СОФИЙСКИ УНИВЕРСИТЕТ
"СВ. КЛИМЕНТ ОХРИДСКИ"

Факултет по Математика и Информатика
Катедра „Информационни Технологии”

Д И П Л О М Н А Р А Б О Т А

Тема: *Графична среда за обработка и извличане на мета-информация от Java байт код*

Дипломант: Росица Панайотова Панайотова*, фак. № М-21332*

Специалност: ***Информатика***

Специализация: ***Информационни Системи***

Научен ръководител: ***доц. д-р Боян Бончев***

София

2007

Съдържание

1. Увод

1.1. Въведение

Развитието на софтуерните системи и технологии, постепенно наложи езика Java като един изключително мощен инструмент, с чиято помощ може да бъде реализирано всяко едно архитектурно решение - било то софтуер за мобилното устройство, което притежава всеки от нас или софтуер за продуктивни системи, обработващи паралелно милиони клиенти, който могат да си позволят да притежават само най-печелившите компании в световен мащаб. Усложняването на Java технологиите и разработването на все по-нови стандарти във всяка една софтуерна ниша, както и повишаването на нивото на абстракция, са едни естествени следствия от еволюцията на езика Java и платформата за разработка на Java приложения.

В най-новите Java стандарти за езика и платформата - Java Development Kit (JDK) 1.5, Java Enterprise Edition (Java EE) 5, е предоставена възможност да се въвежда мета-информация в кода на приложенията (т.нар. анотации). Това са метаданни, които дават възможност на програмиста да свърже допълнителна информация с клас, поле, метод, параметри на метод и въобще с почти всеки компонент на кода. В по-старите версии на платформата, тази информация най-често се съхранява в допълнителни Extensible Markup Language (XML) файлове.

Заданието на дипломната работа произтича от факта, че не винаги е възможно да се използват вградените в Java методи за достъп до метаданните (Reflection API). За да бъде възможно това е задължително да бъдат заредени всички класове (Class Loading) във виртуалната машина - Java Virtual Machine (JVM), които директно или индиректно са използвани в класа, съдържащ желаната мета-информация. Това ограничение е неприемливо в много от сценариите за използване на мета-информацията. Например, клас от едно приложение няма да може да бъде заредено, ако зависи от друг клас в друго приложение, което не е и не може да бъде заредено поради някаква причина.

Това мотивира нуждата от система, която да извлича метаданните от Java байт кода и да ги структурира в удобен за използване формат, независимо от

зареждането им във виртуалната машина. В същото време системата трябва да предлага алтернативен начин за работа, освен чрез програмният й интерфейс, за да не задължава потребителите да пишат програмен код, необходим за използването й. А именно, реализацията на графичен интерфейс, с чиято помощ потребителите ще могат да работят интерактивно със системата. Чрез този графичен интерфейс ще бъде възможно да се избират входни файлове за обработка, да се правят настройки на системата за филтриране на изходният резултат, да се търсят анотации в цялото множество от класове чрез специални филтри, както и да се записват извлечените метаданни в независим формат - XML.

1.2. Цел и задачи на дипломната работа

Целта на дипломната работа е да се проектира и разработи графична среда, която да предоставя механизъм за обработка и извличане на мета-информация (анотации) от Java байт код. Тази графична среда ще дава възможност за използване на системата като самостоятелен инструмент за филтрация, търсене, анализ, визуализация и запазване на резултатите от извлечените метаданни.

Постигането на поставените цели налага последователното решаване на следните задачи:

Задача 1. Проучване на проблемната област и определяне на входните изисквания към разработваната система.

Задача 2. Анализ и описание на сценариите за използване на системата, въз основа на бизнес нуждите, които тя задоволява.

Задача 3. Проектиране на софтуерната система, обект на разработката.

Задача 4. Реализация на софтуерната система, обект на разработката.

Задача 5. Тестване на разработената система.

Задача 6. Описание на разработената система.

Настоящата дипломна работа представя решаването на изброените по-горе задачи.

1.3. Структура на дипломната работа

Дипломната работа се състои от 7 глави: “Увод”, “Анотация и анотиране”, “Анализ на проблемната област и определяне на входните изисквания”, “Проектиране на решението”, “Описание на реализацията”, “Тестване”, “Заключение”.

Уводът (**Глава 1.**) съдържа кратко въведение в областта на дипломната работа. Формулират се целта, поставените задачи и начинът на структуриране на изложението й.

В **Глава 2.** „Анотация и анотиране“ е описано какво е анотация и кои елементи могат да бъдат анотирани. Представени са примери, илюстриращи начините на тяхното използване. Разгледани са всички, валидни в JDK 1.5, вградени мета анотации. Тук се решава част от **Задача 1** на дипломната работа.

В **Глава 3.** „Анализ на проблемната област и определяне на входните изисквания“ са разгледани бизнес сценариите, в които възниква нужда от анализ на метаданните, с цел да се определят изискванията за входния формат на данните към разработваната система. В резултат, употребата й не се ограничава само до инструмент за работа с метаданните на отделни клас файлове, а при направения анализ, се разширява за да може да бъде използвана в J2EE (Java EE 5) технологията - за извличане на мета-информацията от WEB и EJB компоненти на JEE приложения. Тук се решават изцяло **Задача 1** и **Задача 2** на дипломната работа.

В **Глава 4.** „Проектиране на решението” се формулира идейния проект на конкретната програмна система. Подробно са определени отделните компоненти на системата. Дефинирани са техните цели и отговорности. Направено е подробно описание на техните предназначения и функции, които са моделирани въз основа на изискванията и сценариите за използване на системата. Изготвени са необходимите фигури за илюстриране на дефинираните графични модели за работа, с което изцяло се решава **Задача 3**, както и част от **Задача 6**.

В **Глава 5.** „Описание на реализацията” е решена **Задача 4**, като е представена реализацията на проекта описан в **Глава 3**. Направено е детайлно описание на отделните компоненти на системата, съпроводено с необходимите

UML диаграми [10], отразяващи връзките и структурата им. В тази глава изцяло е решена **Задача 6.**

В **Глава 6.** „Тестване” е решена **Задача 5** и е описан начина, по който е тествана системата.

Заключението представя приносите на дипломната работа и възможните перспективи за развитието ѝ.

Основният текст на дипломната работа се съпровожда от:

- „Списък съкращения и речник на специалните термини”
- „Използвана литература”
- „Приложение”, в което е включено „Ръководство на потребителя”

2. Анотация и анотиране

Новата версия 5.0 на Java 2 Platform Standard Edition (J2SE), повече позната под кодовото име “Тигър” (Tiger), е съсредоточена върху ускоряване процеса на разработка, чрез неговото опростяване и въвеждането на множество улеснения. Промените в J2SE засягат множество аспекти на платформата – производителността, самия език Java, виртуалната машина, базовите библиотеки и тези обслужващи интеграционните услуги, потребителския интерфейс, различните инструменти и поддържани архитектури. Едно от нововъведенията в JDK 1.5, с което са постигнати горните цели е механизма на анотациите [1], [3], [4], [5], [6].

2.1. Анотация

Създателите на новата версия на Java разглеждат анотациите (метаданните) като една от най-съществените особености на езика, избавяща програмистите от задължението да пишат стандартни фрагменти код и както те самите обявяват на сайта на Sun "Програмирането базирано на анотации, в много от случаите, ни позволява да избегнем писането на стереотипен код (повтарящ се код, изпълняващ строго специфична функция), като позволява той да бъде автоматично изгенериран от анотациите в сорс кода. Това води до възможността за декларативен стил на програмиране, където програмиста указва какво трябва да прави приложението, а инструментите, които ги разпознават, се грижат да добавят код, на необходимите за целта места, за да бъде изпълнена поставената задача” [1]. Или с други думи, анотациите са механизъм за прикрепяне/асоцииране на метаданни към декларациите в кода (класове, методи, полета на класа и т.н.), които позволяват на компилатора и/или на виртуалната машина на Java (JVM) да извличат програмното поведение от анотираните елементи и да генерират независим код, когато е необходимо.

Важно е да се отбележи, че самите анотациите нямат ефект върху семантиката на програмата. Те по-скоро променят начина, по който програмата се третира от различни инструменти и библиотеки. Те могат да четат и обработват анотации от сорс код, компилиран клас файл и дори по време на работа на самата програма.

2.2. Декларация на анотация (анотиращ тип)

Анотация (анотиращ тип) се декларира като интерфейс. Единствената разлика в декларациите е знака @, предхождащ ключовата дума interface.

Примера по-долу илюстрира примерна дефиниция на анотация:

```
public @interface TestAnnotation {
  int id;
  String author();
  String date() default ”1/4/2007”;
}
```

Елементите на анотацията се наричат методи, атрибути, полета или членове (в анотацията TestAnnotation са дефинирани елементите id, author и date). Те се подчиняват на следните правила:

- Декларациите на елементите не трябва да съдържат никакви параметри
- Декларациите на елементите не трябва да съдържат нито една throws клауза.
- Връщания тип на елемент трябва да бъде един от следните:
    - Примитивен тип (primitives)
    - Тип String (String)
    - Клас (Class)
    - Изброим тип (enum)
    - Масив от горните типове (array)

Всеки елемент, в тялото на анотацията, може да приема стойност по подразбиране. Това става чрез директивата default, следвана от стойността по подразбиране. Горният пример илюстрира тази възможност, като задава на полето date стойността ”1/4/2007”.

Използването на анотацията, а именно нейното прикрепване към елементи на кода, се нарича анотиране.

2.3. Типове анотации

Анотациите, валидни в JDK 1.5, се делят на два основни вида:

- според броя на елементите

- според предназначението

2.3.1. Според броя на елементите

Според броя на съдържащите се елементи, анотациите се разделят на три типа :

- **Маркер (Marker):** Най-простия вид анотация. Не съдържа атрибути.

```
public @interface TestAnnotation {
}
```

В случай, че целевият елемент е метод, то използването на горната анотация изглежда така:

```
@TestAnnotation
public void testMethod() {
   ....
}
```

- **Анотации с единствен атрибут/единствена стойност (Single-Element /Single-Value):** Този тип анотации се различават от маркерите с това, че могат да съхраняват и някакъв тип данни под формата на един атрибут.

```
public @interface TestAnnotation {
    String author();
}
```

Използването на анотацията се съпровожда с двойката атрибут=стойност или със съкратен синтаксис (ако името на атрибута е “value”) - само със стойността на атрибута, заградени в скоби.

```
@TestAnnotation(author = "TestName")
public void testMethod () {
   ....
}
```

- **Пълни анотации (Full-value/multi-value):** Пълните анотации съдържат повече от един атрибут.

```
public @interface TestAnnotation {
       int id;
       String author();
       String date() default ”1/4/2007”;
}
```

Синтаксисът атрибут=стойност се прилага за всеки един от елементите на анотацията.

```
@TestAnnotation(id = 1, author=”TestName”,date = "10/4/2007")
public void testMethod () {
```

```
    ....
}
```

2.3.2. Според предназначението

Според предназначението анотациите се делят на:

- **Прости анотации (Simple annotations):** Това са основните типове поддържани от JDK 1.5, които могат да бъдат използвани само за анотиране на кода. Този тип анотации не могат да бъдат използвани като тип за дефиниране на собствени анотации
- **Мета анотации (Meta annotations):** Това са анотации, които служат за анотиране на анотации или накратко анотации за анотациите.

2.4. Вградени анотации

В платформата JDK 1.5 вградените анотации наследяват java.lang.annotation.Annotation. Поради естеството на дипломната работа ще бъдат разгледани само мета анотациите. Те биват четири типа @Target, @Retention, @Documented, @Inherited.

2.4.1. Вградена анотация @Retention

Анотацията @Retention има за цел да определи жизнения цикъл на анотираната с нея анотация. Тя може да приема три стойности - SOURCE, CLASS и RUNTIME, които са дефинирани в класа RetentionPolicy от същия пакет:

```
package java.lang.annotation;

public enum RetentionPolicy {
  SOURCE,         // Annotation is discarded by the compiler
  CLASS, // Annotation is stored in the class file, but ignored by the VM
  RUNTIME         // Annotation is stored in the class file and read by the VM
}
```

Първата стойност (SOURCE) ограничава съществуването на анотацията само в Java кода на класа, в който е използвана. Мета-информацията, която тя носи няма да съществува в байт кода на компилирания клас (когато компилатора на Java генерира байт кода на дадения клас, той няма да запише в него метаданните зададени с анотацията).

Втората и третата стойности (CLASS и RUNTIME) за разлика от (SOURCE) не ограничават съществуването на анотацията единствено в Java кода на класа, в който е използвана. Метаданните, зададени с тях, ще бъдат записани в

байт кода на компилирания клас, което превръща тези две стойности в обект на разглеждане за настоящата дипломната работа. Разликата между CLASS и RUNTIME се състои в това, че само анотациите анотирани с RUNTIME ще бъдат заредени от JVM, когато се зарежда байт кода на класа, в който са използвани и съответно ще бъдат достъпни през Reflection API.

Ако една анотация не е анотирана с @Retention, тогава стойността по подразбиране е CLASS.

В горния пример, използването на тази анотация ще изглежда по следния начин:

```
@Retention (RetentionPolicy.SOURCE [,или RetentionPolicy.CLASS
или RetentionPolicy.RUNTIME])
public @interface TestAnnotation {
     String id();
     String author() default "TestName";
     String date();
}
```

2.4.2. Вградена анотация @Target

Анотацията @Target определя към кои елементи от даден клас може да може да бъде прикрепяна анотирана с нея анотация. Тези елементи могат да бъдат клас, интерфейс, анотация, изброим тип (enum), метод, конструктор, поле на клас, параметър на метод и локална променлива на метод. @Target може да приема списък от следните стойности - TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE. Тези стойности са дефинирани в класа ElementType от същия пакет:

```
package java.lang.annotation;

public enum ElementType {
  TYPE,                 // Class, interface, or enum (but not annotation)
  FIELD,                // Field (including enumerated values)
  METHOD,               // Method (does not include constructors)
  PARAMETER,            // Method parameter
  CONSTRUCTOR,          // Constructor
  LOCAL_VARIABLE,       // Local variable or catch clause
  ANNOTATION_TYPE,      // Annotation Types (meta-annotations)
  PACKAGE               // Java package
}
```

Ако една анотация не е анотирана с @Target, тогава стойността по подразбиране е списък от всички изброени по-горе константи. В такъв случай, тя може да бъде прикрепена към произволен елемент.

- Анотациите анотирани с @Target (..., ElementType.TYPE, ...) могат да бъдат прикрепени към декларации на клас, интерфейс, анотация и изброим тип (enum).
- Анотациите анотирани с @Target (..., ElementType.FIELD, ...) могат да бъдат прикрепени към декларации на полета на класа.
- Анотациите анотирани с @Target (..., ElementType.METHOD, ...) могат да бъдат прикрепени към декларации на методи на класа.
- Анотациите анотирани с @Target (..., ElementType.PARAMETER, ...) могат да бъдат прикрепени към декларации на параметри на метод.
- Анотациите анотирани с @Target (..., ElementType.CONSTRUCTOR, ...) могат да бъдат прикрепени към декларации на конструктори.
- Анотациите анотирани с @Target (..., ElementType.LOCAL_VARIABLE, ...) могат да бъдат прикрепени към декларации на локални променливи на методи.
- Анотациите анотирани с @Target (..., ElementType.PACKAGE, ...) могат да бъдат прикрепени към декларации на пакети.
- Анотациите анотирани с @Target (..., ElementType.ANNOTATION_TYPE , ...) могат да бъдат прикрепени към декларации на анотации (изброените мета анотациите @Retention, @Target, @Documented, @Inherited могат да бъдат прикрепени само към декларации на анотации, тъй като горе споменатите, по спецификация, са анотирани само с @Target (ElementType.ANNOTATION_TYPE)).

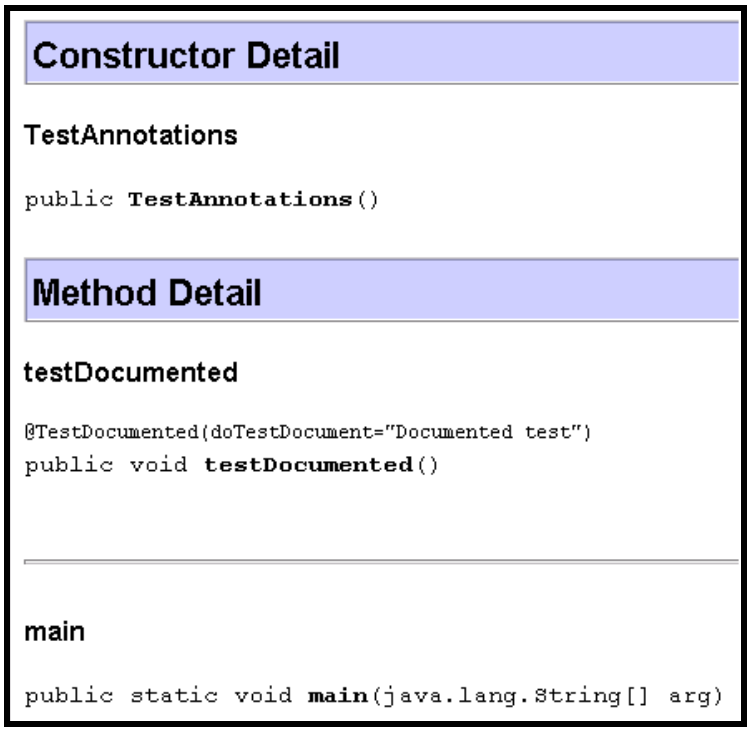
Използването на тази анотация ще изглежда по следния начин:

```
@Target(ElementType.TYPE [,и/или ElementType.FIELD и/или
ElementType.METHOD и/или ElementType.PARAMETER и/или
ElementType.CONSTRUCTOR и/или ElementType.LOCAL_VARIABLE и/или
ElementType.ANNOTATION_TYPE и/или ElementType.PACKAGE])
public @interface TestAnnotation {
      String id();
      String author() default "TestName";
      String date();
}
```

2.4.3. Вградена анотация @Documented

Анотацията @Documented е маркер анотация, а именно анотация без елементи. Тя определя дали анотираните с нея анотации да бъдат включени в Java документацията на класа. По подразбиране анотациите не се включват в Java документите, но ако една анотация е анотирана с @Documented, то тя ще бъде обработена от инструментите за генериране на Java документация и информацията за нейното използване ще бъде включена в изходния документ.

```
@Documented
public @interface TestDocumented {
   String doTestDocument();
}
public class TestAnnotations {

    public TestAnnotations() {}


    @TestDocumented(doTestDocument="Documented test")

    public void testDocumented() { … }


    public static void main(String arg[]) {

       new TestAnnotations().testDocumented();

    }

}
```



Фигура 2.1: Включване на анотация в Java документацията

2.4.4. Вградена анотация @Inherited

Анотираните анотации с @Inherited се използват единствено за анотиране на класове. За разлика от поведението по подразбиране, при което класовете, които наследяват други класове не наследяват техните анотации, @Inherited осигурява автоматичното наследяване на анотираната анотация от супер класа в наследника.

2.5. Как се използват анотациите в Java кода

В тази точка са разгледани примери за анотиране на различни елементи от клас. В примерите се използват вградените анотации Stateless, EJBs, EJB и TransactionAttribute, които се намират в пакета javax.ejb.*, който е част от Java EE 5 платформата. Тези анотации имат следните дефиниции:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Stateless {
    String name() default "";
    String mappedName() default "";
    String description() default "";
```

```
}


@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface EJB {
    String name() default "";
    Class beanInterface() default Object.class;
    String beanName() default "";
    String mappedName() default "";
    String description() default "";
}


@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface EJBs {
    EJB[] value();
}


@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TransactionAttribute {
  TransactionAttributeType value() default REQUIRED;
}
```

От тези дефиниции става ясно, че:

- Анотациите Stateless и EJBs могат да бъдат прикрепени единствено към дефиниция на клас.
- Анотацията EJB допуска да бъде прикрепена към дефиниция на клас, поле и метод.
- На анотацията TransactionAttribute се позволява да бъде прикрепена към дефиниция на клас и метод.

2.5.1. Анотиране на клас

Клас се анотира, като декларацията му се предхожда от сигнатурата на анотацията. Тази анотация трябва да може да бъде прикрепена към декларация на клас. За целта ще бъде използвана анотацията @Stateless, която притежава необходимото свойство по дефиниция:

```
@Stateless
public class AnnotationTest {
    public void field1;

    public void method1(Object param1) {
            //Do nothing
     }
}
```

2.5.2. Анотиране на поле

Поле на клас се анотира, като преди декларацията му се пише сигнатурата на анотация. Дефиницията на анотацията @EJB допуска такова прикрепване:

```
@Stateless
public class AnnotationTest {
    @EJB
    public void field1;

    public void method1(Object param1) {
        //Do nothing
    }
}
```

2.5.3. Анотиране на метод

Метод и конструктор на клас се анотират, по аналогичен начин, като декларацията им се предхожда сигнатурата на анотацията:

```
@Stateless
public class AnnotationTest {
    @EJB
    public void field1;

    @TransactionAttribute (TransactionAttributeType.REQUIRED)
    public void method1(Object param1) {
           //Do nothing
    }
}
```

2.5.4. Анотиране на аргументи на метод

Параметър на метод се анотира, като преди декларацията му се пише сигнатурата на съответната анотация. За целта ще бъде дефинирана анотация, допускаща да бъде прикрепена към параметър на метод:

```
@Target({ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface ParameterAnnotation {
   String value1();
   String value2();
}


@Stateless
public class AnnotationTest {
   @EJB
   public void field1;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
    public void method1(@ParameterAnnotation (value1=”value1”,
                                    value2=”valu2”) Object param1) {
            //Do nothing
     }
  }
```

2.5.5. Вградени една в друга анотации

Анотациите могат да бъдат вграждани една в друга, т.е. една анотация може да бъде атрибут на друга анотация. Такава е анотацията @EJBs, която съдържа списък от @EJB анотации:

```
@EJBs ({@EJB (beanName="beanName1", beanInterface=BeanInterface1.class)
        @EJB (beanName="beanName2", beanInterface=BeanInterface2.class)
      })
public class AnnotationTest {
    @EJB
    public void field1;

    @TransactionAttribute (TransactionAttributeType.REQUIRED)
        public void method1(Object param1) {
            //Do nothing
        }
}
```

2.5.6. Масиви като атрибути на анотация

Стойността на даден атрибут на анотация може да бъде масив. В горния пример анотацията @EJBs има само един атрибут и това е атрибута по под-разбиране “value”. Неговата стойност е масив, изграден от две анотации @EJB, разделени със запетайка.

3. Анализ на проблемната област и определяне на входните изисквания

За да бъде решен даден софтуерен проблем трябва да бъде проучена проблемната област, както и да бъдат изследвани сценариите, в които този проблем е дефиниран. Благодарение на това, могат да бъдат описани рамките на решението [2], [7], [8].

3.1. Анализ на бизнес нуждите за извличане на метаданни от клас файлове

За да се определи рамката, върху която ще функционира разработената система, трябва да се дефинират бизнес сценариите, в които възниква нужда от анализ на метаданните на клас файловете. По този начин ще се придобие цялостна представа върху формата на входните данни, съдържащи клас файлове. Дали е достатъчно да се анализира само отделен клас файл или директория с клас файлове, това е въпрос, на който ще бъде отговорено в тази точка.

Както бе определено в Глава 2, анотациите представляват метаданни асоциирани с елементи на кода. Тяхната цел е да направят възможно прикрепването на информация в самия Java код, вместо да се използват допълнителни конфигурационни файлове, които са трудни за поддръжка и актуализация. Те целят опростяване на модела за работа във всички аспекти и технологии на Java.

Най-широко използваната технология на Java е J2EE. Опростяването не пропуска и нейната най-нова версия Java EE 5, базирана на JDK 1.5. Основните единици, които дефинира тази технология са JEE Applications. Това са приложения, които задоволяват разнообразни бизнес нужди. Те представляват множество от файлове и архиви, пакетирани във файл. Преди да излезе новата спецификация Java EE 5, тези архиви съдържаха множество конфигурационни XML файлове. С излизането на Java EE 5 е намерен алтернативен начин за конфигуриране на тези приложения. XML файловете са заменени от използването на анотации [2].

Всяко едно JEE приложение е изградено от модули. Тези модули имат различна роля в бизнес процеса, който реализира приложението. Има два вида модули, които представляват интерес за разработката. Това са Enterprise JavaBeans (EJB) модули и Web модули. Те представляват множество пакетирани файлове, в това число и Java класове. Именно, в тези класове се пазят нужните конфигурации, под формата на анотации. EJB и Web модулите представляват архивирани файлове с разширения съответно “.jar” и “.war”. Една от целите на дипломната работа е да може да анализира съдържанието на тези файлове.

От казаното по-горе може да се дефинира формата на входните данни, съдържащи клас файлове. Има 4 основни източници на мета информация и това са:

- Отделен клас файл на файловата система.
- Директория на файловата система, съдържаща клас файлове или други директории, които от своя страна съдържат клас файлове.
- Архивен файл с разширение “.jar”.
- Архивен файл с разширение “.war”.

3.2. Анализ на съществуващи решения

Съществуват различни инструменти за четене на Bytecode, но всички те имат един или няколко от следните недостатъци:

- Не са пригодени за работа с байт кода на JDK 1.5
- Не са приспособени специално за работа с анотации, тъй като предоставят интерфейс за работа на доста ниско ниво, най-често използван за модификация на байт код.

По известните инструменти за четене и манипулация на Bytecode са:

- Byte Code Engineering Library – проект на Apache Software Foundation [bcel] [12]

- ASM, Java bytecode manipulation framework – проект на ObjectWeb Consortium [13].
- SERP – проект на Abe White [14].

▪ Byte Code Engineering Library

Библиотека Byte Code Engineering Library (BCEL API) предоставя възможност на потребителите за статичен анализ, динамично създаване или преобразуване на Java клас файлове. Класовете са представени от обекти, които съдържат цялата кодирана информация на дадения клас: методи, полета и в частност, байт код инструкции.

Такива обекти могат да бъдат прочетени от съществуващ файл, да бъдат трансформирани от програма (например клас лоудъра по време на изпълнение) и записани отново във файл. По-интересно приложение е създаването на класове изцяло по време на изпълнение. BCEL API може да бъде полезно и за изучаване на Виртуалната машина на Java (JVM) и структурата на Java клас файловете. BCEL предоставя средство за верификация на байт кода, наречено JustIce, което в повечето от случаите, дава много по-подходяща информация относно грешките в кода, отколкото стандартните JVM съобщения.

BCEL вече се използва успешно в проекти като например компилатори, оптимизатори, генератори на код и инструменти за анализ.

За съжаление не е имало развитие през миналите няколко години.

▪ SERP

Библиотеката SERP може да бъде използвана за интерпретация на различни програмни езици, за да могат да бъдат стартирани с JVM, за създаване на нови класове по време на изпълнение, като инструмент за анализ на технически характеристики, за дебъг, за изменение повишаващо качествата на съществуващи компилирани класове.

Целта на SERP е да предлага пълен набор от Bytecode модификации, като се опитва да понижи времето за изпълнение и използваната памет. Той осигурява API от високо ниво за манипулиране на всички аспекти на байт кода: от по-мащабните структури, като полета на клас, до структурите от най-ниско ниво - отделни инструкции, които изграждат кода на методи.

- **ASM - Java bytecode manipulation framework**

ASM е библиотека за манипулация на байт код. Тя предлага подобна функционалност като BCEL и SERP, но е по-малка и по-бърза от тези инструменти. ASM да бъде използвана за динамично генериране на класове директно в двоична форма, за модифициране на класове по време на зареждането им във виртуалната машина и по-точно, непосредствено преди зареждането им във виртуалната машина.

До момента ASM се изпозва в повече от 40 продукта.

- **APT - Annotation Processing Tool**

Инструментът **APT** има сходни функции с изброените по-горе. Той за разлика от тях обработва java сорс файлове вместо java клас файлове, като предоставя програмен интерфейс подобен на Reflection API. Обикновено се използва в среди за разработка на Java сорс код, като изгражда модел на кода на програмата, който се използва за реализация на функциите на средата (AutoComplete, Methods View, Class Structure View, Find method’s usages и т.н.). Към този инструмент има допълнителни библиотеки, с чиято помощ може да се стартира процес преди компилацията на сорс кода, в който да се генерират допълнителни класове в зависимост от анотациите, които имат останалите входни класове. Тази функция е много удобна за създаването на специфични ANT скриптове.

Основните характеристики на разгледаните съществуващи решения са описани в таблица 3.2.

Характеристика / Име	BCEL	ASM	SERP	APT
Поддръжка на J2SE5.0 Annotations	не	да	не	да
Допълнително натоварване причинено от библиотеката по време на зареждането на клас	700%	60%	1100%	-
Големина на библиотеката	350KB	33KB	150KB	250KB
Работа върху class файлове	да	да	да	не
Работа върху java файлове	не	не	не	да
Програмен интерфейс	да	да	да	да
Графичен интерфейс	не	не	не	не

Таблицата 3.2: Характеристики на разгледаните съществуващи решения

Както се забелязва от описанието тези решения са общи и са насочени към четене и модификацията на байт код. Те нямат функционалност тясно специализирана за четене на анотации. Към настоящия момент, не съществуват инструменти за визуализация и записване в XML формат на метаданните на клас.

4. Проектиране на решението

В тази глава е описано проектирането на решението. Изброени са отделните визуални компоненти и функцията, която ще изпълняват, в това число и действията, които ще се извършат при настъпване на определено събитие (натискане на бутон, меню бар, контекстно меню и т.н.).

Проектирането на графичен интерфейс се състои от една страна в дефинирането на визуалната част, чрез която клиентите ще използват системата (прозорци, текстови полета, визуални дървета, контекстни менюта и т.н.), а от друга в дефинирането на действия (Actions), които ще извършва системата, когато клиента предизвика определени събития (MouseEvent, KeyEvent и т.н.) т. нар. бизнес логика на приложението.

Първо ще бъдат дефинирани всички функции, които трябва да предоставя приложението, след което, с помощта на изображения, ще бъдат онагледени начините, по които тези функции могат да бъдат достъпни чрез възможностите предоставени от графичните компоненти, които ще бъдат използвани.

4.1. Бизнес логика – набор от функции за работа с приложението

Следва проектиране на функциите, които приложението ще предоставя на потребителите. Реализацията на тези функции е описан в **Точка 5.7**.

4.1.1. Функция 1 - Създаване на проект 

Проектът описва множеството от Java класове, които ще бъдат анализирани. Той се съхранява под формата на XML, като дава възможност на потребителя да съхранява направените от него настройки, за да могат да бъдат използвани многократно. Моделирането на проект и грижата за неговия жизнен цикъл произтича от факта, че обикновено потребителят предварително знае местоположението на файловата система на директориите и файловете съдържащи класовете, които той иска да анализира. С други думи работното множество от входни файлове е константно за даден момент във времето. От друга страна, създаването на различни проекти помага на потребителя да групира входните файлове, по желания от него начин, за да може да използва функциите на приложението върху специфицирани от него части от цялото

множество достъпни класове. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-N**, чрез бутона  , чрез менюто на главния прозорец на приложението **File>New Project** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.2. Функция 2 - Зареждане на вече съществуващ проект 

Зареждането на вече съществуващ проект, съхранен на файловата система, под формата на XML, е функция произтичаща от предходната точка. Системата трябва да предоставя възможност на потребителя да навигира във файловата система и да избира предварително съхранения си проект. След което, информацията записана в него трябва да бъде извлечена с помощта на XML Parser и да бъдат заредени всички класове, които потребителят е включил създавайки проекта. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-O**, чрез бутона  , чрез менюто на главния прозорец на приложението **File>Open Project** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.3. Функция 3 - Визуализация и промяна на настройките на зареден проект 

Промяната на настройките на проект е функция в допълнение към предходните. Тя дава възможност на потребителя да включи/изключи определено подмножество от класове към вече зареден проект. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-P**, чрез бутона  , чрез менюто на главния прозорец на приложението **View>Project Properties** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.4. Функция 4 - Затваряне на зареден проект

Функцията затваряне на проект предоставя възможност на потребителя да поддържа набора от заредени проекти, като му дава възможност да изключва тези, които в даден момент не са необходими. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-F4**, чрез бутона  , чрез менюто на главния прозорец на приложението **File>Close Project** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.5. Функция 5 - Затваряне на всички заредени проекти

Функцията затваряне на всички проекти е за улеснение на потребителя. Тя предоставя възможност за затваряне на всички проекти, без да се налага да се затваря приложението и предотвратява затварянето на проектите един по един. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-SHIFT-F4**, чрез бутона , чрез менюто на главния прозорец на приложението **File>Close All** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.6. Функция 6 - Затваряне на приложението

Функцията за изход от приложението може да бъде използвана след като потребителят е приключил с анализа на метаданни. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **ALT-F4**, чрез бутона или чрез менюто на главния прозорец на приложението **File>Exit**.

4.1.7. Функция 7 - Търсене на анотации в произволно подмножество от съдържанието на всички проекти

Това е една от основните функции на приложението, с чиято помощ потребителят може да търси анотирани класове (или техни елементи), отговарящи на определени критерии. Търсенето може да е в множеството от всички заредени проекти, само в даден проект или в произволна част от проект (конкретен входен файл, Java пакет, Java клас). Подробното описание на критериите за търсене и контекстите, в които то е възможно, е направено в **Точки 4.6.1, 4.8, 4.9.1 и 4.10**. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-F**, чрез бутона , чрез менюто на главния прозорец на приложението **Tools>Find Annotations** или чрез контекстните менюта на графичното дърво за визуализация на проектите.

4.1.8. Функция 8 - Записване на метаданните в XML формат

Втората основна функция на приложението е записването, под формата на XML, на клас метаданните, съдържащи се в проектите или техни подмножества, както и метаданните съдържащи се в резултата от търсене. С нейна помощ, извлечената информация от байт кода може да бъде записана в текстов формат, който може да бъде входна точка на други инструменти и в същото

време може да бъде прочетен от самия потребител без да се налага повторно използване на системата. Подробното описание на контекстите, в които е възможно записването на метаданни в XML формат, е направено в **Точки 4.6.1, 4.8, 4.9.1 и 4.10**. Тази функция ще бъде достъпна чрез бърза комбинация от клавиши **CTRL-E**, чрез бутона  , чрез менюто на главния прозорец на приложението **Tools>Export to XML** или чрез контекстните менюта на графичните дървета за визуализация на проектите и резултата от търсене.

4.1.9. Функция 9 - Визуализация на информация за автора на проекта (About) 

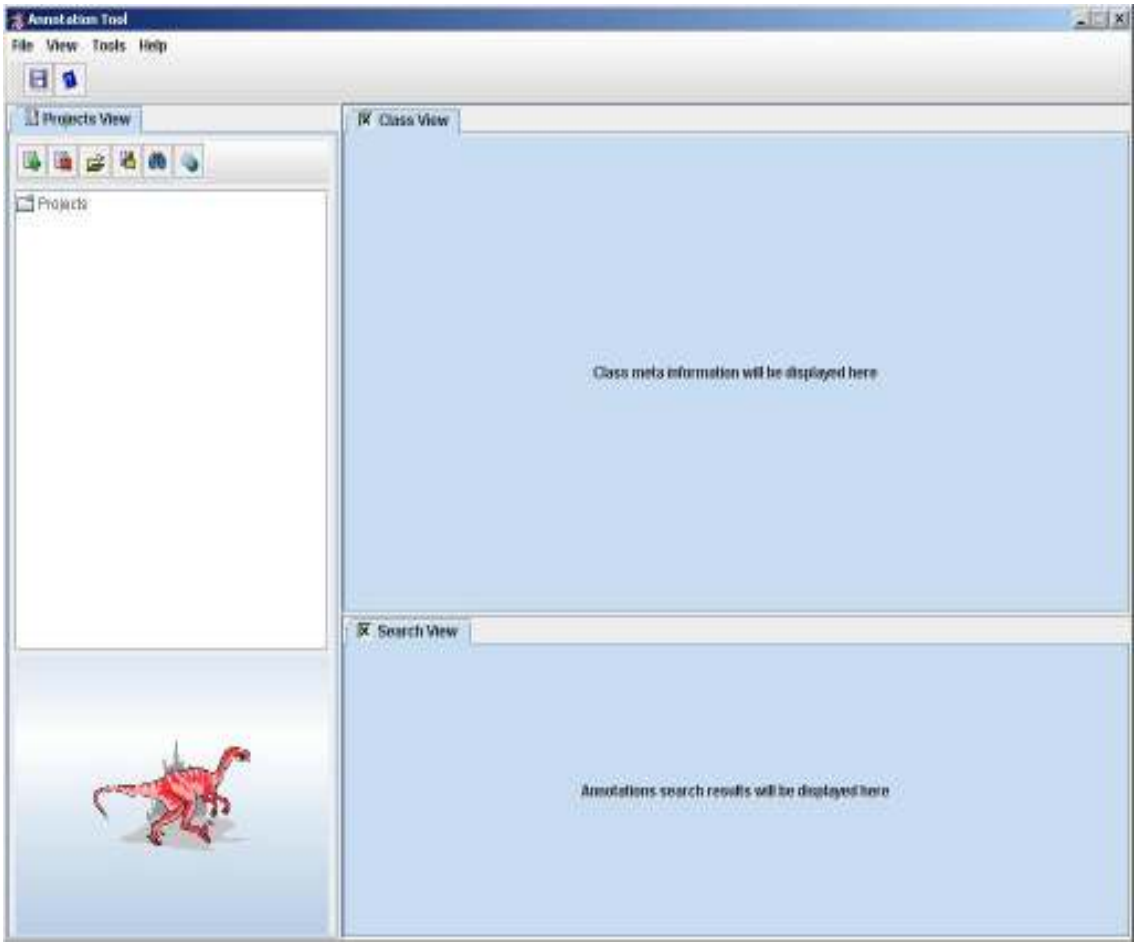
Тази функция е стандартна за графичните приложения и има за цел да визуализира информация свързана с автора на приложението. Ще бъде достъпна чрез менюто на главния прозорец на приложението **Help>About**.

4.1.10. Функция 10 - Визуализация на ръководство на потребителя 

Тази функция също е стандартна за графичните приложения. Тя е в помощ на потребителя, като осигурява необходимата информация по отношение начина на работа със системата. Ще бъде достъпна чрез бърза комбинация от клавиши **F1**, чрез бутона  или чрез менюто на главния прозорец на приложението **Help>Help**.

4.2. Главен прозорец на приложението – Main Frame

При стартиране на приложението трябва да се отвори неговия главен прозорец, от който чрез използването на менюта, бързи бутони и бързи комбинация от клавиши може да се достъпят различните функции описани в **Точка 4.1**:

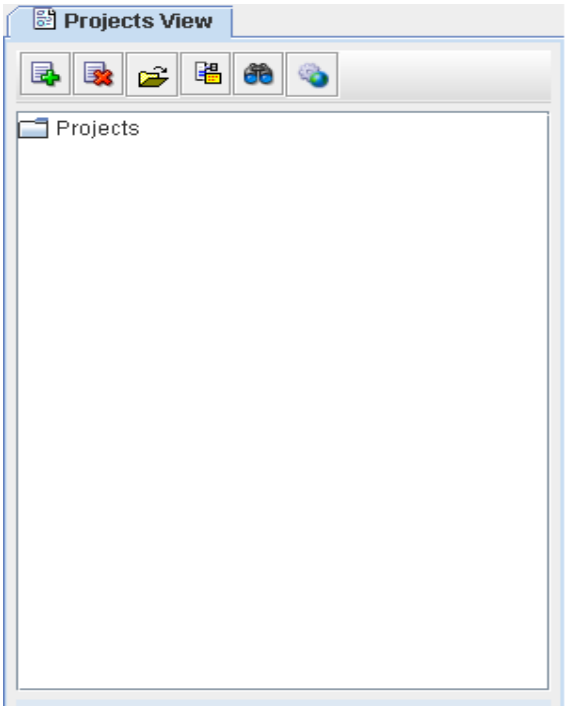


Фигура 4.1: Главен прозорец на приложението

Той съдържа следните графични компоненти:

4.2.1. Графичен компонент за визуализация на заредените проекти – Projects View

Компонента за визуализация на заредените проекти има за цел да отрази графично множеството от активни проекти. Поради естеството на задачата най-подходящия начин за реализация е използването на графично дърво - javax.swing.JTree. Този компонент е разположен в лявата част на главния прозорец и съдържа една от входните точки за използване на функциите описани в **Точка 4.1** – бързите бутони разположени в горната му част, непосредствено над графичното дърво:



Фигура 4.2: Projects View

Подробно описание за структурата на визуалното дърво на компонента е дадено в **Точка 4.6**.

4.2.2. Графичен компонент за визуализация на резултата от търсене по зададени филтри – Search View

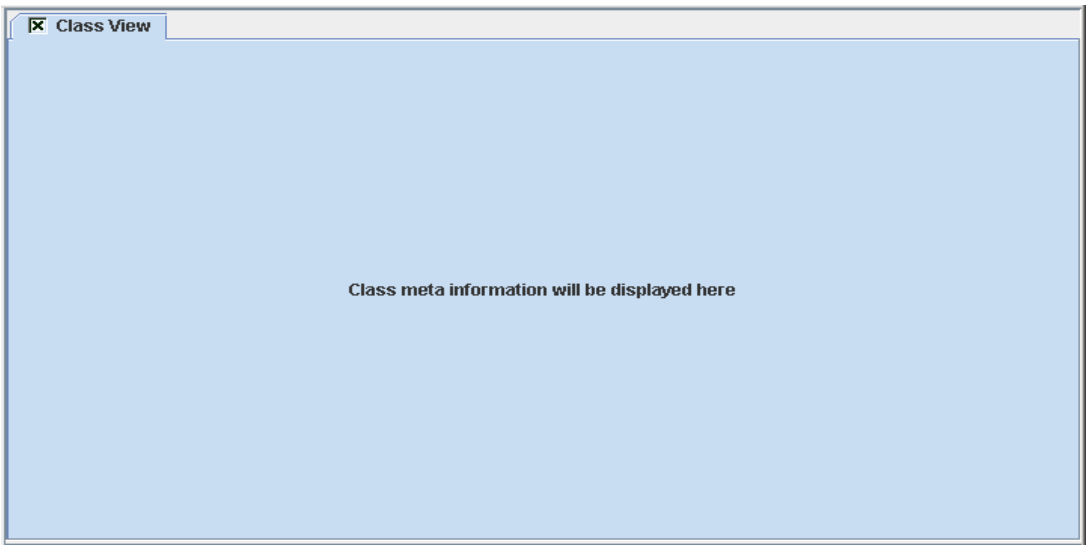
Компонента, който визуализира резултатите от търсенето. Той е реализиран с помощта на javax.swing.TabbedPane, за да може да визуализира едновременно няколко резултата от търсене в различни табове. Резултата във всеки таб е организиран с графично дърво. Компонента е разположен в долният десен ъгъл на главния прозорец.

Фигура 4.3: Search View

Подробно описание за структурата на визуалните дървета на компонента е дадено в **Точка 4.9**.

4.2.3. Графичен компонент за визуализация на метаданните на отделен клас – Class View

Компонента има за цел да визуализира елементите и анотациите на отделен клас. Той е реализиран аналогично на Search View, с помощта на javax.swing.TabbedPane, за да може да визуализира едновременно информацията за няколко класа в различни табове. Резултата във всеки таб е организиран с графично дърво. Компонента е разположен в горния десен ъгъл на главния прозорец.



Фигура 4.4: Class View

Подробно описание за структурата на визуалните дървета на компонента е дадено в **Точка 4.7**.

4.3. Създаване на нов проект – New Project Dialog

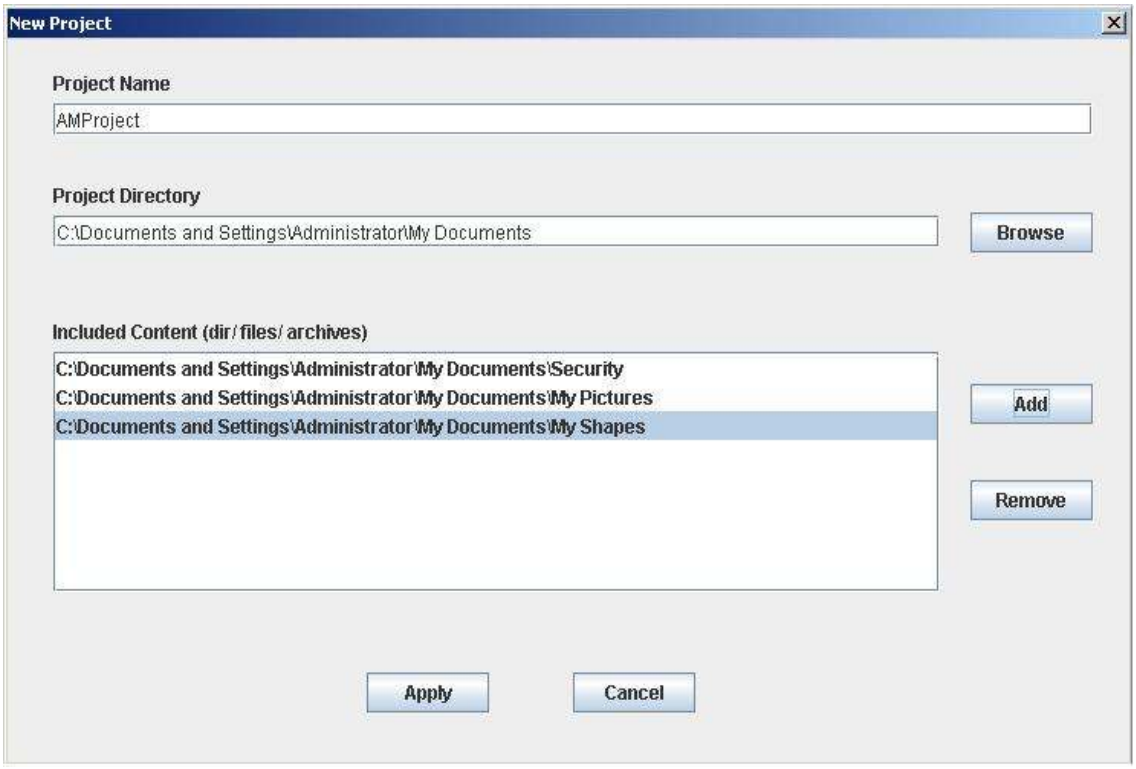
Този графичен компонент обслужва входната точка на системата (**Функция 1**) – създаването на проект, описващ множеството от Java класове, които ще бъдат анализирани (виж. **Фигура 4.5**).

Проектът се състои от:

- *Име,* под което ще бъде записан.

- *Директория,* в която ще бъде записан.
- *Списък от входни файлове и/или директории* съгласно допустимите входни данни описани в **Точка 3.1** (Анализ на бизнес нуждите) - *.jar, *.war, *.class или директория.

Проектът трябва да се съхранява под формата на XML файл, с посочено име и директория, за да може да бъде зареждан повторно.



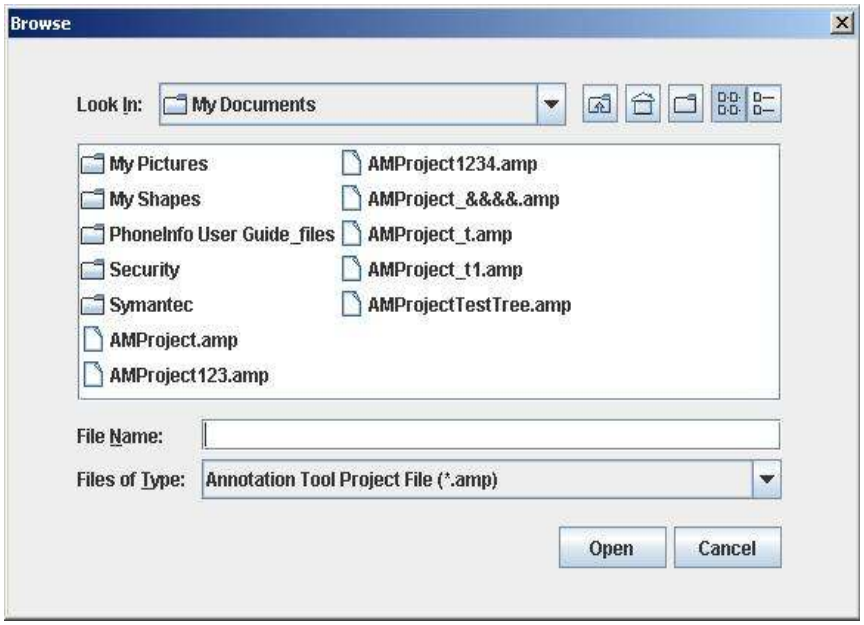
Фигура 4.5: Диалогов прозорец за създаване на проект

След попълване на необходимата информация за създаване на проект е необходимо той да бъде съхранен. Това става чрез бутона **Apply**. Ако всички попълнени параметри отговарят на изискванията (името на файла и директорията не трябва да съдържат недопустими символи, подадените файлове или директории, които трябва да бъдат обработени трябва да са сред допустимото множество – jar, war и т.н.), новосъздадения проект се записва под формата на XML във файл с разширение **.amp** (Annotations Model Project). При отказ на пот-

ребителя, диалоговия прозорец може да бъде затворен чрез натискане на бутона **Cancel**.

4.4. Зареждане на съществуващ проект – Load Project Dialog

За зареждане на проект се използва стандартния за Java/Swing графичен компонент javax.swing.JFileChooser. С негова помощ се реализира **Функция 2**. Той дава възможност на потребителя да навигира из файловата система за да избере вече съществуващ проект, който да бъде зареден:



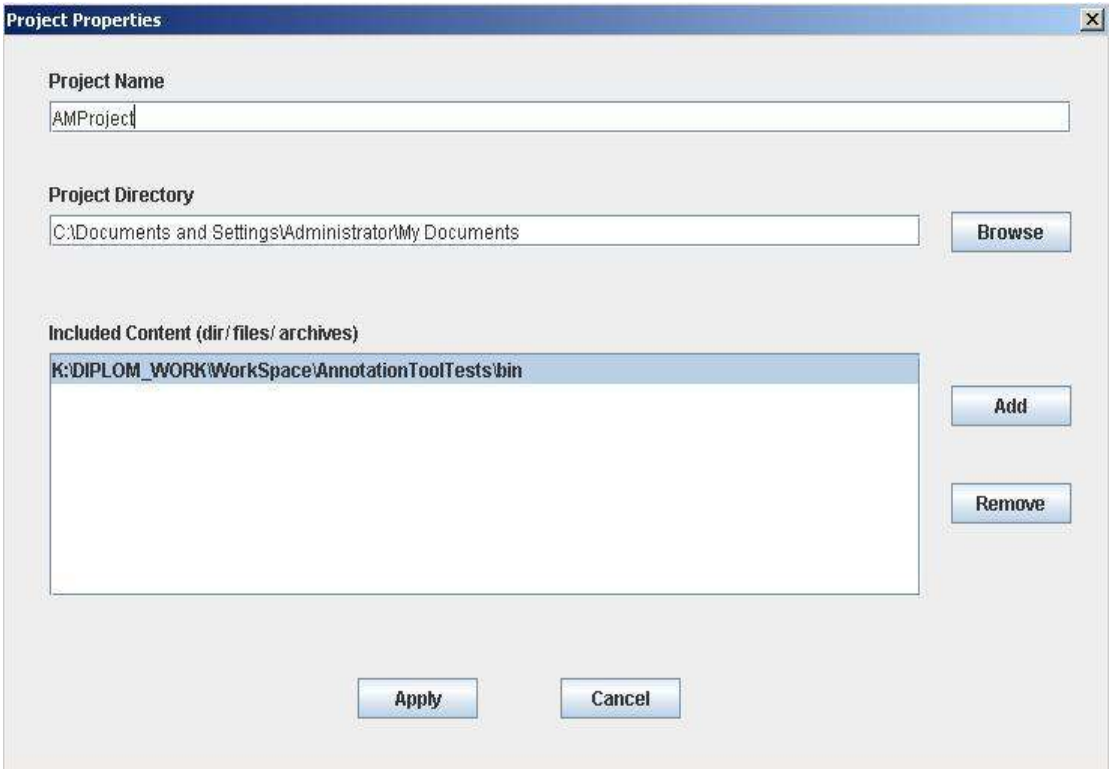
Фигура 4.6: Диалогов прозорец за зареждане на съществуващ проект

След като проекта бъде избран и потребителят натисне бутона **Open**, проекта се зарежда и визуализира в Projects View. Подробно описание за структурата и съдържанието на Projects View е дадено в **Точка 4.6**.

4.5. Модификация на зареден проект – Project Properties Dialog

Чрез този графичен компонент се реализира **Функция 3**. След като даден проект е зареден, той може да бъде модифициран. Под модификация се разбира промяна на името на проекта и/или промяна на директорията, в която проек-

та ще бъде записан. Освен това чрез бутоните **Add** и **Remove** може да се добавят или премахват файлове и директории, съдържащи класовете на проекта (content files):



Фигура 4.7: Диалогов прозорец за модифициране на зареден проект

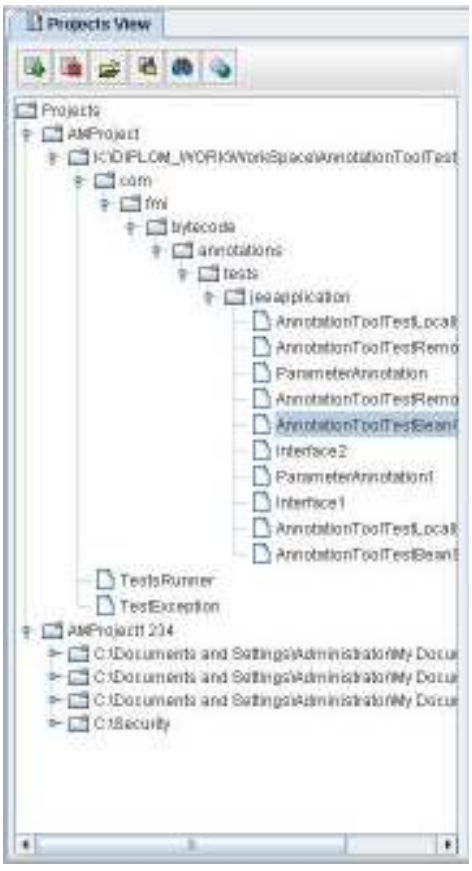
След като потребителят е направил необходимите модификации, може да ги запише с помощта на бутона **Apply**. За отказ на направените промени се използва бутона **Cancel**.

4.6. Контекстни менюта и структура на Projects View

Графичния компонент за визуализация на проекти ще бъде реализиран с графично дърво – максимално интуитивно решение за подобен тип проблеми, което се използва във всички графични инструменти от този род. Структурата на дървото е следната (вж. **Фигура 4.8**):

- Елемента “Projects” е корен на дървото

- Елементите, които са наследници на корена представляват заредените проекти. За имена на тези елементи се взима името на съответния проект.
- Елементите, които са преки наследници на проекта са неговите т.нар. content files – файловете и директориите, специфицирани при създаването на проекта, които съдържат клас файловете на проекта.
- Следват пакетите на класовете, като за всеки под пакет се създава отделен елемент с името му.
- Листата на дървото са класовете, намиращи се в съответния под пакет като за име на елемента се взима името на класа (без името пакета).

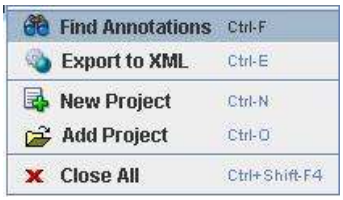


Фигура 4.8: Структура на Projects View

4.6.1. Контекстни менюта на Projects View

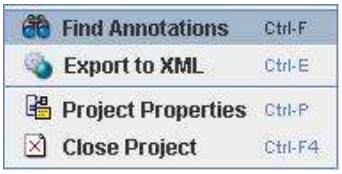
Всеки елемент, от визуалното дърво на Projects View, има собствено контекстно меню. Менюто се визуализира чрез натискане на десен бутон на мишката върху съответния елемент. Функциите, които предлага се асемблират в зависимост от функцията, която изпълнява елемента. Следва описание на функциите присъстващи в менюто на всеки от 5-те вида елементи:

- Контекстно меню на елемента “Projects”:



Реализира следните функции описани в **Точка 4.1** - **Функция 1**, **Функция 2**, **Функция 5**, **Функция 7**, **Функция 8**.

- Контекстно меню на елементите визуализиращи проект



Реализира следните функции описани в **Точка 4.1** - **Функция 3**, **Функция 4**, **Функция 7**, **Функция 8**.

- Контекстно меню на елементите визуализиращи project content.

Реализира следните функции описани в **Точка 4.1** - **Функция 7**, **Функция 8**.

- Контекстно меню на елементите визуализиращи Java пакет

Реализира следните функции описани в **Точка 4.1** - **Функция 7**, **Функция 8**.

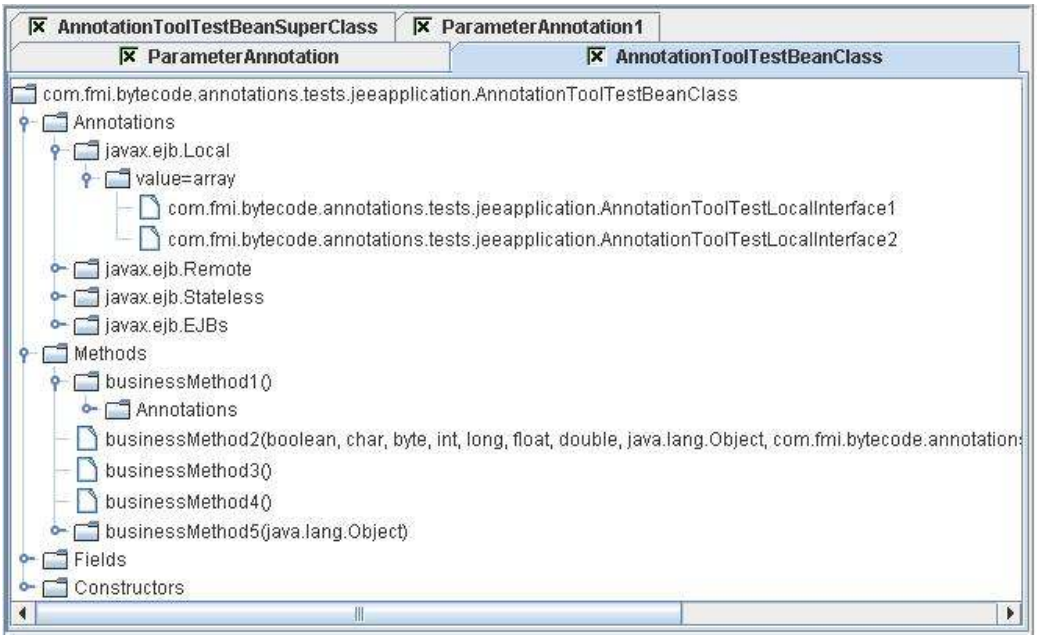
- Контекстно меню на елементите визуализиращи Java клас

Реализира следните функции описани в **Точка 4.1** - **Функция 7**, **Функция 8**.

4.7. Структура на Class View

Графичният компонент Class View служи за визуализация на метаданните на отделен клас файл. За целта, отново подходящо решение е използването на графично дърво, тъй като структурата на класа е йерархична. Визуализацията на клас може да бъде предизвикана по два начина – чрез двойно натискане на левия бутон на мишката върху определен клас от Project View или Search View. Когато визуализацията е инициирана от Search View, съответния елемент асоцииран с конкретния резултат от търсене (притежаващ определена анотация) ще бъде оцветен с червен цвят в Class View (Фигура 4.9).

Class View е реализирано с помощта на класа javax.swing.TabbedPane, с чиято помощ потребителят може да отвори едновременно няколко класа, като за да превключи съответните им визуализиращи дървета е необходимо да фокусира и натисне ляв бутон на мишката на съответния таб, наименован с името на желания клас:



Фигура 4.8: Структура на Class View

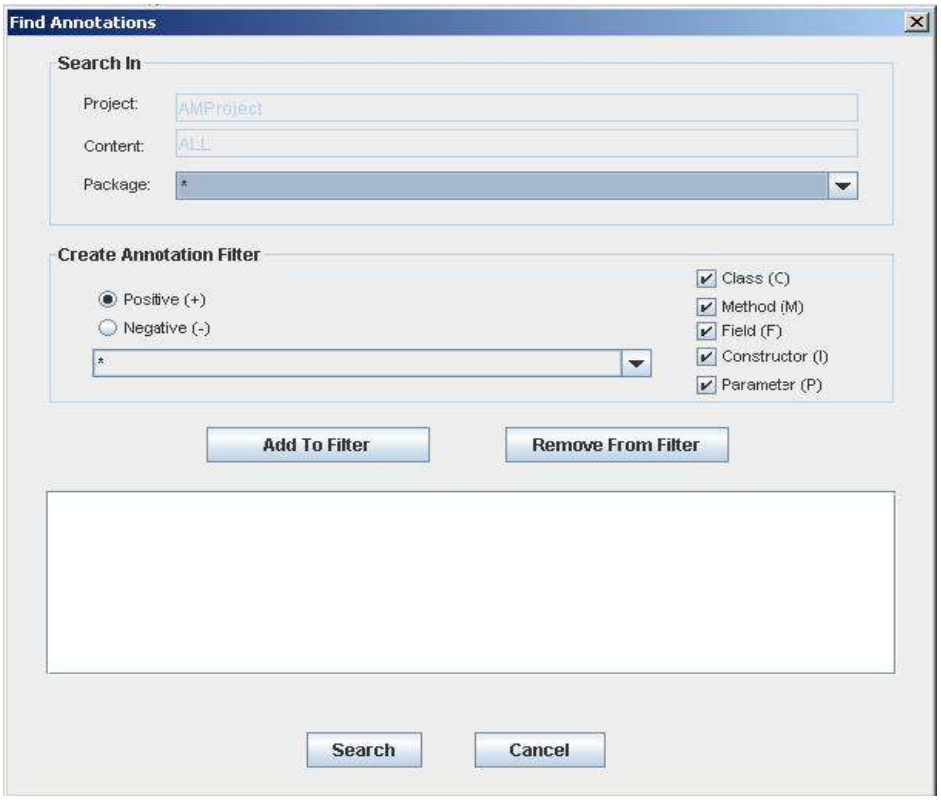
Структурата на визуалното дърво за метаданните на клас е следната:

- Корена на дървото е името на класа (заедно с пакета).
- Елементите, които са наследници на корена представляват групиране на елементите на класа - “Annotations”, “Methods”, “Fields” и “Constructors”.
- Наследниците на “Annotations” представляват анотациите на елемента, който е родител на “Annotations”. Родителите могат да бъдат елементите представляващи клас, метод, поле, конструктор или параметър на метод.
- Наследниците на “Methods” представляват сигнатурата на методите на класа (име и параметри).
- Наследниците на “Fields” представляват сигнатурата на полетата на класа (име и тип).
- Наследниците на “Constructors” представляват сигнатурата на конструкторите на класа (име и параметри).
- Всеки елемент представящ конкретен метод или конструктор, може да има наследник “Parameters”.

- Всеки “Parameters” елемент съдържа един или няколко “Parameter<index>” елементи, представящи параметрите на метода, в случай, че са анотирани. Стойността на <index> се определя от позицията на аргумента в сигнатурата на метода / конструктора.
- Всеки “Parameter” елемент съдържа “Annotations” елемент, представящ анотациите на съответния параметър.

4.8. Търсене на анотации по определени критерии – Find Annotations Dialog

Чрез този графичен компонент се реализира **Функция 7**. След като потребителят е заредил необходимите проекти в приложението, той може да пристъпи към търсене на анотации. Това е една от основните функции на приложението, с чиято помощ потребителят може да търси анотирани класове (или техни елементи), отговарящи на определени критерии. Търсенето може да е в множеството от всички заредени проекти, само в даден проект или в произволна част от проект (конкретен входен файл, Java пакет, Java клас). Както видяхме в Точка 4.6, търсенето може да бъде стартирано в определен възел на Projects View, чрез контекстно меню. Ако в конкретния възел и неговите наследници не се съдържат анотации, то системата уведомява за това. В противен случай, се визуализира диалога за търсене на анотации по зададени критерии - **Find Annotations Dialog** (вж. Фигура 4.9). В горната част на този диалог се визуализират името на проекта и content файла, в който ще бъде осъществено търсено. Те служат за информация и отговарят на пътя до възела, от който е извикан диалоговия прозорец. Съществуват два частни случая - когато търсенето е в корена на дървото или в конкретен проект, тогава съответно в полетата Project и Content на диалога ще се визуализира “*”, “*” или “<project-name>”, “*”.



Фигура 4.9: Диалогов прозорец Find Annotations

Следва падащото меню “Package”, което поддържа списък с наличните пакети в контекста на избрания проект и content. То дава възможност търсенето да бъде ограничено до определен пакет (намиращ се в съответните проект и content). Списъкът с пакетите е неактивен, ако текущия възел, от който е стартиран диалоговия прозорец е пакетен възел. Тогава търсенето ще бъде точно в този пакет.

В секцията **Create Annotation Filter** се съставят критериите за търсене, наречени филтър. Всеки филтър съдържа информация за тип на търсене, целева анотация и ниво на търсене. Целевата анотация може да бъде специфицирана от списъка с наличните анотации. Ако от списъка бъде избран “*”, то всички налични анотации в списъка стават целеви. Типа на търсене дефинира филтъра като позитивен и негативен, което определя наличието или отсъствието в изходното множество, на класа съдържащ целевата анотация от филтъра. Нивото на търсене редуцира изходното множество от класове до такива, в

които целевата анотация е прикрепена към елементи на тези класове от съответното ниво.

Пример 1: Параметри:

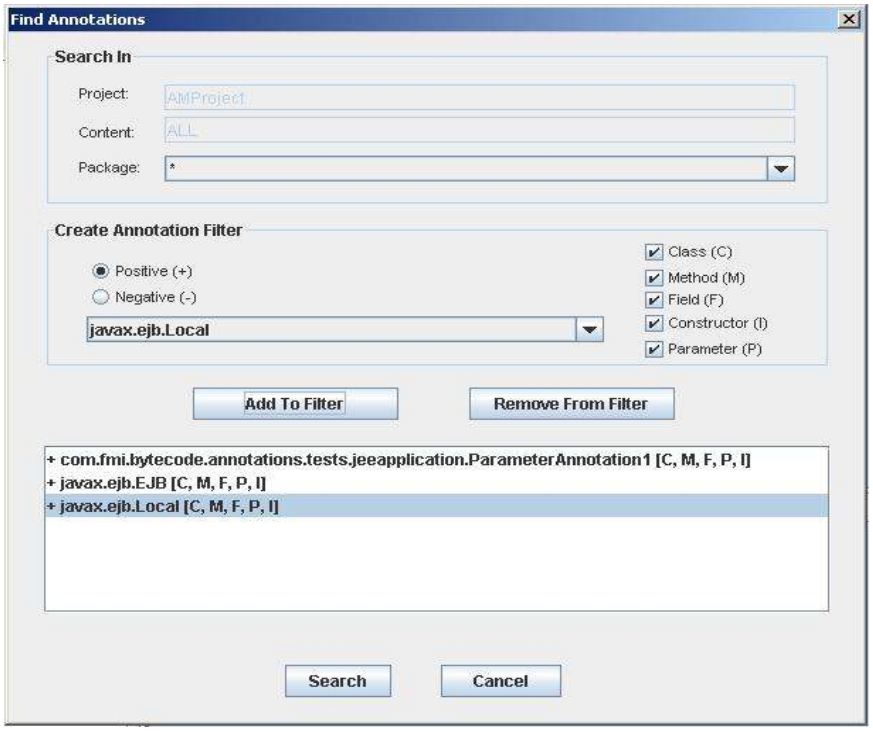
Позитивен (Отрицателен) филтър

Целева анотация – javax.ejb.EJB

Ниво на търсене – метод, поле

В резултат, ще бъдат визуализирани всички класове, в които целевата анотация – javax.ejb.EJB е (не е) прикрепена към метод(и) и/или поле(та) на тези класове.

След като даден филтър бъде изграден, той може да бъде добавен чрез бутона **Add To Filter** към общото множество от филтри – главен филтър. Ако даден филтър се сметне за неуместен, то той може да бъде изваден от главния филтър (наличното множество от филтри). Това става като бъде маркиран и след това отстранен с бутона **Remove From Filter**.

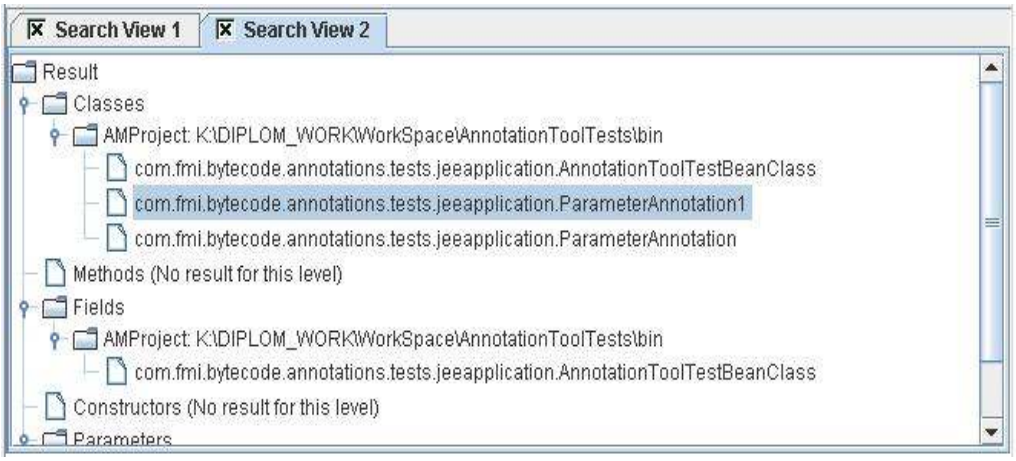


Фигура 4.10: Диалогов прозорец Find Annotations с добавени филтри

Заявката за търсене се образува от текущото сформирано множество от филтри в главния филтър, като се натисне бутона **Search**. След приключване на търсенето, резултатът от него се визуализира в Search View.

4.9. Контекстни менюта и структура на Search View

Графичният компонент Search View служи за визуализация на резултатите от търсене на анотации. За целта ще бъде използвано графично дърво, тъй като структурата на резултата е йерархична. Search View е реализирано с помощта на класа javax.swing.TabbedPane, с чиято помощ потребителя може да отвори едновременно няколко резултата от търсене, като за да превключи съответните им визуализиращи дървета е необходимо да фокусира и натисне ляв бутон на мишката на съответния таб, наименован с автоматично генерираното име на желания резултат:



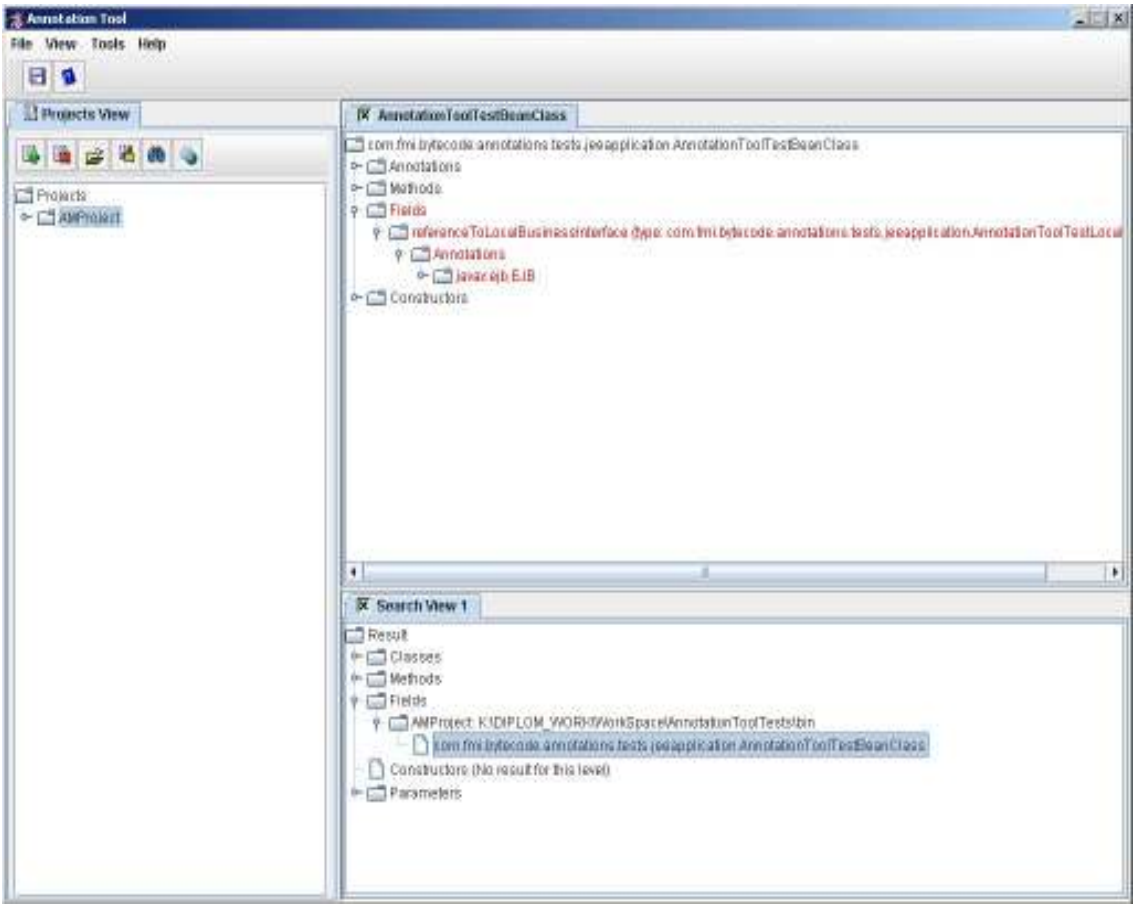
Фигура 4.8: Структура на Search View

Структурата на визуалното дърво за визуализация на резултата от търсене е следната:

- Корена на дървото е възел с име “Result”.
- Елементите, наследници на корена групират резултата от търсене по т.нар. нива на търсене (вж. **Точка 4.8**). Те биват 5 вида – “Classes”, “Methods”, “Fields”, “Constructors” и “Parameters”.

- Наследниците на нивата на търсене, групират намерените класове, отговарящи на критериите за търсене, по проекта и content файла, в който се намират. Те имат за име името на проекта и името на content файла.
- Следват елементите представящи намерените класове от съответните нива, намиращи се в съответния проект и content файл. Те имат за име името на класа (заедно с пакета).

За улеснение на потребителя, когато натисне два пъти ляв бутон върху някой клас в Search View, визуализацията на този клас в Class View е специфична – визуалното дърво на класа се отваря до елемента на класа от съответното ниво на анотиране и пътят до този елемент се оцветява в червено:



Фигура 4.9: Оцветяване на резултата от търсене в Class View

4.9.1. Контекстно меню на Search View

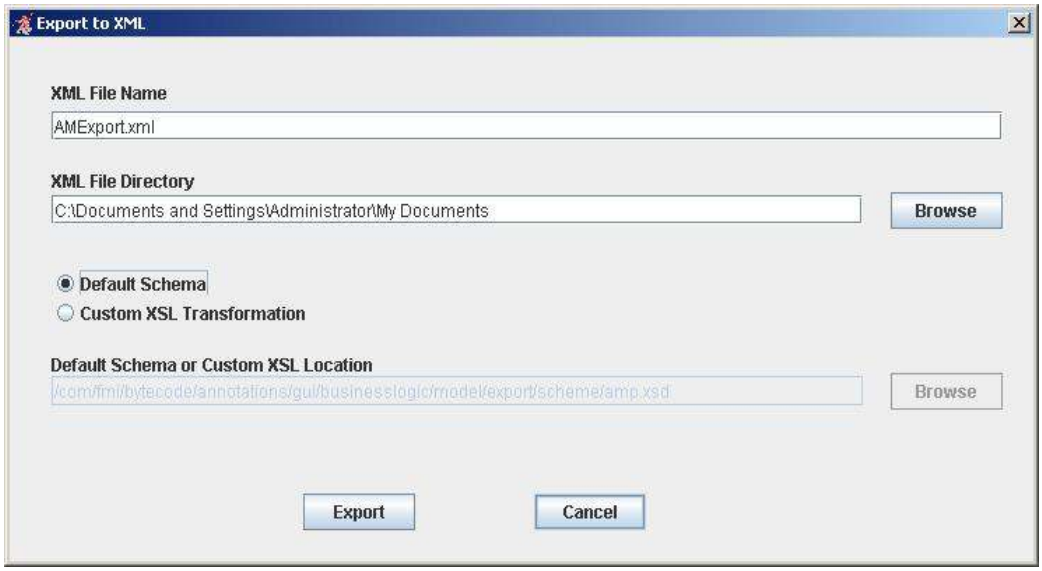
Всеки елемент, от визуалното дърво на Search View, има контекстно меню, даващо възможност за запис на резултата от търсене в XML формат. Менюто се визуализира чрез натискане на десен бутон на мишката върху произволен елемент.

Контекстно меню на елементите във “ Search View”:

Реализира **Функция 8** описани в **Точка 4.1**.

4.10. Записване на метаданни в XML формат – Export to XML Dialog

Метаданните за класовете, които се съдържат пряко или косвено в произволен възел на **Projects View** или **Search View**, могат да бъдат записани като XML файл т.нар. **Функция 8**. За целта се използва диалогов прозорец **Export to XML**:



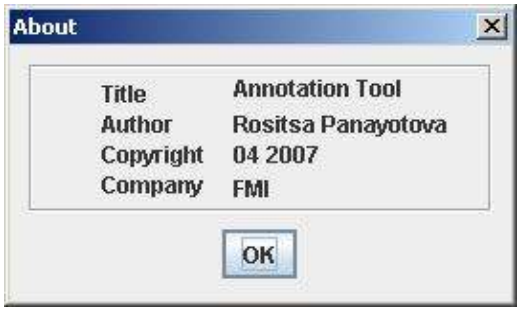
Фигура 4.10: Export to XML Dialog

Потребителят специфицира името на XML файла и директорията, в която ще бъде записан.

XML файла се конструира като се използва схемата по подразбиране (**amp.xsd**), която е част от реализацията. Диалога предоставя и възможност на потребителя да приложи XSLT трансформация върху XML файла, създаден съгласно схемата по подразбиране, чрез собствен XSL документ.

4.11. Визуализация на данни за автора на приложението – About Dialog

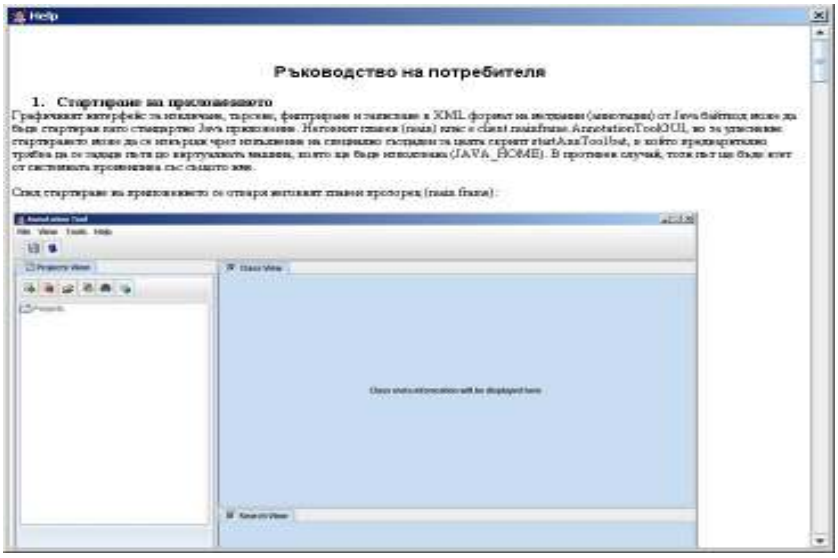
Това е диалога, който визуализира информация за автора на приложението – реализация на **Функция 9**:



Фигура 4.11: About Dialog

4.12. Визуализация на ръководство за потребителя – Help Dialog

Това е диалога, който визуализира ръководството за потребителя – реализация на **Функция 10**:



Фигура 4.12: Help Dialog

5. Описание на реализацията

В тази глава е описана реализацията на проекта описан в предходната глава (Проектиране на реализацията). Тя следва принципите на Обектно Ориентираното Програмиране (ООП), като се използват най-новите шаблони за програмиране и структури данни вградени в езика Java [9], [12], [13].

Съдържанието на главата е организирано по Java пакети на реализираните класове, с цел изложението да върви последователно по отделните логически и функционални компоненти на разработката.

5.1. Архитектура на системата

В тази глава е представена цялостната архитектура на системата (Фигура 5.1). Тя е изградена от 3 основни групи модули:

- Графични компоненти (вж. **Точка 5.2**) – входната точка на приложението. Потребителят взаимодейства с тях с помощта на мишката и клавиатурата. В тази група се причисляват всички графични компоненти – прозорец, диалог, графично дърво, стандартни и контекстни менюта, бутони, табове, текстови полета и т.н. Тези компоненти се намират в подпакети на главния пакет съдържащ графичните компоненти - com.fmi.bytecode.annotations.gui.graphical.*.
- Слушатели на събития (вж. **Точка 5.6**) – междинен слой, вътрешен за системата, имащ за цел да приема и обработва всички събития предизвикани от потребителя (натискане на бутон на мишката върху определен графичен компонент, натискане на специална комбинация от клавиши и т.н.). Към тази група модули спадат всички класове от разработката, които са реализация на стандартните за Java/Swing интерфейси - java.awt.event.ActionListener, java.awt.event.KeyListener и java.awt.event.MouseListener. Местоположението на тези компоненти е пакета com.fmi.bytecode.annotations.gui.businesslogic.actions.*.
- Функции (вж. **Точка 5.7**) – реализацията на всички функции, които предлага разработката, описани в **Точка 4.1**. Тези функции се стартират от слушателите на събития при настъпването на определено събитие инициирано от потребителя. Реализацията на функциите е в подпа-

кетите на пакета com.fmi.bytecode.annotations.gui.businesslogic.* и от части в пакета с помощни класове - com.fmi.bytecode.annotations.gui.utils.*.

- Контекст от данни (вж. **Точка 5.4**) – модели на данните и структури данни, които се предават при комуникация между трите изброени групи модули – Графични модули, Слушатели на събития и Функции.

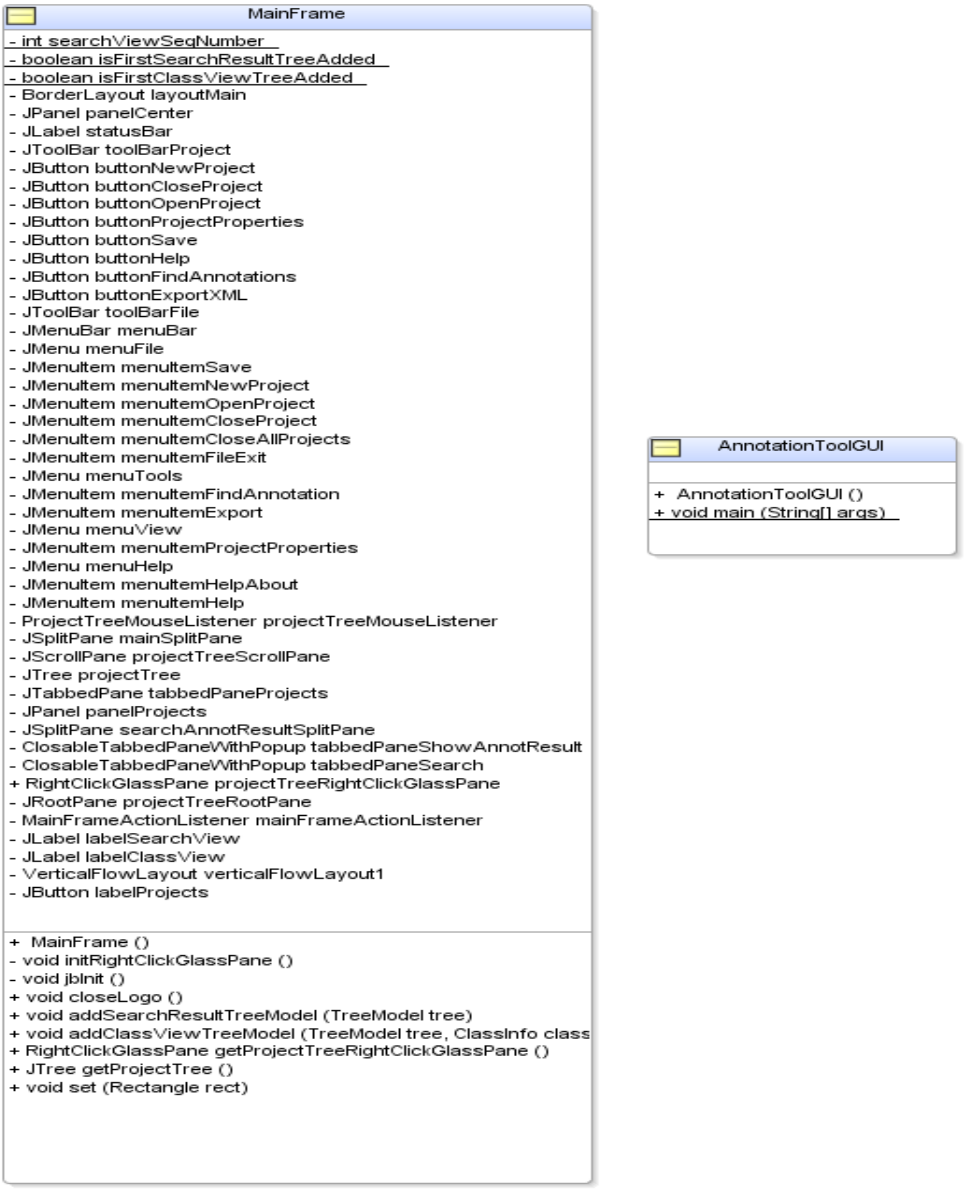
Фигура. 5.1: Архитектура на системата

*5.2. Пакет com.fmi.bytecode.annotations.gui.graphical.**

В този пакет са разположени реализациите на графичните компоненти описани в **Точка 4** – главният прозорец на приложението, диалоговите прозорци за зареждане на проект, създаване на проект, търсене на анотации и записване на метаданните в XML формат:

5.2.1. Клас MainFrame и AnnotationToolGUI

Класът AnnotationToolGUI е т.нар. main клас и с негова помощ може да бъде стартирано приложението. Класът MainFrame представлява главния прозорец на приложението, описан в **Точка 4.2**. Той наследява стандартния клас в Java/Swing - javax.swing.JFrame. В този клас се съдържат инициализациите на ProjectView, SearchView и ClassView, като за целта се използва специално реализирания клас ClosableTabbedPaneWithPopup.



Диаграма. 5.1: MainFrame

5.2.2. Клас ClosableTabbedPaneWithPopup

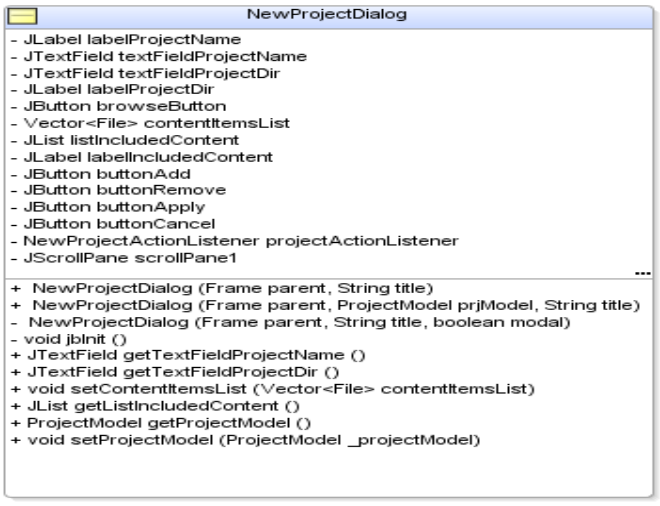
Този клас разширява функциите на стандартния в Swing клас за табулации - javax.swing.JTabbedPane, като добавя функционалност за затваряне на отделните табове (т.нар. X икона на всеки таб), а също така и функция за затваряне на всички табове едновременно, чрез контекстно меню, вж. **Приложение 9.1.2**. Използва се като панел, в който са разположени визуалните дървета на Class View и Search View.

5.2.3. Клас RightClickGlassPane

Този клас е реализацията на използваното в разработката контекстно меню вж. **Приложение 9.1.1**. Използва се при реализацията на менютата на визуалните дървета в Search View и Projects View, както и при реализацията на класа ClosableTabbedPaneWithPopup.

5.2.4. Клас NewProjectDialog и ProjectPropertiesDialog

Класът NewProjectDialog е диалога предназначен за създаване на нов проект, описан в **Точка 4.3**. За целта съдържа текстови полета, бутони, надписи и визуален списък (javax.swing.JList) за композиция на content файловете на новия проект. Той наследява стандартния клас в Java/Swing - javax.swing.JDialog. Същия клас се използва за реализация на ProjectPropertiesDialog, описан в **Точка 4.5**.



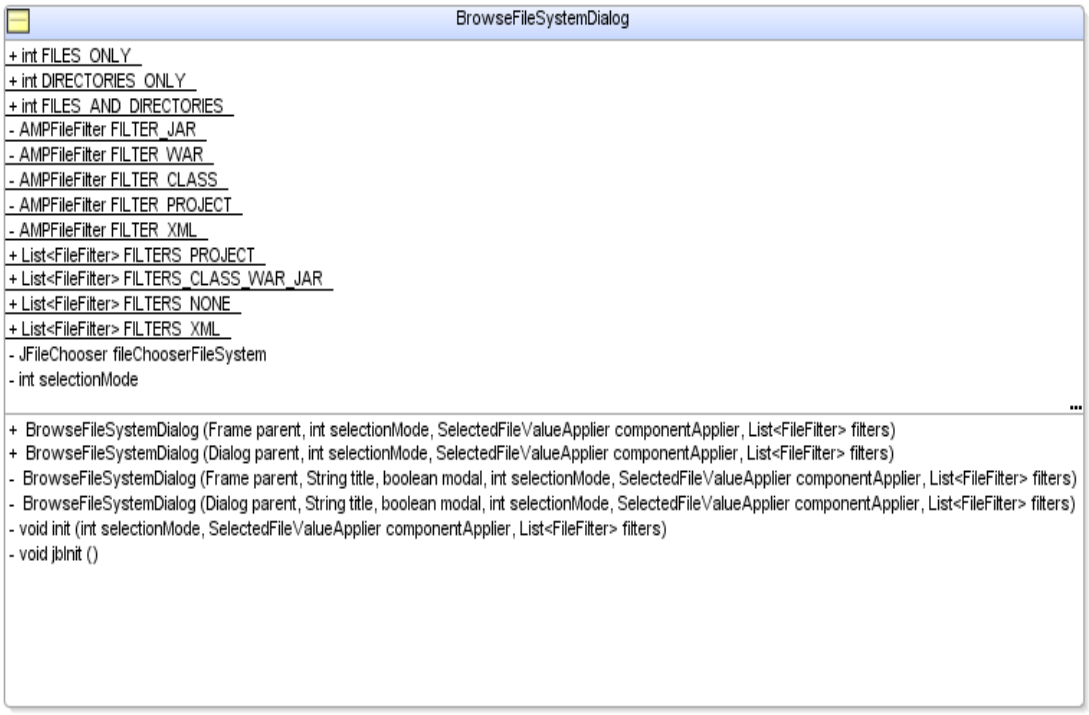
Диаграма. 5.2: NewProjectDialog

5.2.5. Клас BrowseFileSystemDialog

Класът BrowseFileSystemDialog е диалога предназначен за навигиране във файловата система. Той има широка употреба в приложението, а именно използван е при реализацията на:

- LoadProjectDialog за зареждане на проект, описан в **Точка 4.4**.
- NewProjectDialog за избиране на content файловете и директорията, в която ще бъде записан новия проект.
- ExportToXMLDialog за избиране на потребителски XSL файл и директория, в която ще бъде записан резултатния XML файл.

Той наследява стандартния клас javax.swing.JDialog и съдържа в себе си инстанция на стандартния клас javax.swing.JfileChooser, като разширява функциите му.



Диаграма. 5.3: BrowseFileSystemDialog

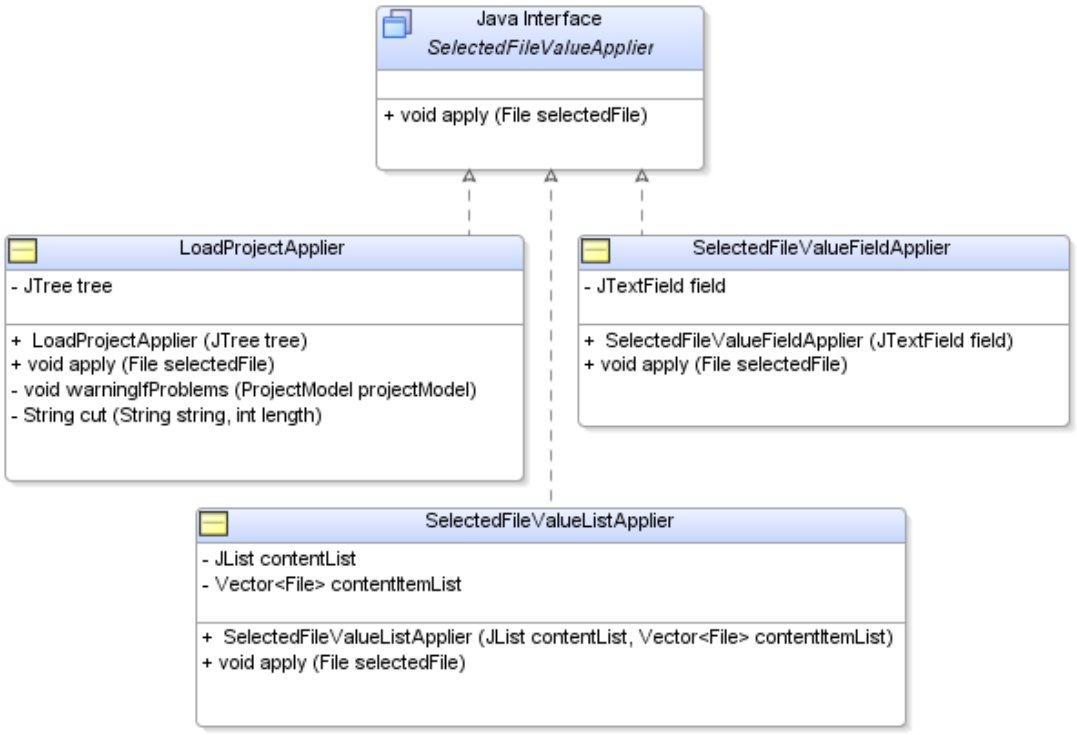
Разширените функции се изразяват в това, че конкретното действие, което трябва да се извърши, когато потребителят избере файл и натисне бутона

Open, са отделени от реализацията, с помощта на интерфейса SelectedFileValueApplier.

5.2.6. Интерфейс SelectedFileValueApplier и реализациите му

Реализациите на интерфейса SelectedFileValueApplier се използват при създаването на BrowseFileSystemDialog. Интерфейсът има един абстрактен метод apply(File selectedFile), в който всеки от реализиращите класове трябва да кодира действието, което трябва да се извърши, когато файла в BrowseFileSystemDialog е избран от потребителят.

```
public interface SelectedFileValueApplier {
    public void apply(File selectedFile) throws Exception;
}
```



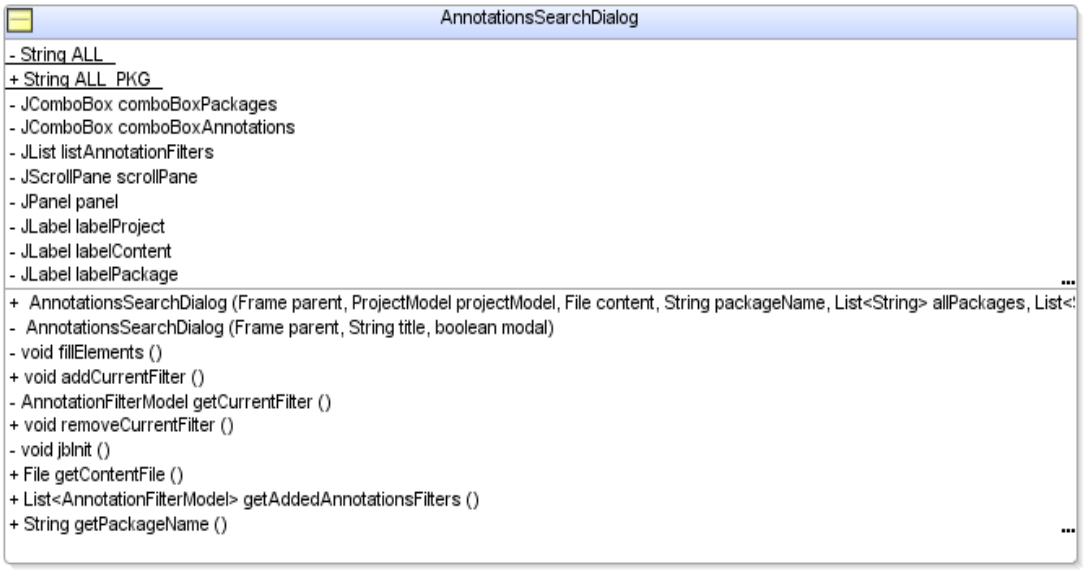
Диаграма. 5.4: SelectedFileValueApplier

Реализациите на интерфейса са общо 3 - LoadProjectApplier, SelectedFileValueFieldApplier и SelectedFileValueListApplier.

- Класа LoadProjectApplier, кодира логиката, която зарежда проект от LoadProejctDialog в ProjectsView, описан в **Точка 4.4**.
- Класа SelectedFileValueFieldApplier, отразява пътят до избрания файл (директория) в текстовите полета на графичните компоненти (NewProjectDialog, ExportToXMLDialog и т.н.).
- Класа SelectedFileValueListApplier, добавя елементи в списъка с content файлове в NewProjectDialog, когато потребителят добавя файлове или директории, с помощта на бутона **Browse**.

5.2.7. Клас AnnotationsSearchDialog

Класът AnnotationsSearchDialog е диалога предназначен за търсене на анотации, описан в **Точка 4.8**. За целта съдържа текстови полета, радио бутони, отметки, падащи менюта, надписи и визуален списък (javax.swing.JList) за композиция на филтрите за търсене. Той наследява стандартния клас в Java/Swing - javax.swing.JDialog.

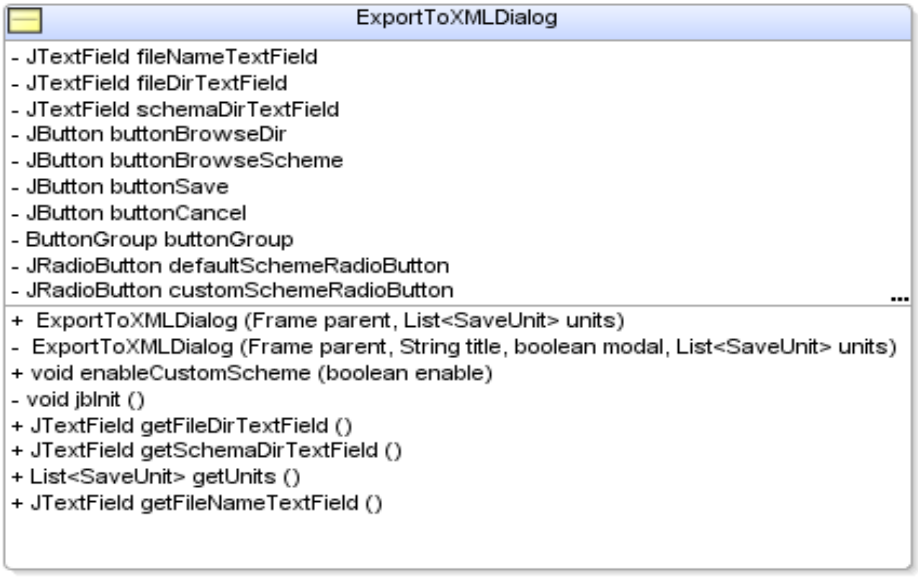


Диаграма. 5.5: AnnotationsSearchDialog

5.2.8. Клас ExportToXMLDialog

Класът ExportToXMLDialog е диалога предназначен за запис на метаданни в XML формат, описан в **Точка 4.10**. За целта съдържа текстови полета, бутони

и радио бутони за избиране на местоположението и името на XML файла, в който ще бъдат записани метаданните. Той наследява стандартния клас в Java/Swing - javax.swing.JDialog.



Диаграма. 5.6: ExportToXMLDialog

5.2.9. Клас AboutDialog

Класът AboutDialog е диалога предназначен за визуализация на информация за автора на проекта, описан в **Точка 4.11**. Той наследява стандартния клас в Java/Swing - javax.swing.JDialog. Съставен е от статични текстови полета, с чиято помощ се визуализира необходимата информация.

5.2.10. Клас HelpDialog

Класът HelpDialog е диалога предназначен за визуализация на ръководство на потребителя, описан в **Точка 4.12**. Той също наследява стандартния клас в Java/Swing - javax.swing.JDialog. За визуализация на ръководството на потребителя, което се намира под формата на HTML в /com/fmi/bytecode/annotations/gui/graphical/dialogs/help/UserGuide.htm, се използва инстанция на стандартния клас javax.swing.JEditorPane.

*5.3. Пакет com.fmi.bytecode.annotations.gui.businesslogic.**

В този пакет е разположена бизнес логиката на приложението – моделите на данните, изразяващи вътрешното им представяне, структурите от данни, с които си служи приложението, както и реализация на действията, които трябва да се предприемат при настъпването на определено събитие, инициирано от потребителя. Пакета е изграден от три под пакета:

- Пакета com.fmi.bytecode.annotations.gui.businesslogic.model.* - моделите на данните, с които работи приложението. В този пакет са реализирани модела на проект, модела на филтрите за търсене и модела на данните, които се записват под формата на XML.
- Пакета com.fmi.bytecode.annotations.gui.businesslogic.treenodes.* - структурите от данни, с които си служи приложението. Използват се предимно за реализацията на визуалните дървета, като по този начин всеки елемент на дървото съдържа контекстните данни, които са необходими за предприемане на бъдещите действия, инициирани от потребителят върху елемента.
- Пакета com.fmi.bytecode.annotations.gui.businesslogic.actions.* - реализация на действията, свързани с определено събитие.

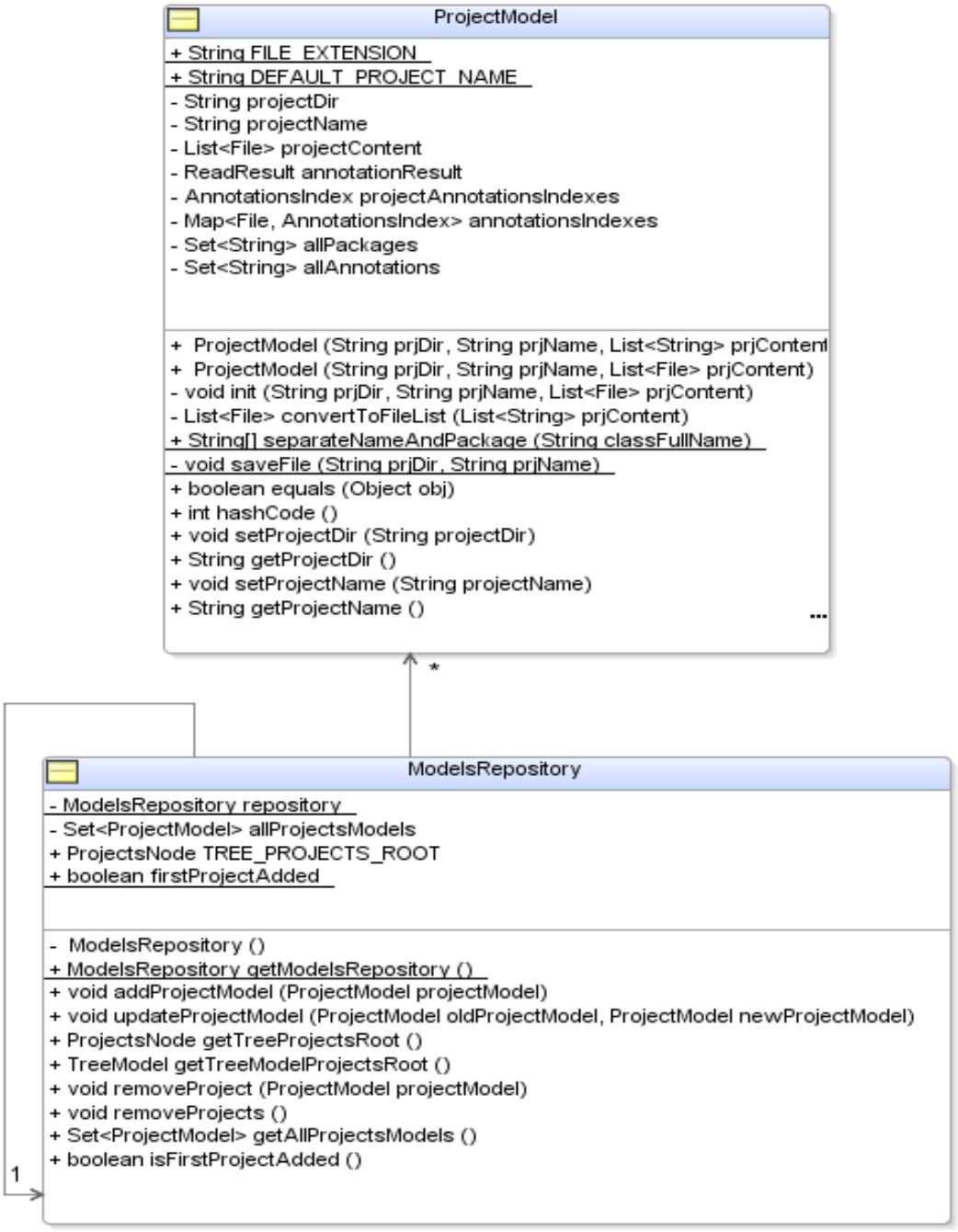
Следва подробно описание на реализираните класове в тези три пакета.

*5.4. Пакет com.fmi.bytecode.annotations.gui.businesslogic.model.**

Тук се намира реализацията на моделите, създадени с цел капсулирането на данните, с които работи приложението. Това са моделите на проект, данни за export и филтрите за търсене. Причината за съществуване на тези модели е нуждата от подаването и съхранението на данните, които те съдържат, между множеството компоненти. Капсулирането на тези данни в модел осигурява интуитивното програмиране на приложението с минимизиран риск от грешки. В моделите е кодирана и логиката за валидация на коректността на модела, вместо тя да бъде писана на повече от едно място в кода. Ако модела не може да бъде създаден, понеже подадените данни са некоректни, се хвърля съответния ModelException.

5.4.1. Класове ProjectModel и ModelsRepository

Класът ProjectModel моделира потребителски проект, създаден чрез приложението. Съдържа името на проекта, директорията, в която ще бъде записан и списък от content файловете му.



Диаграма. 5.7: ProjectModel и ModelsRepository

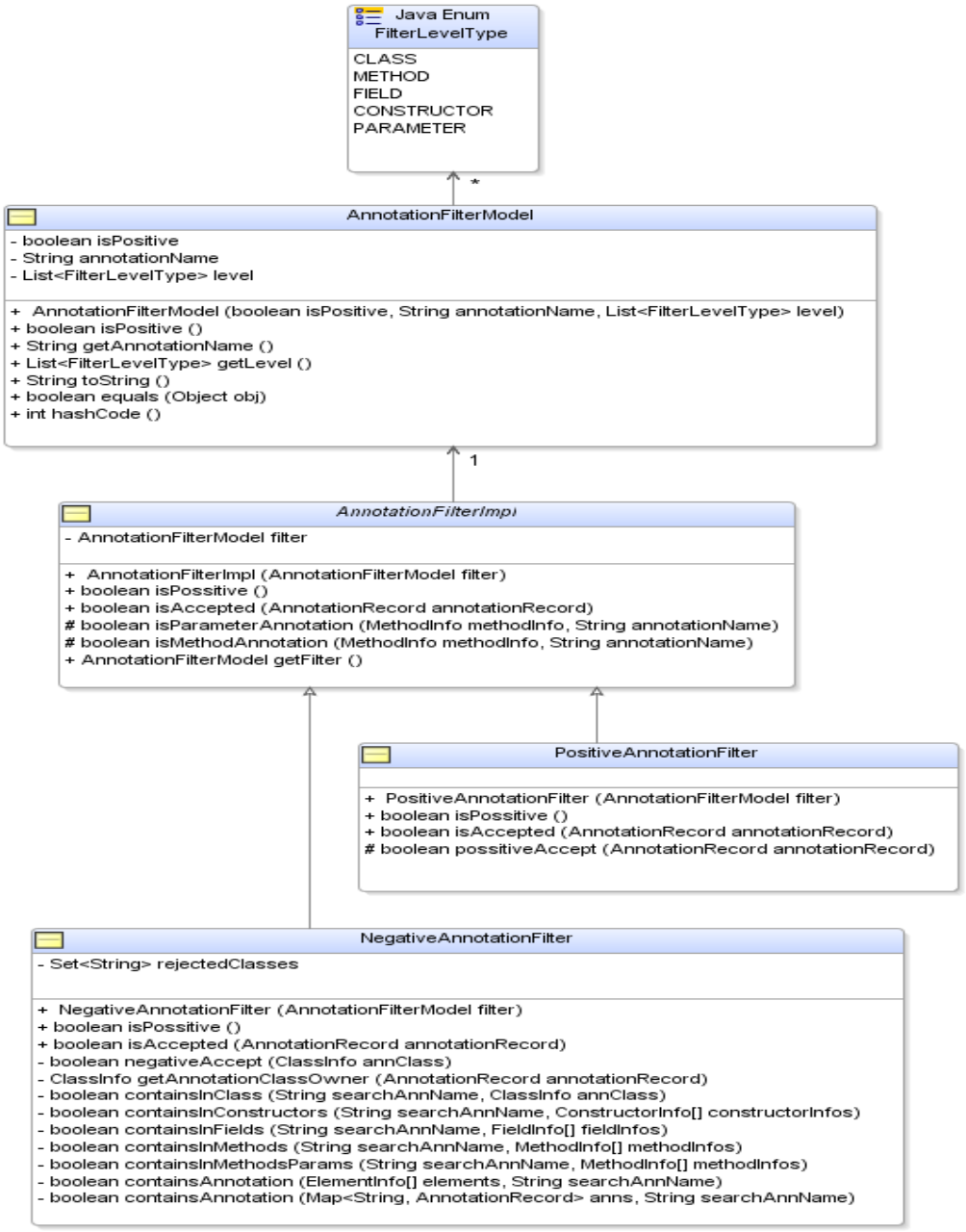
Обекти на този клас се инстанцират при зареждане или създаване на проект. Ако, при извикване на конструктора на ProjectModel, данните, които са подадени са некоректни, се хвърля ModelException. Това изключение се прихваща от графичният компонент за създаване на проект и на екрана се извежда подходящо съобщение за грешка, което сигнализира на потребителя, че е необходима промяна на параметрите, с които е направен опит за създаване на проект.

Класът ModelsRepository има за цел да съхранява моделите на всички коректно заредени проекти. Той реализира т.нар. Singleton Pattern и съответно може да бъде достъпен, на произволно място в приложението, по статичен начин – ModelsRepository.getModelsRepository(). Както се вижда от **Диаграма 5.7**, той предлага следните методи за работа с модела на проект:

- public void addProjectModel(ProjectModel projectModel) – Добавя проект. Използва се за реализация на **Функция 1** и **Функция 2**.
- public void updateProjectModel(ProjectModel oldProjectModel, ProjectModel newProjectModel) throws ModelException – Подменя модела на проект. Използва се при промяна на настройките на проект, с помощта на ProjectPropertiesDialog. Използва се за реализация на **Функция 3**.
- public void removeProject(ProjectModel projectModel) – изтрива проект. Използва се при затваряне на проект. Използва се за реализация на **Функция 4**.
- public Set<ProjectModel> getAllProjectsModels() – връща всички заредени проекти. Използва се за конструиране на Projects View.
- public void removeProjects() – изтрива всички проекти. Използва се при командата за затваряне на всички проекти. Използва се за реализация на **Функция 5**.

5.4.2. Класове, реализиращи модела на филтрите за търсене – FilterLevelType, AnnotationFilterModel, AnnotationFilterImpl, NegativeAnnotationFilter и PositiveAnnotationFilter

Класът AnnotationFilterModel моделира единичен филтър за търсене на анотации.



Диаграма. 5.8: ProjectModel и ModelsRepository

Класът FilterLevelType е от тип enum. Той моделира нивата за търсене на анотации. Съдържа се като поле на AnnotationFilterModel и определя нивото на търсене на конкретната инстанция, към която принадлежи.

Абстрактния клас AnnotationFilterImpl е базова реализация на AnnotationFilterModel, която капсулира общата бизнес логика на позитивните и

негативните филтри (вж. **Приложение 9.1.3**). Той дефинира два абстрактни метода, които трябва да бъдат реализирани в наследниците му:

- public abstract boolean isPossitive() – Флаг, показващ дали филтъра е позитивен или негативен.
- public abstract boolean isAccepted(AnnotationRecord annotationRecord) – метод, определящ дали подадената, като аргумент, анотация отговаря на критериите на филтъра.

Класовете NegativeAnnotationFilter и PositiveAnnotationFilter наследяват AnnotationFilterImpl и реализират съответно негативен и позитивен филтър за търсене на анотации. В тях е и специфичната реализация на абстрактните методи от родителския клас (вж. **Приложение 9.1.4** и **Приложение 9.1.5**).

5.4.3. Класове, реализиращи модела за записване на метаданните като XML файл – ExportModel и SaveUnit

Класът SaveUnit моделира определената част от данните, които ще бъдат записани под формата на XML файл, които принадлежат на даден проект и негов content файл.

Диаграма. 5.9: SaveUnit и ExportModel

Той съдържа модела на проекта, File обект, представящ content файла, и списък от ClassInfo обекти, които моделират множеството от класове, които ще бъдат записани, чрез съдържащия ги SaveUnit обект.

Класът ExportModel моделира данните, които ще бъдат записани под формата на XML файл. Той съдържа списък от всички SaveUnit обекти, свързани с определено записване на метаданни в XML файл, предизвикано от потребителят. Като допълнителна информация се съдържат името и директорията на резултатния XML файл, а също и избрания от потребителят файл (ако има такъв) за специфична XSL трансформация на резултатния XML файл. Обекти на този клас се инстанцират при опит за записване на дадено множество от метаданни под формата на XML. Ако при извикване на конструктора на ExportModel, данните, които са подадени са некоректни, се хвърля ExportException. Това изключение се прихваща от графичният компонент ExportToXMLDialog и на екрана се извежда подходящо съобщение за грешка, което сигнализира на потребителя, че е необходима промяна на параметрите, с които е направен опит за записване и трансформация на резултата.

В пакета com.fmi.bytecode.annotations.gui.businesslogic.model.export.scheme е разположена схемата по подразбиране, съгласно, която се записват данните при конструиране на XML файла. Потребителят може да изисква допълнително прилагането на специфична XSL трансформация с външен *.xsl файл. Тази възможност се дава от диалога ExportToXMLDialog.

В пакета com.fmi.bytecode.annotations.gui.businesslogic.model.export.xsl, са разположени стандартни *.xsl файлове за трансформация, разработени заедно с приложението:

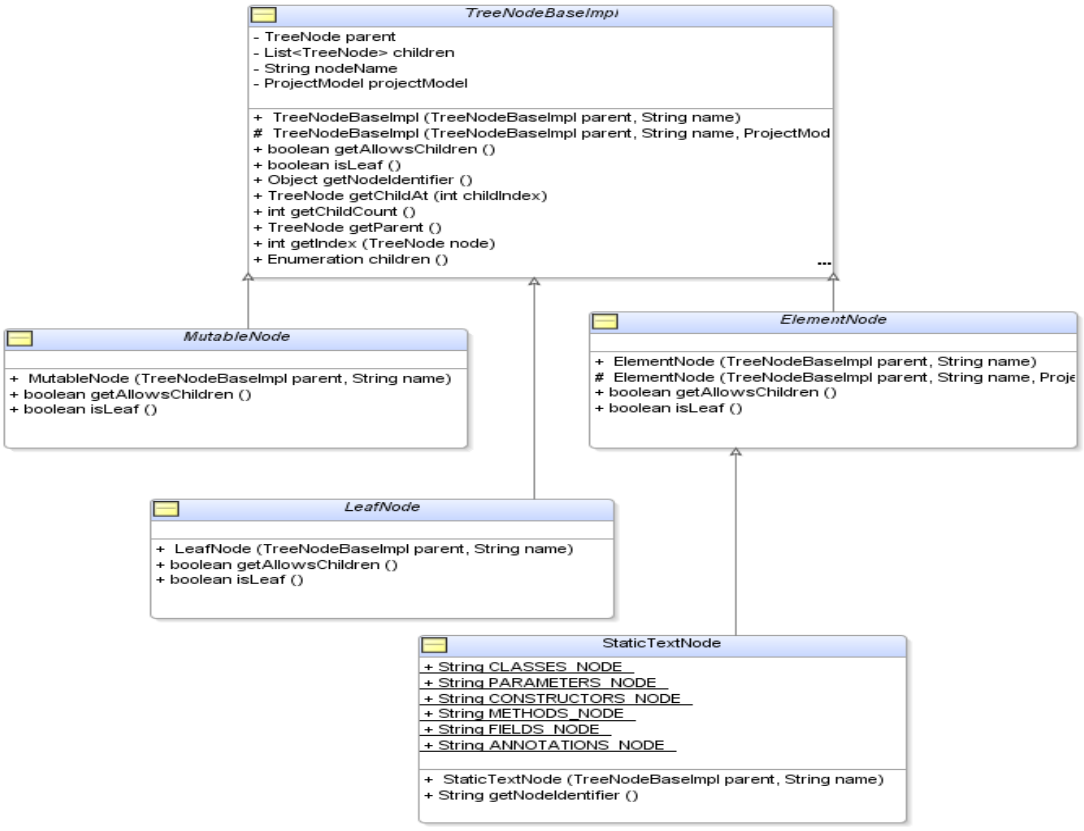
- **replaceTagNames.xsl** (вж. **Приложение 9.1.12**) – Дава възможност на потребителя да дефинира собствени имена на таговете, заграждащи данните, в резултатният XML файл.
- **extractAllAnnotationsAsRoots.xsl** (вж. **Приложение 9.1.13**) – Реорганизира резултатния XML файл, като корените на XML структурата стават анотациите на елементите. Сложна трансформация, програмният код, на която може да служи в помощ на потребителя, при изготвянето на собствени XSL файлове.

*5.5. Пакет com.fmi.bytecode.annotations.gui.businesslogic.treenodes.**

В този пакет се намират реализациите на елементите, които изграждат визуалните дървета в Projects View, Search View и Class View. Всички класове реализират базовия за Swing интерфейс javax.swing.tree.TreeNode. Всеки конкретен клас наследява някои от базовите (вж. **Диаграма. 5.9**) класове TreeNodeBaseImpl (вж. **Приложение 9.1.6**), ElementNode (вж. **Приложение 9.1.7**) или LeafNode (вж. **Приложение 9.1.8**) и съдържа специфична информация, в зависимост от семантиката, която има във визуалното дърво, на което принадлежи.

5.5.1. Базови класове TreeNodeBaseImpl, ElementNode, LeafNode и StaticTextNode

Тези класове са базови за цялостната реализация на елементите изграждащи трите визуални дървета - Projects View, Search View и Class View. Йерархията между тях се представя със следната диаграма:



Диаграма. 5.9: Базови елементи използвани във визуалните дървета на Projects View, Search View и Class View

- Класът TreeNodeBaseImpl е корена на цялата йерархия от общо 22 класа. Той е директна реализация на базовият за Swing интерфейс javax.swing.tree.TreeNode и освен наследените методи, предлага и допълнителни, които се използват за реализиране на целите на системата. Един от най-важните допълнителни методи е public abstract Object getNodeIdentifier(). С негова помощ, всеки елемент на визуалните дървета предоставя идентификатор, който го прави уникален за визуалния му родител (родителя във визуалното дърво, а не този в клас йерархията). Този идентификатор е обект представящ някой от полетата на съответния елемент (ProjectModel, File, String и т.н.).
- Базовите класове ElementNode, MutableNode и LeafNode служат за определяне на това, дали елемент е съответно - вътрешен за дървото, листо на дървото или с променлив характер (може да бъде както вътрешен така и листо на дървото). Реализираните методи в тези класове служат за връщане на стойност към визуалните дървета, които ги съдържат, като определят начина, по който трябва да бъдат визуализирани.
- Класът StaticTextNode е елемент, който съдържа текст и има групираща семантика, без да има конкретен смисъл в термините на моделите, с които си служи приложението. Такива елементи са корените на дърветата (с текст “Projects”, “Result”), както и групиращите (за ниво на търсене на анотация) елементи, които са директни наследници на корена на визуалните дървета в Search View.

5.5.2. Класове, реализиращи елементите на визуалното дърво в Project View

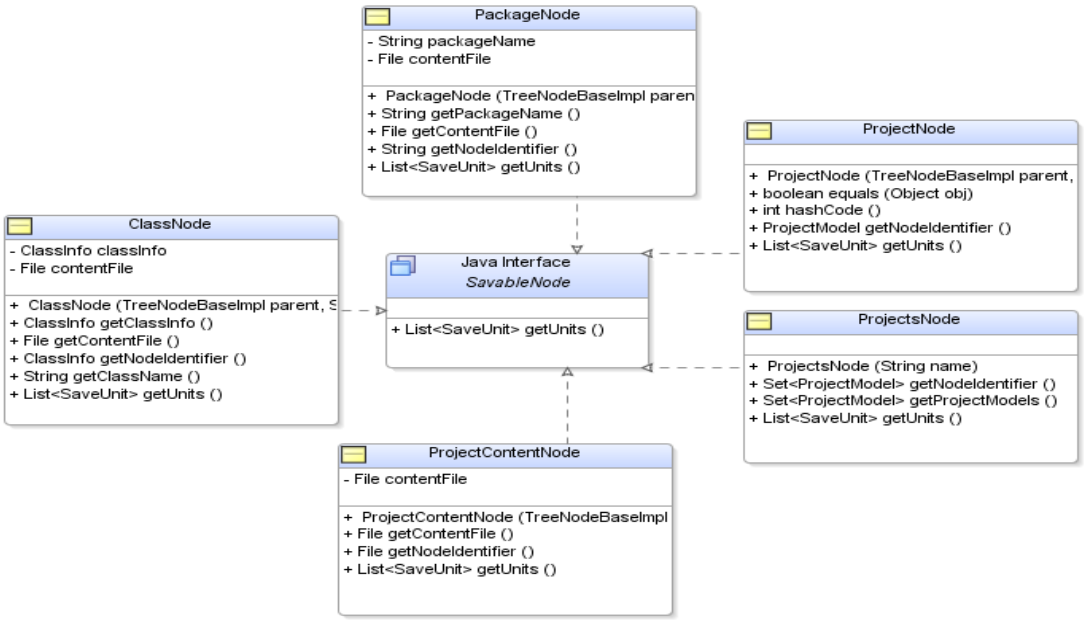
Разположени са в пакета com.fmi.bytecode.annotations.gui.businesslogic.treenodes.project и с тяхна помощ е реализирано визуалното дърво на Project View. Йерархичните зависимости между класовете са представени чрез следната диаграма:

Диаграма. 5.10: Елементи на визуалното дърво в Projects View

- Класът ProjectsNode представя корена на дървото. Съдържа моделите на всички проекти, които са заредени в Projects View.
- Класът ProjectNode представя конкретен проект и е наследник (визуален) на ProjectsNode. Съдържа модела на проекта, който представя.
- Класът ProjectContentNode представя конкретен content файл на проекта, към който принадлежи. Той е наследник (визуален) на ProjectNode. Съдържа модела на проекта, към който принадлежи и content файла, който представлява.
- Класът PackageNode представя конкретен Java пакет, чиито класове се намират в content файла, на който принадлежи. Освен модела на проекта и content файла, към който принадлежи, съдържа и името на подпакета, който представлява.

- Класът ClassNode (вж. **Приложение 9.1.9**) е листо на дървото и пряк наследник (визуален) на PackageNode. Той представя конкретен Java клас от пакета, на който принадлежи. Освен модела на проекта, content файла и пакета на класа, съдържа още името и ClassInfo обекта на класа, който представлява.

И петте изброени класа, чрез контекстни менюта, предлагат възможността за запис на метаданните в XML файл. За целта наследяват интерфейса SavableNode:



Диаграма. 5.11: Елементите на визуалното дърво в Projects View, реализиращи интерфейса SavableNode

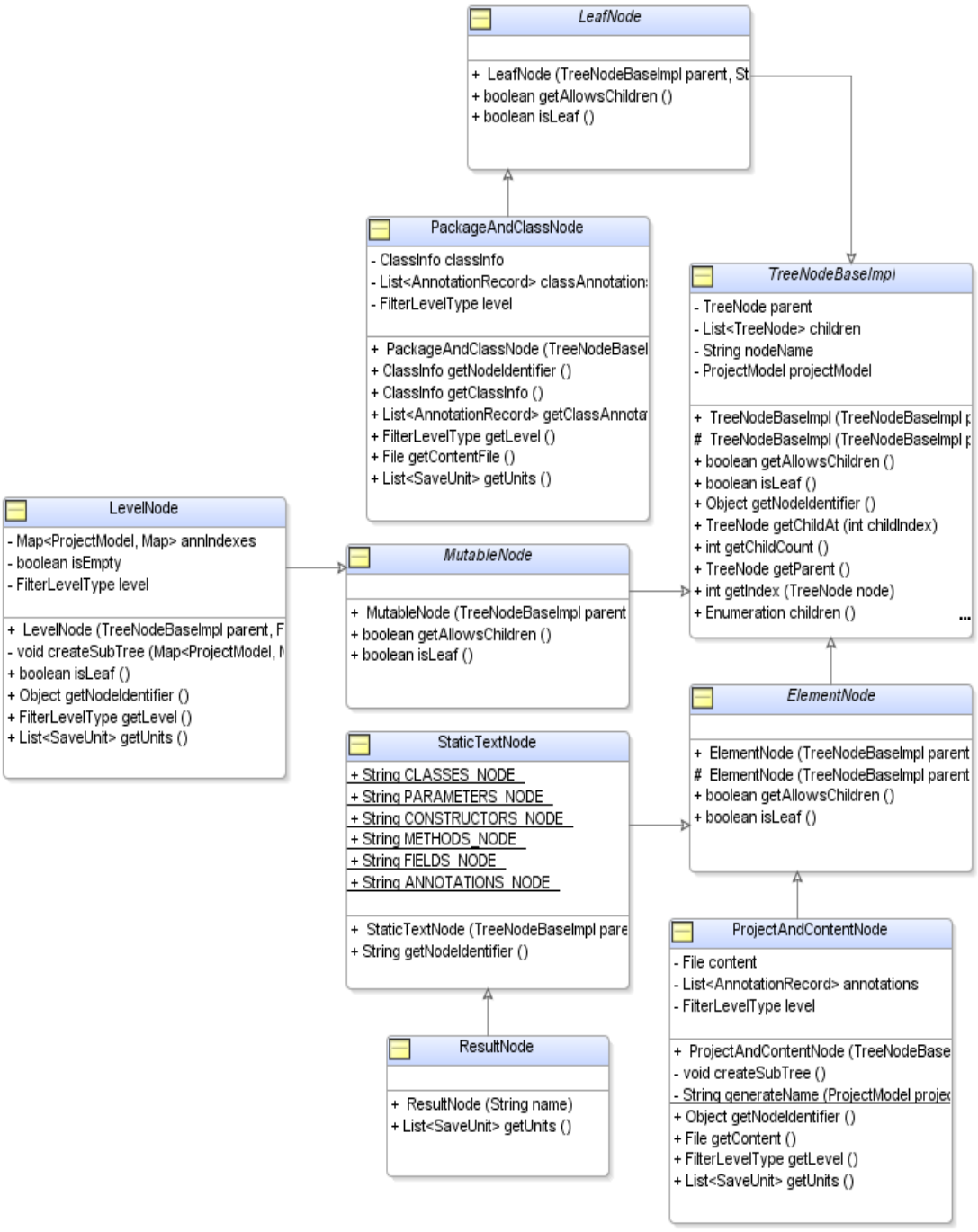
В сигнатурата на интерфейса е дефиниран метода getUnits(), който връща списък от всички SaveUnit обекти, свързани пряко или косвено с конкретния елемент (т.е. SaveUnit обекта на конкретния елемент, заедно с тези на всичките му наследници) :

```
piblic interface SavableNode {
   public List<SaveUnit> getUnits();
}
```

5.5.3. Класове, реализиращи елементите на визуалното дърво в Search View

Разположени са в пакета com.fmi.bytecode.annotations.gui.businesslogic.treenodes.searchresult и с тяхна помощ са реализирани визуалните дървета на Search View. Йерархичните зависимости между класовете са представени чрез **Диаграма. 5.12**, по долу. Освен изброените базови класове (TreeNodeBaseImpl вж. **Приложение 9.1.6**, ElementNode вж. **Приложение 9.1.7**, LeafNode (вж. **Приложение 9.1.8**), тук се появяват и базовите интерфейси MutableNode (вж. **Приложение 9.1.10**) и StaticTextNode (вж. **Приложение 9.1.11**).

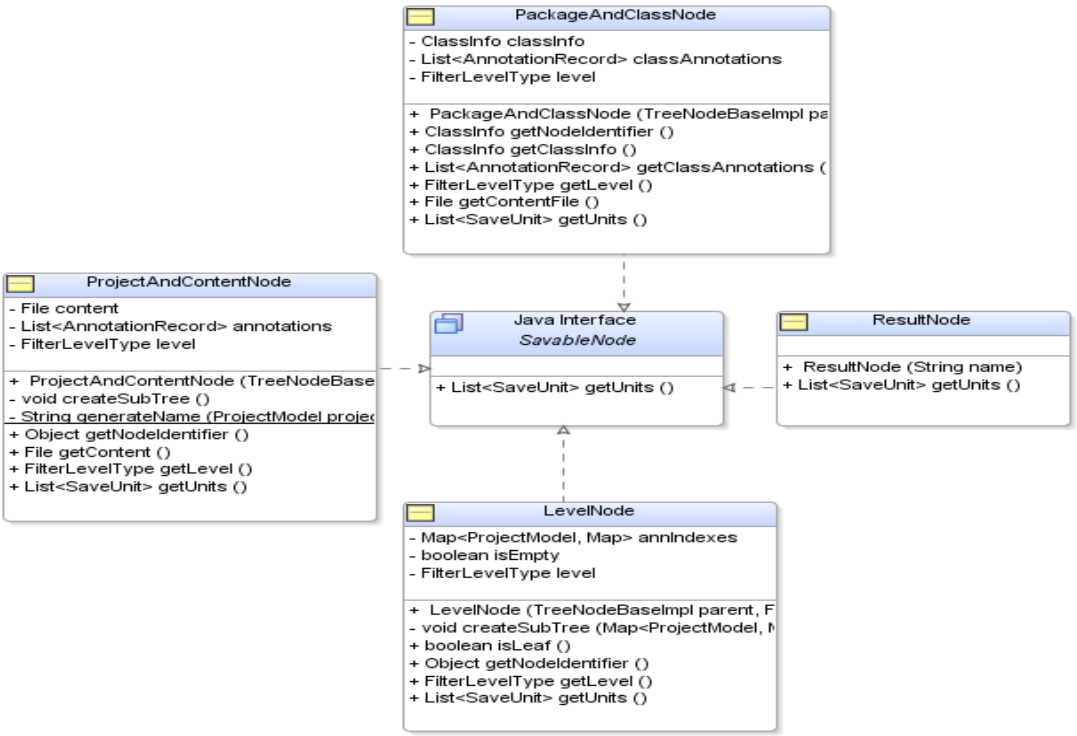
Всички класове на диаграмата, с изключение на базовите, чрез контекстни менюта, предлагат възможността за запис на метаданните в XML файл. За целта наследяват интерфейса SavableNode (вж. **Диаграма. 5.13**), аналогично на елементите във визуалното дърво на Projects View - интерфейсът SavableNode дефинира метода getUnits(), който връща списък от всички SaveUnit обекти, свързани пряко или косвено с конкретния елемент (т.е. SaveUnit обекта на конкретния елемент, заедно с тези на всичките му наследници):



Диаграма. 5.12: Елементи на визуалните дървета в Search View

- Класът ResultNode представя корена на дървото. Съдържа списък от всички SaveUnit обекти, намиращи се в кой да е от елементите на дървото.

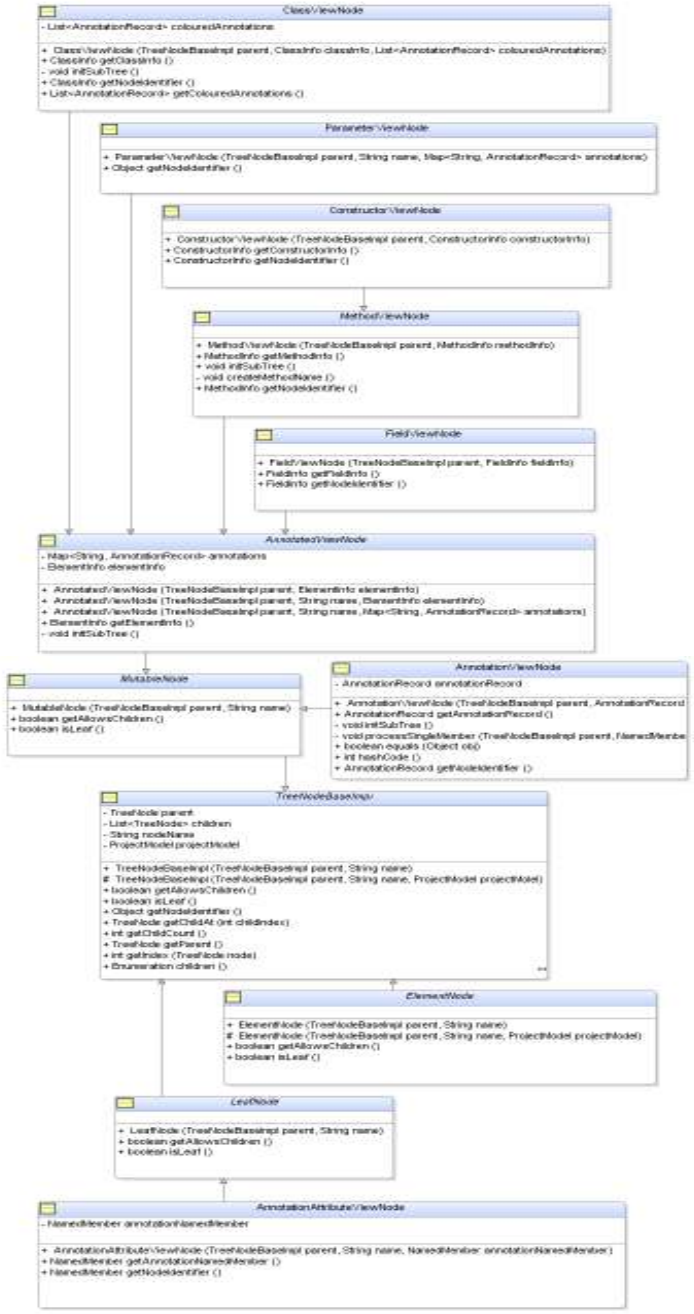
- Класът LevelNode представя наследниците на корена на дървото. Това са елементите визуализиращи т.нар. нива на търсене на анотации (Клас, Метод, Конструктор, Поле, Параметър).
- Инстанциите на класа ProjectAndContentNode представят елементи от дървото, служещи за групиране на всички намерени класове отговарящи на критериите за търсене, които се намират в даден проект и в един и същ негов content файл.
- Елементите на класа PackageAndClassNode са наследници (визуални) на елементите на класа ProjectAndContentNode. Те представят конкретен Java клас, който съдържа анотация, отговаряща на критериите за търсене, като визуализира името и пакета на класа.



Диаграма. 5.13: Елементите на визуалното дърво в Search View, реализиращи интерфейса SavableNode

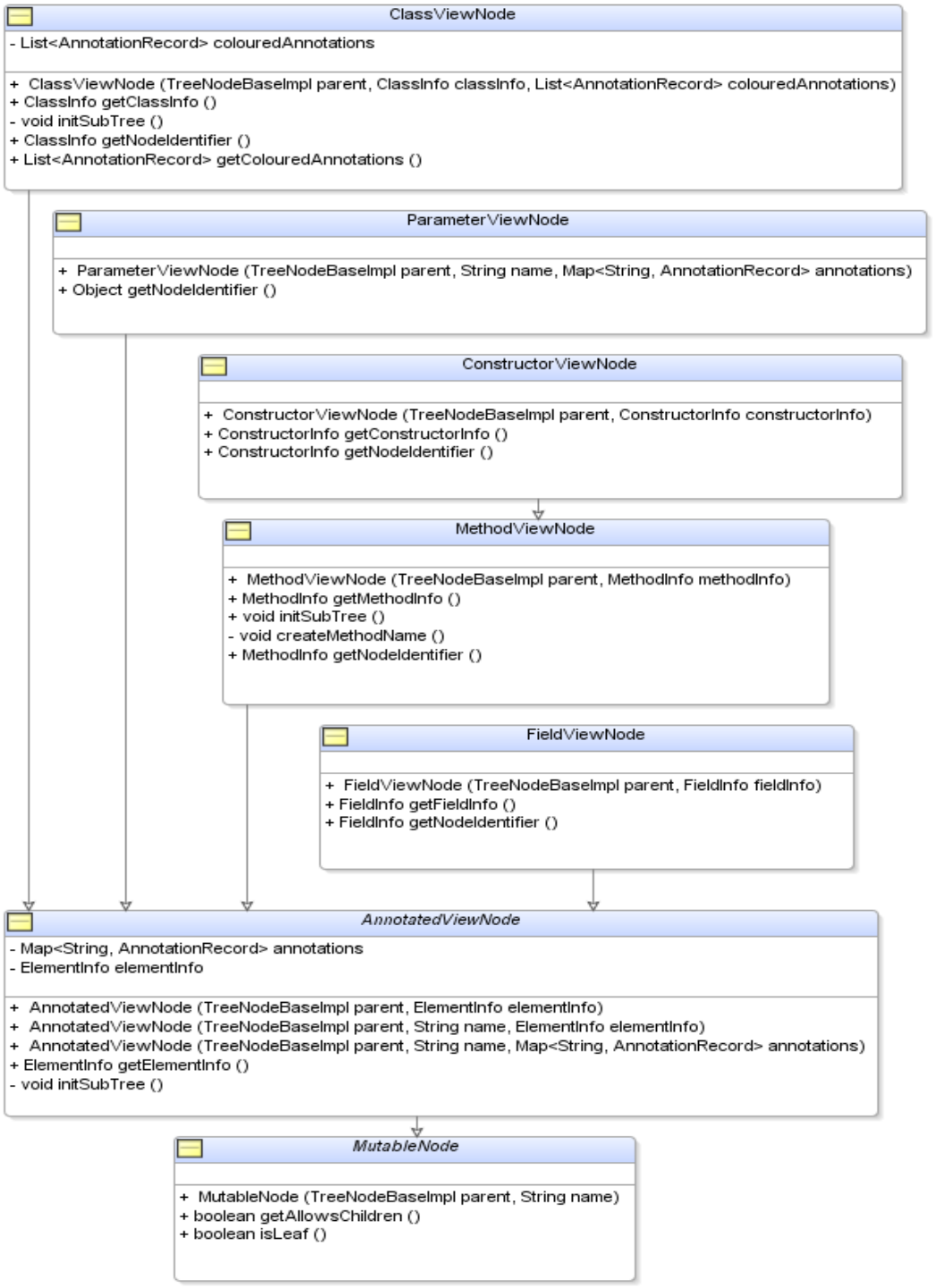
5.5.4. Класове, реализиращи елементите на визуалното дърво в Class View

Тези класове са илюстрирани, чрез следните три диаграми и са детайлно описани по-долу:

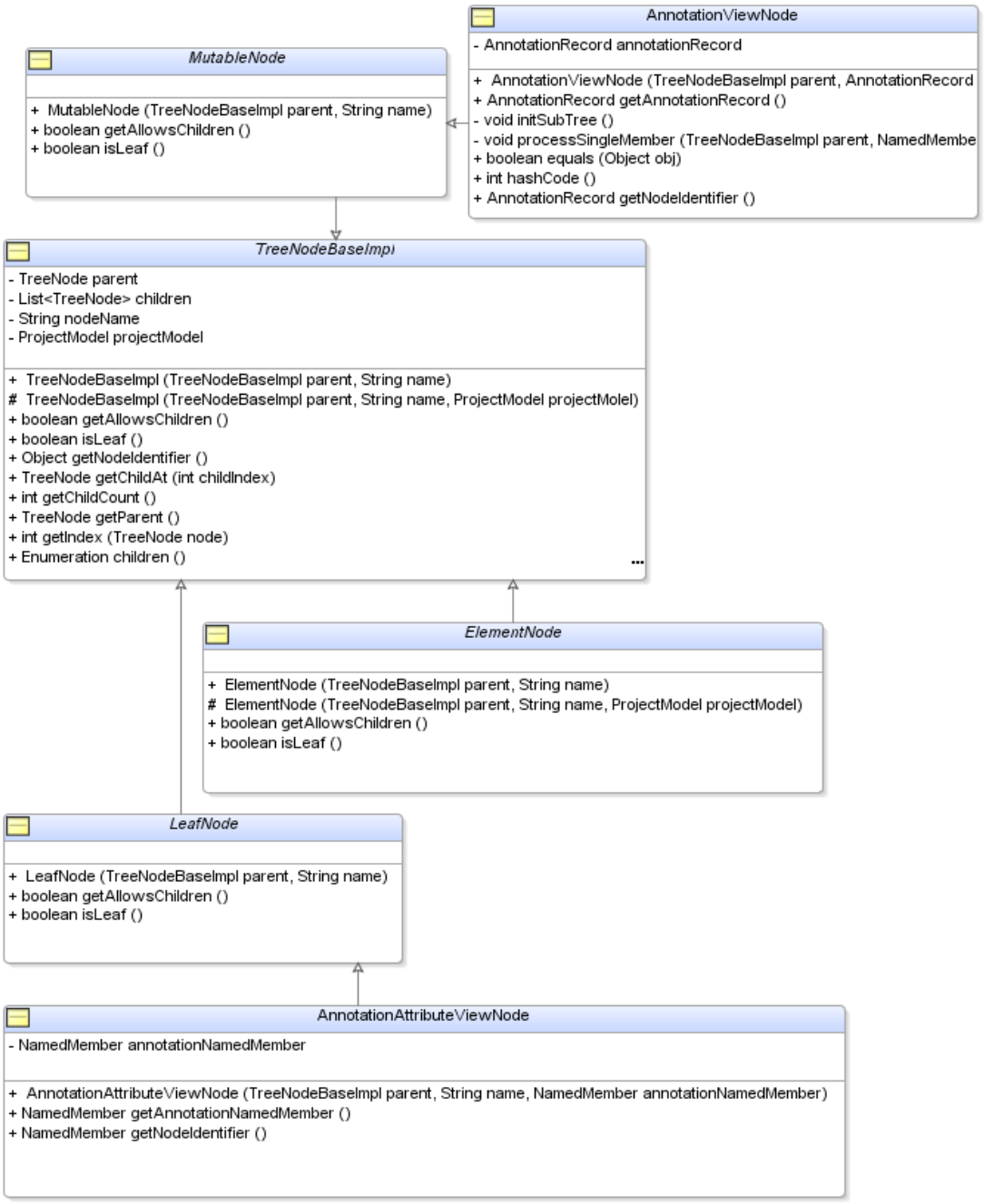


Диаграма. 5.14: Цялостна класова йерархия на елементите изграждащи визуалните дървета в Class View

Следва декомпозиция на диаграмата в две части:



Диаграма. 5.15: Класова йерархия на елементите изграждащи визуалните дървета в Class View като наследници на MutableNode.



Диаграма. 5.16: Клас TreeNodeBaseImpl като прародител на класовете AnnotationAttributeViewNode и AnnotationViewNode, инстанциите, на които са елементите “листа” на визуалните дървета в Class View и представят възлите, визуализиращи отделните анотации и техни атрибути в изобразения, чрез дър-вото, клас.

- Базовият клас **AnnotatedViewNode**, се наследява от всички класове на елементи, имащи възможността да притежават анотации – ClassViewNode, MethodViewNode, ConstructorViewNode, FieldViewNode и ParameterViewNode. В него е реализирана логиката за визуализация на анотациите на тези елементи. Ако съответния елемент има анотации се създава негов наследник с име “Annotations”, в който се структурира цялата информация за анотациите му. Класът съдържа ElementInfo обект, представящ клас елемента, за който се отнася и списък от анотациите на този елемент.
- Класът **ClassViewNode** е корен на дървото и визуализира името (заедно с пакета) на класа, който дървото изобразява. Той съдържа списък с анотациите, които трябва да бъдат оцветени в червен цвят, в случай, че визуализацията на този клас е инициирана от Search View, вместо от Projects View.
- Класът **MethodViewNode** визуализира сигнатурата на даден метод от класа. Освен наследената информация от родителите си, не съдържа допълнителна.
- Класът **ConstructorViewNode** визуализира сигнатурата на даден конструктор от класа. Не съдържа допълнителна информация, а само тази, която наследява от родителите си.
- Класът **FieldViewNode** визуализира сигнатурата на дадено поле на класа. Не съдържа допълнителна информация, а само тази, която наследява от родителите си.
- Класът **ParameterViewNode** визуализира сигнатурата на даден параметър на клас. Не съдържа допълнителна информация, а само тази, която наследява от родителите си.
- Класът **AnnotationViewNode** визуализира името на единична анотация на даден елемент. Тя е визуален наследник на “Annotations” възела на дадения елемент.
- Класът **AnnotationAttributeViewNode** визуализира името и стойността на единичен атрибут на анотация.

*5.6. Пакет com.fmi.bytecode.annotations.gui.businesslogic.actions.**

В този пакет са разположени слушателите на събития (вж. **Фигура 5.1**), които са междинният слой, служещ за връзка между действията, които инициира потребителят върху графичният интерфейс и конкретните функции (вж. **Точка 4.1**) свързани с тях, които трябва да се изпълнят от приложението. Общия брой на класовете, реализиращи слушатели на събития, е 14. Това са класовете, чиито инстанции слушат за събития в главния прозорец на приложението, в диалозите - за нов проект, за зареждане на проект, за модификация на проект, за търсене на анотации и за записване на метаданните в XML, а също и слушателите на събития в контекстните менюта на визуалните дървета в Projects View и Search View. Общото за всички слушатели е, че тяхна инстанция се създава в конструктора на съответния графичен компонент (JDialog или JFrame), за чиито събития ще слушат. Тази инстанция се запазва в съответния графичен компонент и споделя неговият жизнен цикъл.

5.6.1. Слушател за събитията на главния прозорец на приложението (MainFrame) – MainFrameActionListener

Инстанция на този клас, инициализирана при създаването на MainFrame, слуша за всички предварително дефинирани събития, които потребителят може да предизвика върху главният прозорец на приложението, чрез клавиатурата и мишката - кратките комбинации от клавиши дефинирани за всяка от функциите в **Точка 4.1**, както и за събития на мишката, избиращи някой от елементите на главните менюта на приложението. Множеството от командите, които разпознава (чрез вътрешните им кодови наименования) е – NewProject, OpenProject, ProjectProps, CloseAll, CloseProject, AnnotationSearch, Export , Help, About и Exit. Тези команди съответстват на функциите, описани в **Точка 4.1**.

След настъпването на определено събитие, този слушател предприема изпълнението на действията свързани с него - отваряне на някой от диалоговите прозорци на приложението (NewProjectDialog, LoadProjectDialog, FindAnnotationsDialog и т.н.) с подходящите параметри или изход от приложението.

5.6.2. Слушател на събитията на диалоговите прозорци за създаване на нов проект (NewProjectDialog) и модификация на

зареден проект (ProjectPropertiesDialog) - NewProjectActionListener

Този клас слуша за събитията свързани с диалога за създаване на нов проект. Това са събитията свързани с натискането на някой от бутоните на диалоговия прозорец. Множеството от командите, които разпознава е – Browse, Add, Remove, Cancel и Apply. Действията, които се изпълняват, при всяка от подадените от потребителят команди, са следните:

- Browse – Отваря диалога за навигиране във файловата система (BrowseFileSystemDialog), като дава възможност на потребителя да избере директория, в която да се запише новия проект.
- Add – Отваря диалога за навигиране във файловата система (BrowseFileSystemDialog), като дава възможност на потребителя да избере content файл, който да бъде добавен към новия проект.
- Remove – Изтрива текущо избрания content файл в JList компонента на NewProjectDialog.
- Cancel – Затваря диалога NewProjectDialog, без да предприема по-нататъшни действия.
- Apply – Създава модела на новия проект, записва го на файловата система, зарежда го в Projects View и затваря диалога NewProject-Dialog. Ако модела не може да бъде създаден, поради некоректно въведени данни от потребителя, се визуализира подходящо съобщение за грешка и диалога не се затваря.

5.6.3. Слушател на събитията на диалога за навигиране във файловата система (BrowseFileSystemDialog) – BrowseFileSystemActionListener

Този клас слуша за събитията свързани с диалога за навигиране във файловата система. Това са събитията свързани с натискането на някой от бутоните на диалоговия прозорец. Множеството от командите, които разпознава е – ApproveSelection, CancelSelection. Действията, които се изпълняват при всяка от подадените от потребителят команди, са следните:

- ApproveSelection – Взима текущо избрания файл и го подава на асоциирания с диалога SelectedFileValueApplier обект (вж. **Точка**

5.2.6). Действието, което се изпълнява е кодирано в конкретната реализация на SelectedFileValueApplier, подадена при създаването на диалога. В частност, когато реализацията е LoadProjectApplier, действието е зареждането на проекта в Projects View. След, което се затваря диалога.

- CancelSelection – Затваря диалога BrowseFileSystemDialog, без да предприема по-нататъчни действия.

5.6.4. Слушател на събитията на диалога за търсене на анотации (AnnotationsSearchDialog) – AnnotationSearchActionListener

Този клас слуша за събитията свързани с диалога за търсене на анотации. Това са събитията свързани с натискането на някой от бутоните на диалоговия прозорец. Множеството от командите, които разпознава е – Cancel, Add, Remove и Search. Действията, които се изпълняват, при всяка от подадените от потребителят команди, са следните:

- Cancel – Затваря диалога.
- Add – Добавя в списъка с филтри текущо конструирания филтър за търсене на анотации.
- Remove – Премахва текущо избраният филтър от списъка с всички добавени филтри.
- Search – Стартира търсенето на анотации (Функция 7) с текущо сформирания списък от филтри. След което визуализира резултата от търсене в Search View и затваря диалога.

5.6.5. Слушател на събитията на диалога за записване на метаданни в XML файл (ExportToXMLDialog) – ExportToXMLActionListener

Този клас слуша за събитията в диалога за записване на метаданни в XML файл. Това са събитията свързани с натискането на някой от бутоните или радио бутоните на диалоговия прозорец. Множеството от командите, които разпознава е – Export, Cancel, BrowseDir, BrowseXSL, EnableCustom и DisableCustom. Действията, които се изпълняват, при всяка от подадените от потребителят команди, са следните:

- Cancel – Затваря диалога.
- Export – Стартира процеса за записване на данните, с текущо избраните от потребителят настройки (Функция 8).
- BrowseDir – Отваря диалога за навигиране във файловата система (BrowseFileSystemDialog), като дава възможност на потребителя да избере директория, в която да бъде записан XML файла, съдържащ метаданните.
- BrowseXSL – Отваря диалога за навигиране във файловата система, като дава възможност на потребителя да избере специален XLS файл, който да бъде приложен за трансформация на стандартния XML, преди записването му във файловата система.
- EnableCustom – Прави достъпна опцията за избиране на XSL файл. Тази команда се предизвиква при избор на радио бутона enable в диалога.
- DisableCustom – Прави недостъпна опцията за избиране на XSL файл. Тази команда се предизвиква при избор на радио бутона disable в диалога.

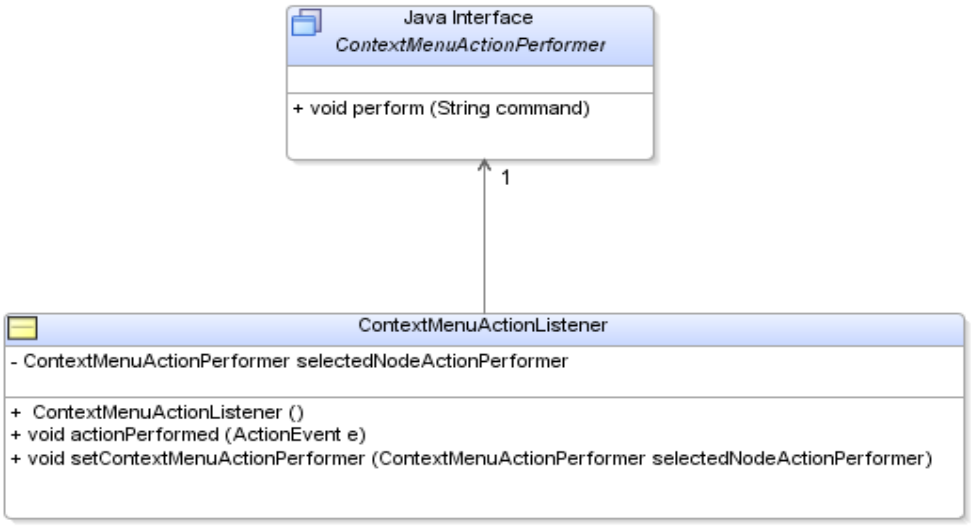
5.6.6. Слушател на събитията на ClosableTabbedPaneWithPopup – TabbedPaneContextMenuActionPerformerImpl

Този клас слуша за събитията в реализирания в приложението клас за табулации. Събитието, за което слуша е десен бутон на мишката. При настъпването на това събитие се подава команда CloseAllTabs, при което слушателят инициира затваряне на всички табове на компонента ClosableTabbedPaneWithPopup .

5.6.7. Клас ContextMenuActionListener и слушатели, реализиращи появяването на контекстните менюта във визуалните дървета на Projects View и Search View – ProjectTreeMouseListener и SearchResultTreeMouseListener

Тези слушатели реализират стандартния в Swing интерфейс java.awt.event.MouseListener. Те слушат за събитието - натискане на десен бутон на мишката. При настъпването на това събитие проверяват дали координатите на мишката съвпадат с някой от елементите на дървото. Ако това е така,

анализират съответния елемент и визуализират контекстното меню свързано с него (всеки елемент има различно контекстно меню, в зависимост от функцията, която изпълнява в дървото). Всички контекстни менюта имат общ слушател за събития – инстанция на ContextMenuActionListener. Той не обработва командите на потребителя, а ги делегира към инстанцията на ContextMenuActionPerformer (вж. **Точка 5.6.8**), която съдържа в себе си:



Диаграма. 5.17: ContextMenuActionListener делегира командите на потребителя към инстанцията на ContextMenuActionPerformer, която съдържа.

Слушателите ProjectTreeMouseListener и SearchResultTreeMouseListener имат отговорността да прикачат коректната инстанция на ContextMenuActionPerformer към ContextMenuActionListener, в момента на появяване на менюто, която да разпознава командите разрешени за елемента, притежаващ визуализираното меню.

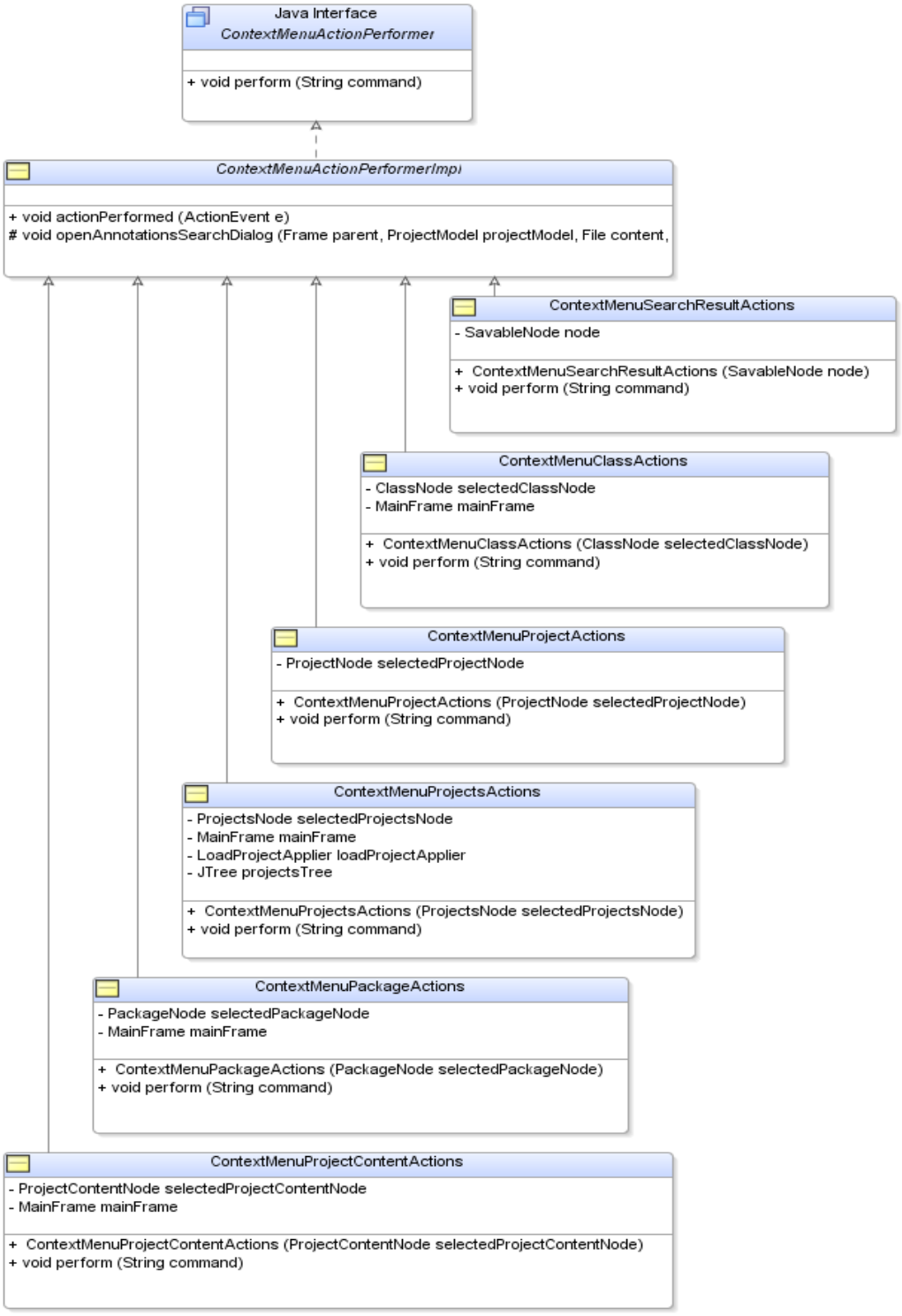
Следва описание на всички слушатели на събития за контекстните менюта на визуалните дървета:

5.6.8. Слушатели на събития за контекстните менюта на визуалните дървета – реализации на ContextMenuActionPerformer

Базовият, за всички слушатели на контекстни менюта, е класа ContextMenuActionPerformer. Той реализира стандартния за Swing интерфейс

java.awt.event.ActionListener. Йерархията на наследяване е представена на **Диаграма. 5.17**, по-долу. Всеки наследник на ContextMenuActionPerformer разпознава определена група от команди, в зависимост от семантиката, която има елемента, към който е прикачен във визуалното дърво:

- **ContextMenuProjectsActions** (в дървото на Projects View) – Разпознава следните команди от контекстното меню на корена на дървото “Projects” – NewProject, CloseAll, AnnotationSearch и Export. Тези команди съответстват на функциите описани в **Точка 4.6.1**, описваща контекстните менюта в Projects View.
- **ContextMenuProjectActions** (в дървото на Projects View) – Разпознава командите от контекстното меню на елементите представящи проект – ProjectProps, CloseProject, AnnotationSearch и Export (вж. **Точка 4.6.1**).
- **ContextMenuPackageActions, ContextMenuProjectContentActions** и **ContextMenuClassActions** (в дървото на Projects View) – Разпознават командите от контекстното меню на елементите представящи съответно проект, content файловете на даден проект или клас принадлежащ на даден content файл – AnnotationSearch и Export (вж. **Точка 4.6.1**).
- **ContextMenuSearchResultActions** (в дървото на Search View) – Използва се от всички елементи на дървото. Разпознава командата Export. Тази команда съответства на функцията описана в **Точка 4.9.1**, описваща контекстните менюта в Search View.



Диаграма. 5.18: Йерархия на слушателите на събития за контекстните менюта на визуалните дървета, като наследници на ContextMenuActionPerformer.

5.7. Пакет com.fmi.bytecode.annotations.gui.utils. - Реализация на функциите, които приложението предоставя на потребителя*

В този пакет са разположени помощните класове на приложението, с чиято помощ са реализирани функциите, дефинирани в точка **Точка 4.1**. Те се състоят от статични методи, които приемат като аргументи всички необходими данни за изпълнението на дадена функция, след което реализират действията за постигане на желания от нея резултат. Както беше показано с **Фигура. 5.1** (Архитектура на системата), тези функции се стартират от описаните в предходната точка - слушатели на събития. При подаването на определена команда, от страна на потребителя, чрез даден графичен компонент, съответният му слушател за събития подготвя контекстните данни, необходими като параметри за извикването на даден статичен метод, изпълнява този метод и визуализира на потребителя статуса, с който е приключило изпълнението на конкретната функция (грешка или успех).

В този пакет е разположен класа ModelUtils, който съдържа метод “ReadResult processContent(ProjectModel projectModel)” с чията помощ се осъществява интеграцията на графичния интерфейс с четеца на байт код от високо ниво [14]. Този метод приема като аргумент модел на проект, извлича съдържащите се в него content файлове и извиква четеца от високо ниво, като подава извлечените content файлове като вход. От своя страна, четецът връща като резултат обект от класа ReadResult, представящ модела на прочетените клас файлове. Информацията съдържаща се в този обект се използва от графичния интерфейс за визуализация на метаданните на прочетените класове.

Следва описание на класовете и статичните методи, които са входни точки за реализацията на определена функция:

5.7.1. Реализация на Функция 1 - Създаване на нов проект

Тази функция се стартира, от слушателя на събития **NewProjectActionListener,** при команда **Apply**. За целта се използват последователно методите:

- ModelsRepository: addProjectModel(ProjectModel projectModel) – Добавя модела на новия проект в склада за модели ModelsRepository.
- MainFrame: getProjectTree().updateUI() – Добавя новия проект във визуалното дърво на Projects View.
- ModelUtils: saveProjectModel(ProjectModel prj) – Записва новия проект в XML файл, като използва информацията, която се съдържа в модела на проекта, подаден като аргумент (в това число името на XML файла и директорията, в която трябва да бъде записан).

5.7.2. Реализация на Функция 2 - Зареждане на вече съществуващ проект

Тази функция се стартира, от слушателя за събития **BrowseFileSystemActionListener**, при команда **ApproveSelection**. В контекста за зареждане на проект, асоциираната с BrowseFileSystemDialog реализация на SelectedFileValueApplier, е LoadProjectApplier. В неговият метод apply(File selectedFile) се използват последователно методите:

- ModelUtils: ProjectModel loadProjectModel(File projectFile) – Зарежда модела на проекта, от специфициран като аргумент XML файл.
- ModelsRepository: addProjectModel(ProjectModel projectModel) – Добавя модела на заредения проект в склада за модели ModelsRepository.
- MainFrame: getProjectTree().updateUI() – Добавя новия проект във визуалното дърво на Projects View.

5.7.3. Реализация на Функция 3 - Визуализация и промяна на настройките на зареден проект

Тази функция се стартира, от слушателя за събития **NewProjectActionListener**, при команда **Apply**. За целта се използват последователно методите:

- ModelsRepository: updateProjectModel (ProjectModel oldProjectModel, ProjectModel newProjectModel) – актуализира модифицирания проект в склада за модели ModelsRepository.

- MainFrame: getProjectTree().updateUI() – актуализира модифицирания проект във визуалното дърво на Projects View.
- ModelUtils: saveProjectModel(ProjectModel prj) – записва модифицирания проект в XML файл.

5.7.4. Реализация на Функция 4 - Затваряне на зареден проект

Тази функция се стартира, от слушателя за събития **MainFrameActionListener**, при команда **CloseProject**. Слушателят проверява кой е избрания за затваряне проект в Projects View, взима неговия модел и използва последователно следните методи:

- ModelsRepository: removeProject(ProjectModel projectModel) – Изтрива проекта от склада за модели.
- MainFrame: getProjectTree().updateUI() – актуализира промяната във визуалното дърво на Projects View.

5.7.5. Реализация на Функция 5 - Затваряне на всички заредени проекти

Тази функция се стартира, от слушателя за събития **MainFrameActionListener,** при команда **CloseAll**. За целта се използват последователно методите:

- ModelsRepository: removeProjects() – Изтрива всички заредени проекти от склада за модели.
- MainFrame: getProjectTree().updateUI() – актуализира промяната във визуалното дърво на Projects View, което остава празно (визуализира се само корена на дървото - “Projects”).

5.7.6. Реализация на Функция 6 - Затваряне на приложението

Тази функция се стартира, от слушателя за събития **NewProjectActionListener**, при команда **Exit**. За целта се използва системния метод System.exit(0).

5.7.7. Реализация на Функция 7 - Търсене на анотации в произволно подмножество от съдържанието на всички проекти

Тази функция се стартира, от слушателя за събития **AnnotationSearchActionListener**, при команда **Search**. За целта се използват последователно методите:

- ***AnnotationSearchActionListener***: private void createProjectAndContentAnnotationIndexes (Map<ProjectModel, Map<File, AnnotationsIndex>> annIndexes, ProjectModel projectModel, File contentFile, String packageName, List<AnnotationFilter> filters) – По подадени модел на проект, негов content файл, Java пакет и списък от филтри за търсене, създава и записва в изходния параметър annIndexes. AnnotationsIndex обект представящ модела на намерените анотации. Той моделира всички анотации на класове или клас елементи, намиращи се в подадения пакет, проект и content файл, които отговарят на критериите описани чрез филтрите за търсене. Този метод се извиква итеративно, когато търсенето е на ниво “Всички проекти” или “Проект”, като се обхождат всички необходими content файлове. При итеративното извикване, резултата се натрупва в изходният параметър annIndexes.
- ***SearchResultUtils***: public static TreeModel createTree(Map<ProjectModel, Map<File, AnnotationsIndex>> annIndexes) – По конструираните обектни модели на намерените анотации (съдържащите се в Map структурата AnnotationsIndex обекти), създава обект от стандартния клас за Swing javax.swing.tree.TreeModel, който се използва за визуализация на резултата от търсене в Search View.
- ***MainFrame***: addSearchResultTreeModel(treeModel) – Визуализира конструирания резултат в Search View на главния прозорец на приложението - MainFrame.

5.7.8. Реализация на Функция 8 - Запис на метаданните в XML формат

Тази функция се стартира, от слушателя за събития **ExportToXMLActionListener**, при команда **Export**. За целта се използват последователно методите:

- ***ExportUtils***: public static Document createExportDoc(ExportModel exportModel, ExportConfiguration exportConfig) throws XMLException – По подаден модел на метаданните за запис в XML и имена по подразбиране на таговете представени чрез ExportConfiguration обект, създава org.w3c.dom.Document (DOM) обект, представляващ стандартния обектен модел на XML документ в Java.
- ***XMLUtils***: public static void save(File output, Document doc) throws IOException – Записва в текстов формат подадения обектен модел на XML документ в специфицирания, като първи аргумент, файл.

5.7.9. Реализация на Функция 9 - Визуализация на информация за автора на проекта

Тази функция се стартира, от слушателя за събития **MainFrameActionListener**, при команда **About**. За целта се използва стандартния метод в Swing:

- JOptionPane.showMessageDialog(parent, new AboutDialog(), "About", JOptionPane.PLAIN_MESSAGE), който визуализира специално реализирания за целта AboutDialog.

5.7.10. Реализация на Функция 10 - Визуализация на ръководство на потребителя

Тази функция се стартира, от слушателя за събития **MainFrameActionListener**, при команда **Help**. За целта се визуализира диалога HelpDialog. В него е реализирана функционалност за визуализация на HTML страници, чрез класа javax.swing.JEditorPane. Съдържанието на документацията в HTML формат се намира в пакета: com.fmi.bytecode.annotations.gui.graphical.dialogs.help.*, като главната страница е UserGuide.htm.

6. Тестване

Процеса на тестване на една софтуерна система може да бъде интегриран по различни начини, в процеса на нейното разработване и внедряване. Трите основни стратегии в това направление са [11]:

1. Тестването върви паралелно с процеса на разработка, като за всеки новоразработен модул се добавят тестове, които да верифицират неговата коректна функционалност.
2. Тестването започва едва след разработването на системата, но преди нейното внедряване за експлоатация.
3. Тестването започва след внедряването на системата, като клиентите, които я използват докладват множество проблеми и по този начин, заявките за поддръжка рязко се повишат.

Практиката показва, че колкото по-рано в процеса на разработката и внедряването на системата се открият проблемите, възпрепятстващи нейната коректна работа, толкова по-малко усилия, време и ресурси са необходими за разрешаването им.

Поради тази причина, избраната стратегия за тестване на разработената система е 1 - Тестването върви паралелно с процеса на разработка, като за всеки новоразработен модул се добавят тестове, които да верифицират неговата коректна функционалност.

Има различни методи за тестване, без значение коя от горните стратегии за интегриране на процеса на тестване е избрана. При тестването на разработената система са използвани техники, които в последно време утвърждават себе си като най ефективните методи за откриване на проблеми в софтуерни системи:

- Тестове на части от кода (Unit tests)
- Тестове базирани на сценариите за използване на системата (Scenario-based testing или Scenario-oriented testing) в комбинация с тестване тип черна кутия (Black Box Testing).

6.1. Тестове на части от кода (Unit tests)

При разработването на всяка една от десетте функции (вж. **Точка 4.1** и **Точка 5.7**) на системата, са реализирани тестове, които имат за цел да верифицират резултата от обработката на дадено множество от входни данни.

В частност, отделните единици, обект на Unit тестване, са класовете на системата, които реализират функциите й. Всеки метод на тези класове е тестван, като е извикан с един или повече набори от входни данни и е верифициран резултата от изпълнението му. Верифицирането на резултата се прави по следния начин:

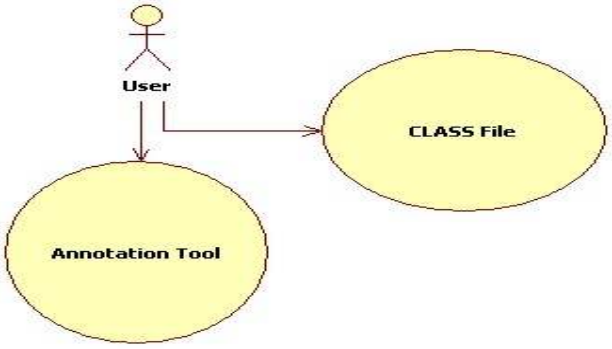
- За методи, които връщат стойност без да променят състоянието на обекта, върху който са извикани – верифицирана е върнатата стойност от метода.
- За методи, които не връщат стойност или които променят състоянието на обекта, върху който са извикани – верифицирането е извършено с помощта на дебъгер, с чиято помощ необходимите стойности са проверени директно в паметта на виртуалната машина (JVM).

6.2. Тестове базирани на сценариите за използване на системата (Scenario-based testing или Scenario-oriented testing) в комбинация с тестване тип черна кутия (Black Box Testing)

По време на разработването на системата, при завършването на един или няколко модула, които реализират цялостен функционален елемент, се разработва тестов сценарий, който проиграва всеки от начините, по които крайният потребител ще работи със системата, използвайки конкретната функция.

По този начин е тествано поведението на системата в множество различни сценарии, в които потребителите ще я използват. Всеки от тези сценарии се свежда до верификация на резултата от обработката на конкретно множество клас файлове, постъпващо като вход на системата, в някой от допустимите форми, описани в **Точка 3** (Анализ на проблемната област и определяне на входните изисквания). Следват диаграми илюстриращи отделните сценарии, след което описание на верификацията за всеки от тях.

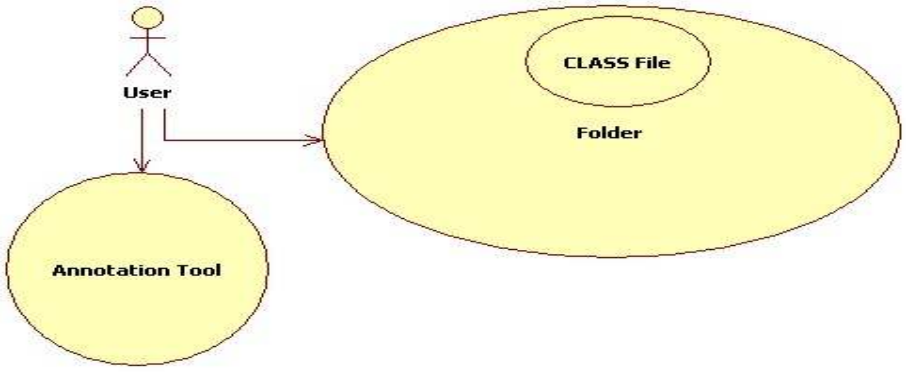
- Създаване на проект, съдържащ само един клас файл:



Фиг. 11: Сценарий за използване на системата, в който създаденият проект съдържа единичен клас файл

При визуализация на новосъздадения проект се появява името му и един content файл с пълния път на класа. Съдържанието на този content файл е само избрания клас.

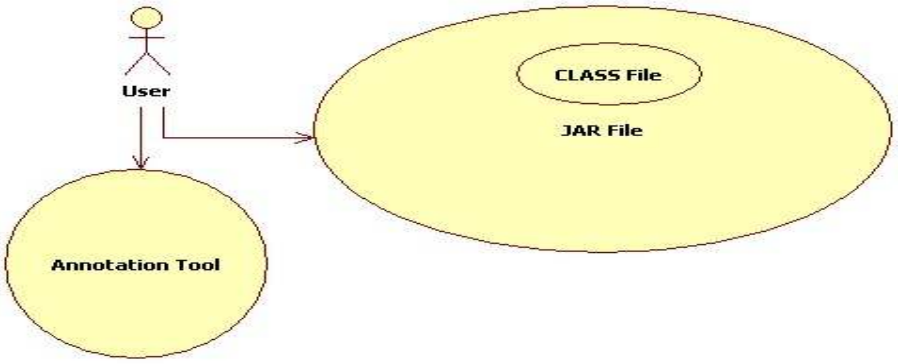
- Създаване на проект, съдържащ клас файловете в дадена директория:



Фиг. 12: Сценарий за използване на системата, в който създаденият проект съдържа клас файлове, намиращи се в дадена директория на файловата система

При визуализация на новосъздадения проект се появява името му и един content файл с пълния път на директорията. Съдържанието на този content файл са всички класове намиращи се в избраната директория.

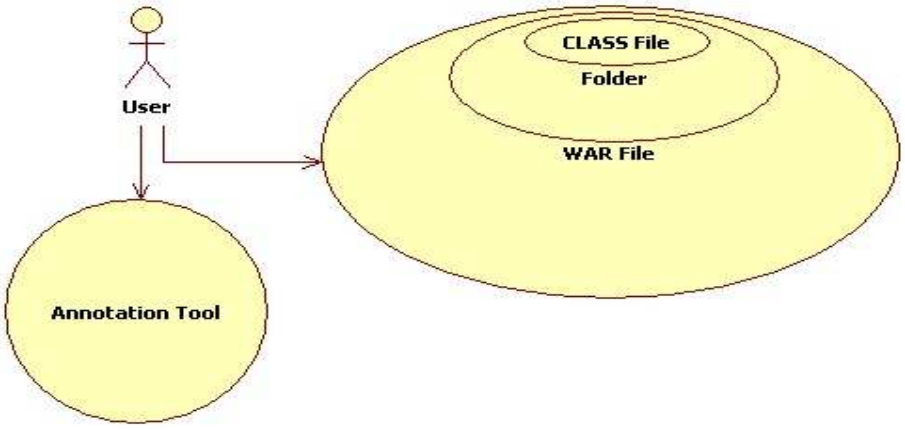
- Създаване на проект, съдържащ jar файл, в който е пакетиран клас файл:



Фиг. 13: Сценарий за използване на системата, в който създаденият проект съдържа jar архив, съдържащ в себе си клас файл.

При визуализация на новосъздадения проект се появява името му и един content файл с пълния път на jar файла. Съдържанието на този content файл е класа намиращ се в избрания jar файл.

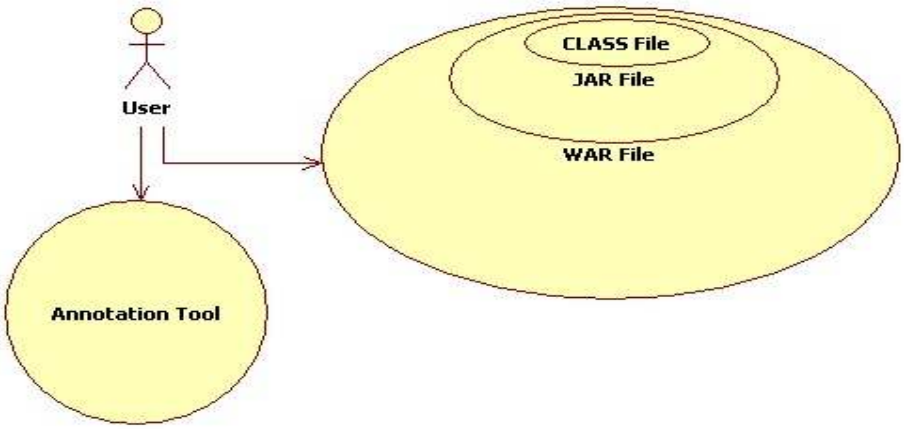
- Създаване на проект, съдържащ war файл, в който е пакетирана директория, в която има клас файл:



Фиг. 14: Сценарий за използване на системата, в който създаденият проект съдържа war архив, пакетиращ в себе си директория с клас файл

При визуализация на новосъздадения проект се появява името му и един content файл с пълния път на war файла. Съдържанието на този content файл е класа намиращ се в директорията на избрания war файл.

- Създаване на проект, съдържащ war файл. В него е пакетиран jar файл, който от своя страна съдържа клас файл:



Фиг. 15: Сценарий за използване на системата, в който създаденият проект съдържа war архив, пакетиращ в себе си jar архив с клас файл

При визуализация на новосъздаденият проект се появява името му и един content файл с пълния път на jar файла. Съдържанието на този content файл е класа намиращ се в него.

Описаните сценарии са верифицирани, чрез използването на тестване тип черна кутия, по следния начин:

- Създаване на проект, който в зависимост от конкретния сценарии, съдържа определен content файл, с определено съдържание.
- Затваряне на проекта.
- Зареждане на проекта от мястото, на което е записан във файловата система.
- Верификация на визуализираното дърво в Projects View.
- Търсене на анотации във всяко от нивата за търсене с позитивни и негативни филтри.
- Верификация на визуализираните дървета в Search View, създадени в резултат на различните търсения.
- Записване на метаданни под формата на XML, от всички възможни възли на визуалните дървета в Projects View и Search View.

- Валидация на записаните XML файлове, чрез схемата по подразбиране **amp.xsd**, с помощта на javax.xml.validation.Validator.
- Трансформиране на резултата с вградените XSL файлове (вж. **Точка 5.4.3**) и механична верификация на резултата с помощта на Browser.

Така протеклото тестване на системата, гарантира нейната коректност във всеки един възможен сценарий за използването ѝ.

7. Заключение

В процеса на разработка на дипломната работа бяха подробно анализирани следните проблеми:

- Проучване на спецификацията описана в J2SE(TM) Development Kit Documentation 5.0, по отношение на нововъведените анотации. Разгледани са техните свойства и е отговорено на въпроса къде и как могат да бъдат използвани.
- Дефиниране на изискванията към разработената система. Определени са всички сценарии за използване на системата, като употребата й не се ограничи само до инструмент за работа с метаданните на отделни клас файлове, а при направения анализ, се разшири за да може да бъде използвана в J2EE (Java EE 5) технологията - за извличане на метаинформацията от WEB и EJB компоненти на JEE приложения.
- Проектиране на решението, в което подробно са определени отделните компоненти на системата. Дефинирани са техните цели и отговорности. Направено е подробно описание на техните предназначения и функции, които са моделирани въз основа на изискванията и сценариите за използване на системата. Изготвени са необходимите фигури за илюстриране на дефинираните графични модели за работа.
- Реализация на решението, която е свързана с написването на програмен код, реализиращ отделните компоненти на системата, както и описанието на всички класове от реализацията, съпроводено с необходимите UML диаграми.
- Тестване на разработената система, в което са вложени необходимите усилия, за да се гарантира коректната й работа.

Като естествено следствие от решаването на всички тези проблеми е изградена гъвкава и стабилна графична среда за обработка и извличане на метаинформация от Java байт код. Обектно ориентирания й модел и реализация позволяват изключително лесна и интуитивна работа с нея. С графичния й интерфейс могат да си служат успешно потребители, които имат бегла представа

от структурата на байт кода. Функционалността, която предлага е несравнима, понеже към настоящия момент не съществуват подобни програмни решения.

Насоките за възможно развитие на дипломната работата са ориентирани към разширяването на функциите, които предлага, по начин, който бъдещите й потребители биха изисквали, за улеснение на работата си с нея.

Списък съкращения и специални термини

JVM	–	Виртуалната машина на Java
JDK	–	Java платформата. Включва JVM, Java компилатор и други инструменти за разработване на Java софтуер.
JDK 1.5	–	Минималната версията на JDK която представлява интерес за разработката. В нея са въведени за пръв път т.нар. анотации
JLS	–	Спецификация на езика Java (Java Language Specification), съгласно документацията J2SE(TM) Development Kit Documentation 5.0.
DOM	–	Document Object Model - стандартния обектен модел на XML документ в Java
Bytecode / Байт код	–	Двоично представяне на клас в Java.
Class Loading	–	Стандартен механизъм на Java за зареждане на класове в JVM от съответният им Bytecode.
API	–	Application programming interface
Reflection API	–	Стандартен интерфейс на Java, който се базира на Class Loading и служи за обектно ориентиран достъп до структурата на даден клас и в частност до неговите метаданни.
Анотация(и)	–	Механизъм за прикрепване на метаданни към даден клас или елемент на клас. Този механизъм е въведен за пръв път в езика Java с JLS.
Анотиране	–	Добавяне на анотация към даден клас или елемент на клас.
UML	–	Unified Modeling Language
XML	–	Extensible Markup Language
J2EE	–	Java 2 Platform, Enterprise Edition
Java EE 5	–	Java Platform, Enterprise Edition version 5
JEE Application	–	Приложение разработено върху Java платформата и базирано на Java технологиите J2EE или Java EE 5.
EJB	–	Enterprise Java Beans

8. Използвана литература

Основни източници:

1. New Features and Enhancements J2SE 5.0 at http://www.mathcs.carleton.edu/courses/course_resources/j2se-1.5.0-docs/relnotes/features.html
2. J2SE 5.0 in a Nutshell at http://java.sun.com/developer/technicalArticles/releases/j2se15/
3. Annotations in Tiger, Part 1: Add metadata to Java code at http://www-128.ibm.com/developerworks/java/library/j-annotate1/
4. What's New in Java 1.5? at http://www.cs.indiana.edu/classes/jett/sstamm/
5. Tiger in NetBeans4 at http://cropcrusher.web.infoseek.co.jp/shufujava/sunone/nb/nb4tiger_en.html
6. Taming the tiger at http://www.denverjug.org/meetings/files/200408_Tiger.pdf
7. http://java.sun.com/javaee/5/docs/tutorial/doc/
8. "Java(TM) Language Specification, 3rd Edition", James Gosling, Bill Joy, Guy Steele, Gilad Bracha
9. "Thinking in Java, 4rd Edition", Bruce Eckel
10. "UML destilled, 3rd edition", Martin Fowler
11. "Managing the Testing Process, Second Edition", John Wiley
12. "Object-Oriented Software Construction, Second Edition", Bertrand Meyer
13. "Data Structures and Algorithms in Java, 4th edition", Michael T. Goodrich, Roberto Tamassia

Дипломна работа:

14. Красимир И. Топчийски (2007) „Система за обработка и извличане на мета-информация от Java байт код"

9. Приложения

9.1. Извадки от кода на приложението

9.1.1. Програмен код на RightClickGlassPane

```
package com.fmi.bytecode.annotations.gui.graphical.dialogs;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class RightClickGlassPane extends JComponent
                implements MouseListener, MouseMotionListener {

   protected JPopupMenu popup;
   protected Container contentPane;

   public RightClickGlassPane(Container contentPane) {
      addMouseListener(this);
      addMouseMotionListener(this);
      this.contentPane = contentPane;
   }

   public void setPopupMenu(JPopupMenu menu) {
      this.popup = menu;
   }

   // draw some text just so we know the glass pane
   // is installed and visible
   public void paint(Graphics g) {
   }

   // catch all mouse events and redispatch them
   public void mouseMoved(MouseEvent e) {
      redispatchMouseEvent(e, false);
   }
   public void mouseDragged(MouseEvent e) {
      redispatchMouseEvent(e, false);
   }
   public void mouseClicked(MouseEvent e) {
      redispatchMouseEvent(e, false);
   }
   public void mouseEntered(MouseEvent e) {
      redispatchMouseEvent(e, false);
   }
   public void mouseExited(MouseEvent e) {
      redispatchMouseEvent(e, false);
```

```
}
public void mousePressed(MouseEvent e) {
   redispatchMouseEvent(e, false);
}
public void mouseReleased(MouseEvent e) {
   redispatchMouseEvent(e, false);
}

public void showPopup(Component comp, int x, int y) {
   if (popup != null) {
      try {
         popup.show(comp,x,y);
      } catch (java.awt.IllegalComponentStateException icse) {
         /**
          * Do nothing.
          * The mouse cursor is moved outside the gui component
          * before the event processing.
          */
      }
   }
}

protected void redispatchMouseEvent(MouseEvent e, boolean repaint) {
   // if it's a popup
   if(e.isPopupTrigger()) {
      // show the popup and return
      showPopup(e.getComponent(), e.getX(), e.getY());
   } else {
      doDispatch(e);
   }
}

protected Component getRealComponent(Point pt) {
   // get the mouse click point relative to the content pane
   Point containerPoint =
      SwingUtilities.convertPoint(this, pt,contentPane);

   // find the component that under this point
   Component component = SwingUtilities.getDeepestComponentAt(
                                 contentPane,
                                 containerPoint.x,
                                 containerPoint.y);
   return component;
}

protected void doDispatch(MouseEvent e) {
   // since it's not a popup we need to redispatch it.

   Component component = getRealComponent(e.getPoint());
```

```
        // return if nothing was found
        if (component == null) {
            return;
        }

        // convert point relative to the target component
        Point componentPoint = SwingUtilities.convertPoint(
            this,
            e.getPoint(),
            component);

        // redispatch the event
        component.dispatchEvent(new MouseEvent(component,
            e.getID(),
            e.getWhen(),
            e.getModifiers(),
            componentPoint.x,
            componentPoint.y,
            e.getClickCount(),
            e.isPopupTrigger()));
    }
}
```

9.1.2. Програмен код на ClosableTabbedPaneWithPopup

```
package com.fmi.bytecode.annotations.gui.graphical.dialogs;

import java.awt.AWTEvent;
import java.awt.Component;
import java.awt.event.MouseEvent;

import javax.swing.Icon;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.JRootPane;
import javax.swing.JTabbedPane;

public class ClosableTabbedPaneWithPopup extends JRootPane {

    private TabbedPane tabbedPane;
    private RightClickGlassPane rightClickGlassPane;

    private JPopupMenu classAllpopup;

    private ContextMenuActionListener ctxMenuListener;
    private ContextMenuActionPerformer selectedNodeActionPerformer;

    public ClosableTabbedPaneWithPopup(Icon closeButtonIcon) {
        this(closeButtonIcon, null);
```

```
    }

    public ClosableTabbedPaneWithPopup(Icon closeButtonIcon,
                                       ContextMenuActionPerformer _selectedNodeActionPerformer) {
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        tabbedPane = new TabbedPane(closeButtonIcon);
        rightClickGlassPane = new RightClickGlassPane(tabbedPane);
        rightClickGlassPane.setVisible(false);
        getContentPane().add(tabbedPane);
        setGlassPane(rightClickGlassPane);

        selectedNodeActionPerformer = _selectedNodeActionPerformer;
        if (selectedNodeActionPerformer == null) {
            selectedNodeActionPerformer = new
TabbedPaneContextMenuActionPerformerImpl(tabbedPane);
        }

        ctxMenuListener = new ContextMenuActionListener();
        ctxMenuListener.setContextMenuActionPerformer(selectedNodeActionPerformer);

        classAllpopup = createClassAllPopup();
    }

    private JPopupMenu createClassAllPopup() {

        classAllpopup = new JPopupMenu();

        JMenuItem closeAllTabsMenuItem = new JMenuItem(" Close All");
        closeAllTabsMenuItem.setActionCommand("CloseAllTabs");
        closeAllTabsMenuItem.addActionListener(ctxMenuListener);
        closeAllTabsMenuItem.setIcon(GUIComponentsRepository.CLOSE_ALL);

        classAllpopup.add(closeAllTabsMenuItem);

        return classAllpopup;
    }

    public RightClickGlassPane getRightClickGlassPane() {
        return rightClickGlassPane;
    }

    private void showGlassPane(boolean show) {
        getRightClickGlassPane().setPopupMenu(classAllpopup);
        getRightClickGlassPane().setVisible(show);
    }

    public JTabbedPane getTabbedPane() {
        return tabbedPane;
    }
```

```
    private class TabbedPane extends JTabbedPane {

        private Icon icon;

        public TabbedPane(Icon closeButtonIcon) {
            enableEvents(AWTEvent.MOUSE_EVENT_MASK);
            icon = closeButtonIcon;
        }

        protected void processMouseEvent(MouseEvent evt) {
            int tabNumber = getUI().tabForCoordinate(this, evt.getX(), evt.getY());

            if (tabNumber > -1) {
                showGlassPane(true);
            } else {
                showGlassPane(false);
            }

            if (evt.getID() == MouseEvent.MOUSE_CLICKED && tabNumber > -1
                && ((IconProxy) getIconAt(tabNumber)).contains(evt.getX(), evt.getY())) {

                int oldSelTab = getSelectedIndex();
                int oldTabCount = getTabCount();
                removeTabAt(tabNumber);

                if (tabNumber < oldSelTab && oldSelTab > 0
                    && oldSelTab != (oldTabCount - 1)) {

                    setSelectedIndex(oldSelTab - 1);
                }
                return;
            }

            super.processMouseEvent(evt);
        }

        public void addTab(String title, Component component) {
            addTab(title, new IconProxy(icon), component);
        }
    }
}
```

9.1.3. Програмен код на AnnotationFilterImpl

```
package com.fmi.bytecode.annotations.gui.businesslogic.model.searchfilters;

import com.fmi.bytecode.annotations.element.AnnotationRecord;
import com.fmi.bytecode.annotations.element.ClassInfo;
import com.fmi.bytecode.annotations.element.ConstructorInfo;
import com.fmi.bytecode.annotations.element.ElementInfo;
import com.fmi.bytecode.annotations.element.FieldInfo;
```

```
import com.fmi.bytecode.annotations.element.MethodInfo;
import com.fmi.bytecode.annotations.tool.indexing.AnnotationFilter;


import java.util.List;
import java.util.Map;

public abstract class AnnotationFilterImpl implements AnnotationFilter {
   private AnnotationFilterModel filter;

   public AnnotationFilterImpl(AnnotationFilterModel filter) {
      this.filter = filter;
   }

   public abstract boolean isPossitive();

   public abstract boolean isAccepted(AnnotationRecord annotationRecord);

   protected boolean isParameterAnnotation(MethodInfo methodInfo, String annotationName)
{
      boolean result = false;
      Map<String, AnnotationRecord>[] parametersAnnotations =
methodInfo.getParameterAnnotations();
      if (parametersAnnotations != null) {
         for (int i = 0; i < parametersAnnotations.length; i++) {
            Map<String, AnnotationRecord> parameterAnnotations =
parametersAnnotations[i];
            if (parameterAnnotations != null) {
               for (String annName: parameterAnnotations.keySet()) {
                  AnnotationRecord currAnnotation = parameterAnnotations.get(annName);
                  if (annotationName == null ||
currAnnotation.getTypeName().equals(annotationName)) {
                     result = true;
                     break;
                  }
               }
            }
         }
      }
      return result;
   }

   protected boolean isMethodAnnotation(MethodInfo methodInfo,
                              String annotationName) {
      boolean result = false;
      Map<String, AnnotationRecord> annotations = methodInfo.getAnnotations();

      if (annotations != null) {
         for (String annName: annotations.keySet()) {
            AnnotationRecord currAnnotation = annotations.get(annName);
```

```
            if (currAnnotation.getTypeName().equals(annotationName)) {
               result = true;
               break;
            }
         }
      }
      return result;
   }

   public AnnotationFilterModel getFilter() {
      return filter;
   }
}
```

9.1.4. Програмен код на PositiveAnnotationFilter

```
package com.fmi.bytecode.annotations.gui.businesslogic.model.searchfilters;

import com.fmi.bytecode.annotations.element.AnnotationRecord;
import com.fmi.bytecode.annotations.element.ClassInfo;
import com.fmi.bytecode.annotations.element.ConstructorInfo;
import com.fmi.bytecode.annotations.element.ElementInfo;
import com.fmi.bytecode.annotations.element.FieldInfo;
import com.fmi.bytecode.annotations.element.MethodInfo;

import java.util.List;

public class PositiveAnnotationFilter extends AnnotationFilterImpl {
   public PositiveAnnotationFilter(AnnotationFilterModel filter) {
      super(filter);
   }

   public boolean isPossitive() {
      return true;
   }

   public boolean isAccepted(AnnotationRecord annotationRecord) {
      return possitiveAccept(annotationRecord);
   }

   protected boolean possitiveAccept(AnnotationRecord annotationRecord) {
      boolean result = false;
      String filterAnnotationName = getFilter().getAnnotationName();
      String annotationName = annotationRecord.getTypeName();
      if (filterAnnotationName == null
            || filterAnnotationName.equals(annotationName)) {

         ElementInfo elInfo = annotationRecord.getOwner();
         List<FilterLevelType> levels = getFilter().getLevel();
         for (FilterLevelType level : levels) {
            if (FilterLevelType.CLASS.equals(level)) {
```

```
                if (elInfo instanceof ClassInfo) {
                    result = true;
                    break;
                }
            } else if (FilterLevelType.CONSTRUCTOR.equals(level)) {
                if (elInfo instanceof ConstructorInfo) {
                    result = true;
                    break;
                }
            } else if (FilterLevelType.FIELD.equals(level)) {
                if (elInfo instanceof FieldInfo) {
                    result = true;
                    break;
                }
            } else if (FilterLevelType.METHOD.equals(level)) {
                if (elInfo instanceof MethodInfo) {
                    MethodInfo methodInfo = (MethodInfo) elInfo;
                    if (isMethodAnnotation(methodInfo, annotationRecord.getTypeName())) {
                        result = true;
                        break;
                    }
                }
            } else if (FilterLevelType.PARAMETER.equals(level)) {
                if (elInfo instanceof MethodInfo) {
                    MethodInfo methodInfo = (MethodInfo) elInfo;
                    if (isParameterAnnotation(methodInfo, annotationRecord.getTypeName())) {
                        result = true;
                        break;
                    }
                }
            }
        }
    }
    return result;
  }
}
```

9.1.5. Програмен код на NegativeAnnotationFilter

```
package com.fmi.bytecode.annotations.gui.businesslogic.model.searchfilters;

import com.fmi.bytecode.annotations.element.AnnotationRecord;
import com.fmi.bytecode.annotations.element.ClassInfo;
import com.fmi.bytecode.annotations.element.ClassMemberInfo;
import com.fmi.bytecode.annotations.element.ConstructorInfo;
import com.fmi.bytecode.annotations.element.ElementInfo;

import com.fmi.bytecode.annotations.element.FieldInfo;

import com.fmi.bytecode.annotations.element.MethodInfo;
```

```
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class NegativeAnnotationFilter extends AnnotationFilterImpl {

   private Set<String> rejectedClasses;

   public NegativeAnnotationFilter(AnnotationFilterModel filter) {
      super(filter);
      rejectedClasses = new HashSet<String>();
   }

   public boolean isPossitive() {
      return false;
   }

   public boolean isAccepted(AnnotationRecord annotationRecord) {
      boolean result = false;
      ClassInfo clazz = getAnnotationClassOwner(annotationRecord);
      String className = clazz.getName();
      if (rejectedClasses.contains(className)) {
         result = false;
      } else {
         boolean accepted = negativeAccept(clazz);
         if (accepted) {
            result = true;
         } else {
            rejectedClasses.add(className);
            result = false;
         }
      }
      return result;
   }

   private boolean negativeAccept(ClassInfo annClass) {
      boolean result = true;
      String filterAnnotationName = getFilter().getAnnotationName();

      List<FilterLevelType> levels = getFilter().getLevel();
      for (FilterLevelType level : levels) {
         if (FilterLevelType.CLASS.equals(level)) {
            if (containsInClass(filterAnnotationName, annClass)) {
               result = false;
               break;
            }
         } else if (FilterLevelType.CONSTRUCTOR.equals(level)) {
            if (containsInConstructors(filterAnnotationName, annClass.getConstructors())) {
```

```
                result = false;
                break;
            }
        } else if (FilterLevelType.FIELD.equals(level)) {
            if (containsInFields(filterAnnotationName, annClass.getFields())) {
                result = false;
                break;
            }
        } else if (FilterLevelType.METHOD.equals(level)) {
            if (containsInMethods(filterAnnotationName, annClass.getMethods())) {
                result = false;
                break;
            }
        } else if (FilterLevelType.PARAMETER.equals(level)) {
            if (containsInMethodsParams(filterAnnotationName, annClass.getMethods())) {
                result = false;
                break;
            }
        }
    }

    return result;
}

private ClassInfo getAnnotationClassOwner(AnnotationRecord annotationRecord) {
    ClassInfo clazz = null;
    ElementInfo element = annotationRecord.getOwner();
    if (element instanceof ClassInfo) {
        clazz = (ClassInfo)element;
    } else if (element instanceof ClassMemberInfo) {
        clazz = (ClassInfo) ((ClassMemberInfo)element).getOwner();
    } else {
        throw new IllegalStateException();
    }
    return clazz;
}

private boolean containsInClass(String searchAnnName, ClassInfo annClass) {
    Map<String, AnnotationRecord> anns = annClass.getAnnotations();
    return containsAnnotation(anns, searchAnnName);
}

private boolean containsInConstructors(String searchAnnName,
                                       ConstructorInfo[] constructorInfos) {
    return containsAnnotation(constructorInfos, searchAnnName);
}

private boolean containsInFields(String searchAnnName, FieldInfo[] fieldInfos) {
    return containsAnnotation(fieldInfos, searchAnnName);
}
```

```
    private boolean containsInMethods(String searchAnnName,
                                      MethodInfo[] methodInfos) {
        return containsAnnotation(methodInfos, searchAnnName);
    }

    private boolean containsInMethodsParams(String searchAnnName,
                                            MethodInfo[] methodInfos) {
        boolean result = false;
        for (int i=0; i<methodInfos.length; i++) {
            MethodInfo method = methodInfos[i];
            if (isParameterAnnotation(method, searchAnnName)) {
                result = true;
                break;
            }
        }
        return result;
    }

    private boolean containsAnnotation(ElementInfo[] elements, String searchAnnName) {
        boolean result = false;
        for (int i=0; i<elements.length; i++) {
            ElementInfo element = elements[i];
            Map<String, AnnotationRecord> anns = element.getAnnotations();
            if (containsAnnotation(anns, searchAnnName)) {
                result = true;
                break;
            }
        }
        return result;
    }

    private boolean containsAnnotation(Map<String, AnnotationRecord> anns, String
searchAnnName) {
        boolean result = false;
        for (String annName: anns.keySet()) {
            if (searchAnnName == null || annName.equals(searchAnnName)) {
                result = true;
                break;
            }
        }
        return result;
    }
}
```

9.1.6. Програмен код на TreeNodeBaseImpl

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Enumeration;

import java.util.List;

import javax.swing.tree.TreeNode;

import com.fmi.bytecode.annotations.gui.businesslogic.model.projects.ProjectModel;

public abstract class TreeNodeBaseImpl implements TreeNode {

   private TreeNode parent;
   private List<TreeNode> children;
   private String nodeName;
   private ProjectModel projectModel;

   public TreeNodeBaseImpl(TreeNodeBaseImpl parent, String name) {
      this.parent = parent;
      children = new ArrayList<TreeNode>();
      nodeName = name;
      if (parent != null) {
         parent.addChild(this);
         this.setProjectModel(parent.getProjectModel());
      }
   }

   protected TreeNodeBaseImpl(TreeNodeBaseImpl parent, String name,
                                       ProjectModel projectMolel) {
      this(parent, name);
      this.setProjectModel(projectMolel);
   }

   public abstract boolean getAllowsChildren();

   public abstract boolean isLeaf();

   public abstract Object getNodeIdentifier(); // data object containig unique inf about node

   public TreeNode getChildAt(int childIndex) {
      return children.get(childIndex);
   }

   public int getChildCount() {
      return children.size();
```

```
}

public TreeNode getParent() {
   return parent;
}

public int getIndex(TreeNode node) {
   return children.indexOf(node);
}

public Enumeration children() {
   return Collections.enumeration(children);
}

public String getNodeName() {
   return nodeName;
}

protected void setNodeName(String nodeName) {
   this.nodeName = nodeName;
}

private void addChild(TreeNode child) {
   children.add(child);
}

public boolean equals(Object obj) {
   boolean result = false;
   if (obj instanceof TreeNodeBaseImpl) {
      TreeNodeBaseImpl node = (TreeNodeBaseImpl) obj;
      if (node.getNodeName().equals(getNodeName())) {
         result = true;
      }
   }
   return result;
}

public int hashCode() {
   return getNodeName().hashCode();
}

public String toString() {
   return this.getNodeName();
}

public void removeChild(Object nodeIdentifier) {
   for (int i = 0; i < children.size(); i++) {
      TreeNodeBaseImpl currChild = (TreeNodeBaseImpl) children.get(i);
      if (currChild.getNodeIdentifier().equals(nodeIdentifier)) {
         children.remove(currChild);
```

```
            break;
        }
      }
    }

    public void removeChildren() {
      children.clear();
    }

    private void setProjectModel(ProjectModel projectModel) {
      this.projectModel = projectModel;
    }

    public ProjectModel getProjectModel() {
      return projectModel;
    }
}
```

9.1.7. Програмен код на ElementNode

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common;

import com.fmi.bytecode.annotations.gui.businesslogic.model.projects.ProjectModel;

public abstract class ElementNode extends TreeNodeBaseImpl {

    public ElementNode(TreeNodeBaseImpl parent, String name) {
      super(parent, name);
    }

    protected ElementNode(TreeNodeBaseImpl parent, String name,
                                    ProjectModel projectModel) {
      super(parent, name, projectModel);
    }

    public boolean getAllowsChildren() {
      return true;
    }

    public boolean isLeaf() {
      return false;
    }
}
```

9.1.8. Програмен код на LeafNode

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common;

public abstract class LeafNode extends TreeNodeBaseImpl {
```

```
    public LeafNode(TreeNodeBaseImpl parent, String name) {
        super(parent, name);
    }

    public boolean getAllowsChildren() {
        return false;
    }

    public boolean isLeaf() {
        return true;
    }
}
```

9.1.9. Програмен код на ClassNode

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.project;

import com.fmi.bytecode.annotations.element.ClassInfo;

import java.io.File;

import java.util.ArrayList;
import java.util.List;

import com.fmi.bytecode.annotations.gui.businesslogic.treenodes.SavableNode;

import com.fmi.bytecode.annotations.gui.businesslogic.model.export.SaveUnit;

import com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common.LeafNode;
import
com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common.TreeNodeBaseImpl;

public class ClassNode extends LeafNode implements SavableNode {
    private ClassInfo classInfo;
    private File contentFile;

    //String name -> Name of тхе class without package
    public ClassNode(TreeNodeBaseImpl parent, String name, ClassInfo classInfo,
                                            File contentFile) {
        super(parent, name);
        this.classInfo = classInfo;
        this.contentFile = contentFile;
    }

    public ClassInfo getClassInfo() {
        return classInfo;
    }

    public File getContentFile() {
        return contentFile;
```

```
    }

    public ClassInfo getNodeIdentifier() {
        return getClassInfo();
    }

    public String getClassName() {
        return getClassInfo().getName();
    }

    public List<SaveUnit> getUnits() {
        List<SaveUnit> saveUnits = new ArrayList<SaveUnit>();
        List<ClassInfo> classInfos = new ArrayList<ClassInfo>();
        classInfos.add(classInfo);
        SaveUnit unit = new SaveUnit(getProjectModel(), getContentFile(), classInfos);
        saveUnits.add(unit);
        return saveUnits;
    }
}
```

9.1.10. Програмен код на MutableNode

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common;

public abstract class MutableNode extends TreeNodeBaseImpl {

    public MutableNode(TreeNodeBaseImpl parent, String name) {
        super(parent, name);
    }

    public boolean getAllowsChildren() {
        return !isLeaf();
    }

    public boolean isLeaf() {
        return getChildCount() == 0;
    }
}
```

9.1.11. Програмен код на StaticTextNode

```
package com.fmi.bytecode.annotations.gui.businesslogic.treenodes.common;

//roots
public class StaticTextNode extends ElementNode {

    public static final String CLASSES_NODE = "Classes";
    public static final String PARAMETERS_NODE = "Parameters";
```

```
    public static final String CONSTRUCTORS_NODE = "Constructors";
    public static final String METHODS_NODE = "Methods";
    public static final String FIELDS_NODE = "Fields";
    public static final String ANNOTATIONS_NODE = "Annotations";

    public StaticTextNode(TreeNodeBaseImpl parent, String name) {
        super(parent, name);
    }

    public String getNodeIdentifier() {
        return getNodeName();
    }
}
```

9.1.12. XSLT трансформация replaceTagNames.xsl

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>


<xsl:template match="projects">
    <projects-data> <xsl:apply-templates/> </projects-data>
</xsl:template>


<xsl:template match="project">
    <project-data projectName="{@name}">
        <xsl:apply-templates/>
    </project-data>
</xsl:template>


<xsl:template match="content">
    <content-data contentName="{@name}">
        <xsl:apply-templates/>
    </content-data>
</xsl:template>


<xsl:template match="package">
    <package-data packageName="{@name}">
        <xsl:apply-templates/>
    </package-data>
</xsl:template>


<xsl:template match="class">
    <class-data className="{@name}">
        <xsl:apply-templates/>
```

```
    </class-data>
</xsl:template>


<xsl:template match="annotations">
    <annotations-data> <xsl:apply-templates/> </annotations-data>
</xsl:template>

<xsl:template match="methods">
    <methods-data> <xsl:apply-templates/> </methods-data>
</xsl:template>

<xsl:template match="constructors">
        <constructors-data> <xsl:apply-templates/> </constructors-data>
</xsl:template>

<xsl:template match="fields">
        <fields-data> <xsl:apply-templates/> </fields-data>
</xsl:template>

<xsl:template match="method">
    <method-data methodName="{@name}" methodType="{@type}">
        <xsl:apply-templates/>
    </method-data>
</xsl:template>

<xsl:template match="annotation">
    <annotation-data annotationName="{@name}" >
        <xsl:apply-templates/>
    </annotation-data>
</xsl:template>

<xsl:template match="constructor">
    <constructor-data constructorName="{@name}">
        <xsl:apply-templates/>
    </constructor-data>
</xsl:template>


<xsl:template match="field">
    <field-data fieldName="{@name}" fieldType="{@type}">
        <xsl:apply-templates/>
    </field-data>
</xsl:template>

<xsl:template match="parameters">
        <parameters-data> <xsl:apply-templates/> </parameters-data>
</xsl:template>

<xsl:template match="parameter">
```

```
    <parameter-data parameterType="{@type}">
        <xsl:apply-templates/>
    </parameter-data>
</xsl:template>

<xsl:template match="annotation-attribute">
    <annotationAttribute-data annotationAttributeName="{@name}"
annotationAttributeValue="{@value}">
        <xsl:apply-templates/>
    </annotationAttribute-data>
</xsl:template>

<xsl:template match="array-attribute">
    <arrayAttribute-data arrayAttributeOrder="{@order}" arrayAttributeValue="{@value}">
        <xsl:apply-templates/>
    </arrayAttribute-data>
</xsl:template>

</xsl:stylesheet>
```

9.1.13. XSLT трансформация extractAllAnnotationsAsRoots.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>


<xsl:template match="projects">
        <annotations> <xsl:apply-templates/> </annotations>
</xsl:template>


<xsl:template match="/projects/project/content/package/class">
        <xsl:for-each select="annotations">
                <xsl:apply-templates>
                        <xsl:with-param name="pathToClass" select=".."/>
                </xsl:apply-templates>
        </xsl:for-each>

        <xsl:for-each select="methods/method | constructors/constructor">
                <xsl:for-each select="annotations">
                        <xsl:apply-templates>
                                <xsl:with-param name="pathToClass" select="../../.."/>
                        </xsl:apply-templates>
                </xsl:for-each>
                <xsl:for-each select="parameters/parameter/annotations">
                        <xsl:apply-templates>
                                <xsl:with-param name="pathToClass" select="../../../../.."/>
                        </xsl:apply-templates>
                </xsl:for-each>
        </xsl:for-each>
```

```
</xsl:template>


<xsl:template match="annotation">
	<xsl:param name="pathToClass" select="."/>


	<xsl:variable name="annotatedElementName" select="local-name(../..)"/> <!--method,
field, constructor, parameter, class-->

	<!--xsl:element name="{$annotatedElementName}">
		<xsl:attribute name="parent"/>
	</xsl:element-->


	<xsl:variable name="projectName" select="$pathToClass/../../../@name"/>
	<xsl:variable name="contentName" select="$pathToClass/../../@name"/>
	<xsl:variable name="packageName" select="$pathToClass/../@name"/>
	<xsl:variable name="className" select="$pathToClass/@name"/>
	<xsl:variable name="parentName" select="local-name(..)"/>


	<annotation name="{@name}">
		<xsl:if test="$parentName='annotations'">
			<xsl:call-template name="createLocation">
			<xsl:with-param name="projectName" select="$projectName"/>
			<xsl:with-param name="contentName" select="$contentName"/>
			   <xsl:with-param name="packageName" select="$packageName"/>
			<xsl:with-param name="className" select="$className"/>
				<xsl:with-param name="annotatedElementName"
select="$annotatedElementName"/>
			</xsl:call-template>
		</xsl:if>

		<xsl:element name="attributes">
			<xsl:apply-templates/>
	</xsl:element>
	</annotation>
</xsl:template>


<xsl:template name="createLocation">
	<xsl:param name="projectName"/>
	<xsl:param name="contentName"/>
	<xsl:param name="packageName"/>
	<xsl:param name="className"/>
	<xsl:param name="annotatedElementName"/>

	<xsl:element name="location">
		<xsl:call-template name="createElement1attr">
```

```
<xsl:with-param name="elName">project</xsl:with-param>
        <xsl:with-param name="attrName">name</xsl:with-param>
   <xsl:with-param name="attrValue" select="$projectName"/>
   </xsl:call-template>
   <xsl:call-template name="createElement1attr">
        <xsl:with-param name="elName">content</xsl:with-param>
       <xsl:with-param name="attrName">name</xsl:with-param>
    <xsl:with-param name="attrValue" select="$contentName"/>
   </xsl:call-template>
   <xsl:call-template name="createElement1attr">
        <xsl:with-param name="elName">package</xsl:with-param>
       <xsl:with-param name="attrName">name</xsl:with-param>
    <xsl:with-param name="attrValue" select="$packageName"/>
</xsl:call-template>
   <xsl:call-template name="createElement1attr">
        <xsl:with-param name="elName">class</xsl:with-param>
       <xsl:with-param name="attrName">name</xsl:with-param>
    <xsl:with-param name="attrValue" select="$className"/>
</xsl:call-template>



   <xsl:if test="not($annotatedElementName='class')">
        <xsl:variable name="name" select="../../@name"/>


        <xsl:if test="$annotatedElementName='method' or
$annotatedElementName='field'">
            <xsl:call-template name="createElement2attr">
                <xsl:with-param name="elName"
select="$annotatedElementName"/>
                <xsl:with-param name="attrName1">name</xsl:with-
param>
                <xsl:with-param name="attrValue1" select="$name"/>
                <xsl:with-param name="attrName2">type</xsl:with-
param>
                <xsl:with-param name="attrValue2"
select="../../@type"/>
            </xsl:call-template>
        </xsl:if>
        <xsl:if test="$annotatedElementName='constructor'">
            <xsl:call-template name="createElement1attr">
                <xsl:with-param name="elName"
select="$annotatedElementName"/>
                <xsl:with-param name="attrName">name</xsl:with-
param>
                <xsl:with-param name="attrValue" select="$name"/>
            </xsl:call-template>
        </xsl:if>
```

```
                        <xsl:if test="$annotatedElementName='parameter'">
                                <xsl:call-template name="createElement2attr">
                                        <xsl:with-param name="elName" select="local-
name(../../../..)"/>
                                        <xsl:with-param name="attrName1">name</xsl:with-
param>
                                        <xsl:with-param name="attrValue1"
select="../../../../@name"/>
                                        <xsl:with-param name="attrName2">type</xsl:with-
param>
                                        <xsl:with-param name="attrValue2"
select="../../../../@type"/>
                                </xsl:call-template>


                                <xsl:call-template name="createElement2attr">
                                        <xsl:with-param name="elName"
select="$annotatedElementName"/>
                                        <xsl:with-param name="attrName1">type</xsl:with-
param>
                                        <xsl:with-param name="attrValue1"
select="../../@type"/>
                                        <xsl:with-param name="attrName2">order</xsl:with-
param>
                                        <xsl:with-param name="attrValue2"
select="../../@order"/>
                                </xsl:call-template>
                        </xsl:if>
                </xsl:if>
        </xsl:element>
</xsl:template>


<xsl:template name="createElement1attr">
        <xsl:param name="elName"/>
        <xsl:param name="attrName"/>
        <xsl:param name="attrValue"/>
        <xsl:element name="{$elName}">
                <xsl:attribute name="{$attrName}">
                        <xsl:value-of select="$attrValue"/>
                </xsl:attribute >
        </xsl:element>
</xsl:template>


<xsl:template name="createElement2attr">
        <xsl:param name="elName"/>
        <xsl:param name="attrName1"/>
        <xsl:param name="attrValue1"/>
        <xsl:param name="attrName2"/>
```

```
        <xsl:param name="attrValue2"/>

        <xsl:element name="{$elName}">
                <xsl:attribute name="{$attrName1}">
                        <xsl:value-of select="$attrValue1"/>
                </xsl:attribute >
                <xsl:attribute name="{$attrName2}">
                        <xsl:value-of select="$attrValue2"/>
                </xsl:attribute >
        </xsl:element>
</xsl:template>


<xsl:template match="annotation-attribute">
        <annotation-attribute name="{@name}" value="{@value}">
        <xsl:apply-templates/>
        </annotation-attribute>
</xsl:template>

<xsl:template match="array-attribute">
        <array-attribute order="{@order}" value="{@value}">
        <xsl:apply-templates/>
        </array-attribute>
</xsl:template>


</xsl:stylesheet>
```