametereditor
CheckStartStopNotifier	Parametereditor
BEASTCommunicator	Parametereditor
PropertyChecker	CBMCPropertyChecker

Ein weiteres sehr wichtiges Element der Software ist der Eigenschafteneditor. Er wird über die Eigenschaftenliste aufgerufen und gibt dem Benutzer die Möglichkeit, die Eigenschaften in einem Texteditor zu bearbeiten.

2 Zentrale Datentypen

2.1 Interne Typen

Zunächst wird beschrieben wie die verschiedenen Typen, welche in der Beschreibung des Wahlverfahrens und der formalen Eigenschaften vorkommen modelliert werden. Diese Klassen werden von allen der folgenden Datentypen verwendet. Zum einen müssen die diversen Typen, welche die symbolischen Variablen annehmen können repräsentiert werden:

- Wähler
- Kandidat
- Sitz

Hinzu kommen die verschiedenen Arten der Stimmabgabe:

- Single-choice : Kandidat
- Zustimmungswahl : Liste von Ja/Nein (Zustimmung pro Kandidat)
- Gewichtete Wahl : Liste von Integern (Zahlenwert pro Kandidat)
- Präferenzwahl : Liste von Kandidaten (Ranking)

und für die verschiedenen Arten an Wahlergebnissen:

- Kandidat
- Liste von Integern (Anzahl zugeteilter Sitze pro Partei)

Das Makro `VOTE_SUM_FOR_CANDIDATE()` sowie die Konstanten V, C und S geben ebenfalls einen Zahlenwert zurück. Eine Modellierung dieser Typen muss folgende Probleme lösen:

- Die boolschen Ausrücke müssen auf Korrektheit untersucht werden können:
  - Es dürfen nur Vergleichbare Typen miteinander verglichen werden
  - Es darf nur bei Listen von Typen auf Listenelemente zugegriffen werden. Die symbolische Variable welche als Index verwendet wird ist vom richtigen Typ.
- Bei der Codeerzeugung müssen die Typen richtig repräsentiert und verglichen werden.

Diese Probleme lassen sich durch folgende Architektur lösen:

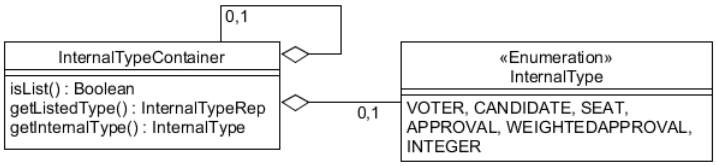


Abbildung 2.1: Die interne Modellierung der verschiedenen Typen

Da getestet wird ob zwei Typen miteinander verglichen werden müssen ist es unumgänglich deren Typinformation konkret zu speichern. Dadurch können die verschiedenen Teile des Systems welche diese Information benötigen sie unkompliziert erfragen.

2.2 Beschreibung der Wahlverfahren

Die Repräsentation der Beschreibung der Wahlverfahren wird intern übernommen durch die Klasse `ElectionDescription`. Dargestellt wird der Wahlvorgang in der Programmiersprache C. Genauer muss in dem C Code eine Funktion namens voting definiert sein. Argumente und Rückgabetyp dieser Funktion hängen von der Art der Stimmenabgabe und des Wahlergebnisses ab. Diese werden repräsentiert durch die Klasse `ElectionTypeContainer`. Ein Problem ist dass deren Felder lowerBound und upperBound nur für die gewichtete Wahl von Belang sind. Eine andere Möglichkeit wäre gewesen für jede Art von Input und Output Klassen zu modellieren welche abstrakte Methoden überschreiben. Diese würden dann in der für die Codegenerierung zuständigen Klasse die korrekten Methoden aufrufen um die zusätzlich nötigen Vorbedingungen zu generieren. Durch diese Verwendung abstrakter Methoden würde die Abfrage des Inputtypes dem Compiler überlassen werden. Allerdings müsste dafür der `ElectionInputType` über all seine Verwendungszwecke Bescheid wissen. Da seine Verantwortung vorerst darin liegt zwischen CEditor, Eigenschafteneditor und Überprüfungsinstanz zu kommunizieren entschieden wir uns für diesen Entwurf. Hinzu kommt dass auch der alternative Entwurf bei einführen eines neuen Input Typs Veränderungen an der für Codegenerierung zuständigen Klasse fordern würde.

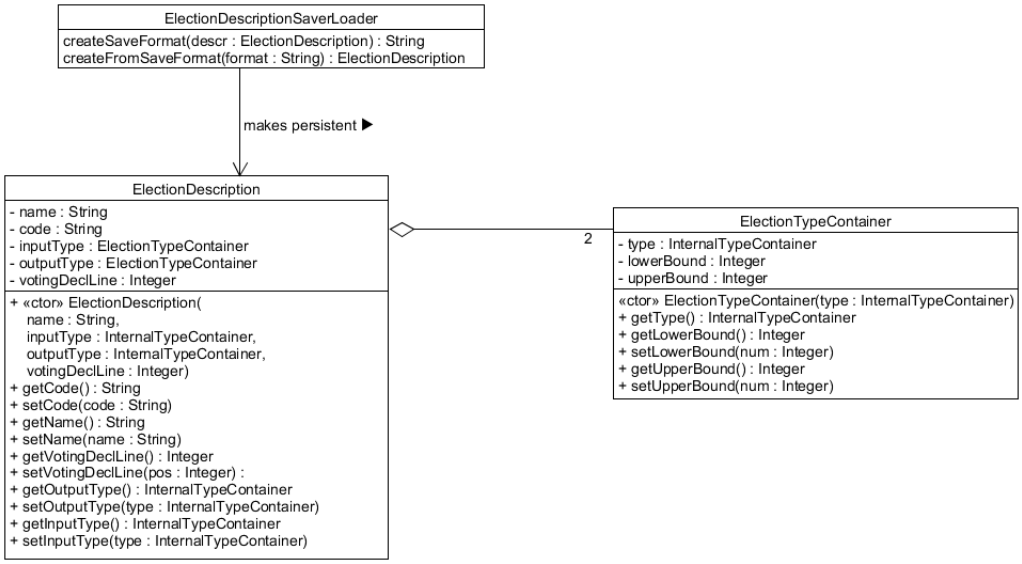


Abbildung 2.2: Die Modellierung der ElectionDescription

`ElectionDescription` ist eine der Datenstrukturen, welche laut Pflichtenheft Persistenz erlauben müssen. Dies wird durch die Klasse `ElectionDescriptionSaverLoader` übernommen.

2.3 AST-Darstellung boolscher Ausdrücke

Im Pflichtenheft werden folgende Anforderungen an die Syntax zur Beschreibung boolscher Ausdrücke gestellt:

- binäre Operatoren `&&`, `||`, `==>`, `<==>`
- aus C bekannte Vergleichsoperatoren `==`, `!=`, `>`, `<`, `>=`, `<=`
- symbolische Variablen vom Typ Wähler, Kandidat oder Sitz
- Quantoren in Form von Makros
- Zugriff auf Stimmen bzw. Ergebnisse mehrerer Wahldurchläufe
- Konstanten: Anzahl Wähler, Kandidaten und Sitze
- Abfrage der Stimmsumme für einen Kandidaten durch `VOTE_SUM_FOR_CANDIDATE()`

Hinzu kommt jetzt noch die Möglichkeit, einen boolschen Ausdruck zu Verneinen. Dazu dient das Token `!` gefolgt von einem boolschen Ausdruck.

Die genaue Beschreibung der Antlr-Grammatik ist Implementierungsdetail. Es wird dann aus dem von Antlr erstellten Syntax-Baum ein AST generiert, welcher die Erzeugung des C-Codes erleichtert. Dieser AST muss in der Lage sein, alle hier aufgezählten Konstrukte zu ermöglichen.

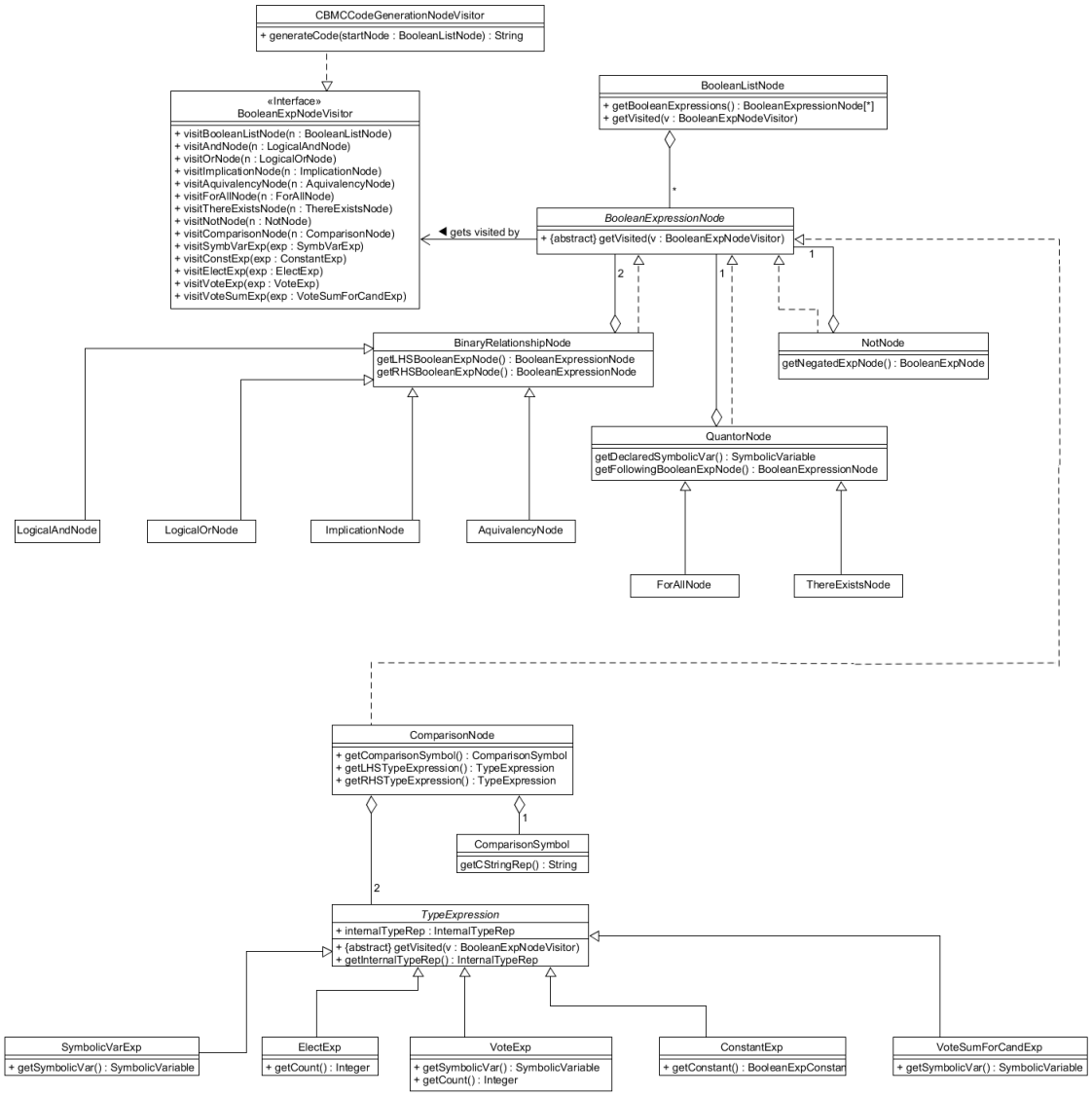


Abbildung 2.3: Der AST, welcher aus boolschen Ausdrücken erstellt wird zusammen mit Visitor

Dieser AST ermöglicht die Codeerzeugung per Visitor Pattern. Alle oben genannten Anforderungen sind abgedeckt:

Anforderung	Klasse
binäre Operatoren	BinaryRelationshipNode
Vergleichsoperatoren	ComparisonNode
symbolische Variablen	SymbolicVarExp
Zugriff auf Stimmen und Ergebnisse	VoteExp, ElectExp
Konstanten	ConstantExp
Abfrage der Stimmsummen	VoteSumForCandExp
Quantoren	QuantorNode
Verneinung	NotNode

`VOTE_SUM_FOR_CANDIDATE()` gibt je nach Art der Stimmabgabe etwas anderes zurück:

- Jeder wählt einen Kandidat: Anzahl Personen, welche für den Kandidaten gestimmt haben
- Jeder listet jeden Kandidaten nach absteigender Präferenz: Plätze aufsummiert
- Jeder bewertet jeden Kandidaten einmal mit Ja und einmal mit Nein: Anzahl Wähler, die den Kandidaten mit Ja bewerten
- Jeder gibt jedem Kandidaten eine Zahl zwischen gewählter Ober- und Untergrenze: Summe der Bewertungen

2.4 Beschreibung formaler Eigenschaften

Eine Beschreibung formaler Eigenschaften teilt sich auf in Vorbedingungen und Nachbedingungen. Diese müssen auch bei der C-Codeerzeugung unterschiedlich gehandhabt werden (siehe). Sowohl in Vor- als auch Nachbedingungen können symbolische Variablen verwendet werden. Diese haben einen Namen sowie einen Typ. Dieser Typ repräsentiert entweder einen Wähler, Kandidaten oder Sitz. Der Name muss in der formalen Eigenschaft einzigartig sein. Das zur Benennung erlaubte Format ist mit der Variablenbenennung in C identisch. Eine Beschreibung hat ebenfalls einen einzigartigen Namen. Verwendet wird der Datentyp von der Eigenschaftenliste, dem Eigenschafteneditor und der Überprüfungsinstanz. Die Eigenschaftenliste kann neue Beschreibungen erzeugen und den Namen bereits existierender Beschreibungen verändern. Der Eigenschafteneditor gibt dem Benutzer die Möglichkeit Vor- und Nachbedingungen sowie die symbolische Variablen zu editieren. Die Überprüfungsinstanz verbindet zu C-Code welcher von CBMC überprüft werden kann. Dazu benötigt es Zugriff auf die AST-Darstellung der Vor- und Nachbedingungen. Wie bei allen internen Datentypen muss es die Möglichkeit geben sie zu Speichern und zu Laden.

Um all diesen Anforderungen gerecht zu werden sind die verschiedenen Verantwortungsbereiche auf verschiedene Klassen aufgeteilt. Zur internen Repräsentation der Beschreibung formaler Eigenschaften dient die Klasse `PostAndPrePropertiesDescription`. Vor- und Nachbedingungen werden repräsentiert von `FormalPropertiesDescription`. Symbolische Variablen werden repräsentiert von `SymbolicVariable`. Die Klasse `SymbolicVariableList` ist dafür verantwortlich dass nur symbolische Variablen mit korrekten Namen erstellt werden. `PostAndPrePropertiesDescription` wird zwischen den Packages Eigenschaf-

tenliste, Eigenschafteneditor und Überprüfungsinstanz gereicht und dient als Interface zu den anderen genannten Klassen.

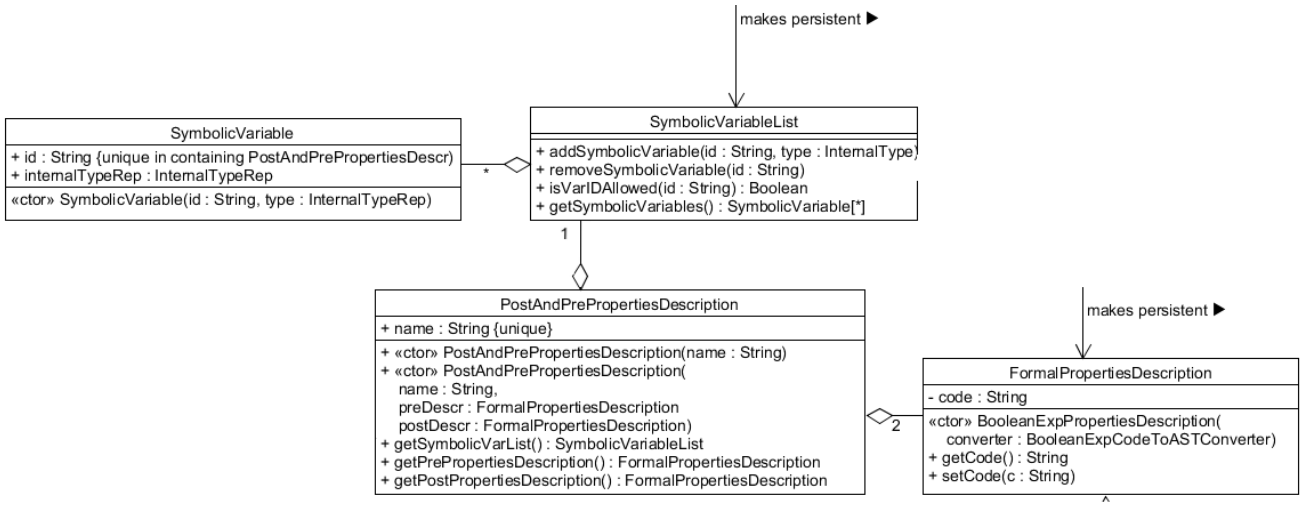


Abbildung 2.4: Klassen welche die Beschreibung formaler Eigenschaften Repräsentieren

Die Klasse `FormalPropertiesDescription` enthält keine Instanz der AST-Darstellung (siehe) der von ihr repräsentierten formalen Eigenschaften. Stattdessen ist das Feld `code` die Repräsentation der Eigenschaft als Code. Die Klasse verwendet eine weitere Klasse, `BooleanExpToASTConverter` um bei Bedarf den AST zu erzeugen. Da die genaue Grammatik Implementierungsdetail ist und daher die Knotennamen des Syntaxbaumes noch unbekannt sind, sind die verschiedenen visit-Methoden noch nicht angegeben.

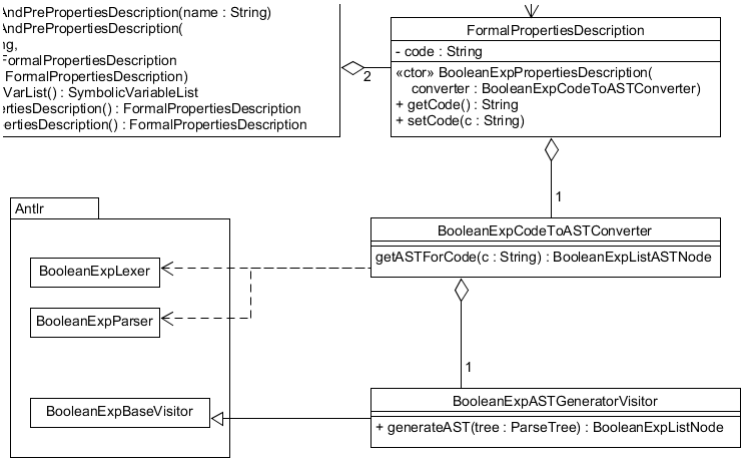


Abbildung 2.5: Klassen welche die Erzeugung der AST-Darstellung der Beschreibung formaler Eigenschaften übernehmen

Diese Klasse wiederum verwendet `BooleanExpASTGeneratorVisitor`. Diese erbt von der Klasse `BooleanExpBaseVisitor`. Letztere wird von ANTLR automatisch erzeugt. Diese Aufteilung hat mehrere Vorteile:

- Erleichtertes Speichern und Laden der Klasse `FormalPropertiesDescription` da diese nun als einziges Feld einen String beinhaltet
- Erzeugung des AST nur wenn er wirklich benötigt wird. Dies spart Ressourcen.
- Trennung der Abstraktionen. `FormalPropertiesDescription` dient eher zur internen Repräsentation eines abstrakten Objektes. Die Erstellung eines ASTs ist ein konkretes Verfahren welches mit dieser Aufgabe nur indirekt zusammenhängt.

Das Speichern und Laden eines `PostAndPrePropertiesDescription` Objektes wird, ähnlich wie bei den anderen Datentypen, ebenfalls von spezialisierten Klassen übernommen. Dabei wird die Hierarchie der repräsentierenden Klassen widergespiegel. `PostAndPrePropertiesDescriptionSaverLoader` verwendet `FormalPropertiesDescriptionSaverLoader` sowie `SymbolicVariableListSaverLoader`. Diese sind jeweils dafür verantwortlich aus den entsprechenden Objekten String-Repräsentationen zu erzeugen oder aus entsprechenden String-Repräsentationen Objekte zu erzeugen.

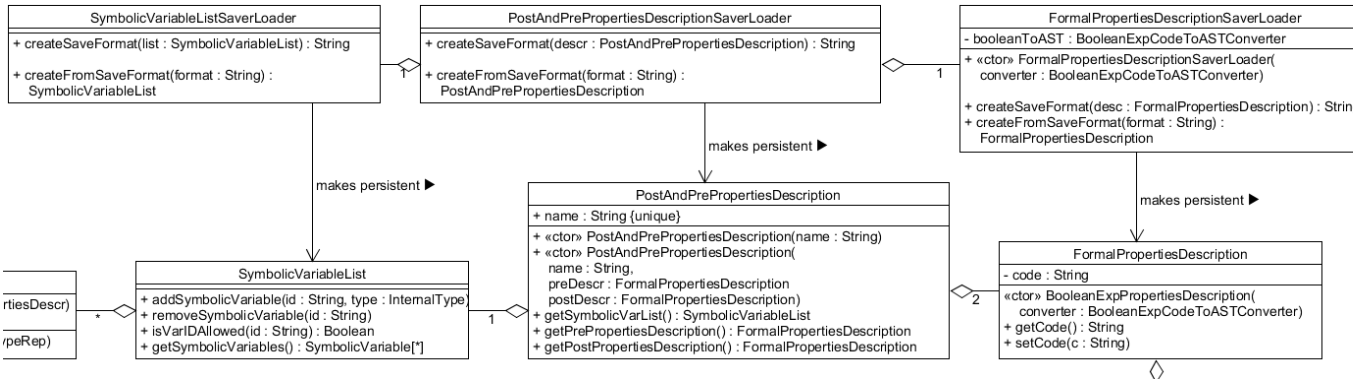
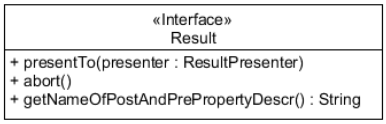


Abbildung 2.6: Klassen welche korrekt formatierte String-Repräsentationen der entsprechenden Objekte erstellen, durch welche diese Objekte erzeugt werden können. Diese Strings werden dann gespeichert.

2.5 Ergebnisse der Überprüfung

Ergebnisse einer Überprüfung werden über eine Klasse kommuniziert welche das `Result`-Interface implementiert. In unserem Fall ist dies die Klasse `CBMCResult`.



2.6 Parameter

Die Parameter werden in der Klasse `ElectionCheckParameter` übergeben.

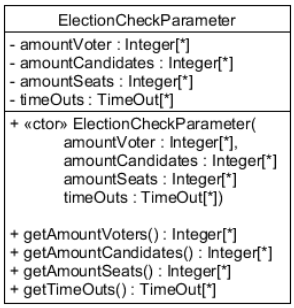
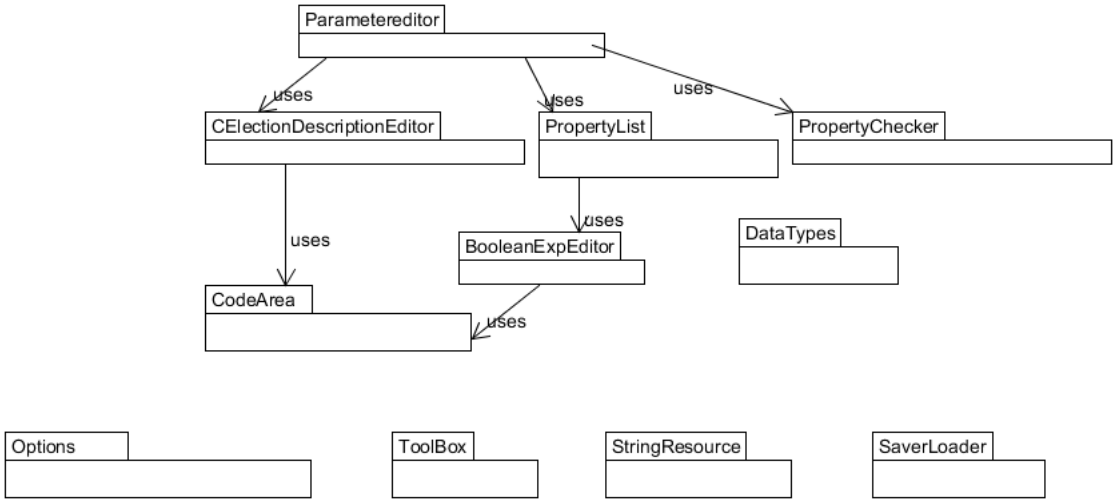


Abbildung 2.7: Klasse welche Parameter repräsentiert

3 Packages

3.1 Überblick und Kommunikation



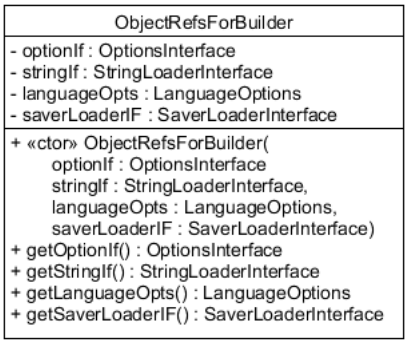
Es existieren folgende Packages:

- `DataTypes`: Die Datentypen welche geladen und gespeichert werden können. Werden auch zur Kommunikation zwischen den Packages verwendet.
- `Options`: Vom Benutzer konfigurierbares Verhalten `SaverLoader` Stellt Funktionalität zum Laden und Speichern aller Datentypen sowie Optionen bereit
- `StringResource`: Stellt Funktionen bereit `String`s zu laden welche dem Benutzer präsentiert werden und sich je nachgewählter Sprache ändern.
- `CodeArea`: Funktionalität welche sowohl `CElectionDescriptionEditor` als auch `BooleanExpEditor` benötigen
- `CElectionDescriptionEditor`: Editieren der `ElectionDescription`. Dient als `ElectionDescriptionSource`
- `BooleanExpEditor`: Editieren der `PostAndPrePropertiesDescription`
- `PropertyList`: Anzeigen und Editieren mehrerer `PostAndPrePropertiesDescription`. Dient als `PostAndPrePropertiesDescriptionSource`. Zeigt `Result`-Objekte an.
- `ParameterEditor`: Editieren der `ElectionCheckerParameter`.
- `PropertyChecker`: Überprüfen der `ElectionCheckerParameter`, `ElectionDescription` und `PostAndPrePropertiesDescription`
- `ToolBox`: Diverse Hilfsklassen

4 Architektur

4.1 Allgemeines

Im Folgenden wird erläutert wie die Klassen des Softwaresystems die Anforderungen des Pflichtenheftes erfüllen. Zunächst werden Muster und Methoden erklärt welche sich immer gleich verhalten. Auf diese wird danach nicht mehr extra eingegangen.
getter und setter: Diese geben bzw. setzen das gleichnamige Attribut.
Erstellen von Objekten: Alle Objekte welche als Attribut ein `JFrame` haben werden nach dem Builder-Pattern erstellt. Dabei nimmt die Builder-Klasse ein Objekt des Types `ObjectRefsForBuilder` entgegen.

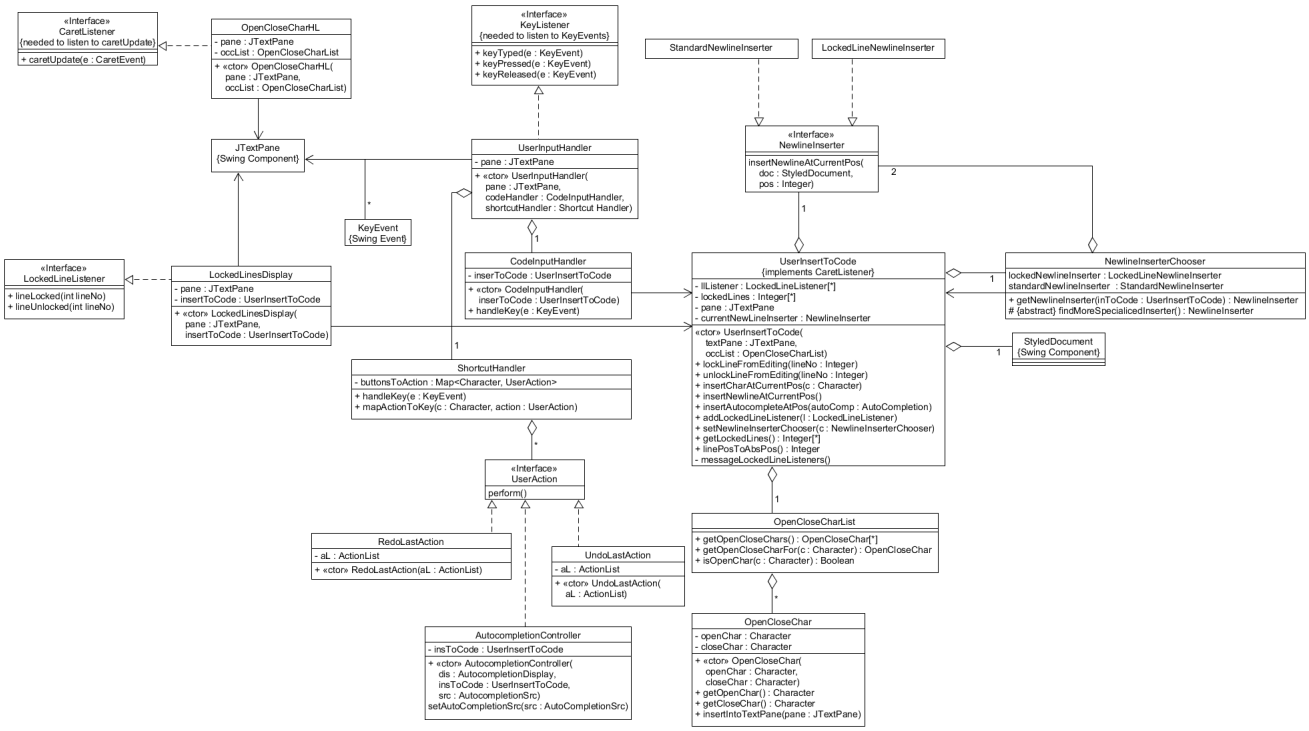


Dieses gibt die zum Erstellen dieser Objekte notwendigen Interfaces. Zur Erstellen von Menüs und Toolbars werden `ToolBarHandler` sowie `MenuBarHandler` verwendet
Vom Benutzer startbare Aktionen: Da diese auf verschiedene Arten wie Shortcuts, Menü- oder ToolBar Klicks gestartet werden können implementieren alle das Interface `UserAction`. Diese verwenden dann `ActionListener` welche mit den entsprechenden `JMenuItem`s bzw. `JButton`s verlinkt sind.

4.2 CodeArea

Erfüllte FAs: /FM1010/, /FM1020/, /FS1070/, /FS1080/, /FS1090/, /FS1100/, /FK1120/, /FK1130/, /FK1150/, /FM2010/, /FM2020/, /FS2130/, /FS2140/, /FS2150/, /FK2140/
Das Package CodeArea wird von den Packages CElectionEditor und BooleanExpEditor verwendet. Es übernimmt Aufgaben welche beiden zufällt. Zusätzlich bietet es Möglichkeiten Funktionalität zum Syntax Highlighting und zur Fehlerfindung hinzuzufügen. Als Interface nach außen dient die Klasse `CodeArea`.

4.2.1 Umgang mit Eingaben des Benutzers



Erfüllte FAs: /FM1020/, /FS1100/
Die Klasse `UserInputHandler` verwendet das Observer-Pattern um auf KeyEvents der JTextPane zu hören. Dazu implementiert er das KeyListener Interface. Er implementiert die Logik nach welcher die Benutzereingaben entweder an `CodeInputHandler` oder `ShortcutHandler` weitergeleitet werden.

CodeInputHandler

Erfüllte FAs: /FM1020/
Methoden:

- `handleKey` : Leitet den von `UserInputHandler` gegebenen Codeinput an `UserInsertToCode` weiter. Handelt es sich bei dem Input um Enter, so ruft er `insertNewlineAtCurrentPos` auf, ansonsten übergibt er den in dem KeyEvent enthaltenen Character an `insertCharAtCurrentPos`.

UserInsertToCode

Erfüllte FAs: /FK1120/, /FK1130/, /FK2140/, /FS1110/
Diese Klasse benutzt das Policy-Pattern um Neue Zeilen einzufügen. Ihre Verantwortung ist das korrekte Umwandeln von Benutzereingaben in Code.
Attribute:

- `NewLineInserter`: Übernimmt das Einfügen neuer Zeilen.

- `NewlineInserterChooser`: Wählt je nach Kontext den korrekten `NewLineInserter` aus.
- `OpenCloseCharList`: Wird verwendet um schließbare Zeichen wie offene Klammern zusammen mit ihrem Komplement einzufügen.
- `llListener`: Werden benachrichtigt sobald sich der Sperrstatus einer Zeile ändert. Wird verwendet von der `LockedLinesDisplay` Klasse.
- `StyledDocument`: Wird verwendet um dem Benutzer den erstellten Code zu präsentieren

Methoden:
- `lockLineFromEditing`: Wird von `CElectionEditor` benötigt um sicher zu gehen dass die `voting`-Definition nicht willkürlich geändert wird. Verhindert dass der Benutzer die angegebene Zeile verändert.
- `insertCharAtCurrentPos`: Fügt an der momentanen Caret-Position das gegebene Zeichen ein. Handelt es sich um ein schließbares Zeichen hängt es automatisch das Komplement an.
- `InsertNewlineAtCurrentPos`: Fügt an der momentanen Caret-Position eine neue Zeile ein. Der genaue Vorgang wird im momentanen `NewLineInserter` implementiert.
- `getCodeAtCurrentPos`: Gibt die Zeichen zurück welche links und rechts der momentanen Caret-Position sind.
- `insertAutoCompleteAtPos`: ruft die `insertIntoStyledDoc` Methode des übergebenen `AutoCompletion`-Objektes auf.
- `addLockedLineListener`: Fügt das übergebene Objekt der `llListener`-Liste hinzu
- `setNewlineInserterChooser`: Wird von `BooleanExpCodeArea` und `CElectionCodeArea` verwendet um einen spezialisierten `NewlineInserterChooser` zu injizieren.
- `MessageNewlineListener`: Benachrichtigt alle llListener falls sich der Sperrstatus einer Zeile ändert.

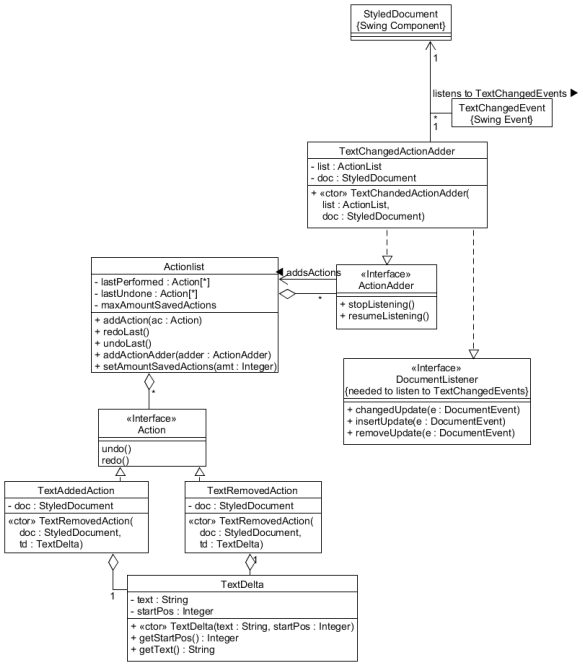
ShortcutHandler

Erfüllte FAs: /FS1100/
Attribute:
- `buttonsToAction`: Repräsentiert die Abbildung von Tastaturcodes auf entsprechende Aktionen (`UserAction`)
- `UserAction`: Implementiert das Ausführen der durch den Shortcut aufzurufenden Aktion

Methoden:
- `handleKey`: Ruft die entsprechende `UserAction` auf oder tut nichts falls der Shortcut nicht existiert.
- `mapActionToKey`: Legt einen neuen Shortcut fest. Beim gleichzeitigen Drücken von STRG und dem gegebenen Zeichen wird die Aktion ausgeführt.

4.2.2 Aktionen Rückgängig machen und wiederholen

Erfüllte FAs: /FS1100/
Polymorphie wird verwendet um die Reihenfolge der ausgeführten Aktionen unabhängig von deren genauen Typ und Implementierung zu machen. Da die `Actionlist` Aktionen nur in umgekehrter Reihenfolge ihrer Ausführung rückgängig macht ist garantiert dass das System immer in einem korrekten Zustand ist. Die Verwendung von Polymorphie ermöglicht leichtes Hinzufügen neuer Funktionalität durch `CElectionEditor`. Dieser führt noch die neue Aktion `changeElectionType` ein. Da es beim Rückgängig machen von Aktionen zu Problemen kommt wenn gleichzeitig neue Aktionen generiert werden können der `Actionlist ActionAdder` hinzugefügt werden. Diese werden benachrichtigt sobald sie das Generieren neuer Aktionen unterbrechen oder fortsetzen sollen. Der `TextChangedActionAdder` wird durch das Observer-Pattern von dem StyledDocument benachrichtigt sobald sich dessen Inhalt ändert.



Action

Methoden

- `undo`: Die von der Klasse repräsentierte Aktion wird Rückgängig gemacht
- `redo`: Die von der Klasse repräsentierte Aktion wird erneut ausgeführt

TextAddedAction und TextRemovedAction

Implementieren die Logik zum hinzufügen von beliebigem Text and beliebiger Stelle.
Attribute

- `doc`: das `StyledDocument` in welchem der Text Hinzugefügt bzw. Entfernt werden soll

TextDelta

Repräsentiert eine Veränderung eines Textes
Attribute

- `text`: Der Text welcher hinzugefügt oder entfernt wurde
- `pos`: Die Stelle an welcher `text` hinzugefügt oder entfernt wurde

ActionAdder

Klassen welche dieses Interface implementieren können benachrichtigt werden sobald sie aufhören sollen Aktionen zu generieren und zu der Liste hinzuzufügen.

Methoden

- `stopListening`: Der `ActionAdder` soll das Generieren von Aktionen unterbrechen
- `resumeListening`: Der `ActionAdder` soll das Generieren von Aktionen fortsetzen

TextChangedActionAdder

Übernimmt das Generieren von `TextAddedAction` und `TextRemovedAction`

Actionlist

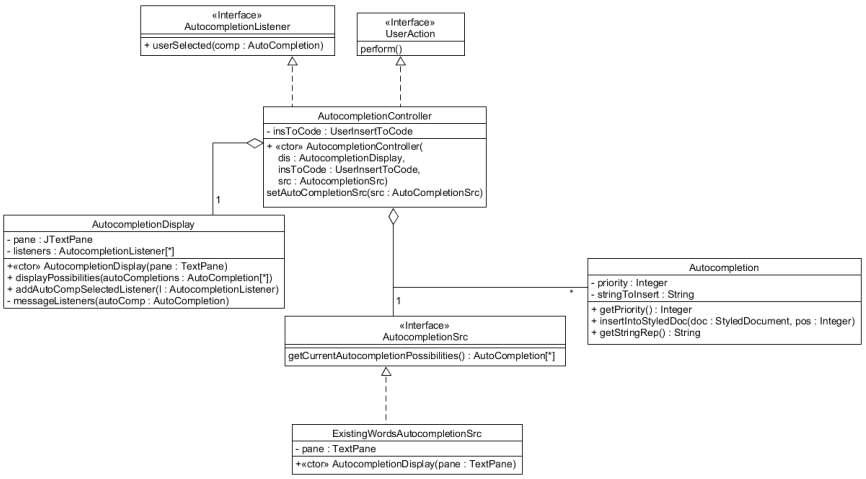
Attribute:

- `ActionAdder`: Fügen der `Actionlist` `Action`s hinzu. Bei Rückgängig machen bzw. Wiederholen einer Aktion müssen diese benachrichtigt werden diese Aktionen nicht nachzuverfolgen
- `lastUndone`: zuletzt Rückgängig gemachte Aktionen. Wird von `redoLast` verwendet
- `lastPerformed`: zuletzt ausgeführte Aktionen. Wird von `undoLast` verwendet

Methoden:

- `addAction`: Fügt die übergebene `Action` an das Ende von `lastPerformed`.
- `redoLast`: Führt die zuletzt rückgängig gemachte Aktion erneut aus
- `undoLast`: Führt die zuletzt ausgeführte Aktion erneut aus
- `addActionAdder`: Der übergebene `addActionAdder` wird zukünftig benachrichtigt, sobald er das generieren von Aktionen unterbrechen oder fortsetzen soll
- `setMaxAmountSavedActions` Legt fest bis zu welcher Anzahl alle Aktionen gespeichert werden.

4.2.3 Codevervollständigung



Erfüllte FAs: /FK1130/, /FK2140/

Die Aufteilung der Verantwortungen folgt dem MVC Prinzip. Model (`AutocompletionSrc`), View (`AutocompletionDisplay`) und Controller

(AutocompletionController). Die Implementierung der
(AutocompletionSrc) kann von außen injiziert werden. Dadurch kann die Codevervollständigung von der CElectionCodeArea und der BooleanExpCodeArea angepasst werden. Kommunikation von (AutocompletionDisplay) zu (AutocompletionController) erfolgt durch das Observer-Pattern.

AutocompletionController

Diese Klasse ermöglicht ein Anzeigen der Autocompletion Vorschläge per Shortcut (siehe 4.2.1). Dazu übergibt sie der Klasse
AutocompletionDisplay eine von
AutocompletionSrc erhaltene Liste an
Autocompletion.
Attribute

- insToCode: Die UserInsertToCode-welche die gewählte Autompletion in den Code einfügen wird

Autocompletion

Diese Klasse dient der Kommunikation zwischen AutocompletionSrc und AutocompletionDisplay
Attribute

- priority: Bestimmt wie an welcher Position in der Liste der Vorschlag angezeigt wird
- stringToInsert: Der String welcher in den Code eingefügt werden soll

Methoden

- insertIntoStyledDoc: Fügt stringToInsert in das übergebene StyledDocumentan der Stelle pos ein.

AutocompletionDisplay

Attribute

- listeners: Werden signalisiert sobald der Nutzer eine Autocompletion wählt
- pane: Die JTextPane in welcher die Vorschläge angezeigt werden sollen

Methoden

- displayPossibilities: Wird von AutocompletionController aufgerufen wenn der Benutzer signalisiert dass er zwischen den verfügbaren Autocompletion Vorschlägen wählen möchte.
- addAutoCompSelectedListener Der übergebeneAutocompletionListener wird zur Liste listeners hinzugefügt.

AutocompletionSrc

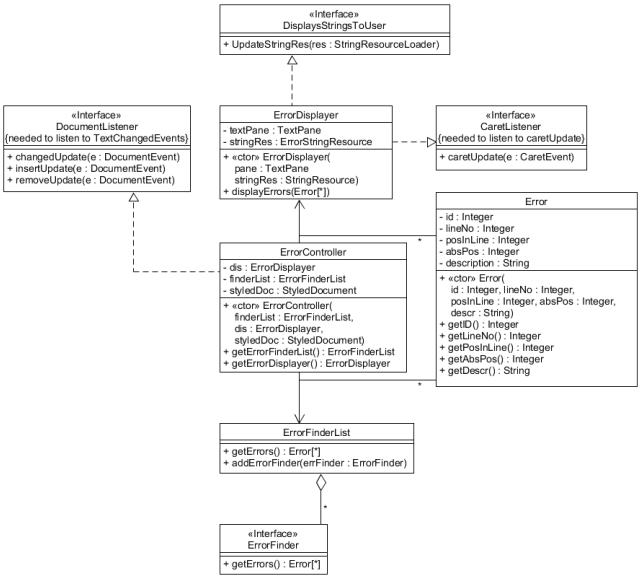
Klassen welche dieses Interface implementieren sind dafür verantwortlich, passende Autocompletion Vorschläge zu finden

ExistingWordsAutocompletionSrc

Dies ist die Implementierung des Interfaces `AutocompletionSrc` welche standardmäßig von `CodeArea` verwendet wird. Sie gibt als Vorschläge eine Liste bereits verwendeter Wörter aus, wobei die Priorität der Häufigkeit entspricht. Attribute

- `pane`: Die TextPane in welche den Text enthält dessen Worte als Autocompletion Vorschläge verwendet werden.

4.2.4 Asynchrones Auffinden und Anzeigen von Fehlern



Erfüllte FAs: /FS1090/, /FS2140/
Da der CodeArea von vornherein nicht die Sprache des angezeigten Codes bekannt ist kann diese nicht das Finden von Fehlern übernehmen. Dies wird durch Implementierungen von `ErrorFinder` injiziert.

ErrorController

Übernimmt die Interaktion zwischen `ErrorDisplayer` und `ErrorFinderList`

ErrorDisplayer

Stellt die gegebenen `Error`-Objekte in der `JTextPane` dar

ErrorFinderList

Leitet Signale von `ErrorDisplayer` an die `ErrorFinder` weiter

ErrorFinder

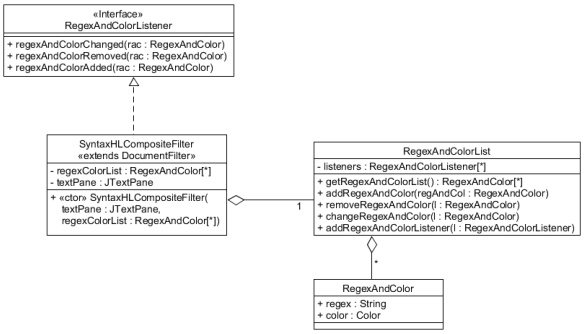
Klassen welche dieses Interface implementieren finden und erstellen `Error`-Objekte

Error

Repräsentiert einen gefundenen Fehler

- `id`: Fehlernummer zur Identifikation des Fehlers
- `ineNo`: Zeile in welcher der Fehler auftritt
- `posInLine`: Position innerhalb der Zeile in welcher der Fehler auftritt
- `absPos`: Absolute Position des Fehlers
- `description`: String zur Identifikation des Fehlers

4.2.5 Syntax Highlighting



Erfüllte FAs: /FS1070/, /FS2130/
Syntax Highlighting auf einer `JTexTPane` funktioniert durch Implementieren eines DocumentFilters. Dieser kann dann beliebige reguläre Ausdrücke farblich hervorheben. Das Observer Pattern wird verwendet um beim Hinzufügen neuer Regulärer Ausdrücke informiert zu werden

RegexAndColor

Regulärer Ausdruck und die Farbe in welcher dieser hervorgehoben werden soll

RegexAndColorList

Wird verwendet um neue `RegexAndColor` hinzuzufügen

RegexAndColorListener

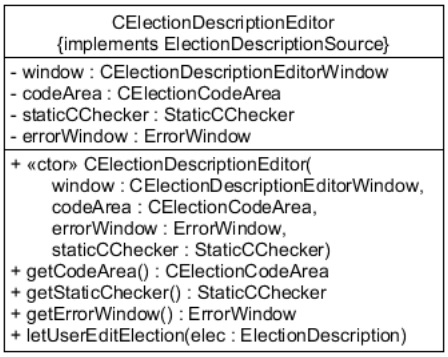
Wird verwendet bei Änderungen in der `RegexAndColorList` informiert zu werden

SyntaxHLCompositeFilter

Implementiert das Syntax Highlighting

4.3 CElectionDescriptionEditor

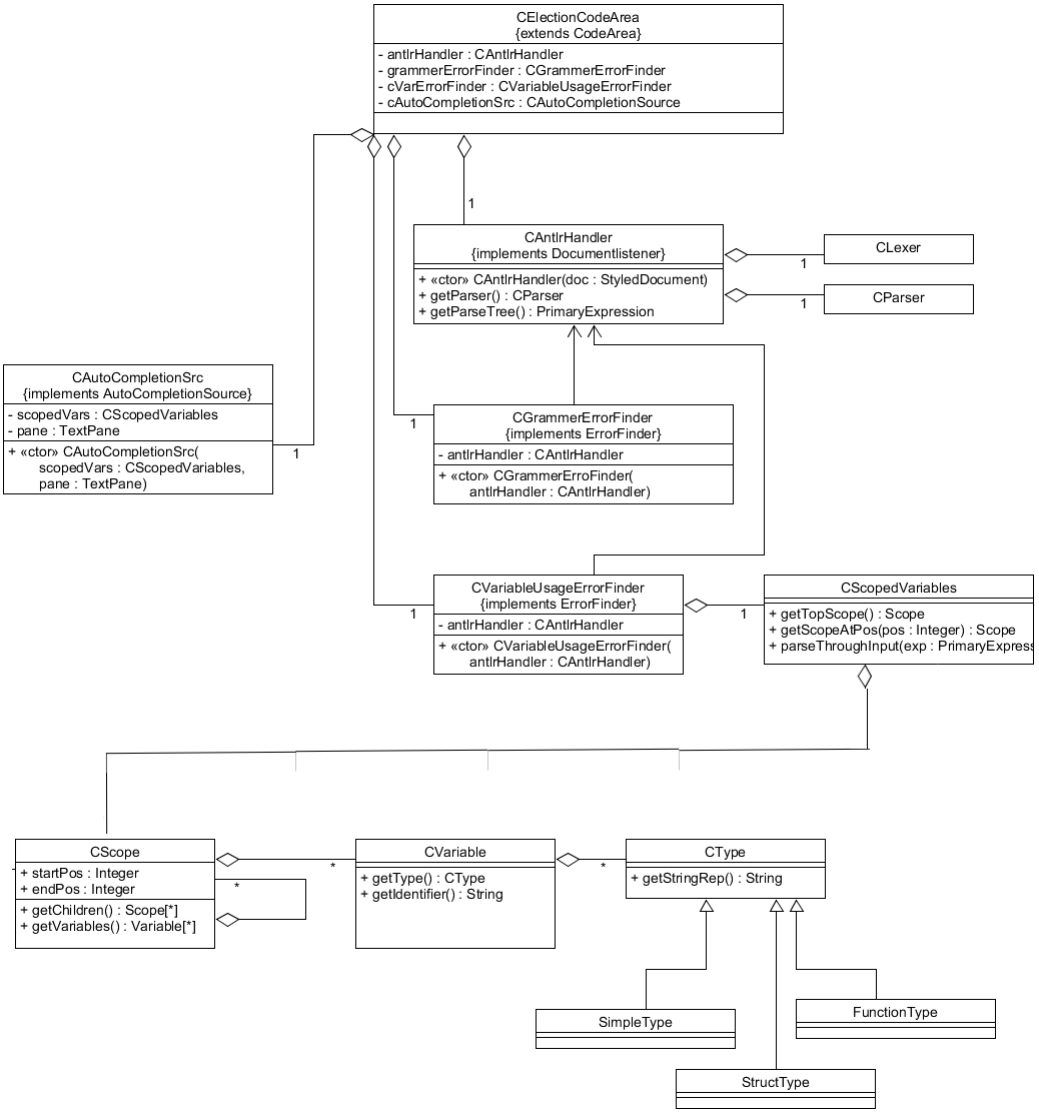
Erfüllte FAs: /FM1050/, /FM1030/, /FM1040/, /FS1070/, /FS1110/, /FK1130/, /FK1150/



Als Interface dieses Packages dient die Klasse CElectionDescriptionEditor.
Methoden:

- `letUserEditElection` Öffnet die gegebene `ElectionDescription` an

4.3.1 Erweitern der CodeArea-Funktionalität



CElectionCodeArea

Diese Klasse erbt von `CodeArea` und erweitert diese durch Zugriff auf deren `protected` Felder um AutoCompletion und Fehlerfindung.

CAntlrHandler

Diese Klasse ist für das generieren des Antlr-Syntaxbaumes zuständig

CGrammerErrorFinder

Diese Klasse findet syntaktische Fehler im C Code

CVariableUSageErrorFinder

Dies Klasse findet Fehler welche durch falsche Nutzung der deklarierten Variablen entstehen. Dazu verwendet es die Klassen `CScopedVariables`, `CScope`, `CVariable` und `CType`.

CAutoCompletionSrc

Diese Klasse findet AutoCompletion Vorschläge welche im Kontext der Sprache C Sinn ergeben

4.4 Eigenschaften-Editor

4.4.1 Übersicht

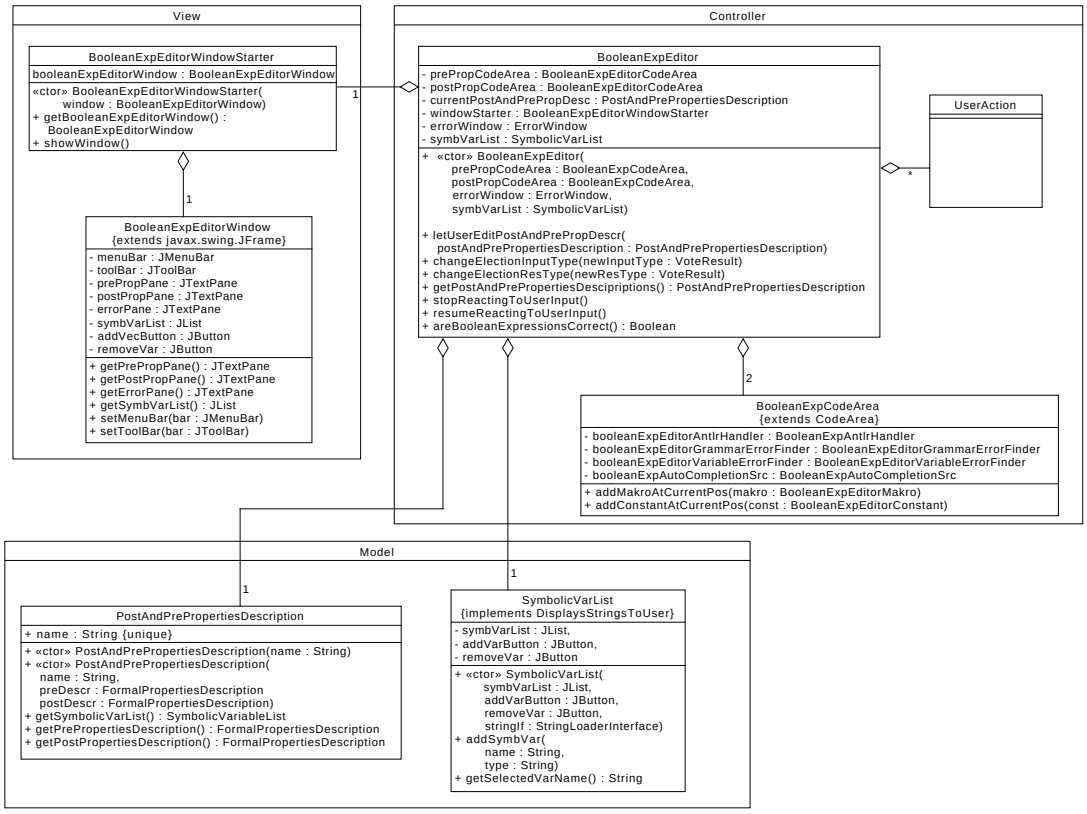
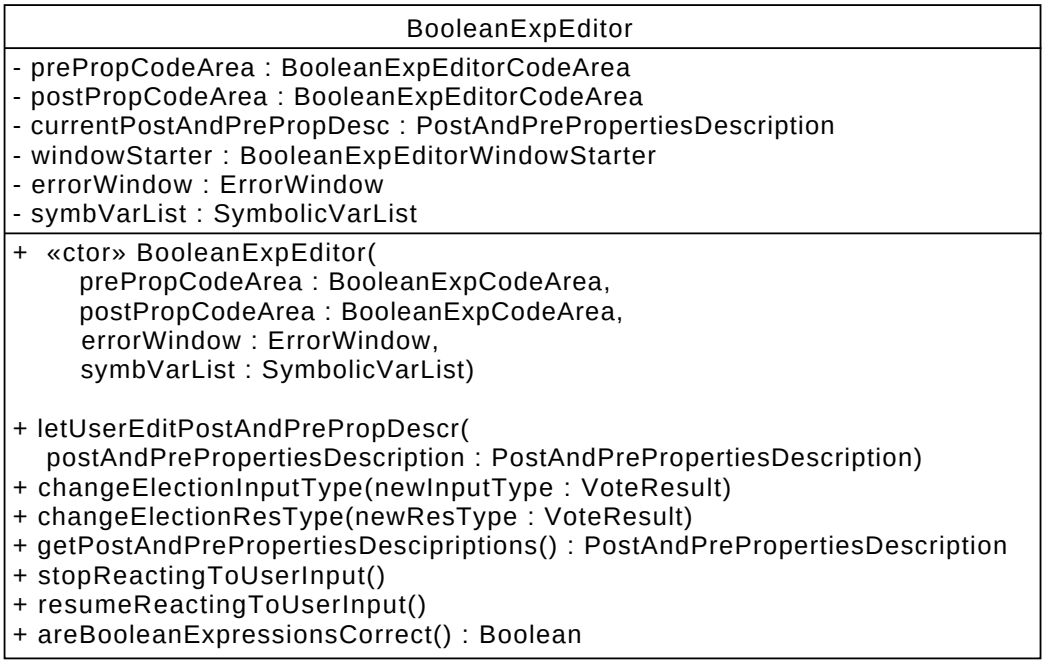


Abbildung 4.8: Übersicht des BooleanExpEditor Packages als MVC.

ErfÜllte FAs: /FM2010/, /FM2020/, /FM2030/, /FM2040/, /FM2050/, /FM2060/, /FM2070/, /FM2071/, /FM2072/, /FM2073/, /FM2080/, /FM2090/, /FM2100/, /FM2110/, /FM2120/, /FS2130/, /FS2140/, /FS2150/, /FK2140/.
Im Package `BooleanExpEditor` wird mithilfe des Model-View-Controller Entwurfsmusters der Eigenschafteneditor modelliert. Die Klasse `BooleanExpEditor` stellt neben der Funktion als Schnittstelle fÜr die Eigenschaftenliste, hier zusammen mit den `UserActions`-Klasssen den Controller da, der vom View `BooleanExpEditorWindow` angesprochen wird sobald der User mit Buttons oder MenÜpunkten interagiert. Die Interaktionsverarbeitung von Usereingaben in die TextPanes für Vor- und Nachbedingungen wird an eine `BooleanExpEditorCodeArea`-Klasse delegiert. Diese stattet die TextPanes außerdem mit Zusatzfunktionen aus, wie z.B. Syntaxhighlighting und Code-Completion. Der Model-Teil des Patterns ist eine `PostAndPrePropertiesDescription`- und eine `SymbolicVarList`- Instanz im `BooleanExpEditor` die die zurzeit geladene formale Eigenschaft enthÄlt.

4.4.2 Klassen

BooleanExpEditor

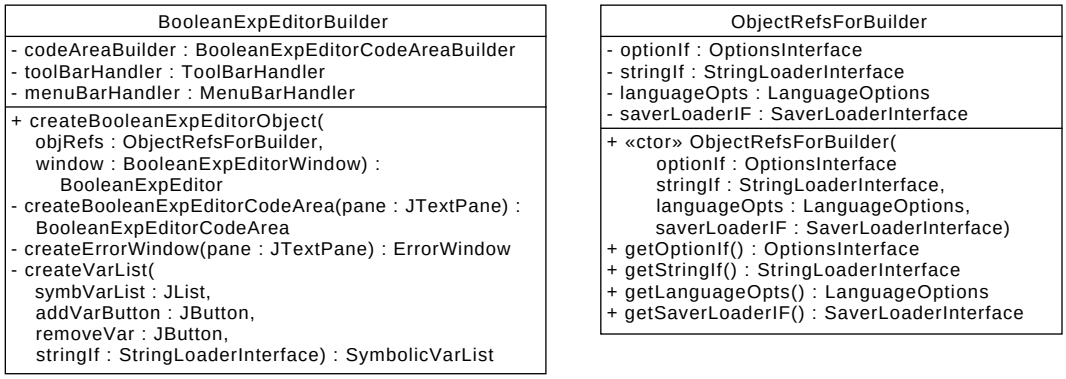


Die BooleanExpEditor Klasse bildet das Interface nach außen und den Controller im MVC-Pattern. Sie verfÜgt Über

- `prePropCodeArea`- und `postProptiesCodeArea`-Klassen fÜr die Vor- und Nachbedingungstextpanes.
- `windowStarter`, eine Instanz der `BooleanExpEditorWindowStarter`-Klasse, wel-

che ein `BooleanExpEditorWindow` erstellt und zugriff auf dieses bietet.
- `errorWindow`, eine `ErrorWindow`-Instanz die dem Anzeigen von Fehlern dient.
- `symbVarList`, die Liste der symbolischen Variablen die in den Eigenschaften verwendet werden können.
- `letUserEditPostAndPrePropertiesDescriptionObject()` das ein `PostAndPrePropertiesDescription` Objekt in den Editor lÄdt.
- `changeElectionInputType()` und `changeElectionResType()` damit der Editor von außerhalb Über eine Änderung dieser Attribute im Wahlverfahren benachrichtigt werden kann.
- `getPostAndPrePropertiesDescription()` damit von außerhalb auf das derzeit geladene PostAndPrePropertiesDescription Objekt zugegriffen werden kann.
- `stopReactingToUserInput()`, die im Falle des Testens von Eigenschaften den Eigenschafteneditor sperrt.
- `resumeReactingToUserInput()`, die den Eigenschafteneditor entsperrt.
- `areBooleanExpressionsCorrect()` damit angefragt werden kann ob die derzeit geladenen Eigenschaften fehlerfrei sind.

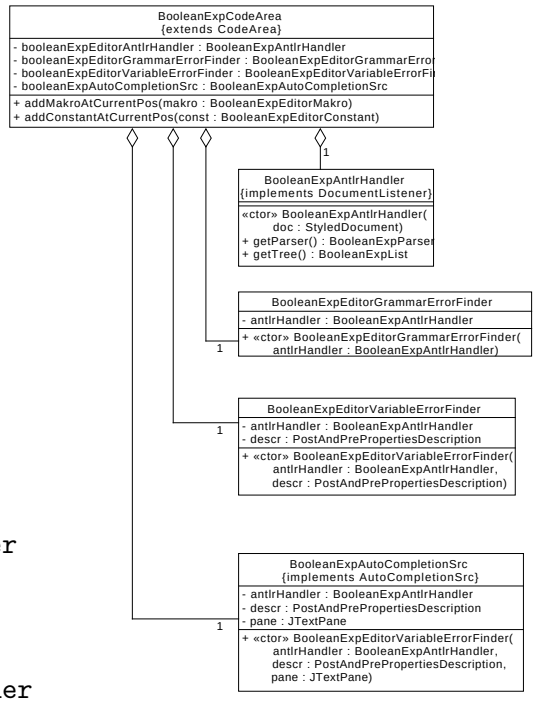
BooleanExpEditorBuilder und ObjectRefsForBuilder



Der `BooleanExpEditorBuilder` baut mit dem Builder Entwurfsmuster eine `BooleanExpEditor`-Instanz auf. Er benutzt dafÜr folgende Attribute und Methoden.
- `createBooleanExpEditorObject()` die aufgerufen werden kann um eine `BooleanExpEditor`-Instanz zu erstellen und zurückzugeben. Sie benötigt dafür als Parameter einen `ObjectRefsBuilder`, der die Optionsinterfaces bereitstellt und ein `BooleanExpEditorWindow`.
- `createBooleanExpEditorCodeArea()` um die `BooleanExpEditorCodeAreas` zu erstellen.
- `createErrorWindow()` um das `ErrorWindow` zu erstellen.
- `createVarList()` die die `SymbolicVarList` erstellt.

BooleanExpCodeArea

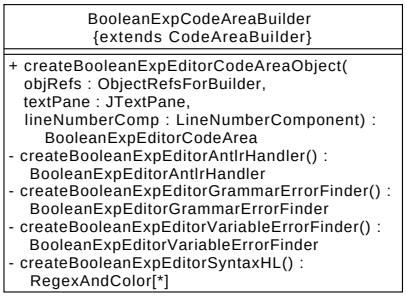
Die `BooleanExpEditor` Klasse hat zwei BooleanExpCodeArea Instanzen fÜr die beiden JTextPanes fÜr Vor- und Nachbedingungen die jeweils von der Klasse CodeArea erben. Diese geben den Vor- und Nachbedingungs-Textpanes zusÄtzliche Funktionen und verarbeiten Usereingaben in diese. Sie besitzen zusÄtzlich zu den von CodeArea geerbten folgende Attribute und Methoden:



- Einen `BooleanExpAntlrHandler` den die beiden folgenden Attribute fÜr die Kommunikation mit ANTLR benutzen.
- Einen `BooleanExpGrammarErrorFinder` fÜr das finden von syntaktischen Fehlern in den angegebenen Eigenschaften.
- Einen `BooleanExpVariableErrorFinder` fÜr das finden von fehlerhaft benutzten Variablen.
- Eine `BooleanExpAutoCompleteSrc` die für die Auto-Completion mithilfe von ANTLR zuständig ist.
- Die Methode `addMakroAtCurrentPosition()`, die ein bestimmtes Makro and der derzeitigen Cursorposition einfÜgt.
- Die Methode `addConstantAtCurrentPos()`, die eine Konstante an der derzeitigen Cursorposition einfÜgt.

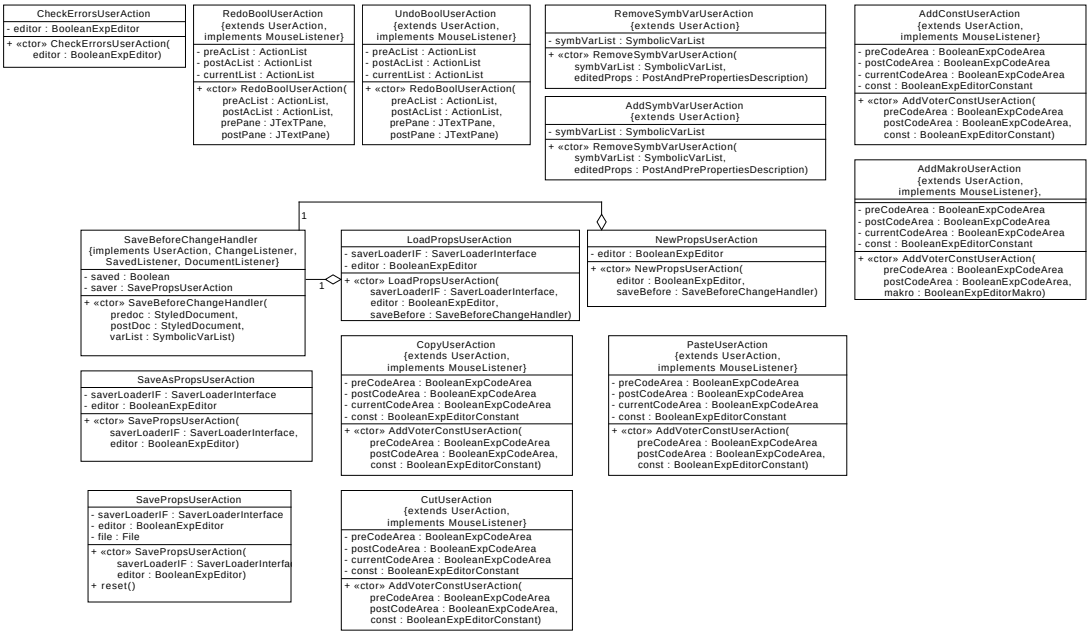
Der `BooleanExpEditorCodeAreaBuilder` baut mit dem Builder Entwurfsmuster eine `BooleanExpEditorCodeArea`-Instanz auf. Er benutzt folgende Funktionen:

- `createBooleanExpEditorCodeAreaObject()`, die eine `BooleanExpCodeArea` Instanz erstellt und zurÜck gibt.
- `createBooleanExpEditorAntlrHandler()`, die eine `BooleanExpEditorAntlrHandler` Instanz erstellt und zurÜck gibt.



- `createBooleanExpEditorGrammarErrorFinder()`, die eine `BooleanExpEditorGrammarErrorFinder`-Instanz erstellt und zurück gibt.
- `createBooleanExpEditorVariableErrorFinder()`, die eine `BooleanExpEditorVariableErrorFinder`-Instanz erstellt und zurück gibt.
- `createBooleanExpEditorSyntaxHL()`, die eine `RegexAndColor`-Liste mit entsprechendem Inhalt für Syntax-Highlighting in der CodeArea bietet.

UserActions



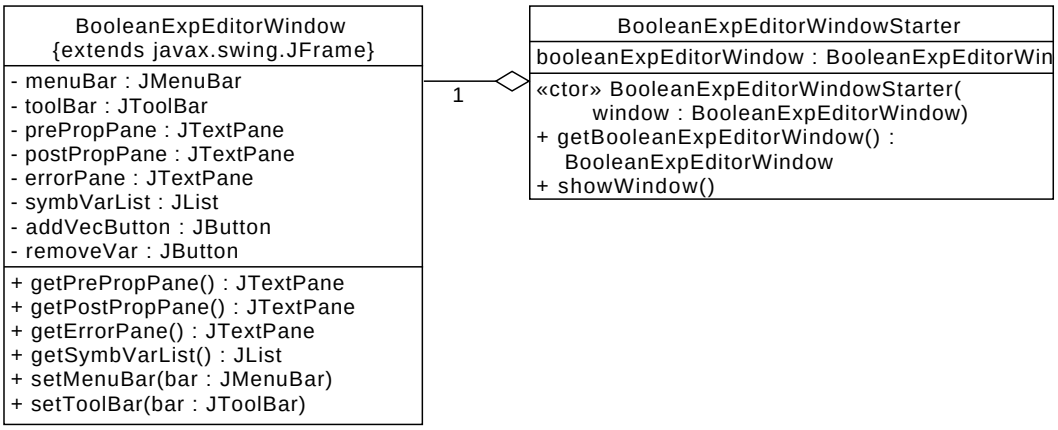
Das Package `BooleanExpEditor` verfügt über folgende `UserAction`-Subklassen:

- `RedoBoolUserAction` und `UndoBoolUserAction` um Aktionen zu rückgÄngig zu machen oder zu wiederholen.
- `RemoveSymbVarUserAction` und `AddSymbVarUserAction` um symbolische Variablen hinzuzufügen oder zu entfernen.
- `AddConstUserAction` und
- `AddMakroUserAction` um Konstanten oder Makros dem derzeit ausgewÄhltem `JTextPane` hinzuzufügen.
- `NewPropsUserAction`, `LoadPropsUserAction`, `SavePropsUserAction` und `SaveAsPropsUserAction` für das Erstellen/Laden/Speichern einer formalen Eigenschaft.
- `CopyUserAction`, `PasteUserAction`, `CutUserAction` um in den `JTextPanes` Text

einzufügen, zu entfernen oder auszuschneiden.

- `CheckErrorsUserAction` für die statische Fehleranalyse.
- `EditorOptionsUserAction`, öffnet einen Optionsdialog einer Option für beide `JTextPanes`.

BooleanExpEditorWindow und BooleanExpEditorWindowStarter

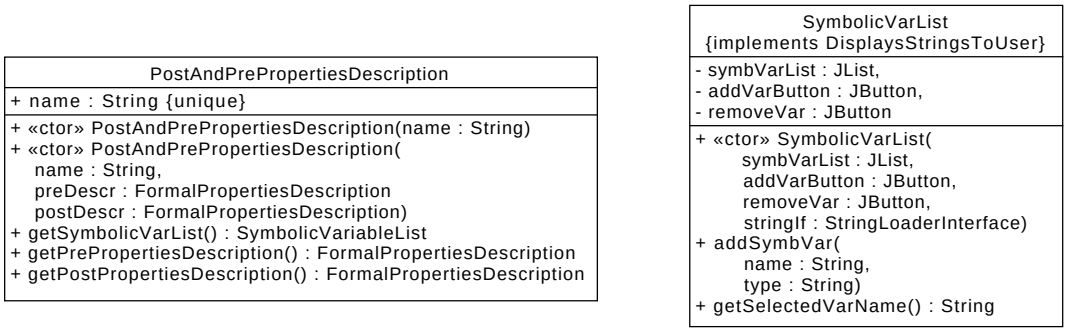


`BooleanExpEditorWindow` und `BooleanExpEditorWindowStarter` bilden den View bzw. die Möglichkeit zur Erstellung und Ansteuerung des Views im `BooleanExpEditor`-Package. Die `BooleanExpEditorWindow` ist ein `JFrame` und enthÄlt folgende `Swing`-Elemente:

- Eine `JMenuBar`.
- Eine `JToolBar`.
- Zwei `JTextPanes` für Vor- und Nachbedingungen.
- Eine `JTextPane` zur Fehlerdarstellung.
- Eine `JList` zum Darstellen der symbolischen Variablen.
- Zwei `JButtons` zum hinzufügen und entfernen von symbolischen Variablen.

Die drei `JTextPanes` und die `JList` haben Getter. Die `JMenuBar` und die `JToolBar` haben Setter damit diese wÄhrend der Laufzeit erstellt werden können.

SymbolicVarList und PostAndPrePropertiesDescription



Die Instanzen dieser beiden Klassen bilden das Modell der vom User eingegebenen formalen Eigenschaft im Eigenschafteneditor. NÄhere Informationen zu diesen Klassen befinden sich im Datentypen Kapitel.

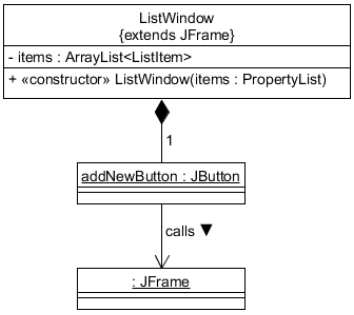
4.5 Eigenschaftenliste

Die Eigenschaftenliste stellt eine veränderbare Liste von formalen Eigenschaften bereit. Der Fokus liegt auf einer übersichtlichen Auflistung und einfacher Bearbeitbarkeit der Liste.
Ein Aufrufer, der ein Analyseergebnis angezeigt haben möchte, muss lediglich ein Objekt übergeben, das das Interface `Result` implementiert. Damit muss der Aufrufer keine Details der Implementierung der Eigenschaftenliste kennen.
Alternativ hätte man auch die Rückgabe des SAT-Solvers direkt in der Eigenschaftenliste verarbeiten können. Dies hätte allerdings eine schlechte Wiederverwendbarkeit des Codes zur Folge. Um es zu ermöglichen, dass Änderungen an der Eigenschaftenliste rückgängig gemacht werden können, wurde das Befehlsmuster verwendet. Der Klient, der Änderungen möchte, wird durch die Listener-Klassen im Controller repräsentiert. Diese Klassen rufen über die abstrakte Klasse ListChangeCommand dann den konkreten Befehl auf.
Durch dieses Muster lassen sich Befehle loggen und damit auch rückgängig machen. Außerdem wird Funktionalität in eigenen Klassen gekapselt, was im Sinne der Objektorientierung geschieht. Die Klasse ListChangeCommand ist kein Interface, da eine Schablonenmethode verwendet wird. Nach der Ausführung eines Befehls wird die GUI aktualisiert. Dazu wird nach jedem Befehl die Methode updateListWindow() aufgerufen. Dadurch wird die Steuerung der Eigenschaftenliste innerhalb des Pakets Controller gehalten. Da es nur Sinn macht, eine einzige Liste vorzuhalten, wird die Klasse PropertyList als Einzelstück implementiert. Die Instanz wird zu Beginn des Programms generiert. Die Beschreibung der formalen Eigenschaften werden zusätzlich zur Klasse PostAndPrePropertiesDescription innerhalb der Eigenschaftenliste gekapselt (in der Klasse PropertyItem).
Diese Datenstruktur fügt der Beschreibung lediglich die Flag hinzu, ob die Eigenschaft auch tatsächlich analysiert werden soll. Durch die Kapselung wird die Datenstruktur

aber innerhalb der Eigenschaftenliste zusätzlich erweiterbar, falls dies in der Zukunft gewünscht wird.

4.5.1 View

ListWindow



Erfüllt: /FM3010/

Die Instanz listWindow bildet das zentrale Fenster in der GUI des Pakets. Sie besteht aus einem Menüstreifen, einer Liste der Klasse `ListItem` und einem JButton zum Hinzufügen neuer Eigenschaften, wie im Pflichtenheft im Kapitel GUI beschrieben.

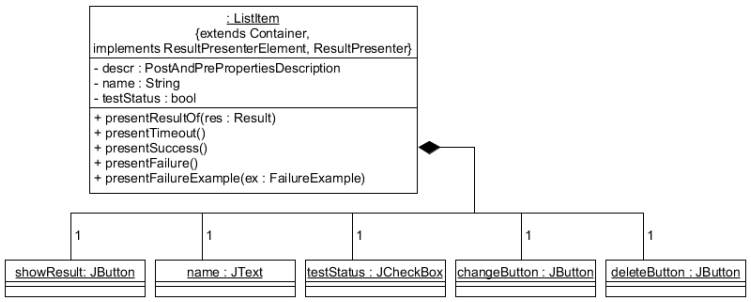
Attribute:

- **- addNewButton : JButton**
  Dieser `JButton` ruft einen Dialog auf, in dem der addNewDescription-Button die Bearbeitung einer neuen Eigenschaft öffnet. Im Dialog kann auch in der standardDescriptions-Liste aus, von den Entwicklern vorgesehenen, formalen Eigenschaften gewählt werden und per addStandardDescription-Button der Liste hinzugefügt werden.

Methoden:

- **+ «constructor» ListWindow(items : PropertyList)**
  Der Konstruktor erwartet ein Argument des Typs `PropertyList` (siehe Model), oder null, falls eine leere Liste angelegt werden soll.

ListItem



Die Klasse `ListItem` ist vom Typ Container und implementiert die Interfaces ResultPresenterElement und ResultPresenter, damit in ihr die Ergebnisse der Analyse von CBMC für jeweils eine formale Eigenschaft dargestellt werden können.
Jedes `ListItem` beinhaltet die für die Interaktion mit der formalen Eigenschaft nötigen Elemente.

Attribute:

- **- showResult : JButton**
  Dieser `JButton` schaltet das Ergebnis der Analyse (die `JTextPane` result) auf sichtbar oder unsichtbar.
- **- name : JText**
  Dieser `JText` zeigt den Name der `PostAndPrePropertiesDescription` an. Eine

Veränderung des Namens, die mit Enter bestätigt wird und nicht mit einem anderen Eigenschaftennamen in Konflikt gerät, wird in der PostAndPrePropertiesDescription gespeichert.

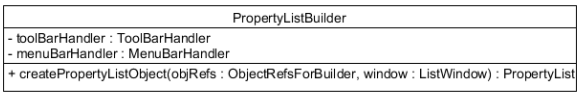
- **- testStatus : JCheckBox**
  Diese `JCheckBox` zeigt an, ob die formale Eigenschaft überprüft werden soll oder nicht.
- **- changeButton : JButton**
  Mit diesem `JButton` wird die Bearbeitung der formalen Eigenschaft gestartet.
- **- deleteButton : JButton**
  Mit diesem `JButton` wird die Eigenschaft von der Liste entfernt.

Methoden:

- **+ presentResultOf(res : Result)**
  Die Implementierung des Interfaces `ResultPresenter` ermöglicht das Anzeigen des Ergebnisses des SAT-Solvers. Das Argument muss das Interface Result implementieren.
- **+ presentTimeout()**
  Das `ListItem` zeigt graphisch an, dass die Analyse der formalen Eigenschaft durch einen Timeout oder durch den Benutzer abgebrochen wurde.
- **+ presentSuccess()**
  Das `ListItem` zeigt graphisch an, dass die formale Eigenschaft erfüllt ist.
- **+ presentFailure()**
  Das `ListItem` zeigt graphisch an, dass die Analyse der formalen Eigenschaft ergeben hat, dass die Eigenschaft nicht erfüllt ist.
- **+ presentFailureExample(ex : FailureExample)**
  Das `ListItem` lädt das Gegenbeispiel des SAT-Solvers in das dafür vorgesehene Element.

4.5.2 Controller

PropertyListBuilder



Diese Klasse sorgt dafür, dass sich die GUI aufbaut und das Einzelstück `PropertyList` erzeugt wird.

Methoden:

- **+ createPropertyListObject(objRefs : ObjectRefsForBuilder, window : ListWindow) : PropertyList**
  Die Referenzen auf Objekte für Optionen usw. und ein ListWindow wird dem Builder mitgegeben. Das Einzelstück wird zurückgegeben.

PropertyListSaverLoader

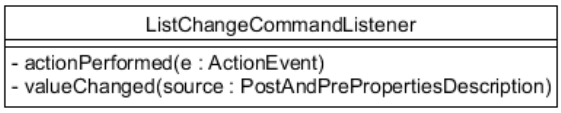
Erfüllt: /FM3060/, /FM3070/

Die Klasse `PropertyListSaverLoader` ist für das Laden und Speichern von Listen von Eigenschaften zuständig.

Methoden:

- **+ createFromSaveFormat(format : String) : ArrayList<PropertyItem>**
  Diese Methode liefert eine Liste von `PropertyItem` zurück und lädt sie in das in der GUI vorgesehene Fenster.
- **+ createSaveFormat(list : PropertyList) : String**
  Diese Methode speichert die gegenwärtige Liste von Eigenschaften ab.

ListChangeCommandListener

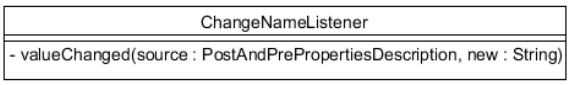


Dieser Listener hört auf alle Ereignisse in der GUI, die keiner gesonderten Überprüfung bedürfen.

Methoden:

- **- actionPerformed(e : ActionEvent)**
  Alle Buttons der GUI werden abgehört.
- **- valueChanged(source : PostAndPrePropertiesDescription)**
  Diese Methode soll aufgerufen werden, wenn der `testStatus` einer formalen Eigenschaft sich geändert hat.

ChangeNameListener

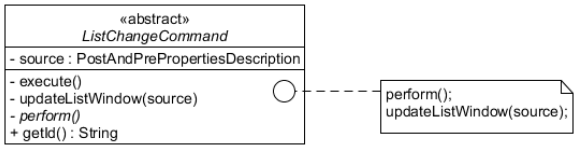


Dieser Listener hört die Änderung in der Eigenschaftenliste ab, die einer Überprüfung bedarf.

Methoden:

- **- valueChanged(source : PostAndPrePropertiesDescription, new : String)**
  Diese Methode soll aufgerufen werden, wenn der `name` einer PostAndPrePropertiesDescription sich geändert hat. Falls die Änderung zulässig war, wird sie durchgeführt.

«abstract» ListChangeCommand



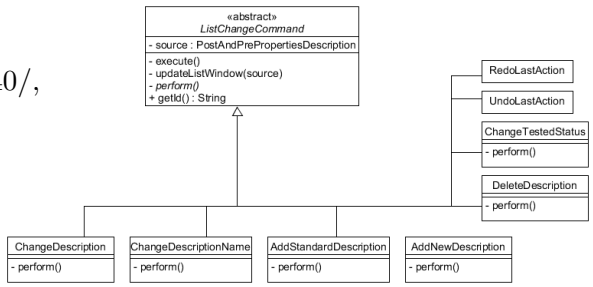
Für die Änderungen an der Eigenschaftenliste wird die abstrakte Klasse vom konkreten Befehl jeweils überschrieben. Da eine Schablonenmethode verwendet wird, ist diese Klasse kein Interface, wie normalerweise im Entwurfsmuster Befehl.
Die Klasse wird in der Klasse `alterList` aggregiert, damit Aktionen rückgängig und wiederhergestellt werden können.

Methoden:

- **- execute()**
  Diese Methode wird von den Listener-Attributen aufgerufen. Sie ist eine Schablonenmethode. Zuerst wird der konkrete Befehl mit perform() ausgeführt. Danach werden vorgenommene Änderungen an der Eigenschaftenliste durch updateListWindow() in der GUI angezeigt.
- **- «abstract» perform()**
  Diese Methode wird vom konkreten Befehl jeweils überschrieben.
- **- updateListWindow(source : PostAndPrePropertiesDescription)**
  Aktualisiert die Anzeige der Eigenschaftenliste, indem die entsprechende Eigenschaft aktualisiert wird.
- **+ getId() : String**
  Um Aktionen wieder rückgängig zu machen oder zu wiederholen, wird die Befehls-ID zurückgegeben.

extends ListChangeCommand

Erfüllt: /FM3020/, /FM3030/, /FM3040/, /FM3050/



Implementiert jeweils die Methode perform() der abstrakten Klasse `ListChangeCommand`.

Unterklassen:

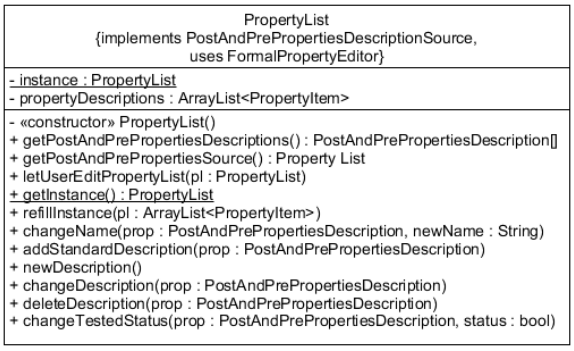
- **ChangeDescription**
  Ruft in der `PropertyList` die Methode changeDescription(PostAndPrePropertiesDescription prop) auf.
- **ChangeDescriptionName**
  Ruft in der `PropertyList` die Methode changeName(PostAndPrePropertiesDescription prop, String newName) auf. Die Namensänderung muss gültig sein.
- **AddStandardDescription**
  Ruft in der `PropertyList` die Methode addStandardDescription(PostAndPrePropertiesDescription prop) auf. Die ausgewählte Standardeigenschaft wird hinzugefügt.
- **AddNewDescription**
  Ruft in der `PropertyList` die Methode newDescription() auf. Damit wird eine komplett neue Eigenschaft bearbeitet.
- **DeleteDescription**
  Ruft in der `PropertyList` die

Methode deleteDescription(PostAndPrePropertiesDescription prop) auf.

- **ChangeTestedStatus**
  Ruft in der `PropertyList` die Methode changeTestedStatus(PostAndPrePropertiesDescription prop, Boolean status) auf.

4.5.3 Model

PropertyList



Die Klasse `PropertyList` ist ein Einzelstück und speichert eine Liste formaler Eigenschaften des Typs `PropertyItem` ab. Sie bietet für andere Pakete über das Interface PostAndPrePropertiesDescriptionSource eine Liste aller zu analysierenden Eigenschaften und bietet einen Getter für die eigene Datenstruktur.

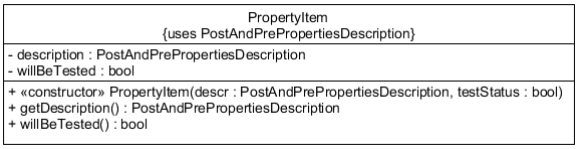
Methoden:

- **+ getPostAndPrePropertiesDescriptions() : PostAndPrePropertiesDescription[]**
  Gibt ein Array aller zu analysierenden Eigenschaften zurück.
- **+ getPostAndPrePropertiesSource() : Property List**
  Gibt die Instanz dieser Klasse zurück.
- **+ «static» getInstance() : PropertyList**
  Gibt ebenfalls die Instanz dieser Klasse zurück.
- **+ refillInstance(pl : ArrayList<PropertyItem>)**
  Befüllt die Instanz der Klasse neu mit der gegebenen Liste von `PropertyItem`. Die alte Liste wird überschrieben.
- **+ changeName(prop : PostAndPrePropertiesDescription, newName : String)**
  Verändert den Namen der übergebenen PostAndPrePropertiesDescription prop zu newName.
- **+ addStandardDescription(prop : PostAndPrePropertiesDescription)**
  Fügt die übergebene PostAndPrePropertiesDescription prop der `PropertyItem` Liste hinzu.
- **+ newDescription()**
  Fügt der `PropertyItem` Liste eine neue Beschreibung hinzu.
- **+ changeDescription(prop : PostAndPrePropertiesDescription)**
  Ruft den Eigenschafteneditor auf, um die Beschreibung der formalen Eigenschaft abzuändern.
- **+ deleteDescription(prop : PostAndPrePropertiesDescription)**

Löscht das entsprechende `PropertyItem`.

- **+ changeTestedStatus(prop : PostAndPrePropertiesDescription, status : bool)**
  Ändert, ob die formale Eigenschaft vom SAT-Solver überprüft werden soll.

PropertyItem



Die Klasse `PropertyItem` erweitert die Beschreibung der formalen Eigenschaft in der Klasse PostAndPrePropertiesDescription um die Angabe, ob die Eigenschaft analysiert werden soll.

Methoden:

- **+ «constructor» PropertyItem(descr : PostAndPrePropertiesDescription, testStatus : bool)**
  Der Konstruktor erwartet ein Objekt der Klasse PostAndPrePropertiesDescription und die Angabe, ob die formale Eigenschaft analysiert werden soll.
- **+ getDescription() : PostAndPrePropertiesDescription**
  Gibt die formale Eigenschaft im Typ PostAndPrePropertiesDescription zurück.
- **+ willBeTested() : bool**
  Gibt zurück, ob die Eigenschaft analysiert werden soll.

4.5.4 Entwurfsentscheidungen

- **Aufrufen des Analyseergebnisses:**
  Ein Aufrufer, der ein Analyseergebnis angezeigt haben möchte, muss lediglich ein Objekt übergeben, das das Interface `Result` implementiert. Damit muss der Aufrufer keine Details der Implementierung der Eigenschaftenliste kennen.
  Alternativ hätte man auch die Rückgabe des SAT-Solvers direkt in der Eigenschaftenliste verarbeiten können. Dies hätte allerdings eine schlechte Wiederverwendbarkeit des Codes zur Folge.
- **Befehlsmuster bei Änderungen der Liste:**
  Um es zu ermöglichen, dass Änderungen an der Eigenschaftenliste rückgängig gemacht werden können, wurde das Befehlsmuster verwendet. Der Klient, der Änderungen möchte, wird durch die Listener-Klassen im Controller repräsentiert. Diese Klassen rufen über die abstrakte Klasse ListChangeCommand dann den konkreten Befehl auf.
  Durch dieses Muster lassen sich Befehle loggen und damit auch rückgängig machen. Außerdem wird Funktionalität in eigenen Klassen gekapselt, was im Sinne der Objektorientierung geschieht.
- **Schablonenmethode in ListChangeCommand:**
  Die Klasse ListChangeCommand ist kein Interface, da eine Schablonenmethode verwendet wird. Nach der Ausführung eines Befehls wird die GUI aktualisiert. Dazu wird nach jedem Befehl die Methode updateListWindow() aufgerufen. Dadurch

wird die Steuerung der Eigenschaftenliste innerhalb des Pakets Controller gehalten.

- **PropertyList als Einzelstück:**
  Da es nur Sinn macht, eine einzige Liste vorzuhalten, wird die Klasse PropertyList als Einzelstück implementiert. Die Instanz wird zu Beginn des Programms generiert.
- **Kapselung der Eigenschaften in Klasse PropertyItem**
  Die Beschreibung der formalen Eigenschaften werden zusätzlich zur Klasse PostAndPrePropertiesDescription innerhalb der Eigenschaftenliste gekapselt (in der Klasse PropertyItem).
  Diese Datenstruktur fügt der Beschreibung lediglich die Flag hinzu, ob die Eigenschaft auch tatsächlich analysiert werden soll. Durch die Kapselung wird die Datenstruktur aber innerhalb der Eigenschaftenliste zusätzlich erweiterbar, falls dies in der Zukunft gewünscht wird.

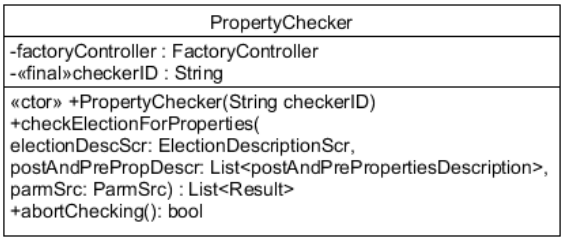
4.6 CBMC

Die Schnittstelle zu CBMC ist dahingend designed, dass sich leicht andere Programme zur Überprüfung einfügen lassen. Der Fokus dieses Pakets liegt darauf, relativ einfach neue Programme, die Wahlen überprüfen können hinzufügen zu können, und dabei so gut wie gar keine Änderungen an bestehenden Klassen machen zu müssen.
Als Entwurfsmuster wird für die Checker-Objekte das Fabrikmodell verwendet. Dies ist daher gut, da sich so leicht neue Checker hinzufügen lassen, und eine kleine Änderung an einem Checker keine großen Änderungen nach sich zieht, was bei guter Objektorientierung der Fall sein sollte.
Attribute werden nur beschrieben, falls sie mit Interaktionen mit anderen Klassen verbunden sind.

4.6.1 Klassen

PropertyChecker

Diese Klasse bildet die Schnittstelle von der Nutzereingabeverarbeitung zum ausgewählten Programm, welches die Wahl überprüfen soll.



Attribute:

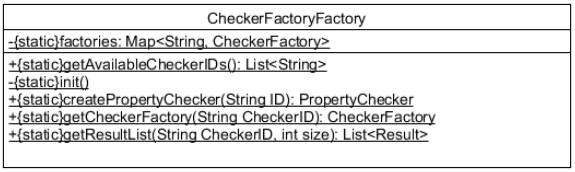
- **FactoryController factoryController**
  Objekt Referenz auf einen `FactoryController`
  um ihm beispielsweise später zum Stoppen aufzufordern
- **final String checkerID**
  Speichert die ID des Checkers, der verwendet werden soll.

Methoden:

- **checkElectionForProperties**
  Startet die Überprüfung der mitgegebenen Parameter, indem es mit diesen Parametern einen neuen `FactoryController` startet, und die von diesem zurückgegebene Liste von Result Objekten dem Aufrufer zurück gibt.
- **bool abortChecking**
  Stoppt die Überprüfung der Eigenschaften, indem es den Befehl an seinen FactoryController weiterleitet.

CheckerFactoryFactory

Diese Klasse beinhaltet nur statische Klassen, die für die Erstellung von CheckerFactories bestimmt sind. Für eine bessere Kapselung in ein anderes Paket gibt es noch die Klasse `CheckerList`, welche jedoch nur ein wrapper für 2 Methoden von `CheckerFactoryFactory`



Attribute:

- **Map<String, CheckerFactory> factories**
  Diese List beinhaltet eine Map, in der alle verfügbaren `CheckerFactories` auf ihre IDs abgebildet und gespeichert sind. Dies wird für die Optionen benötigt, sodass der Nutzer zwischen `Checkern` wählen kann.

Methoden:

- **static List<String> getAvailableCheckerIDs**
  Dies liefert alle verfügbaren IDs, die auf Checkerfactoriess verweisen. Die IDs sind die Namen die die CheckerFactories besitzen.

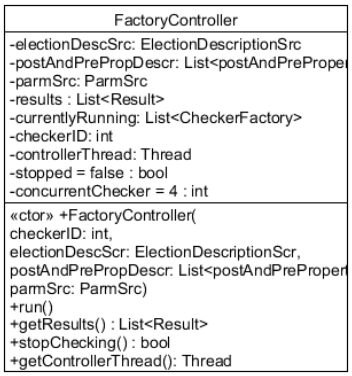
- **static PropertyChecker createPropertyChecker**
  Diese Methode wird vom GUI Paket aus aufgerufen, und liefert einen neuen `PropertyChecker` der dann später die Überprüfung durchführt.

- **static CheckerFactory getCheckerFactory**
  Liefert eine `CheckerFactory` zurück. Wird vom Factory Controller verwendet um die passenden `CheckerFactories` zu erhalten.

- **static List<Result> getResultList**
  Diese Methode liefert eine Liste von bestimmter Länge von `Result` Objekten die zum gewählten `Checker` passen.

FactoryController

Diese Klasse ist für das erstellen der Überprüfungen verantwortlich.



Attribute:

- **ElectionDescriptionSrc electionDescSrc**
  Beschreibt das allgemeine Wahlverfahren dieser Überprüfung.
- **ElectionDescriptionSrc electionDescSrc**
  Beschreibt die eventuell mehreren Eigenschaften, auf welche die Wahl untersucht werden soll.
- **ElectionDescriptionSrc electionDescSrc**
  Beschreibt die Parameter unter denen diese Wahl ablaufen soll.
- **Result results**
  Referenz auf die Liste der `Result` Objekte, in welche die Ergebnisse herein geschrieben werden.
- **List<ChckerFactory> currentlyRunning** Zeigt auf alle `CheckerFactories` die im Moment am arbeiten sind. Diese Information wird Beispielsweise dafür gebraucht, falls man die Überprüfung vorzeitig abbrechen will.
- **String checkerID** Spezifiziert die ID über die bestimmt wird welche Implementation der `CheckerFactory` verwendet wird.
- **Thread controllerThread** Der Thread der in dieser `CheckerFactory` läuft. Wird zum vorzeitigen abbrechen der Überprüfung gebraucht.
- **bool stopped** Flag die anzeigt, dass die Überprüfung vorzeitig abgebrochen werden soll.
- **int concurrentChecker** Gibt an, wie viele `CheckerFactories` gleichzeitig parallel laufen dürfen.

Methoden:

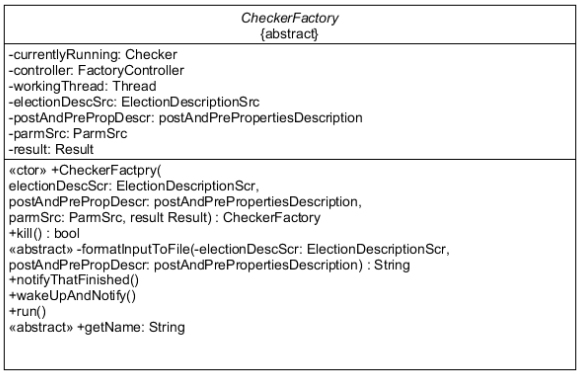
- **«constructor» FactoryController**
  Erstellt den `FactoryController` mit den angegebenen Parametern, holt sich die zum spezifizierzen Checker passenden Result Objekte von der `CheckerFactoryFactory`, in denen später die Ergebnisse stehen werden und startet den `FactoryController` dann in einem neuen Thread.
- **run**
  Der Threaded Teil des `Controllers`. Hier werden so viele `CheckerFactories` vom gewünschten Typ von der `CheckerFactoryFactory` erstellt und immer so viele gleichzeitig laufen gelassen, wie es in dem Attribut “concurrentChecker” steht. Falls eine fertig ist wird sofort die nächste gestartet.
- **getResults**

Getter-Methode, mit der verschiedene Klassen die Referenz auf die Liste der `Result` Objekte erhalten kann.

- **stopChecking**
  Stoppt die Überprüfung der Eigenschaften, indem es aufhört `CheckerFactories` zu erstellen und den momentan Laufenden mitteilt mit der Überprüfung aufzuhören.
- **getControllerThread**
  Getter-Methode welche den Thread des `FactoryControllers` liefert, sodass man ihn beispielsweise interrupten kann.

CheckerFactory

Diese abstrakte Klasse ist dazu da die gewünschten `Checker` für eine spezielle Eigenschaft und alle möglichen Parameter zu starten. Hierbei werden die Variablen, die die Wahl und die Eigenschaft beschreiben in für den Checker verwendbaren Code verwandelt, welcher dann in einer Datei abgespeichert wird. Diese Datei mit dem Code wird nun allen Aufrufen mit den unterschiedlichen Parametern mitgegeben.



Attribute:

- **currentlyRunning**
  Eine Referenz auf den `Checker`, welcher momentan läuft. Sie wird zum Beispiel zum beendigen einer noch laufenden Überprüfung gebraucht.
- **controller**
  Eine Referenz auf den `FactoryController`.
  Diese wird gebraucht um dem Controller nach Abschluss aller Überprüfungen mitzuteilen, dass er eine neue CheckerFactory starten kann.
- **Thread workingThread**
  Eine Referenz auf den Thread der in der `Checkerfactory` arbeitet. Diese wird dafür gebraucht, um der `CheckerFactory` mitzuteilen dass der Prozess fertig ist und sie den nächsten starten kann.

Methoden:

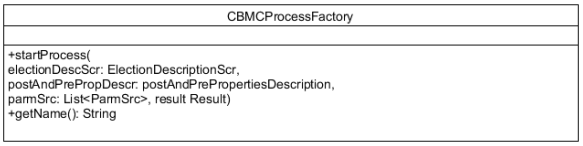
- **void run**
  Startet die Überprüfung der Parameter die vom Konstruktor gespeichert wurden,

indem es diese zu einer Datei die vom Checker akzeptiert wird transformiert und damit dann nach einander `Checker` startet. Sollte eine Überprüfung fehlschlagen werden keine weiteren mehr ausgeführt und in das Result Objekt wird das Ergebnis geschrieben und es auf "valid" und "finished" gesetzt. Dieses Result Objekt kann nun zum anzeigen des Gegenbeispiels genutzt werden.

- **abstract formatInputToFile**
  Erstellt aus den eingegebenen Daten eine Datei, mit welcher der `Checker` dann arbeiten kann.
- **Thread getWorkingThread**
  Liefert den Thread zurück, der in dieser Klasse weitere Threads startet. Dies ist wichtig, da man ihn so aufwecken kann um ihm beispielsweise mitzuteilen, dass die Überprüfung durch einen `Checker` abgeschlossen ist.
- **abstract String getName**
  Diese abstrakte Methode sorgt dafür dass alle Implementationen der `CheckerFactory` einen Namen besitzen. Aus diesem Namen wird dann von der `CheckerFactoryFactory` eine indeutige ID für jeden `Checker` erstellt, worüber die Checker dann später aufgerufen werden.
-

CBMCProcessFactory

Diese Klasse ist eine Implementierung der `CheckerFactory`, welche CBMC zum überprüfen von Wahlverfahren nutzt. Beim starten des Prozesses wird darauf geachtet, dass CBMC für das richtige Betriebssystem gestartet wird.



Attribute:

Methoden:

- **String formatInputToFile** Diese Methode erstellt aus den gegebenen Parametern eine Datei die von CBMC überprüft werden kann. Zurückgegeben wird der Pfad zur Datei.

Checker

Diese Klasse ist die Oberklasse für alle Programme, die in unserem Sinne fähig sind Wahlverfahren zu überprüfen, wie in unserem Falle CBMC.

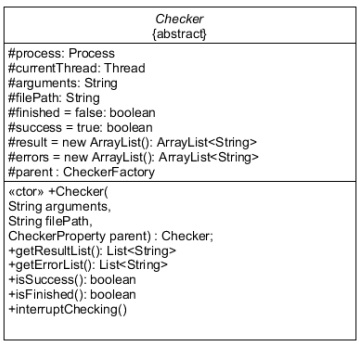
Attribute:

- **Process process**
  Der Prozess in welchem der `Checker` vom Betriebssystem ausgeführt wird.
- **Thread currentThread**
  Zeigt auf den Thread der in diesem `Checker` erstellt wurde, um ihn aufwecken zu können.

- **String arguments**
  Die Argumente die bei der Überprüfung mit an das überprüfende Programm gegeben werden sollen.
- **String filePath**
  Ein Zeiger auf die Datei, welche den Code beinhaltet der überprüft werden soll.
- **bool finished**
  Zeigt an, ob die Überprüfung aufgehört hat (entweder durch vollständige Berechnung oder aber durch einen Abbruch)
- **bool success**
  Zeigt an, ob die Überprüfung richtig ausgeführt wurde und es zu keinem Interrupt von Außerhalb kam.
- **List<String> result**
  All die normale Ausgabe die der Prozess zurück liefert, in der Reihenfolge wie er kommt gespeichert.
- **List<String> errors**
  Wie oben, jedoch für den ErrorStream des Prozesses.
- **CheckerFactory parent**
  Eine Referenz auf die `CheckerFactory` die diesen `Checker` gestartet hat, damit dieses Objekt Bescheid geben kann, dass es seine Überprüfung abgeschlossen hat.
-

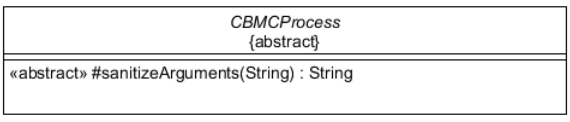
Methoden:

- **Checker()**
  Setzt die Daten die zum späteren startet des Prozesses gebraucht werden.
- **getter/setter**
  Verschiedene getter und setter zum setzen und Abfragen des Status und der Abfrage der Ergebnisse.
- **interruptChecking**
  Stopp beim Aufruf die Analys des momentanen Wahlverfahrens.
-



CBMCProcess

Diese Klasse ist eine abstrakte Implementierung der `Checker` Klasse für CBMC. Sie bildet das Grundgerüst für die betriebssystemabhängigen Aufrufe von CBMC.



Attribute:

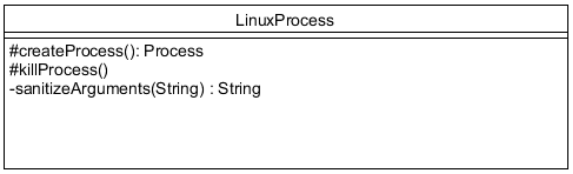
Methoden:

- **abstract void sanitizeArguments** Da die Argumente die man an CBMC übergeben kann Unterschiede aufweisen werden die Argumente in dieser Methode rich-

tig angepasst.

Linux/Mac-Process

Diese beiden Klassen sind Implementierungen von CBMCPRocess um CBMC auf Mac oder Linux zu starten.



WindowsProcess

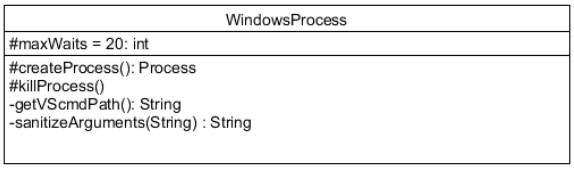
Diese Klasse implementiert `CBMCProcess` für Windows.

Attribute:

- **int maxWaits**
  Da sich der Prozess auf Windows nicht so schnell ausstellen lässt warten wir maximal 20 Sekunden darauf, dass sich der Prozess beendet.

Methoden:

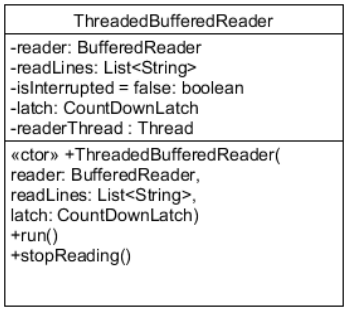
- **String getVScmdPath** Unter Windows benötigt CBMC die VisualStudioCMD, nach welcher mit dieser Methode gesucht und auch gefragt wird, falls der Computer sie nicht findet.



ThreadedBufferedReader

Attribute:

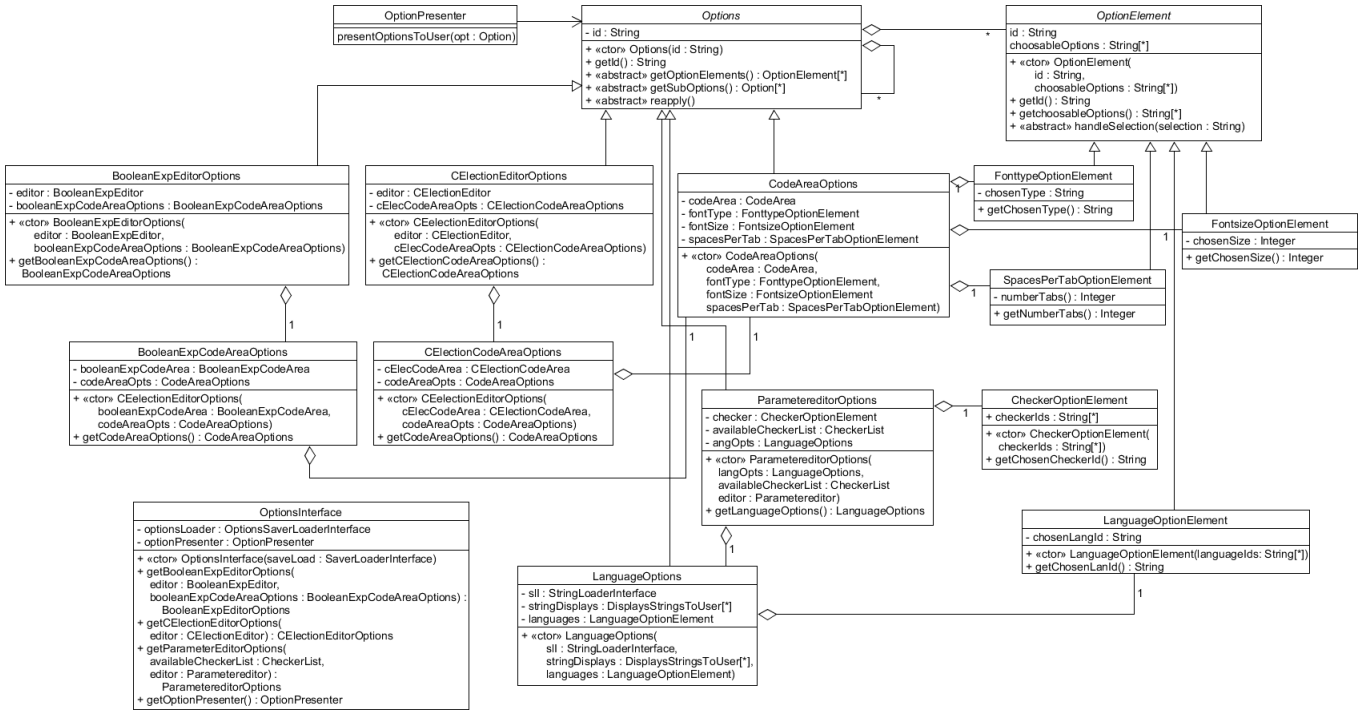
- **BufferedReader reader**
  Der Reader der die ganze Zeit laufen soll.
- **List<String> readLines**
  Alle Strings die der reader gelesen hat werden hier gespeichert.
- **boolean isInterrupted**
  Falls das Programm abgebrochen wird sollte man auch die `Reader` interrupted. Diese Flag zeigt den Status davon an.
- **Latch CountDownLatch**
  Wenn die Überprüfung abgeschlossen ist müssen wir warten bis alle `Reader` auch fertig s1ind. Hierfür verwenden wir ein `CountDownLatch`, welches im `Checker` erstellt wird.
- **Thread readerThread**
  Der Thread der im `Reader\verb` läuft. Die Referenz wird zum vorzeitigen beenden gebraucht



Methoden:

- **ThreadedBufferedReader** Nimmt die Referenzen zu dem `BufferedReader` von dem es lesen soll, eine Referenz auf eine Liste in der er die Ergebnisse speichern soll und ein Latch, auf das er sich am Ende synchronisieren soll entgegen und sppeichert sie.
- **void run** Startet den Thread, der dann den `Reader` so lange ausführt bis all der Output gelesen wurde, oder aber der Reader unterbrochen wird.
- **void stopReading** Bringt den Thread dazu aufzuhören zu lesen.

4.7 Vom Nutzer konfigurierbares Verhalten



Erfüllte FAs: /FK1140/
Jede Klasse welche Eigenschaften aufweist die vom Benutzer verändert werden können hat ein dazugehörige `Options`-Klasse. Die Funktionalität, eine Klasse zu konfigurieren wird von diesen `Options`-Klassen implementiert. Jede `Options`-Klasse kann dem Benutzer von der Klasse `OptionsPresenter` präsentiert werden. Dabei wird ein Fenster erstellt welches für jede enthaltene `subOptions` einen Reiter enthält. Diese Reiter enthalten pro `OptionItem` ein label welches die Bezeichnung enthält und eine JComboList welche die wählbaren Einstellungen enthält.

Options

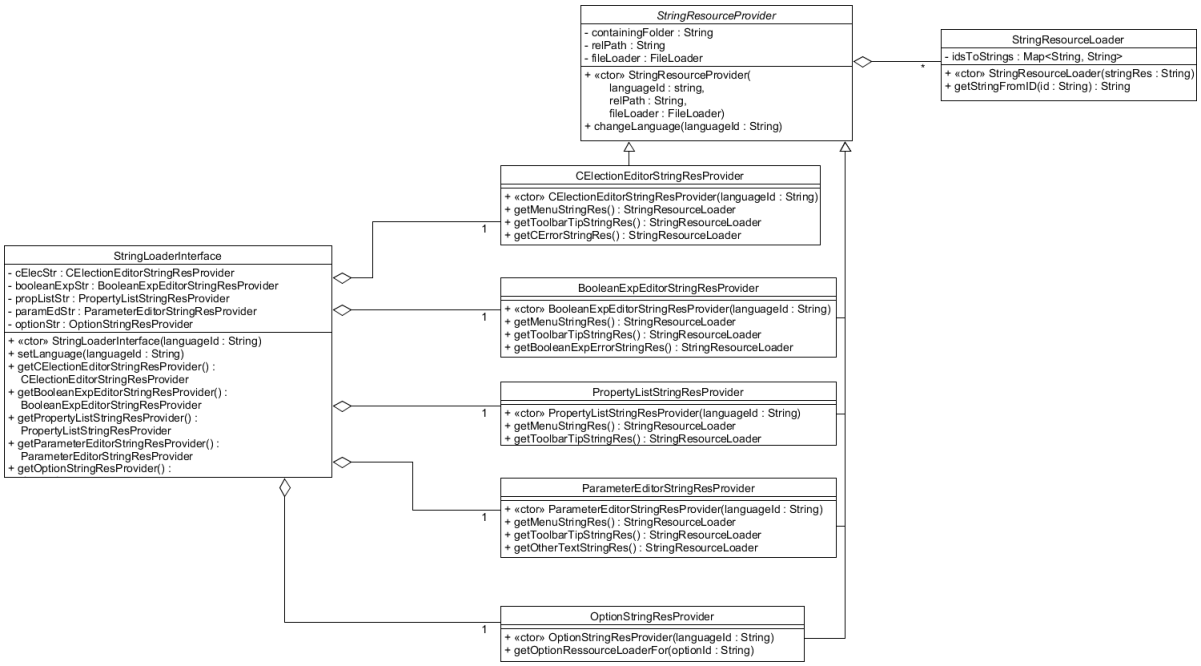
Methoden

- `reapply` Wird nach setzen neuer Einstellungen vom `Optionpresenter` aufgerufen und von den erbenden Klassen implementiert. Sorgt dafür dass die gewählten Einstellungen übernommen werden

OptionElement

Diese Klasse repräsentiert ein wählbares Element und dessen Optionen.

4.8 Umgang mit String-Ressourcen welche von der Sprache abhängen

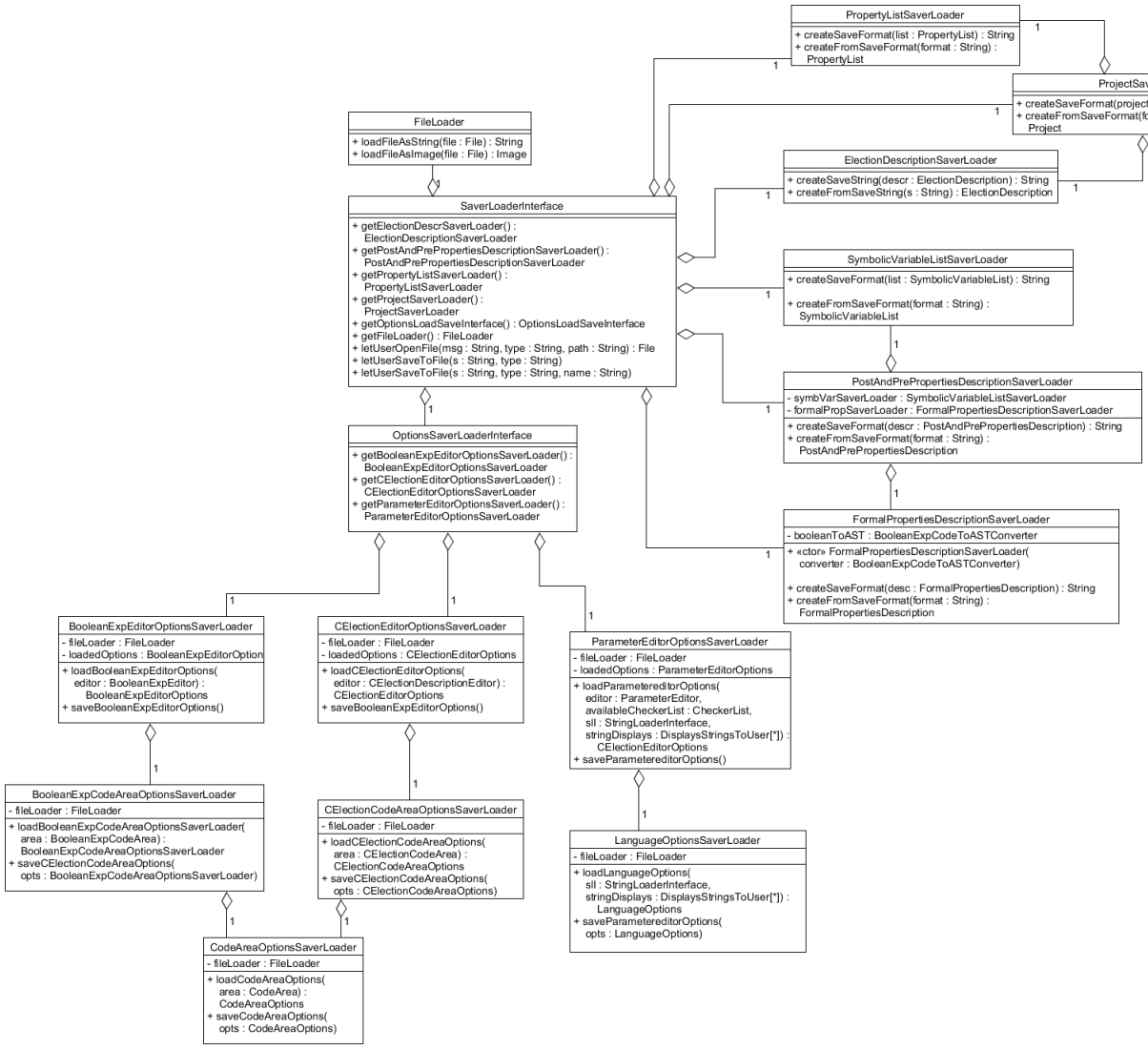


Ändert der Benutzer die Sprache hat dies Auswirkungen auf alle Teile des Systems welche Strings verwenden um mit dem Benutzer zu kommunizieren. Dies sind `ToolBarHandler`, `MenuBarHandler`, `ErrorDisplayer`, `ElectionChangeWizard`, `NewElectionWizard`, Diese Klassen implementieren alle das `DisplaysStringsToUser` Interface. Um das Hinzufügen neuer Sprachen leicht zu ermöglichen werden alle betroffenen `String`s mit einer Id vom Typ `String` assoziiert. Pro Sprache wird dann eine Textdatei angelegt welche diese Ids mit den `String`s als Key-Value Pairs enthält. Textdateien in diesem Format werden von der Klasse `StringResourceLoader` geladen. Die Klassen welche von `StringResourceProvider` erben enthalten die Logik wo und wie diese Files zu laden

sind. `StringLoaderInterface` dient als Schnittstelle zu diesem Package.

4.9 Persistenz von Objekten

Erfüllte FAs: /FM1030/, /FM1040/, /FS1060/, /FM2100/, /FM2110/, /FM3060/, /FM3070/, /FM4050/



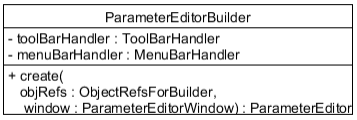
Wie im Pflichtenheft festgelegt müssen Optionen und Datentypen lad- und speicherbar sein. Dies ist im Package SaverLoader implementiert. Jedes Objekt welches gespeichert werden muss hat eine analoge SaverLoader Klasse. Diese nimmt das Objekt entgegen und erzeugt einen String, aus welchem sie das Objekt wieder erzeugen könnte. Diese String Formate sind ähnlich wie JSON Schlüssel-Wert Paare, wobei die Schlüssel die Variablennamen und die Werte die Repräsentation des Wertes sind.

4.10 Parametereditor

Erfüllte FAs: /FM0030/, /FM0040/, /FM0050/, /FM4010/, /FM4011/, /FM4020/, /FM4030/, /FM4040/, /FM4050/, /FM4060/, /FM4070/, /FM4080/
Der Parametereditor stellt Eingabemöglichkeiten für die Parameter Wähler, Sitze, Kandidaten, Dauer, maximale Anzahl von Prozessen und benutzerdefinierte Argumente für CBMC bereit. Als Interface nach außen dient die Klasse `ParameterEditor`.

4.10.1 Aufbau der Komponenten und Interface

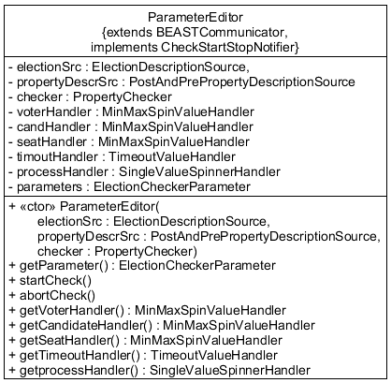
ParameterEditorBuilder



Diese Klasse übernimmt das Erstellen des `ParameterEditor` und seiner Attribute nach dem Builder-Pattern.
Methoden:

- `create` : Die übergebene `ObjectRefsForBuilder` enthält alle benötigten Referenzen zur Erstellung eines `ParameterEditor`. Das `ParameterEditorWindow` liefert das Fenster, über das der Benutzer die Parameter eingeben kann.

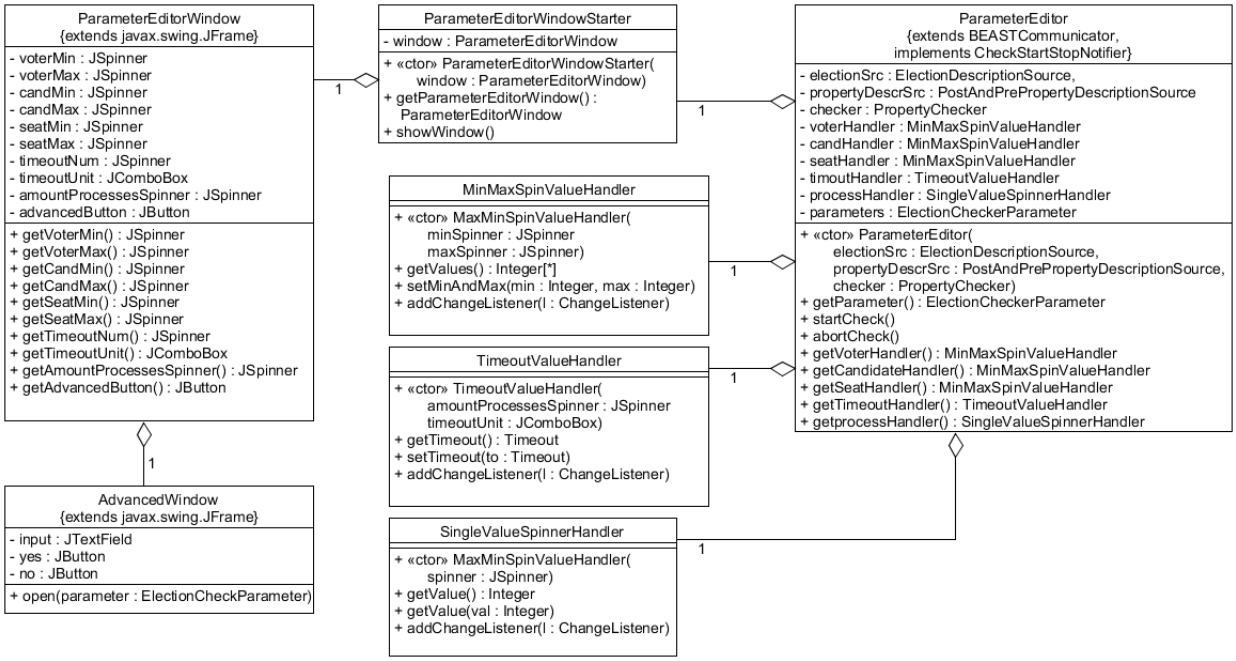
ParameterEditor



Die Klasse `ParameterEditor` dient als Interface für das Package ParameterEditor.
Methoden:

- `startCheck`: Startet die Überprüfung des Wahlverfahrens.
- `abortCheck`: Beendet die Überprüfung des Wahlverfahrens.

4.10.2 Eingabe von Parametern durch den Benutzer



ParameterEditorWindow

Erfüllte FAs: /FM0030/, /FM0040/, /FM0050/, /FM4010/, /FM4011/, /FM4030/, /FM4040/, /FM4050/, /FM4060/, /FM4070/, /FM4080/
Die Klasse `ParameterEditorWindow` stellt die grafische Benutzeroberfläche bereit über die Parameter eingegeben werden und Überprüfungen gestartet und gestoppt werden können.

ParameterEditorWindowStarter

Erfüllte FAs: /FM0030/
Die Klasse `ParameterEditorWindowStarter` startet und speichert das `ParameterEditorWindow`.
Methoden:

- `showWindow`: Zeigt das `ParameterEditorWindow`.

AdvancedWindow

Erfüllte FAs: /FM4040/
Die Klasse `AdvancedWindow` stellt das Dialogfenster zur Eingabe von benutzerdefinierten Argumenten für CBMC bereit.
Methoden:

- `open`: Öffnet das `AdvancedWindow` nach Betätigung des `advancedButton` des `ParameterEditorWindow`

MinMaxSpinValueHandler

Erfüllte FAs: /FM0030/, /FM4010/, /FM4011/, /FM4020/
Die Klasse `MinMaxSpinValueHandler` verwendet das Observer-Pattern um auf ChangeEvents der JSpinner, die Wähler, Kandidaten und Sitze festlegen, zu hören. Die entsprechenden Daten werden dann nach Überprüfung auf Sinnhaftigkeit im `ParameterEditor` geändert.
Methoden:

- `getValues`: Holt sich die Werte des geänderten Parameters.
- `setMinAndMax`: Setzt die neuen Werte des geänderten Parameters.
- `addChangeListener`: Fügt GUI-Elementen einen `ChangeListener` hinzu.

TimeoutValueHandler

Erfüllte FAs: /FM0030/, /FM4020/, /FM4030/
Die Klasse `TimeoutValueHandler` verwendet das Observer-Pattern um auf ChangeEvents des JSpinners und der ComboBox, die den Timeout festlegen, zu hören. Die entsprechenden Daten werden dann nach Überprüfung auf Sinnhaftigkeit im `ParameterEditor` geändert.
Methoden:

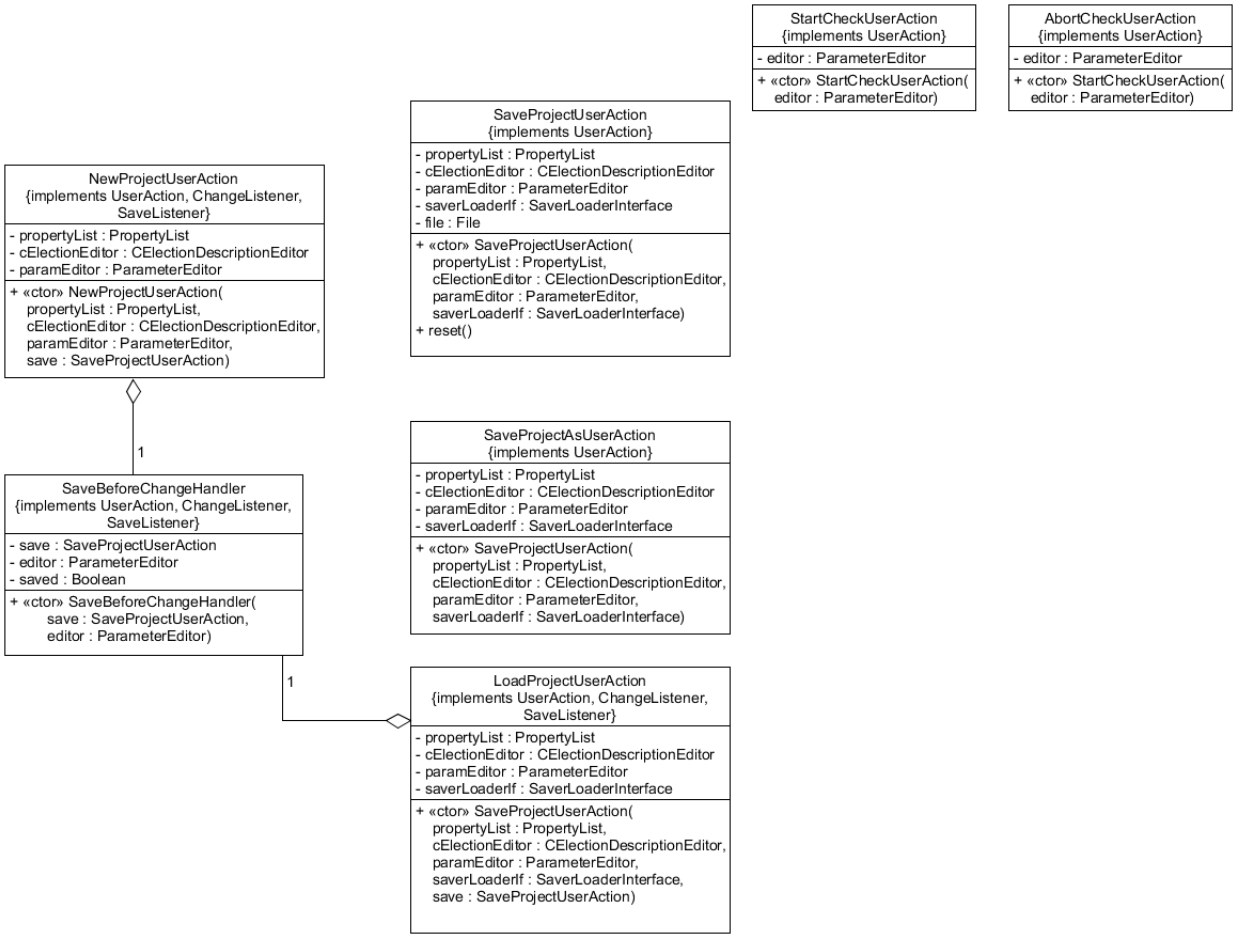
- `getTimeout`: Holt sich die Werte des Timeouts.
- `setTimeout`: Setzt die neuen Werte des Timeouts.
- `addChangeListener`: Fügt dem JSpinner und der ComboBox einen `ChangeListener` hinzu.

SingleValueSpinnerHandler

Erfüllte FAs: /FM0030/, /FM4020/
Die Klasse `SingleValueSpinnerHandler` verwendet das Observer-Pattern um auf ChangeEvents des JSpinners, der die maximale Anzahl an Prozessen festlegen, zu hören. Das entsprechende Datum wird dann nach Überprüfung auf Sinnhaftigkeit im `ParameterEditor` geändert.
Methoden:

- `getValue`: Holt sich den Wert des Parameters.
- `setValue`: Setzt den neuen Wert des Parameters.
- `addChangeListener`: Fügt GUI-Elementen einen `ChangeListener` hinzu.

4.10.3 Verarbeitung von Icon- und Menüklicks



Die Verarbeitung von Icon- und Menüklicks wird durch Klassen geregelt, die `UserAction` implementieren.

SaveProjectUserAction

Erfüllte FAs: /FM0040/, /FM4050/
Die Klasse `SaveProjectUserAction` speichert die Parameter zusammen mit dem Wahlverfahren und den Eigenschaften als Projekt ab. Sollte das Projekt noch nicht vorher gespeichert worden sein, so öffnet sie den Dialog zur Auswahl des Speicherorts.
Methoden:

- `reset`: Setzt die Referenz auf die zuletzt gespeicherte Datei zurück, damit ein neu erzeugtes Projekt nicht das vorhergehende beim nächsten Speichern überschreibt.

SaveProjectAsUserAction

Erfüllte FAs: /FM0040/, /FM4050/
Die Klasse `SaveProjectAsUserAction` öffnet den Dialog zur Auswahl eines Speicherorts und speichert dort das Projekt ab.

NewProjectUserAction

Erfüllte FAs: -
Die Klasse `NewProjectUserAction` .

LoadProjectUserAction

Erfüllte FAs: /FM0040/, /FM4050/
Die Klasse `LoadProjectUserAction` öffnet den Dialog zur Auswahl des Speicherorts einer Projektdatei und lädt diese.

SaveBeforeChangeHandler

Erfüllte FAs: -
Die Klasse `SaveBeforeChangeHandler` vermeidet das versehentliche Verwerfen eines Projekts bei Erstellen eines neuen. Sie erreicht dies, indem sie den Benutzer in einem Dialogfenster fragt, ob er sein vorhergehendes Projekt speichern möchte.

StartCheckUserAction

Erfüllte FAs: /FM4070/
Die Klasse `StartCheckUserAction` startet die Überprüfung des Wahlverfahrens auf die Eigenschaften.

AbortCheckUserAction

Erfüllte FAs: /FM4080/
Die Klasse `AbortCheckUserAction` stoppt die Überprüfung des Wahlverfahrens.

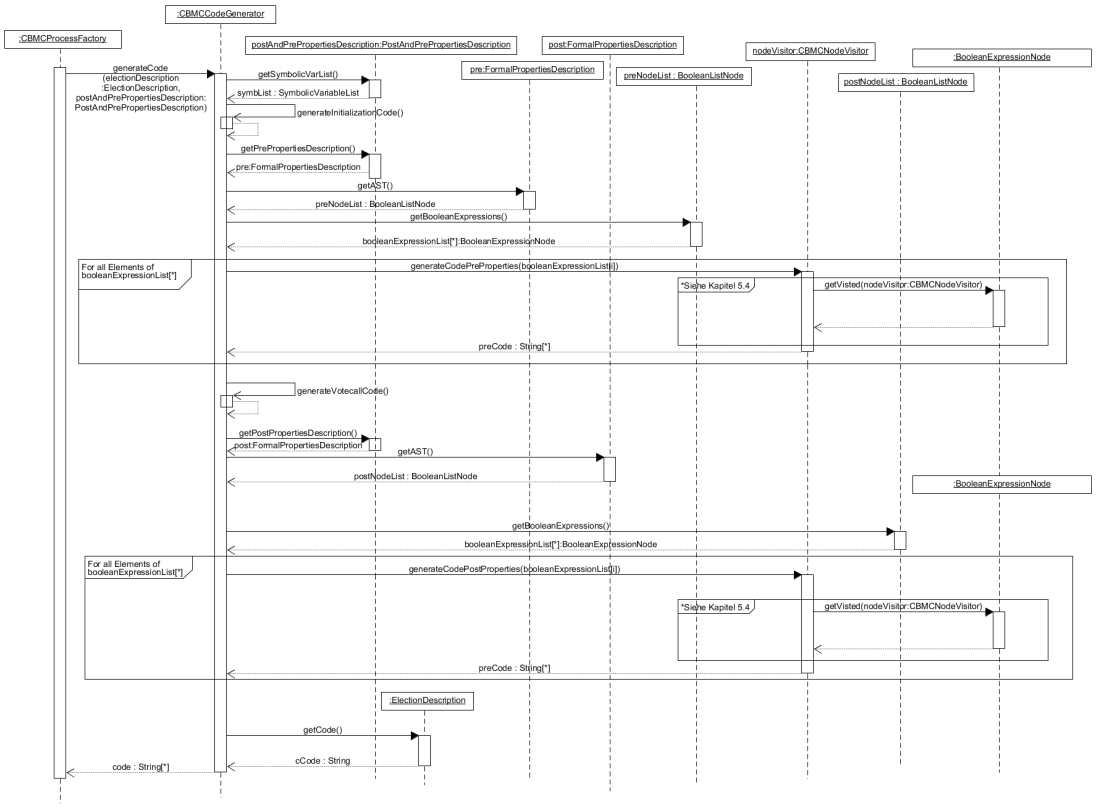
5 Algorithmen

5.1 Zusammenstellung des Quellcodes

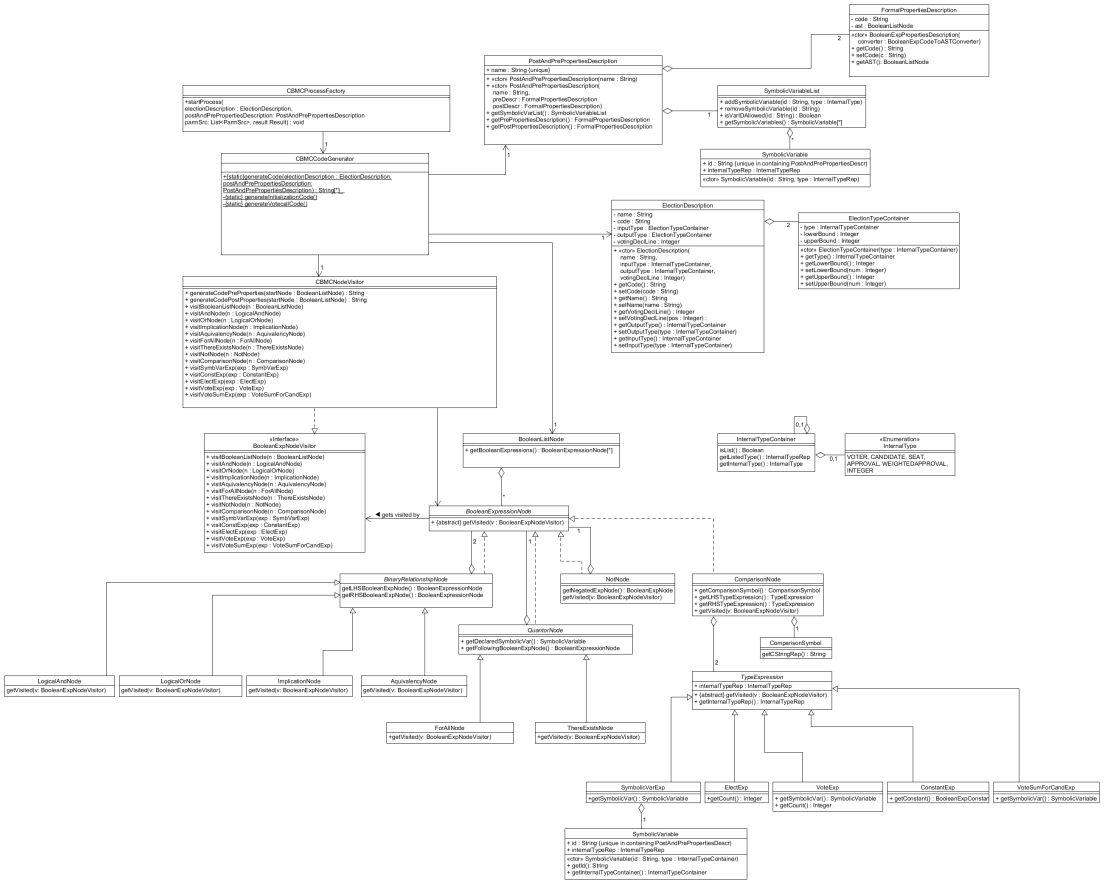
Der gesamte Quellcode, welcher CBMC weitergegeben wird stellt sich aus 4 Teilen zusammen:

- Initialisierungen und Includes
- Mainmethode
- Die Methode, welche die formale Eigenschaft beschreibt
- Die Methode voting und hierfür eventuell Hilfsmethoden, die der Benutzer beschrieben hat

Die Aufgabe der Codezusammenstellung übernimmt die Klasse CBMCCodeGenerator. Sie befindet sich im Package CBMC-Schnittstelle. Die Zusammenstellung des Quellcodes wird erst gestartet, wenn die Überprüfung des Wahlverfahrens auf formale Eigenschaften gestartet wird.
Die genaue Generierung der Mainmethode sowie der Initialisierung und Includes sind Implementierungsdetail. Sie sollen durch private Methoden in der Klasse CBMCCodeGenerator realisiert werden.
Das folgende Sequenzdiagramm zeigt, wie die Datenstrukturen zur Codegenerierung verwendet wird.



Das nachfolgende Klassendiagramm zeigt alle Klassen auf, die zur Codegenerierung (ausgenommen vom Aufbauen der AST) verwendet werden.

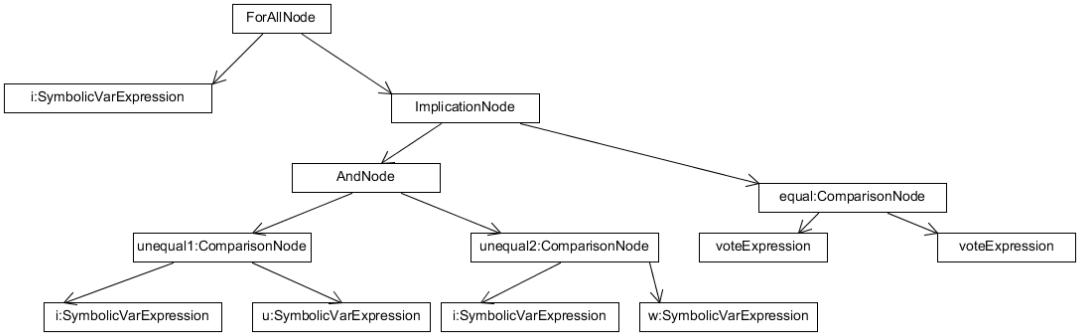


5.2 Umwandlung von formalen Eigenschaften zu Code

Dieses Unterkapitel erklärt die Umwandlung der zuvor erstellten AST zu c-Quellcode. Hierfür wird das Visitorpattern verwendet.
Für jede einzelne Vor- und Nachbedingung ist ein eigener AST vorhanden. Ein Beispiel für die Beschreibung der Vorbedingung der formalen Eigenschaft Anonymität:
u und w sind hierbei symbolische Variablen vom Typ Voter

```
FOR_ALL_VOTERS(i)(i!=u && i!=w) ==> VOTES1[i] = VOTES2[i];
```

Der AST dieser Vorbedingung ist:



Das Visitorpattern wurde hier verwendet, da jeder einzelne Knoten ein zu generierenden Codeteil beschreibt, der unabhängig von den anderen Knoten generiert werden kann.
Der generierte Code variiert je danach, ob es sich um eine Vor- oder Nachbedingung handelt. Assume wird in der Vorbedingung verwendet. Bei der Nachbedingung hingegen wird assert verwendet.
Da der Visitor für Nach- und Vorbedingungen unterschiedlichen aufgerufen wird, muss er diese Information intern speichern.
Pro Knoten im AST wird eine neue Variable vom Typ int deklariert. Die Variablennamen werden im Visitor auf einem Stack gespeichert. Diese Variablen müssen unique implementiert werden. Der Ablauf der Umwandlung des AST zu Code sieht folgendermaßen aus:

1. Der Visitor besucht den Knoten
2. Abhängig von der Knotenart wird ein Teil des Codes des Knoten erstellt
3. Die Variable des gerade Besuchten Knoten wird auf den Stack gepusht
4. Falls es einen tieferliegenden Knoten gibt, wird dieser besucht (Wiederholung der Punkte 1-3)
5. Falls es keinen tieferliegenden Knoten gibt, wird für den gerade besuchten Knoten der restliche Code erstellt. Je nach Knotenart werden 1, 2 oder keine Variablen vom Stack gepopt um den Code zu erzeugen
6. Der Knoten wird verlassen. (Punkt 5-6) werden so lange wiederholt, bis der Stack nur noch ein Element enthält
7. Der Letzte Knoten wird verlassen
8. Bei einer Vorbedingung wird die letzte Variable vom Stack geholt und in einen assume Befehl geschrieben. Bei einer Nachbedingung wird die letzte Variable vom Stack geholt und in einen assert Befehl geschrieben.

Betrachten wir nun den erzeugten Code aus dem vorher genannten Beispiel. Die Kommentare im Code erklären, wie der generierte Code entsteht.

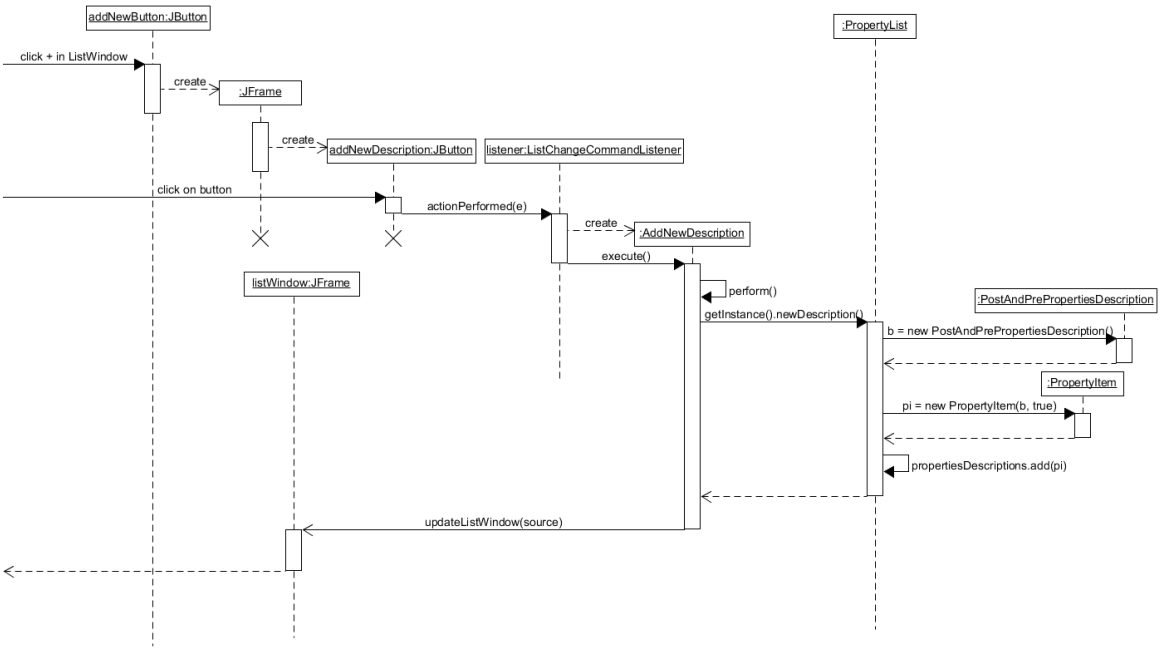
```
//Die ForAllNode wird vom Visitor besucht
int forAll1 = 1;
unsigned int i;
for(i = 0; i < V && forAll1; i++){
        // Variable forAll1 wird auf den Stack gepusht
        // Die ImplicationNode wird vom Visitor besucht
```

```
        // Variable implicationNode1 wird auf den Stack gepusht
        // Die AndNode wird vom Visitor besucht
        // andNode1 wird gevisited
        // Variable andNode1 wird auf den Stack gepusht
        // Die unequal1Node wird vom Visitor besucht
        // Variable unequal1 wird auf den Stack gepusht
        // Es gibt nur noch die 2 zu vergleichenden Werte
         int unequal1 = i != u;
        // unequal1Node wird vom Visitor verlassen
        // Die unequal2Node wird vom Visitor besucht
        // Variable unequal2 wird auf den Stack gepusht
        // Es gibt nur noch die 2 zu vergleichenden Werte
        int unequal2 = i != w;
        // unequal2Node wird vom Visitor verlassen
        // es gibt keine niedrigere Node mehr
        // 2 Elemente werden vom Stack gepopt.
        int andNode1 = unequal1 && unequal2;
        // andNode1 wird vom Visitor verlassen
        // equal1 wird auf den Stack gepusht
        // Es gibt nur noch die 2 zu vergleichenden Werte
        int equal1 = votes1[i] == votes2[i];
        // es gibt keine niederigere Node mehr
        // equal1 wird vom Visitor verlassen
        // 2 Elemente werden vom Stack gepopt.
        int implicationNode1 = !andNode1 || equal1;
        // es gibt keine niedrige Node mehr
        // implicationNode1 wird vom Visitor verlassen
        // 1 Elemente wird vom Stack gepopt.
        forAll1 = implicationNode1;
}
// es gibt keine niedrige Node mehr
// forAll1 wird vom Visitor verlassen
// 1 Elemente wird vom Stack gepopt.
assume (forAll1);
```

6 Anwendungsfälle

6.1 Anwendungsfall für Testfall /T530/

Dieses Sequenzdiagramm stellt den Ablauf von Testfall 530 aus dem Pflichtenheft dar. Eine neue Eigenschaft wird in die Liste aufgenommen und die Darstellung erfolgt in Listenform.



7 Abweichungen zum Pflichtenheft

Vom Pflichtenheft zum Entwurf hat sich nicht sehr viel geändert, hier eine kleine Aufzählung über die kleinen Änderungen:

- Im Parametereditor ist es nun möglich aus allen verfügbaren Checkern einen auszuwählen, der verwendet werden soll.
- Im Parametereditor ist es nun möglich einzustellen, wie viele Checker gleichzeitig parallel ablaufen können.
- Der Eigenschafteneditor stellt nun auch die Verneinung eines boolschen Ausdrucks zur Verfügung

8 Implementierungsplanung

Hier ist die Aufteilung der Packages mit der Person, die diese Teile der Packages realisieren:
Die Packages (hier fett) sind mit allen ihren Meilensteinen jeweils einem der Phasenverantwortlichen zugeordnet.

Was	Wer
C-Editor	Schnell
GUI-Dummy	Klein
GUI-Funktionalität	Klein
Interfaces	Klein
Speichern und Laden	Klein
Fehlererkennung	Klein
Kannkriterien	Klein
CodeArea	Schnell
Basisimplementierung	Schnell und Klein
Syntaxhighliting	Hecht
Code-Completion	Schnell und Klein
CElectionDescriptionEditor	Schnell
Interfaces	Schnell
Fehlererkennung	Klein
Eigenschafteneditor	Schnell
Interfaces	Schnell
GUI-Dummy	Schnell
GUI-Funktionalität	Schnell
Speichern und Laden	Schnell
ANTlrHandler (Erstellung des AST, Fehlererkennung in den formalen Eigenschaften)	Schnell
Kannkriterien	Schnell
Eigenschaftenliste	Hanselmann
Interfaces	Hecht
GUI-Dummy	Hecht
GUI-Funktionalität	Hecht
Speichern und Laden	Hecht
Ergebnisdarstellung	Hecht
Optionen	Hanselmann
Interfaces	Hecht
Restliche Implementierung	Stapelbroek
StringResource	Hecht
Textdatein mit Sprachwahl	Hanselmann

Was	Wer
PropertyChecker	Hanselmann
Interfaces	Hanselmann
Verwendete Datentypen	Stapelbroek mit Hilfe von Hanselmann
CBMC-Ansteuerung	Stapelbroek
Codegenerierung	Hanselmann und Stapelbroek
Eventuelle Multithreadingverbesserung	Stapelbroek
Parametereditor	Hanselmann
Interfaces	Wohnig
GUI-Dummy	Wohnig
GUI-Funktionalität	Wohnig
Speichern und Laden	Wohnig
Sonstiges	Hanselmann
Eine zusammenhängende Jar aus Code generieren	Wohnig
Präsentationsplanung	Schnell und Hanselmann
Änderungsdokument erstellen	Wohning

In dem folgenden Diagramm sieht man die Zeitplanung: