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: *Система за обработка и извличане на мета-информация от Java байт код*

Дипломант: Красимир Иванов Топчийски*, фак. № М-21218*

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.

1.2. Цел и задачи на дипломната работа

Целта на дипломната работа е да се проектира и разработи система, която да предоставя механизъм за работа с метаданни (анотации), въз основа на тяхното обектно ориентирано представяне, по максимално удобен за използване начин (Аналогично на Reflection API) [4], [7], [8].

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 8 глави: “Увод”, “Анотация и анотиране”, “Анализ на проблемната област и определяне на входните изисквания”, “Проектиране на решението”, “Описание на реализацията”, “Тестване”, “Заключение”, “Използвана литература”.

Уводът (**Глава 1.**) съдържа кратко въведение в областта на дипломната работа. Формулират се целта и задачите на работата 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 вградените анотации @Retention и @Target. Тук се решава част от **Задача 1** на дипломната работа.

В **Глава 3.** „Анализ на проблемната област и определяне на входните изисквания“ е направено резюме на клас структурата. Разгледани са бизнес сценариите, в които възниква нужда от анализ на метаданните, с цел да се определят изискванията за входния формат на данните към разработваната система. 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. Описани са моделите използвани при нейната реализация. Приложени са Unified Modeling Language (UML) клас диаграми, които онагледяват свойствата и връзките на отделните компоненти, с което изцяло се решава **Задача 3**, както и част от **Задача 6** [5].

В **Глава 5.** „Описание на реализацията” е решена **Задача 4**, като е представена реализацията на проекта описан в **Глава 3**. Описани са техническите детайли и са обяснени по-интересни фрагменти от кода. 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 й.

Основният текст на дипломната работа се съпровожда от „Списък съкращения и речник на специалните термини” и списък с „Използвана литература”.

2. Анотация и анотиране

За да могат да бъдат извлечени метаданните (анотациите) от даден Byte-code, първо трябва да бъде дефинирано какво е анотация и кои елементи в Java кода могат да бъдат анотирани.

2.1. Какво е анотация?

Анотацията е механизъм за прикрепване на метаданни към даден клас и/или елемент на клас.

За да се използва една анотация, тя първо трябва да бъде дефинирана. Анотацията се дефинира като стандартен Java клас. За целта се използва ключовата дума @interface предхождаща името на класа на анотацията. Примера по-долу илюстрира дефиницията на анотация MyAnnotation:

```
public @interface MyAnnotation {
    String id();
    String name() default "unassigned";
    String someText() default "empty";
}
```

В класа MyAnnotation са дефинирани полетата id, name и someText. Те се наричат атрибути на анотацията. Всяко от тези полета може да приема стойност по подразбиране, която да му бъде присвоена, ако полето не бъде специфицирано, когато се използва анотацията (в примера полетата name и someText приемат стойност по подразбиране съответно “unassigned” и “empty”).

2.2. Елементи на клас, които могат да бъдат анотирани

В платформата JDK 1.5 има вградени анотации, които се намират в пакета java.lang.annotation. Някои от тях се използват при дефиниране на анотации, за да бъдат описани техните свойства. Ще бъдат разгледани вградените анотации @Retention и @Target, които имат отношение към естеството на дипломната работа.

2.2.1. 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. 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 MyAnnotation {
      String id();
      String name() default "unassigned";
      String someText() default "empty";
}
```

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 байткода на компилирания клас (когато компилатора на Java генерира байткода 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 байткода 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 байткода на класа, в който са използвани и съответно ще бъдат достъпни през Reflection API.

Въз основа на горе казаното, може да се приеме, че програмното решение, което е предмет на дипломата работа, ще разшири функциите на Reflection API, тъй като ще предоставя не само метаданните на анотации анотирани с @Retention (RetentionPolicy.RUNTIME), а също така и метаданните на такива, които са анотирани с @Retention (RetentionPolicy.CLASS).

2.2.2. Вградена анотация @Target

Анотацията @Target определя към какви елементи може да бъде прикрепена анотираната с нея анотация. Тези елементи са 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. 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.

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 MyAnnotation {
      String id();
      String name() default "unassigned";
      String someText() default "empty";
}
```

- 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 (разгледаните анотациите @Retention и @Target могат да бъдат прикрепени само към декларации на анотации, тъй като горе споменатите, по спецификация, са анотирани само с @Target (ElementType.ANNOTATION_TYPE)).

2.3. 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 [3]. 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.

2.3.1. 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
     }
}
```

2.3.2. 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
     }
}
```

2.3.3. 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
        }
    }
```

2.3.4. 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
    }
}
```

2.3.5. 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
        }
}
```

2.3.6. 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. Това е и предмета на настоящата глава, а именно под каква форма може да се среща клас файла и каква е неговата структурата [1], [2], [9], [11].

3.1. Структура (формат) на клас файл

Структурата на клас файла, който се разпознава от JVM е дефиниран подробно в спецификацията на виртуалната машина “The Java™ Virtual Machine Specification, Second Edition” [2].

Използвайки спецификацията, структурата на клас файла ще бъде описана, като се използва псевдокод подобен на C (C-like structure notation). С негова помощ ще бъдат дефинирани структури, които описват неговото съдържание. Елементите на тези структури ще бъдат записани в двоичното представяне на клас, в реда, в който са декларирани. Терминът таблица, който ще бъде използван по-долу, означава последователност от един или повече елемента от различен тип. Таблиците ще бъдат представяни като масиви, чрез псевдо кода, като допълнително ще бъде описана структурата на всяка таблица.

Всеки клас файл съдържа дефиниция на отделен клас или интерфейс. Той представлява последователност от порции данни по 8 бита (1 байт). За да се прочете тип, който е 16, 32 или 64 бита трябва да бъдат прочетени последователно съответно 2, 4 или 8 байта, като старшите битове винаги са на по-предни позиции. Например, ако трябва да бъде прочетен тип short, който е изграден от 2 байта, то трябва да се прочетат последователно 2 байта, от предварително известна позиция в клас файла, като първия байт ще съдържа по-старшите битове. Това представяне е известно още като “big-endian order”. Числовите типове, които се използват в представянето на клас файла са без знак и са изградени от 1, 2 или 4 байта. Бележим ги съответно с u1, u2 и u4. Като използва-

ме тези номенклатури на типовете, структурата на клас файла изглежда по следния начин:

3.1.1. Клас структура

```
ClassFile {
   u4 magic;
   u2 minor_version;
   u2 major_version;
   u2 constant_pool_count;
   cp_info constant_pool[constant_pool_count-1];
   u2 access_flags;
   u2 this_class;
   u2 super_class;
   u2 interfaces_count;
   u2 interfaces[interfaces_count];
   u2 fields_count;
   field_info fields[fields_count];
   u2 methods_count;
   method_info methods[methods_count];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Елементите на клас файл структурата са както следва:

- magic - числото 0xCAFEBABE идентифицира клас файл формата. Всеки валиден клас файл трябва да започва с него.
- minor_version, major_version - Поради различните версии на JDK и съответно компилаторите към тях е важно да се знае с коя версия на компилатора е генериран дадения клас файл. Чрез тези две числа се определя версията. Например, ако имаме minor_version = 2 и major_version = 0, тогава версията ще бъде 2.0. Това позволява версиите да могат да бъдат наредени лексикографски 1.5 < 2.0 < 2.1.
- constant_pool_count - Задава броя на елементите в constant_pool таблицата.
- constant_pool[] – Таблица от структури (вж. constant_pool), които представят различни константи. Те могат да бъдат реферирани, както от структурите представящи самия клас, така и от структурите представящи неговите елементи. Формата на всеки елемент от таблицата се определя от

неговия първи байт т.е. първият байт служи за определяне на типа на елемента.

- access_flags – Маска от стойности, които определят атрибутите на класа или интерфейса. Интерпретацията на всеки флаг се определя от следната таблица:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; may not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic ; Not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

Таблица 3.1.1: Интерпретация на атрибутите на клас или интерфейс, които са записани в байткода

- this_class – Индекс във constant_pool[] таблицата, който реферира CONSTANT_Class_info структура, представяща класа или интерфейса дефиниран от този клас файл.
- super_class - Индекс в constant_pool[] таблицата, който реферира CONSTANT_Class_info структура, представяща директния супер клас или интерфейс на класа или интерфейса дефиниран от този клас файл. Ако стойността е 0, тогава клас файла представя java.lang.Object класа. Последният е единствения клас в Java, който няма супер клас.
- interfaces_count – Задава броя на директните интерфейси, които има класа, а от там и броя на елементите в interfaces[] масива.
- interfaces[] – Масив с индекси в constant_pool[] таблицата. Всеки от тях реферира CONSTANT_Class_info структура, представяща директен интерфейс на класа.

- fields_count - Задава броя на полетата, които има класа (без тези от супер класовете), а от там и броя на елементите в масива fields[ ].
- fields[] – Масив с field_info структури, които изцяло описват полетата на класа.
- methods_count - Задава броя на методите, които има класа (без наследените от супер класовете), а от там и броя на елементите в methods [] масива.
- methods[] - Масив с method_info структури, които изцяло описват методите на класа.
- attributes_count - Задава броя на атрибутите, които има класа, а от там и броя на елементите в attributes [] масива.
- attribute[] - Масив с attribute_info структури, които изцяло описват атрибутите на класа. Именно, в този масив се пазят анотациите прикрепени към класа или към негови елементи.

В описанието на клас файла бяха използвани структурите - constant_pool, CONSTANT_Class_info, field_info, method_info и attribute_info, чиято дефиниция е следната:

3.1.2. Структура constant_pool

Всеки от елементите на constant_pool таблицата има следния формат:

```
cp_info {
   u1 tag;
   u1 info[];
}
```

Съгласно дефиницията всеки елемент започва с tag – байт, който определя типа на елемента записан в него. Съдържанието и дължината на масива info зависи от стойността на първия байт. Валидните стойности за елемента tag са изброени в следната таблица:

Constant Type	*Value*
CONSTANT_Class	7
CONSTANT_Fieldref	9

CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Таблица 3.1.2.1: Списък от константи, определящи типа на елементите записани в байткода

Имената на типовете на класове, полета, аргументи, връщани стойности от метод и локални променливи на методи се записват в таблицата constant_pool на клас файла. Те се придържат към определен формат, който е различен от този, в който се срещат в Java кода, дефиниран от Java Language Specification (JLS). Таблицата по-долу задава този формат:

BaseType **Character**	**Type**	**Interpretation**
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L<classname>;	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

Таблица 3.1.2.2: Интерпретация на символите, представящи типовете в Java, записани в байткода

Под <classname> (на 8-ми ред от таблицата) се има в предвид името на класа, заедно с пакета. Вместо със символа “.” отделните подстрингове на пакета са разделени със символа “/”.

Следващите примери имат за цел да онагледят информацията, която се съдържа в таблицата.

float – F

float[ ] – [F

float[ ][ ] – [[F

byte[ ] – [B

char – C

char [ ] [ ] [ ] – [[[C

com.fmi.diplom.SomeClass – Lcom/fmi/diplom/SomeClass;

com.fmi.diplom.SomeClass [ ] – [Lcom/fmi/diplom/SomeClass;

java.lang.String [ ] - [Ljava/lang/String;

3.1.3. Структура CONSTANT_Class_info

Структурата се използва за представяне на име на клас или интерфейс. Тя се дефинира по следния начин:

```
CONSTANT_Class_info {
          u1 tag;
          u2 name_index;
}
```

Стойността на tag е константата 7, която съответства на стойността CONSTANT_Class, описана в таблица 3.1.2.1. Елемента name_index е индекс в constant_pool таблицата, на който е записано името на класа или интерфейса, съгласно формата дефиниран в таблица 3.1.2.2**.**

3.1.4. Структура field_info

Структурата се използва за представяне на поле на класа. Тя се дефинира по следния начин:

```
field_info {
         u2 access_flags;
```

```
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

Елементите на field_info структурата са следните:

- access_flags - Маска от стойности, които определят правилата за достъп до полето. Интерпретацията на всеки флаг се определя от следната таблица:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared `private`; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared `protected`; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared `static`.
ACC_FINAL	0x0010	Declared `final`; no further assignment after initialization.
ACC_VOLATILE	0x0040	Declared `volatile`; cannot be cached.
ACC_TRANSIENT	0x0080	Declared `transient`; not written or read by a persistent object manager.
ACC_ENUM	0x4000	Declared as an element of an `enum`.

Таблица 3.1.4: Интерпретация на атрибутите на поле на клас, които се записват в байткода

- name_index – Индекс в таблицата constant_pool, където е записано името на полето, под формата на Java идентификатор (същото име, с което полето фигурира в съответния Java код).
- descriptor_index - Индекс в таблицата constant_pool, където е записан типа на полето, съгласно формата дефиниран в таблица 3.1.2.2.
- attributes_count - Задава броя на атрибутите на полето, а от там и броя на елементите в attributes [] масива

- attributes[] - Масив с attribute_info структури, които изцяло описват атрибутите на полето. Точно в този масив се пазят анотациите прикрепени към полето.

3.1.5. Структура method_info

Структурата се използва за представяне на метод на клас. Тя се дефинира по следния начин:

```
method_info {
          u2 access_flags;
          u2 name_index;
          u2 descriptor_index;
          u2 attributes_count;
          attribute_info attributes[attributes_count];
}
```

Елементите на method_info структурата са следните:

- access_flags - Маска от стойности, които определят правилата за достъп до метода. Интерпретацията на всеки флаг се определя от следната таблица:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared `private`; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared `protected`; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared `static`.
ACC_FINAL	0x0010	Declared `final`; may not be overridden.
ACC_SYNCHRONIZED	0x0020	Declared `synchronized`; invocation is wrapped in a monitor lock.
ACC_BRIDGE	0x0040	`A bridge method, generated by the compiler.`
ACC_VARARGS	0x0080	`Declared with variable number of arguments.`
ACC_NATIVE	0x0100	Declared `native`; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared `abstract`; no implementation is provided.
ACC_STRICT	0x0800	Declared `strictfp`; floating-point mode is FP-strict

Таблица 3.1.5: Интерпретация на атрибутите на метод или конструктор на клас, които се записват в байткода

- name_index – Индекс в таблицата constant_pool, където е записано името на метода под формата на Java идентификатор (същото име, с което полето фигурира в съответния Java код) или някоя от стойностите за специален метод - <init> и <clinit>.
- descriptor_index - Индекс в таблицата constant_pool, където е записан дескриптора на метода. Дескриптора на метода се задава с граматиката MethodDescriptor: (ParameterDescriptor*) ReturnDescriptor, където ParameterDescriptor и ReturnDescriptor са съответно типовете на аргументите и връщания от метода резултат. Тези типове се форматират съгласно таблица 3.1.2.2**.** Например, дескриптора на метода “Object myMethod(String arg1, double [ ] arg2)” има вида – (Ljava/lang/String[D)Ljava/lang/Object;
- attributes_count - Задава броя на атрибутите на метода, а от там и броя на елементите в attributes [] масива
- attributes[] - Масив с attribute_info структури, които изцяло описват атрибутите на метода. Именно, в този масив се пазят анотациите прикрепени към метода.

3.1.6. Структура attribute_info

Структурата се използва за представяне на атрибути на клас, метод или поле. Тя се дефинира по следния начин:

```
attribute_info {
            u2 attribute_name_index;
            u4 attribute_length;
            u1 info[attribute_length];
}
```

Елементите на method_info структурата са следните:

- attribute_name_index – Индекс в таблицата constant_pool, където е записано името на атрибута.

- attribute_length – Дължина в байтове на информацията, носена от дадения атрибут. В тази дължина не се включват шестте байта, заети от attribute_name_index и attribute_length.
- info - Информацията носена от дадения атрибут

3.1.7. Атрибут Signature - носител на анотациите в клас файл формата

Някои атрибути са дефинирани като част от спецификацията за клас файл формата. Такъв атрибута е “Signature”, чрез който се реализира представянето на анотациите в клас файла. За него е характерно, че стойността записана в constant_pool таблицата на индекс attribute_name_index е низа “Signature”, а стойността записана в attribute_length елемента е числото 2. Елемента info на този вид атрибути носи информацията свързана с анотацията, която представят.

3.2. 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 [3].

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 разработката 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”.

3.3. Анализ на съществуващи решения за четене на Bytecode.

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

- Не са пригодени за работа с байткода 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 байткод.

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].

3.3.1. BCEL - 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 байткода, 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.

3.3.2. 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 байткода: от помащабните структури, като полета на клас, до структурите от най-ниско ниво - отделни инструкции, които изграждат кода на методи.

3.3.3. ASM - Java bytecode manipulation framework

ASM е библиотека за манипулация на байткод. 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.

Основните характеристики на разгледаните съществуващи решения са описани в таблица 3.3.

Характеристика / Име	BCEL	ASM	SERP
Поддръжка на J2SE5.0 Annotations	не	да	не
Допълнително натоварване причинено от библиотеката по време на зареждането на клас	700%	60%	1100%
Големина на библиотеката	350KB	33KB	150KB

Таблицата 3.3: Характеристики на разгледаните съществуващи решения

Както се забелязва от описанието на тези решения, те са общи и са насочени към четене и модификацията на байткод. Те нямат функционалност тясно специализирана за четене на анотации.

Архитектурата на разработената система ще бъде модулна, за да може да се адаптира лесно, без промяна на код, към произволен инструмент от ниско ниво за четене на Bytecode, подобен на BCEL, SERP и ASM.

4. Проектиране на решението

Както вече видяхме в точка 3 (Анализ на проблемната област и определяне на входните изисквания), формата, в който се пазят типовете на данните в байткода съвпада с формат на вътрешното представяне на типовете в JVM, описан в таблица 3.1.2.2. Характерното за това представяне е, че то няма за цел да бъде лесно и интуитивно възприемано от разработчиците, които използват езика за програмиране Java. Изборът на това представяне, при реализацията на JVM, е направен с цел по-бързо и еднозначно определяне на някои основни характеристики на типа на данните (дали са прости типове или клас, дали е масив и колко е размерността му и т.н.). Част от целите на настоящото решение е да бъдат конвертирани типовете, които се извличат от байткода до типовете, които се използват в програмния код на езика Java. Това налага използването на два различни модела за представяне на извлечените от байткода данни: Модел на данните от ниско ниво, в който данните се представят в байткода и модел на данните от високо, в който данните се представят в езика Java. Модела на данните от високо ниво се изгражда на базата на модела на данните от ниско ниво, като се правят необходимите трансформации. Структурите и на двата модела са изградени по аналог на вградения в Java програмен интерфейс Reflection API, който е утвърдил себе си, с това, че е интуитивен и съответно изключително лесен за използване.

4.1. Модел на данните от ниско ниво

Моделът на данните от ниско ниво представлява обектно ориентирана структура. Целта му е да представя релацията между прочетените от байткода елементи (класове, методи, конструктори, полета и съответните им анотации) като всички данни, които съдържа са в суровият вид, в който са прочетени. Интерфейсите на модела се намират в пакета com.fmi.bytecode.annotations.input.* на реализацията. Йерархията на използваните интерфейси в този пакет е илюстрирана с Фиг. 1:

Фиг. 1: Клас диаграма представяща модела на данните от ниско ниво

4.1.1. Интерфейс AnnotationModel

Интерфейсът AnnotationModel моделира отделна анотация, прочетена от байткода, която може да принадлежи на клас, метод, поле или параметър на метод. Предоставя следните операции:

- String getTypeName() – Връща низ, който представлява JVM представянето на името на анотацията. От своя страна JVM представянето на името на анотацията е името на нейния клас, съгласно формата дефиниран в таблица 3.1.2.2. Например за анотацията @EJB, върнатата стойност ще бъде “Ljavax.ejb.EJB;”.
- boolean isRuntimeVisible() – Връща булева стойност true, ако анотацията е анотирана с @Retention (RetentionPolicy.RUNTIME) и false, ако е анотирана с @Retention (RetentionPolicy.CLASS) – вж. точка 2.2.1 (Вградена анотация @Retention).
- Map<String, NamedMemberModel> getNamedMembersMap() – Връща java.util.Map структура, съдържаща атрибутите на анотацията, ако има такива. В противен случай, връща празна java.util.Map структура. Ключовете в структурата са имената на атрибутите, а стойностите са обекти, реализиращи интерфейса NamedMemberModel, дефиниран по-долу.

4.1.2. Интерфейс NamedMemberModel

Интерфейсът NamedMemberModel моделира отделен атрибут на анотацията, прочетен от байткода. Предоставя следните операции:

- String getName() – Връща низ, който представлява името на атрибута на анотацията. Например, за анотацията @EJB (beanName=" TheNameOfTheReferedBean"), върнатата стойност за атрибута beanName ще бъде “beanName”.
- Object getMemberValue() – връща обект, който представлява стойността на атрибута на анотацията. Типа на върнатия обект се определя от константата, която връща метода int getMemberTag(), описан по-долу. Например, за анотацията @EJB (beanName="TheNameOfTheReferedBean"), върнатата стойност за атрибута

beanName ще бъде от тип String и ще има стойност “TheNameOfTheReferedBean”.

- int getMemberTag() – Връща константа, която представлява типа на атрибута на анотацията. Константите за тип са дефинирани като статични полета с публичен достъп в същия клас. Те имат аналогични стойности на дефинираните в таблица 3.1.2.2:

```
static final int TAG_BYTE = 'B';
static final int TAG_CHAR = 'C';
static final int TAG_DOUBLE = 'D';
static final int TAG_FLOAT = 'F';
static final int TAG_INT = 'I';
static final int TAG_LONG = 'J';
static final int TAG_SHORT = 'S';
static final int TAG_BOOLEAN = 'Z';
static final int TAG_STRING = 's';
static final int TAG_ENUM = 'e';
static final int TAG_CLASS = 'c';
static final int TAG_ANNOTATION = '@';
static final int TAG_ARRAY = '[';
static final int TAG_REF = 'L';
```

Например, при анотацията @EJB (beanName = ”TheNameOfTheReferedBean"), върнатата стойност ще бъде число, представящо уникода на символа ‘s’, съответстващ на типа String.

4.1.3. Интерфейс EnumTypeModel

Интерфейсът EnumTypeModel моделира стойност на атрибут на анотацията, който е от тип enum - стандартен тип в езика Java, въведен с версия 1.5. Предоставя следните операции:

- String getEnumerationTypeName() – Връща низ, който представлява JVM представянето на името на enum типа на атрибута на анотацията. От своя страна, този тип е името на класа на enum, съгласно формата дефиниран в таблица 3.1.2.2. Например, за атрибута по подразбиране с име “value” на анотацията @TransactionAttribute (TransactionAttributeType.REQUIRED) или @TransactionAttribute (value=TransactionAttributeType.REQUIRED), върнатата стойност ще бъде “Ljavax.ejb.TransactionAttributeType;”.
- String getEnumerationLiteral() – Връща низ, който представлява стойността на атрибута на анотацията. В горния пример - @TransactionAttribute (TransactionAttributeType.REQUIRED), върнатата стойност ще бъде “REQUIRED”.

4.1.4. Интерфейс AnnotatableElementModel

Интерфейсът AnnotatableElementModel моделира всеки елемент в Java, който може да бъде анотиран. В частност, това може да бъде клас, поле, метод, конструктор или аргумент на метод. Предоставя следните операции:

- String getName() – Връща низ, който представлява името на елемента. В случай, че този елемент е клас, връща JVM представянето на името на класа.
- int getAccessFlags() – Връща маска от флаговете, определяща правилата за достъп. В зависимост от това, дали елемента е клас, поле или метод/конструктор, маските се определят съгласно дефинираните флагове на елементите в таблиците съответно 3.1.1, 3.1.4 и 3.1.5.
- Map<String, AnnotationModel> getAnnotations() - Връща java.util.Map структура, съдържаща анотацията на елемента, ако има такива. В противен случай връща празна java.util.Map структура. Ключовете в структурата са имената на класовете на анотациите, съгласно таблица 3.1.2.2, а стойностите са обекти реализиращи интерфейса AnnotationModel, дефиниран по-горе.

4.1.5. Интерфейс FieldModel

Интерфейсът FieldModel моделира поле на клас, в това число и статичните полета. Той е наследник на AnnotatableElementModel и съответно наследява и операциите му. Интерфейсът предоставя следните собствени операции:

- boolean isGeneric() – Връща булева стойност true, ако типа на полето е Generic, false в противен случай.
- String getRawType() - Връща низ, който представлява JVM представянето на типа на полето, съгласно формата дефиниран в таблица 3.1.2.2.
- String getSignature() - Връща низ, който представлява сигнатурата на Generic типа на полето, ако то е Generic и null в противен случай.

4.1.6. Интерфейс MethodModel

Интерфейсът MethodModel моделира метод на клас. Той е наследник на AnnotatableElementModel и съответно наследява операциите му. Интерфейсът предоставя следните собствени операции:

- String getRawReturnType() - Връща низ, който представлява JVM предстaвянето на типа на връщаната от метода стойност, съгласно формата дефиниран в таблица 3.1.2.2.
- String[] getRawParams() - Връща масив от низове, представляващи JVM представянето на типа на аргументите на метода, в реда, в който се срещат в сигнатурата на метода, съгласно формата дефиниран в таблица 3.1.2.2.
- String[] getExceptions() - Връща масив от низове, представляващи JVM представянето на декларираните изключения на метода, в реда, в който се срещат в сигнатурата на метода, съгласно формата дефиниран в таблица 3.1.2.2.
- Map<String,AnnotationModel>[] getParameterAnnotations() - Връща масив от java.util.Map структури, всяка от които съдържа анотациите на съответния параметър на метода, ако има такива. В противен случай връща

празна java.util.Map структура. Ключовете в структурата са имената на класовете на анотациите, съгласно таблица 3.1.2.2, а стойностите са обекти, реализиращи интерфейса AnnotationModel, дефиниран по-горе. Реда на структурите с анотациите в масива съвпада с реда на съответните им параметри в сигнатурата на метода.

4.1.7. Интерфейс ConstructorModel

Интерфейсът ConstructorModel моделира конструктор на клас. Той е наследник на MethodModel и съответно наследява операциите му. Интерфейсът не предоставя собствени операции. Характерното за него е, че извикването на getRawReturnType() винаги връща void. Също така, името върнато от наследения метод getName() винаги съвпада с името на класа, на който принадлежи конструктора. Това се дължи на спецификата на езика Java, по отношение на дефинирането на конструктори.

4.1.8. Интерфейс ClassModel

Интерфейсът ClassModel моделира клас. Той е наследник на AnnotatableElementModel и съответно наследява операциите му. Интерфейсът предоставя следните собствени операции:

- String getSuperClassName() - Връща низ, който представлява JVM представянето на името на супер класа, съгласно формата дефиниран в таблица 3.1.2.2.
- String getPackageName() - Връща низ, който представлява JVM представянето на пакета на класа.
- boolean isParametrised() – Връща булева стойност true, ако типа на класа е Generic и false в противен случай.
- boolean isAnnotation() - Връща булева стойност true, ако класа представя дефиниция на анотация и false в противен случай.
- boolean isInterface() - Връща булева стойност true, ако класа представя дефиниция на интерфейс, false в противен случай.
- boolean isEnum() - Връща булева стойност true, ако класа представя клас от тип enum, false в противен случай.

- String[] getInterfaceNames() - Връща масив от низове, който представлява JVM представянето на типа на интерфейсите, които са реализирани в класа. Имената им са съгласно формата дефиниран в таблица 3.1.2.2.
- MethodModel[] getMethods() - Връща масив от обекти, които са реализация на интерфейса MethodModel. Тези обекти са модел на методите на класа, в това число и статичните методи. Трябва да се отбележи, че в масива не се съдържат моделите на наследените методи от супер класовете.
- FieldModel[] getFields() - Връща масив от обекти, които са реализация на интерфейса FieldModel. Тези обекти са модел на полетата на класа, в това число и статичните полета. В масива не се съдържат моделите на наследените полета от супер класовете.
- ConstructorModel[] getConstructors() - Връща масив от обекти, които са реализация на интерфейса ConstructorModel. Тези обекти са модел на конструкторите на класа. Тук отново в масива не се съдържат моделите на наследените конструктори от супер класовете.

4.1.9. Интерфейс ByteCodeReader

Интерфейсът ByteCodeReader моделира четец от ниско ниво, който по зададен Bytecode на клас, под формата на поток от байтове, да създаде обект реализация на ClassModel. Интерфейсът предоставя следната операция:

- ClassModel parseClassInformation (InputStream bytecode) – връща обект, който е реализация на интерфейса ClassModel, по зададен като аргумент Bytecode на клас, под формата на поток от байтове.

4.1.10. Клас ByteCodeReaderFactory

Класа ByteCodeReaderFactory предоставя възможност за създаване на обекти, които реализират интерфейса ByteCodeReader, по зададено име на класа на реализацията. Този клас, заедно с интерфейса ByteCodeReader, придават модулна структура на разработката, като я правят независима от избора и реализацията на използвания четец на Bytecode от ниско ниво. Това се постига чрез дефинирането на междинен слой, за обмен на данни, между четеца на Bytecode от ниско ниво и реализацията на разработката. Този междинен слой е именно модела от ниско ниво, описан в тази глава.

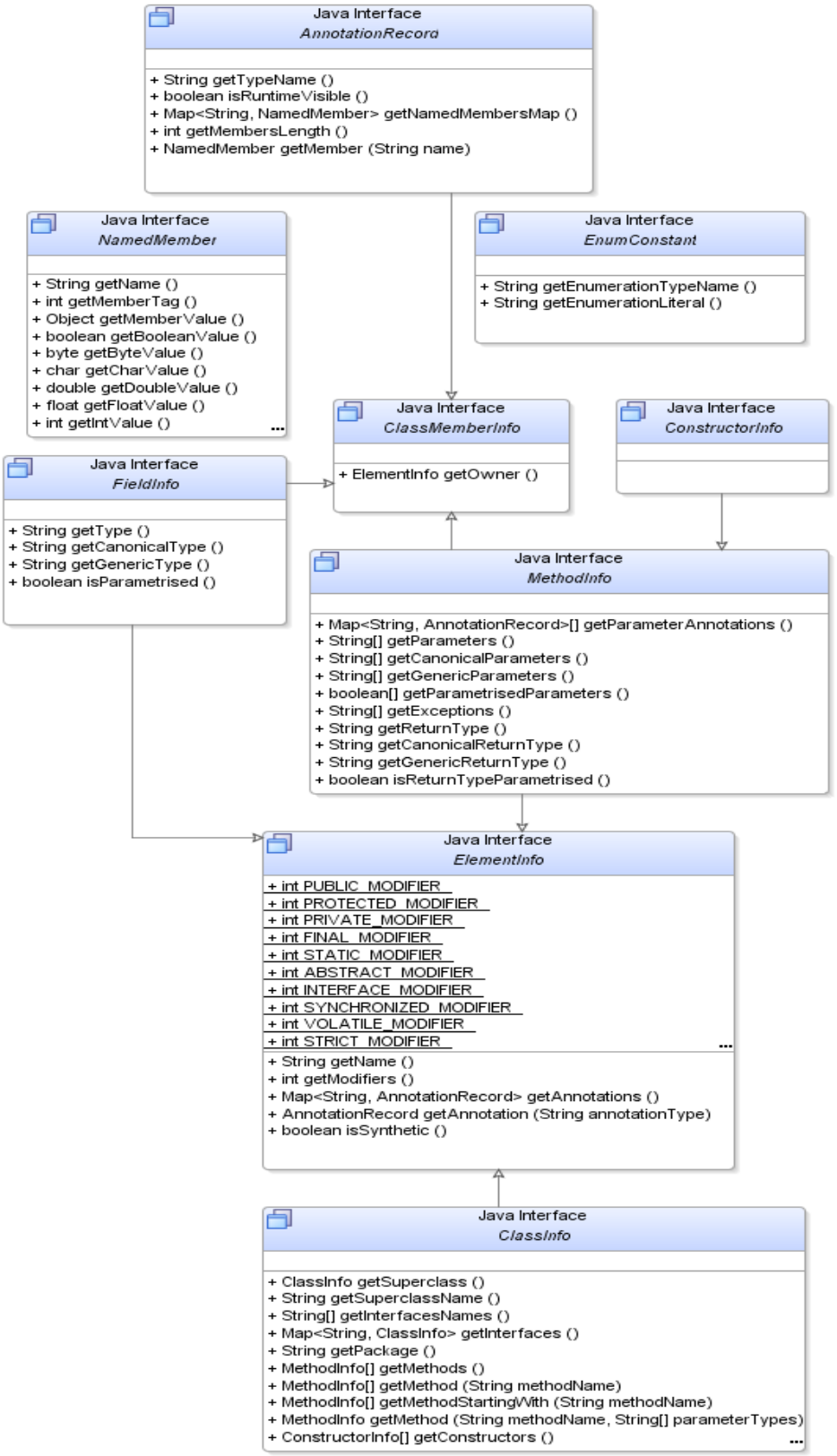
Класа предоставя следната операция за създаване на ByteCodeReader обект:

- public static final ByteCodeReader createByteCodeReader(String byteCodeReaderClassName) – връща обект, който е реализация на интерфейса ByteCodeReader, по зададено име на класа реализатор.

4.2. Модел на данните от високо ниво

Моделът на данните от високо ниво представлява обектно ориентирана структура, направена с цел да представя релацията между прочетените от байткода елементи (класове, методи, конструктори, полета и съответните им анотации). За разлика от модела от ниско ниво, описан в предходната точка, всички данни, които съдържа са във вида, в който се използват в програмния код на езика Java. Моделът от високо ниво има аналогична функционалност и структура като модела от ниско ниво. С оглед да бъде избегнато, където е възможно, дублирането на информацията от изложението по-горе, в следващите подточки ще бъдат описани само функциите, с които модела от високо ниво разширява този от ниско ниво.

Интерфейсите на модела се намират в пакета com.fmi.bytecode.annotations.element.* на реализацията. Йерархията между интерфейсите в този пакет се представя с Фиг. 2:



Фиг. 2: Клас диаграма представяща модела на данните от високо ниво

4.2.1. Интерфейс AnnotationRecord

Интерфейсът AnnotationRecord моделира отделна анотация на базата на AnnotationModel, която може да принадлежи на клас, метод, поле или параметър на метод. Интерфейсът AnnotationRecord притежава всички операции на AnnotationModel с тази разлика, че връщаните стойности, представящи типове, не са в JVM формата дефиниран в таблица 3.1.2.2, а са във вида, в който се използват в езика Java (специфициран в JLS). Интерфейсът предоставя и следните допълнителни операции:

- int getMembersLength() – Връща число, съответстващо на броя на атрибутите на анотацията.
- NamedMember getMember(String name) – Връща атрибут на анотацията, по зададено име и null, ако не съществува атрибут с това име.

4.2.2. Интерфейс EnumConstant

Интерфейсът EnumConstant моделира стойност на атрибут на анотацията, на базата на EnumTypeModel. Той притежава всички операции на EnumTypeModel с тази разлика, че връщаните стойности, представящи типове, не са в JVM формата дефиниран в таблица 3.1.2.2, а са във вида в който се използват в езика Java (специфициран в JLS). Интерфейсът не предоставя допълнителни операции.

4.2.3. Интерфейс NamedMember

Интерфейсът NamedMember моделира отделен атрибут на анотацията на базата на MemberModel. Той притежава всички операции на MemberModel с тази разлика, че връщаните стойности, представящи типове, не са в JVM формата, дефиниран в таблица 3.1.2.2, а са във вида, в който се използват в езика Java (специфициран в JLS). Интерфейсът предоставя и следните допълнителни операции, за достъп до стойността на атрибута, когато предварително е известен нейния тип:

- boolean getBooleanValue() – Връща boolean стойност, съответстваща на стойността на атрибута.
- byte getByteValue() – Връща byte стойност, съответстваща на стойността на атрибута.

- char getCharValue() – Връща char стойност, съответстваща на стойността на атрибута.
- double getDoubleValue() – Връща double стойност, съответстваща на стойността на атрибута.
- float getFloatValue() – Връща float стойност, съответстваща на стойността на атрибута.
- int getIntValue() – Връща int стойност, съответстваща на стойността на атрибута.
- long getLongValue() – Връща long стойност, съответстваща на стойността на атрибута.
- short getShortValue() – Връща short стойност, съответстваща на стойността на атрибута.
- String getStringValue() – Връща String стойност, съответстваща на стойността на атрибута.
- EnumConstant getEnumConstantValue() – връща обект, който е реализация на EnumConstant, съответстващ на стойността на атрибута, в случай че тази стойност е от тип enum.
- AnnotationRecord getAnnotationValue() – връща обект реализация на AnnotationRecord, съответстващ на стойността на атрибута (Използва се при вградени една в друга анотации - @EJBs ({@EJB (beanName="beanName1")).
- int getMemberArrayLength() – Връща броя на елементите на масива представящ стойността на атрибута (Използва се при стойности на атрибути, които са масиви).
- NamedMember[] getMemberArrayValue() – Връща масив от обекти, които са реализация на NamedMember (Използва се при стойности на атрибути, които са масиви).
- boolean isValueDefaulted() – Връща булева стойност true, ако стойността на атрибута е стойността му по подразбиране, специфицирана при дефинирането на анотацията, на която принадлежи.

4.2.4. Интерфейс ElementInfo

Интерфейсът ElementInfo моделира всеки елемент в Java който може да бъде анотиран на базата на AnnotatableElementModel. В частност, това може да бъде клас (наследникът ClassInfo), поле (наследникът FieldInfo), метод (наследникът MethodInfo), конструктор (наследникът ConstructorInfo) или аргумент на метод. Той притежава всички операции на AnnotatableElementModel с тази разлика, че връщаните стойности, представящи типове, не са в JVM формата дефиниран в таблица 3.1.2.2, а са във вида, в който се използват в езика Java (специфициран в JLS). Интерфейсът предоставя и следните допълнителни операции:

- AnnotationRecord getAnnotation(String annotationType) – връща обект, който е реализация на AnnotationRecord, по зададено име на анотация. Ако даденият елемент няма анотация с това име, върнатата стойност е null.
- boolean isSynthetic() – Връща булева стойност true, ако даденият елемент е поле, метод или конструктор, изгенериран от компилатора по време на компилация (Ако елемента не съществува в Java кода). Подобни елементи се генерират от компилатора със служебна цел, за нуждите на JVM.

4.2.5. Интерфейс ClassMemberInfo

Интерфейсът ClassMemberInfo моделира елемент на клас (поле, метод и конструктор). Предоставя следната операции:

- ElementInfo getOwner() – Връща базов обект, реализация на ElementInfo, който може да е FieldInfo, MethodInfo или ConstructorInfo.

4.2.6. Интерфейс FieldInfo

Интерфейсът FieldModel моделира поле на клас, в това число и статичните полета на базата на FieldModel. Той е наследник на ElementInfo и ClassMemberInfo и съответно наследява и операциите им. Интерфейсът предоставя следните собствени операции:

- public String getType() - Връща низ, който представлява типа на полето. Върнатата стойност е същата, както при извикване на метода Class.getName() от Reflection API.
- public String getCanonicalType() - Връща низ, който представлява типа на полето. Върнатата стойност е същата, както при извикване на метода Class. getCanonicalName() от Reflection API (JLS формата).
- public String getGenericType() – Връща низ, който представлява сигнатурата на Generic типа на полето, ако то е Generic или null в противен случай.
- public boolean isParametrised() – Връща булева стойност true, ако типа на полето е Generic, false в противен случай.

4.2.7. Интерфейс MethodInfo

Интерфейсът MethodModel моделира метод на клас на базата на MethodModel. Той е наследник на ElementInfo и ClassMemberInfo и съответно наследява техните операции. Интерфейсът предоставя следните собствени операции:

- public String[] getExceptions() - Връща масив от низове, представляващи JLS представянето на декларираните изключения на метода, в реда, в който се срещат в сигнатурата на метода.
- public Map<String,AnnotationRecord>[] getParameterAnnotations() - Връща масив от java.util.Map структури, всяка от които съдържа анотациите на съответния параметър на метода, ако има такива. В противен случай връща празна java.util.Map структура. Ключовете в структурата са имената на класовете на анотациите, съгласно JLS, а стойностите са обекти реализиращи интерфейса AnnotationRecord, дефиниран по-горе. Реда на структурите с анотациите в масива съвпада с реда на съответните им параметри в сигнатурата на метода.
- public String[] getParameters() – Връща масив от низове, представляващи типовете на параметрите, в реда, в който се срещат в сигнатурата на метода. Формата на имената на типовете са същите, както при извикване на метода Class. getName() от Reflection API.

- public String[] getCanonicalParameters() – Връща масив от низове, представляващи JLS представянето на типовете на параметрите, в реда, в който се срещат в сигнатурата на метода.
- public boolean[] getParametrisedParameters() – Връща масив от булеви стойности, които са true, ако съответните им параметри на метода са с Generic тип.
- public String[] getGenericParameters() – Връща масив от низове, които представляват сигнатурата на Generic типа на съответните параметри на метода. Ако съответния параметър не е с Generic тип, то стойността на този индекс е null.
- public String getReturnType() – Връща низ, представляващ типа на връщания резултат на метода. Формата на името на типа е същият, както при извикване на метода Class. getName() от Reflection API.
- public String getCanonicalReturnType() – Връща низ, представляващ JLS представянето на типа на връщания резултат на метода.
- public boolean isReturnTypeParametrised() – Връща булева стойност true, ако типа на връщания резултат на метода е Generic, false в противен случай.
- public String getGenericReturnType() – Връща низ, който представлява сигнатурата на Generic типа на връщания резултат на метода, ако той е Generic или null в противен случай.

4.2.8. Интерфейс ConstructorInfo

Интерфейсът ConstructorInfo моделира метод на клас на базата на ConstructorModel. Той е наследник на MethodInfo и съответно наследява неговите операции. Интерфейсът не предоставя собствени операции. Характерното за него е, че извикването на коя да е от наследените операции за взимане на връщаната стойност, дава резултат void. Също така, името върнато от наследения метод getName() винаги съвпада с името на класа, на който принадлежи конструктора. Това се дължи на спецификата на езика Java, по отношение на дефинирането на конструктори.

4.2.9. Интерфейс ClassInfo

Интерфейсът ClassInfo моделира клас на базата на ClassModel. Той притежава всички операции на ClassModel с тази разлика, че връщаните стойности, представящи типове, не са в JVM формата дефиниран в таблица 3.1.2.2, а са във вида, в който се използват в езика Java (специфициран в JLS). Интерфейсът ClassInfo е наследник на ElementInfo и съответно наследява неговите операции. Предоставя и следните допълнителни операции:

- public ClassInfo getSuperclass() – връща обект, който е реализация на ClassInfo и представя родителя на класа.
- public Map<String,ClassInfo> getInterfaces() – Връща java.util.Map структура, съдържаща ClassInfo обекти, представящи интерфейсите реализирани от класа, ако има такива. В противен случай връща празна java.util.Map структура. Ключовете в структурата са имената на интерфейсите, а стойностите са обекти реализиращи интерфейса ClassInfo.
- public MethodInfo[] getMethods() – Връща масив от обекти, които са реализация на интерфейса MethodInfo. Тези обекти са модел на методите на класа, в това число и статичните методи. Трябва да се отбележи, че в масива не се съдържат моделите на наследените методи от супер класовете.
- public MethodInfo[] getMethod(String methodName) – Връща масив от обекти, които са реализация на интерфейса MethodInfo. Тези обекти са модел на методите на класа с име <methodName>, в това число и статичните методи. В този масив не се съдържат моделите на наследените методи от супер класовете.
- public MethodInfo[] getMethodStartingWith(String methodName) – Връща масив от обекти, които са реализация на интерфейса MethodInfo. Тези обекти са модел на методите на класа, чието име започва с <methodName>, в това число и статичните методи. В масива не се съдържат моделите на наследените методи от супер класовете.
- MethodInfo getMethod(String methodName, String[] parameterTypes) – връща обект, който е реализация на интерфейса MethodInfo. Този обект е модел на метода на класа, чието име е <methodName> и аргументите

му са <parameterTypes>. Ако в класа няма метод отговарящ на тези критерии се връща стойност null.

- ConstructorInfo[] getConstructors() – Връща масив от обекти, които са реализация на интерфейса ConstructorInfo. Тези обекти са модел на конструкторите. В масива не се съдържат моделите на наследените конструктори от супер класовете.
- ConstructorInfo getConstructor(String[] parameterTypes) – връща обект, който е реализация на интерфейса ConstructorInfo. Този обекти е модел на конструктора на класа, чийто параметри са <parameterTypes>. Ако в класа няма конструктор отговарящ на този критерий се връща стойност null.
- FieldInfo[] getFields() – Връща масив от обекти, които са реализация на интерфейса FieldInfo. Тези обекти са модел на полетата на класа. В масива не се съдържат наследените полетата от супер класовете.
- FieldInfo getField(String fieldName) – връща обект, който е реализация на интерфейса FieldInfo. Този обект е модел на поле на класа, чието име е < fieldName >. Ако в класа няма поле с такова име се връща стойност null.
- boolean isAnonymousClass() – Връща булева стойност true, ако класа е анонимен.
- FileInfo[] getContainingFiles() – Връща масив от обекти, които са реализация на FileInfo интерфейса, описан в следващата точка. Това са всички анализирани входни файлове, които съдържат дадения клас.

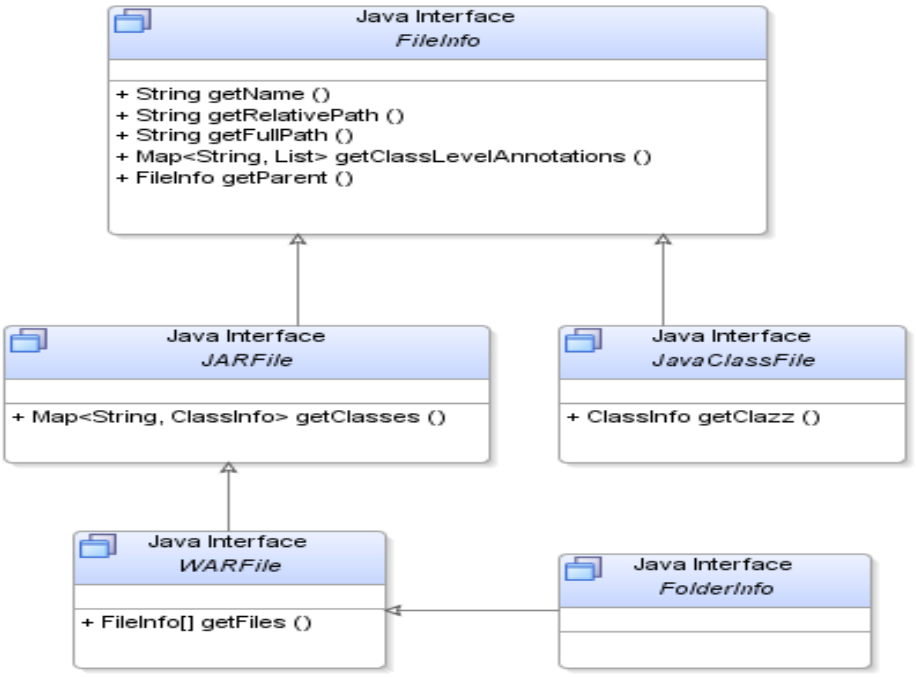
4.3. Модел на входните данни от файловата система

В предходната глава (Анализ на проблемната област и определяне на входните изисквания) е дефиниран възможния формат на входните данни, съдържащи клас файлове. Имаме 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”.

Този входен формат е реализиран чрез моделиране на изброените елементи. Те могат да бъдат класифицирани най-общо като файлове. Поради тази причина, всички те наследяват общ интерфейс FileInfo, в който са изнесени всички общи операции. Интерфейсите на модела се намират в пакета com.fmi.bytecode.annotations.file.* на реализацията. Следва описанието на тези интерфейси, както и UML клас диаграма представяща йерархията им (Фиг. 3):



Фиг. 3: Клас диаграма представяща модела на входните данни

4.3.1. Интерфейс FileInfo

Интерфейсът FileInfo е базов интерфейс, който моделира всички изброени конкретни представители на входен файл – клас файл, директория, jar файл и war файл. Наследява се от съответните интерфейси – JavaClassFile, FolderInfo, JARFile и WARFile и предоставя следните операции:

- String getName() – Връща низ, който представлява името на файла (без пътя до него).
- String getRelativePath() – Връща низ, който представлява името на файла, заедно с относителният му път. Относителнят път от своя страна е спрямо архива, на който принадлежи файла, ако има такъв, или спрямо някоя от началните директории, подадена като вход на програмата. Формата на върнатата стойност е както следва:
  1. За файлове намиращи се в war или jar файл - <име на архива>!/<път в архива>/<име на файла>.

2. За файлове намиращи се в директория - /<относителен път спрямо началната директория подадена на програмата>/<име на файла>.
3. За файлове, които са подадени директно на програмата – null.

- String getFullPath() – Връща низ, който представлява абсолютния път на файла във файловата система, заедно с името му.
  1. За файлове намиращи се в war или jar файл - <абсолютния път на архива>!/<път в архива>/<име на файла>.
  2. За файлове или директории - /<абсолютния път на файла>/<име на файла>.
- Map<String, List<AnnotationRecord>> getClassLevelAnnotations() – Връща всички намерени анотации, с които са анотирани класовете в текущия файл. Ключа е името на анотацията, а стойността е списък от обекти, реализация на AnnotationRecord, които представят всички намерени анотации с това име.
- FileInfo getParent() – връща обект, реализация на FileInfo, който представя jar/war файла или директорията, която съдържат този файл. Ако този метод се извика директно на входен файл (подаден на програмата), ще се върне стойност null.

4.3.2. Интерфейс JavaClassFile

Интерфейсът JavaClassFile представя конкретен клас файл. Той е наследник на интерфейса FileInfo и съответно наследява всички негови операции. Интерфейсът предоставя и следните собствени операции:

- ClassInfo getClazz() – връща обект, който е реализация на интерфейса ClassInfo, описан подробно по-горе.

4.3.3. Интерфейс JARFile

Интерфейсът JARFile представя конкретен jar файл. Той е наследник на интерфейса FileInfo и съответно наследява всички негови операции. Интерфейсът предоставя и следните собствени операции:

- Map<String,ClassInfo> getClasses() – Връща структура с обекти, които са реализация на ClassInfo. Тези обекти представят клас файловете,

които се съдържат в архива. Ключа в структурата е името на класа, а стойността е обект, реазлизация на ClassInfo, представящ този клас.

4.3.4. Интерфейс WARFile

Интерфейсът WARFile представя конкретен war файл. Той е наследник на интерфейса JARFile и съответно наследява всички негови операции. Интерфейсът предоставя и следните собствени операции:

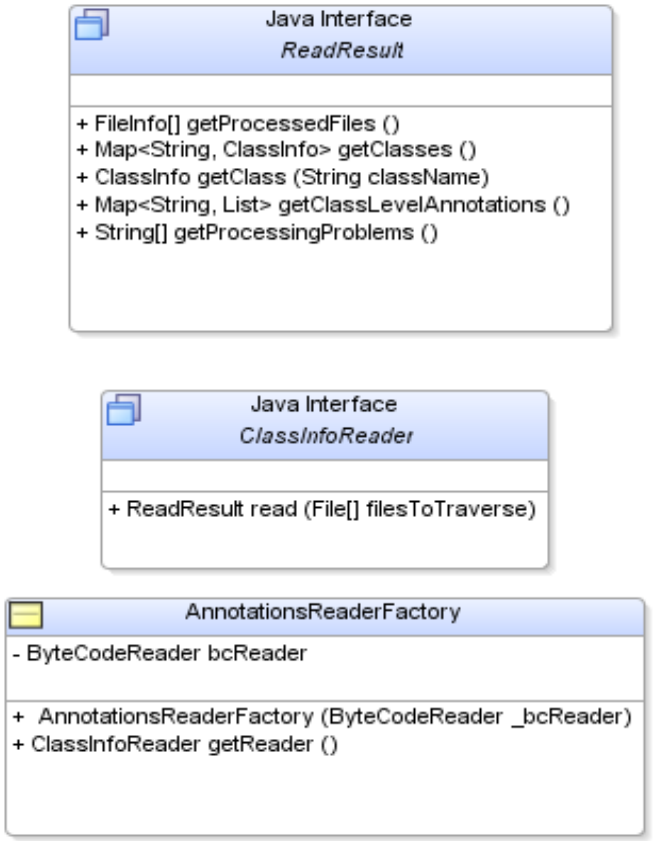
- FileInfo[] getFiles() – Връща списък от обекти, които са реализация на FileInfo и представят всички намерени в архива клас файлове, jar файлове и директории, съответно като конкретни реализации на JavaClassFile, JARFile и FolderInfo интерфейсите.

4.3.5. Интерфейс FolderInfo

Интерфейсът JavaClassFile представя конкретна директория. Той е наследник на интерфейса WARFile и съответно наследява всички негови операции. Интерфейсът не предоставя собствени операции.

4.4. Изходящ модел за работа с реализацията

Проектирането на системата завършва с модел, чрез който се представя интерфейс за работа с нея. Това е входната точка на реализацията. С негова помощ, от една страна може да се специфицират входните файлове, чиято мета-информация трябва да бъде извлечена, а от друга да се получи изходящ модел на мета-информацията, представящ резултата от обработката им. Интерфейсите и класовете на изходящия модела се намират в пакета com.fmi.bytecode.annotations.tool.* на реализацията. Следва описанието на тези класове и интерфейси, както и UML клас диаграма представяща йерархията им (Фиг. 4):



Фиг. 4: Клас диаграма представяща изходящия модел

4.4.1. Интерфейс ReadResult

Интерфейсът ReadResult представя изходящия модел на мета-информацията, който е получен в резултат на обработката на всички входни файлове. Интерфейсът предоставя следните операции:

- FileInfo[] getProcessedFiles() – Връща масив, съдържащ обекти, които са реализация на FileInfo. Те представляват всички входни файлове, които са обработени от реализацията, за да бъде получен ReadResult обекта.
- ClassInfo getClass(String className) – връща обект, който е реализация на ClassInfo и представлява модела от високо ниво за класа с име

<className>, подадено като аргумент. Ако в резултата няма информация за този клас, върнатата стойност ще бъде null.

- Map<String, List<AnnotationRecord>> getClassLevelAnnotations() – Връща всички намерени анотации, с които са анотирани класовете във входните файлове. Ключа е името на анотацията, а стойността е списък от обекти, реализация на AnnotationRecord, които представят всички намерени анотации с това име. Ако никой от класовете, съдържащи се във входящите файлове, не съдържа анотация на ниво клас, върнатата стойност ще е празна Map структура.
- String[] getProcessingProblems() – Връща масив от низове, които представляват съобщения за грешки, възникнали по време на обработката. Тези грешки могат да се дължат на срещането на един и същ клас файл с еднаква версия, в различни архивни файлове посочени като вход.

4.4.2. Интерфейс ClassInfoReader

Интерфейсът ClassInfoReader представя модел на четецът на мета-информация - по зададени входни файлове той изгражда изходящият и модел. Предоставя следната операция:

- ReadResult read(File[] filesToTraverse) throws ReadingException – връща обект, реализация на ReadResult, по зададен списък от входни файлове за обработка. Тези файлове могат да бъдат клас файлове, директории, jar файлове, war файлове или произволна комбинация от изброените. При възникване на грешка (обработка на невалиден клас файл или срещането на клас файл на няколко места във входните файлове, но с различна версия) ще бъде хвърлено изключението ReadingException.

4.4.3. Клас AnnotationsReaderFactory

Класа AnnotationsReaderFactory е входната точка на реализацията. Той предоставя възможност за създаване на ClassInfoReader, чрез който може да бъде обработен произволен набор от входни файлове. Това става чрез следната статична операция:

ClassInfoReader getReader(ByteCodeReader bcReader) – връща обект, който е реализация на ClassInfoReader, по зададен като аргумент ByteCodeReader (с помощта на bcReader, се създава модела от ниско ниво на всеки клас файл, намерен при обработка на входните файлове).

5. Описание на реализацията

В тази глава е описана реализацията на разработката. Тя се изразява, от една страна в реализирането на интерфейсите, описани в предходната глава (Проектиране на реализацията), а от друга, в разработката на допълнителни помощни класове, имащи за цел да капсулират логиката, използвана на повече от едно място в реализацията, за да се избегне дублирането на код.

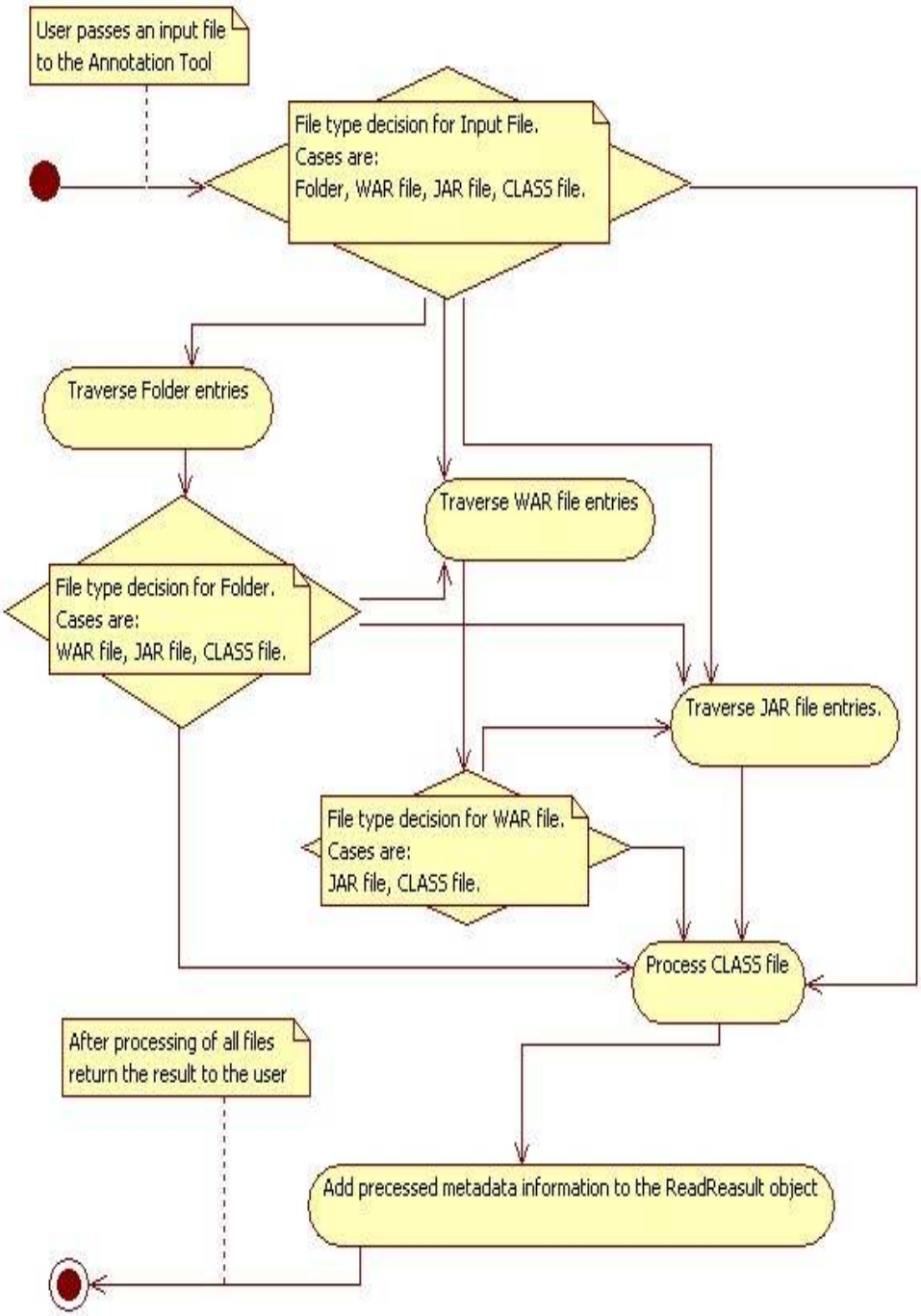
Съдържанието на главата е организирано по Java пакети на реализираните класове, с цел изложението да върви последователно по отделните логически и функционални компоненти на разработката.

5.1. Архитектура на системата

В тази глава е представена цялостната архитектура на системата (Фиг. 5). Тя е изградена от 4 основни модула:

- Модул за обхождане на входните файлове, имащ за цел да извлече всички клас файлове, които се съдържат директно във входните файлове или в техни поддиректории и архиви (Фиг. 6).
- Модул за четене на байткода на клас файл и създаване на неговия модел от ниско ниво.
- Модул за трансформация на модела от ниско ниво към модел от високо ниво.
- Модул за складиране и структуриране на моделите от високо ниво, на обработениете клас файлове, в изходящия модел.

Фиг. 5: Архитектура на системата за извличане и структуриране на мета-информация

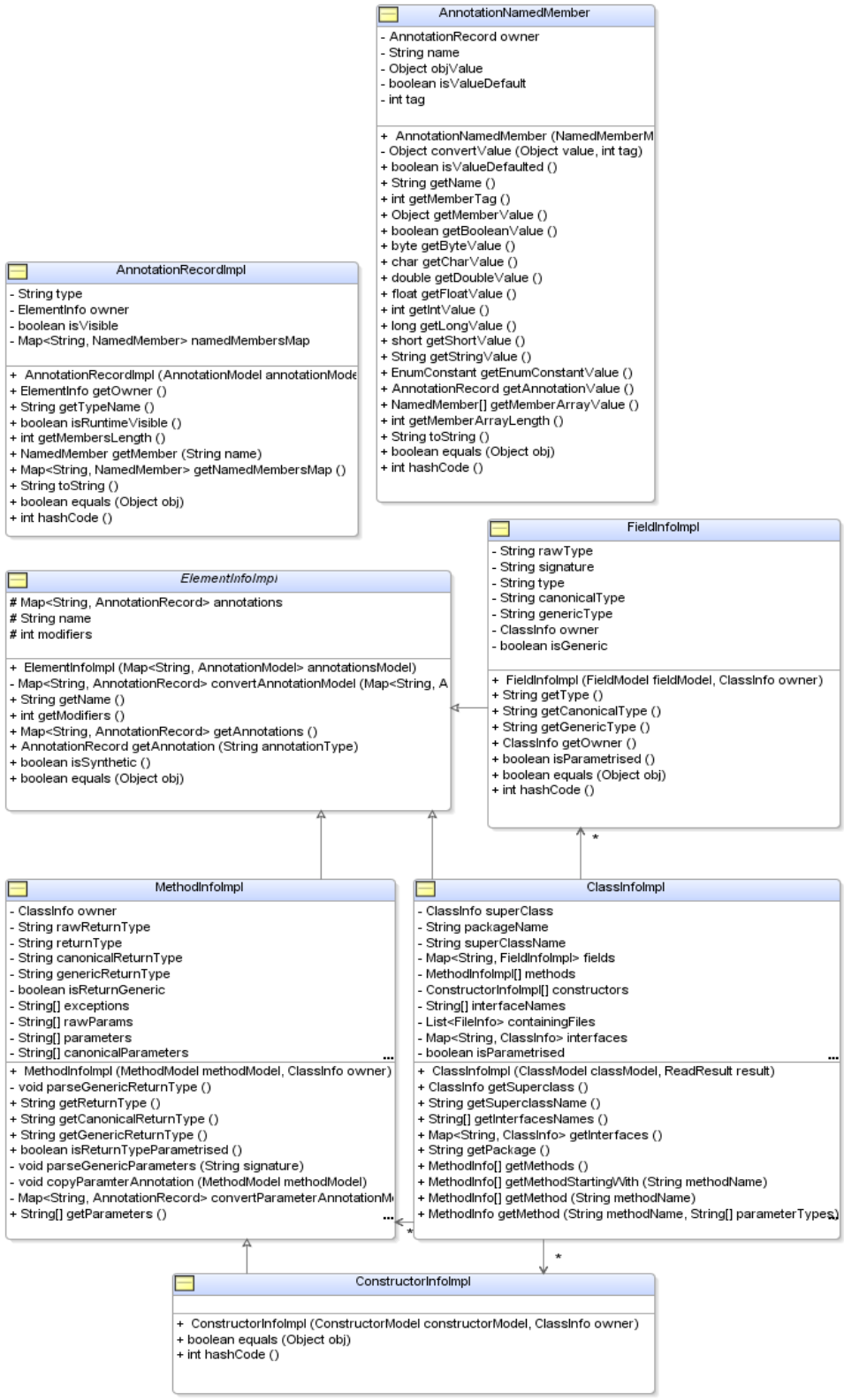


Фиг. 6: Диаграма на работен процес (UML Activity Diagram) на модула за обхождане на входните файлове

*5.2. Пакет com.fmi.bytecode.annotations.element.impl.**

В този пакет се намират всички класове, които реализират интерфейсите от пакета com.fmi.bytecode.annotations.element.*, описани подробно в предходната глава (Проектиране на реализацията). Тези класове реализират модела на данните от високо ниво, като предлагат обектно ориентирана структура, направена с цел да представя релацията между прочетените от байткода елементи (класове, методи, конструктори, полета и съответните им анотации).

В тази точка, ще бъдат представени йерархията и асоциациите между класовете, чрез UML клас диаграма (Фиг. 7). Ще бъде направено описание на всеки от тях.



Фиг. 7: Клас диаграма представяща класовете в пакета com.fmi.bytecode.annotations.element.impl.*

5.2.1. Клас AnnotationNamedMember

Класа AnnotationNamedMember моделира отделен атрибут на анотация. Той реализира операциите на интерфейса com.fmi.bytecode.annotations.element.NamedMember.

5.2.2. Клас AnnotationRecordImpl

Класа AnnotationRecordImpl моделира отделна анотация. Той реализира операциите на интерфейса com.fmi.bytecode.annotations.element.AnnotationRecord.

5.2.3. Абстрактен клас ElementInfoImpl

Класа ElementInfoImpl е абстрактен и моделира всеки елемент в Java, който може да бъде анотиран. В частност това може да бъде клас (наследникът ClassInfoImpl), поле (наследникът FieldInfoImpl), метод (наследникът MethodInfoImpl), конструктор (наследникът ConstructorInfoImpl) или аргумент на метод. Той реализира операциите на интерфейса com.fmi.bytecode.annotations.element.ElementInfo.

5.2.4. Клас FieldInfoImpl

Класа FieldInfoImpl моделира поле на класа. Той наследява класа ElementInfoImpl и реализира операциите на интерфейса com.fmi.bytecode.annotations.element.FieldInfo.

5.2.5. Клас MethodInfoImpl

Класа MethodInfoImpl моделира метод на класа. Той наследява класа ElementInfoImpl и реализира операциите на интерфейса com.fmi.bytecode.annotations.element.MethodInfo.

5.2.6. Клас ConstructorInfoImpl

Класа ConstructorInfoImpl моделира конструктор на класа. Той наследява класа ElementInfoImpl и реализира операциите на интерфейса com.fmi.bytecode.annotations.element.ConstructorInfo.

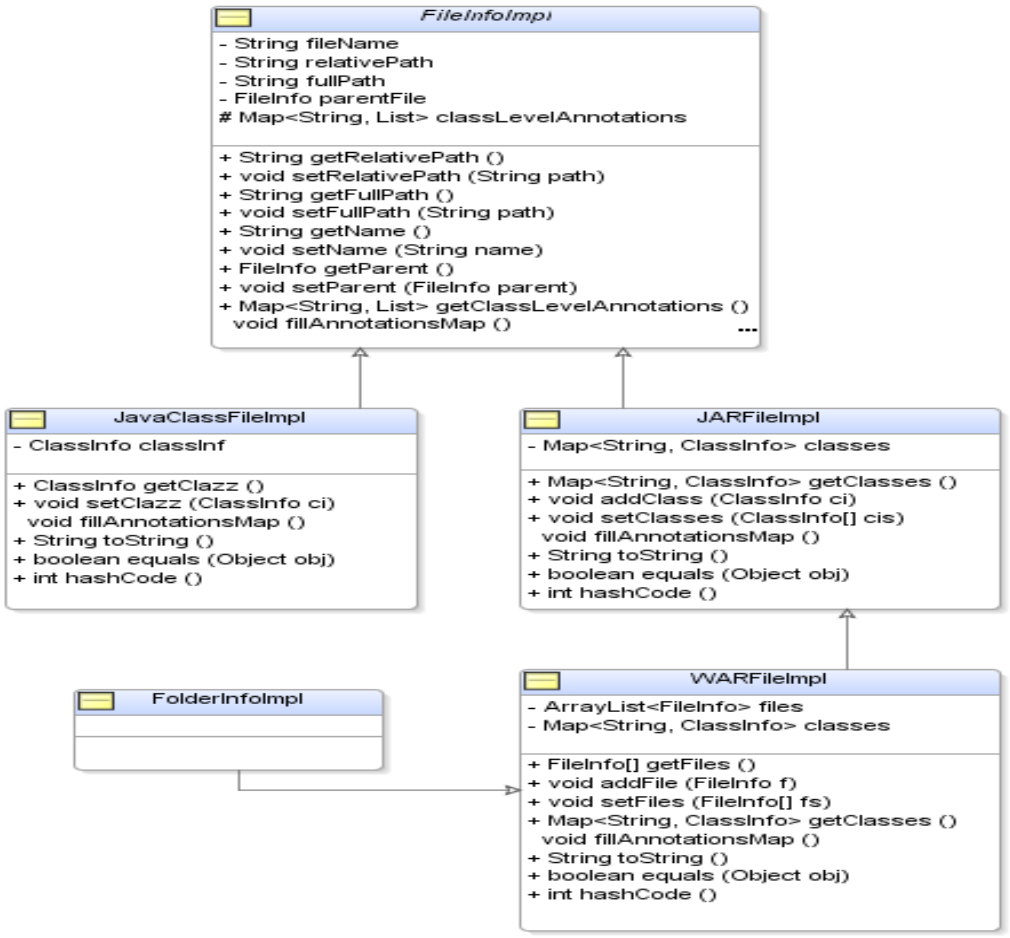
5.2.7. Клас ClassInfoImpl

Класа ClassInfoImpl моделира клас. Той наследява класа ElementInfoImpl и реализира операциите на интерфейса com.fmi.bytecode.annotations.element. ClassInfo.

*5.3. Пакет com.fmi.bytecode.annotations.file.impl.**

В този пакет се намират всички класове, които реализират интерфейсите от пакета com.fmi.bytecode.annotations.file.*, описани подробно в предходната глава (Проектиране на реализацията). Тези класове реализират модела на входните данни съдържащи клас файлове. Това са четирите източници на мета-информация – клас файл (JavaClassFileImpl), директория (FolderInfoImpl), jar файл (JARFileImpl) и war файл (WARFileImpl). Те могат да бъдат класифицирани най-общо като файлове и поради тази причина всички те наследяват общ абстрактен клас FileInfoImpl, в който са изнесени всички общи операции.

В тази точка, ще бъдат представени йерархията и асоциациите между класовете, чрез UML клас диаграма (Фиг. 8). Ще бъде направено описание на всеки от тях.



Фиг. 8: Клас диаграма представяща класовете в пакета com.fmi.bytecode.annotations.file.impl.*

5.3.1. Абстрактен клас FileInfoImpl

Класа FileInfoImpl е абстрактен и моделира всички изброени конкретни представители на входен файл – клас файл, директория, jar файл и war файл. Наследява се от съответните класове – JavaClassFile, FolderInfo, JARFile и WARFile. Той дефинира абстрактната операция abstract void fillAnnotationsMap(). Всеки от наследниците реализира тази операция, като кодира специфичната логика за структуриране на анотациите, които съдържа пряко (ако е клас файл) или косвено (ако е директория, jar или war файл).

Класа FileInfoImpl реализира операциите на интерфейса com.fmi.bytecode.annotations.file.FileInfo.

5.3.2. Клас JavaClassFileImpl

Класа JavaClassFileImpl представя конкретен клас файл. Той наследява абстрактния клас FileInfoImpl, като реализира неговата абстрактна операция abstract void fillAnnotationsMap() представена в Приложение 9.1.

Първо се инициализира Map структурата, която пази анотациите на клас файла (`classLevelAnnotations`). След това от асоцииранияят ClassInfo обект (`classInf`), се взимат всички анотации на текущия клас файл и се добавят в структурата, като за ключ се използва името на класа на анотацията.

Класа JavaClassFileImpl реализира операциите на интерфейса com.fmi.bytecode.annotations.file.JavaClassFile.

5.3.3. Клас JARFileImpl

Класа JARFileImpl представя конкретен jar файл. Той наследява абстрактния клас FileInfoImpl, като реализира неговата абстрактна операция abstract void fillAnnotationsMap() представена в Приложение 9.2.

Първо се инициализира Map структурата, която пази анотациите на jar файла (`classLevelAnnotations`). След това се обхожда асоцииранияят списък с ClassInfo обекти (`classes`), представляващи клас файловете, които се съдържат в този jar файл. Аналогично на реализацията в JavaClassFileImpl, анотациите на всеки ClassInfo обект се добавят в Map структурата, като за ключ се използва името на класа на анотацията. За стойност се пази списък от AnnotationRecord обекти, съответстващи на всички анотации на ниво клас с това име, намерени в jar файла.

Класа JARFileImpl реализира операциите на интерфейса com.fmi.bytecode.annotations.file.JARFile.

5.3.4. Клас WARFileImpl

Класа WARFileImpl представя конкретен war файл. Той наследява абстрактния клас FileInfoImpl, като реализира неговата абстрактна операция abstract void fillAnnotationsMap() представена в Приложение 9.3.

Първо се инициализира Map структурата, която пази анотациите на war файла (`classLevelAnnotations`). След това се обхожда асоцииранияят списък с

FileInfo обекти (`files`), представляващи файловете, които се съдържат в този war файл. Те могат да бъдат клас файлове, директории и jar файлове. Взимат се анотациите на всеки асоцииран FileInfo обект и се добавят в Map структурата, като за ключ се използва името на класа на анотацията. За стойност се пази списък от AnnotationRecord обекти, съответстващи на всички анотации на ниво клас с това име, намерени в war файла.

Класа WARFileImpl реализира операциите на интерфейса com.fmi.bytecode. annotations.file.WARFile.

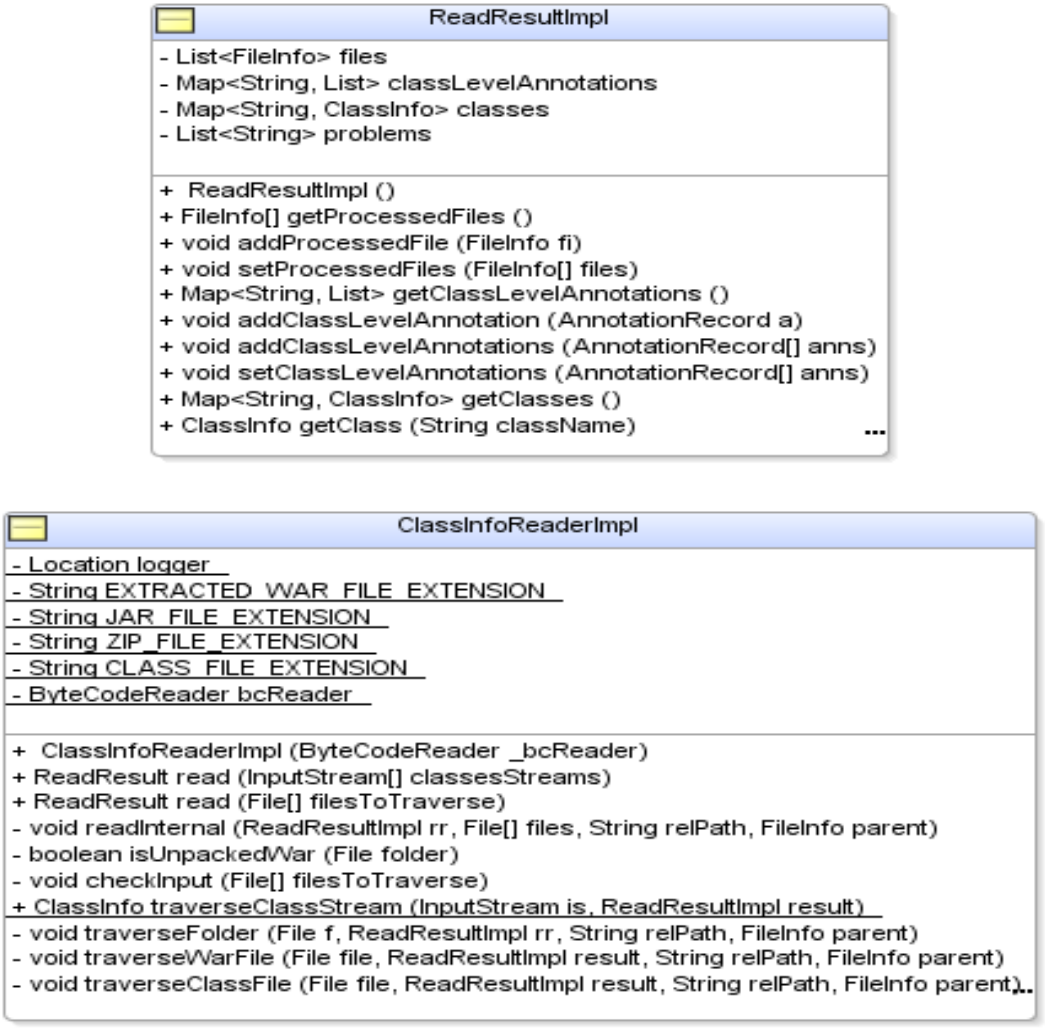
5.3.5. Клас FolderInfoImpl

Класа FolderInfoImpl представя конкретна директория. Той наследява класа WARFileImpl и реализира операциите на интерфейса com.fmi.bytecode. annotations.file.FolderInfo.

*5.4. Пакет com.fmi.bytecode.annotations.tool.impl.**

В този пакет се намират всички класове, които реализират интерфейсите от пакета com.fmi.bytecode.annotations.tool.*, описани подробно в предходната глава (Проектиране на реализацията). Тези класове реализират модела за работа с разработената система. Това е входната точка на реализацията. С нейна помощ, от една страна може да се специфицират входните файлове, чиято мета-информация трябва да бъде извлечена, а от друга да се получи изходящ модел на мета-информацията, представящ резултата от обработката им.

В тази точка са представени класовете, чрез UML клас диаграма (Фиг. 9). Направено е описание на всеки от тях.



Фиг. 9: Клас диаграма представяща класовете в пакета com.fmi.bytecode.annotations.tool.impl.*

5.4.1. Клас ReadResultImpl

Класа ReadResultImpl представя изходящия модел на мета-информацията получен в резултат на обработката на всички входни файлове.

Всеки обработен клас файл, намерен във входните файлове, се добавя в ReadResultImpl обекта, като се използва операцията addClass(ClassInfo ci) представена в Приложение 9.4.

С помощта на тази операция се проверява дали вече не е добавена мета-информация, за подадения като аргумент клас файл. Ако този файл вече съ-

ществува в резултата, се добавя съобщение за грешка, в което се описват местата, на които се среща във входното множество от файлове.

Класа ReadResultImpl реализира операциите на интерфейса com.fmi.bytecode.annotations.tool.ReadResult.

5.4.2. Клас ClassInfoReaderImpl

Класа ClassInfoReaderImpl представя модел на четеца на мета-информация - по зададени входни файлове той изгражда изходящият и модел.

Той реализира операцията ReadResult read(File[] filesToTraverse) на интерфейса com.fmi.bytecode.annotations.tool.ClassInfoReader, като използва директна и косвена рекурсия, чрез пет помощни операции:

- void readInternal (File[] files, ReadResultImpl result, String relPath, FileInfo parent) – Обхожда списъка от файлове (files), подаден като аргумент. За всеки файл проверява дали е клас файл, директория, jar или war файл и извиква съответно traverseClassFile, traverseFolder, traverseZipFile и traverseWarFile.
- void traverseClassFile (File file, ReadResultImpl result, String relPath, FileInfo parent) – Обработва клас файла подаден като аргумент (file) и добавя създаденият ClassInfo обект към резултата (result). Тази операция е крайна точка на всяка последователност от извиквания на останалите операции. С други думи всяка от останалите операции, пряко или косвено, се свежда до многократното извикване на traverseClassFile.
- void traverseZipFile (File file, ReadResultImpl result, String relPath, FileInfo parent) – Обхожда всички клас файлове, които се съдържат в jar файла подаден като аргумент (file) и извиква traverseClassFile за всеки от тях.
- void traverseFolder (File file, ReadResultImpl result, String relPath, FileInfo parent) – Обхожда списъка от файлове, които се съдържат в подадената директория (file). За всеки файл проверява дали е директория, клас файл, jar или war файл. За намерените поддиректории извиква себе си в директна рекурсия. За файловете, които са клас файл, jar

или war файл извиква в косвена рекурсия съответно traverseClassFile, traverseZipFile и traverseWarFile.

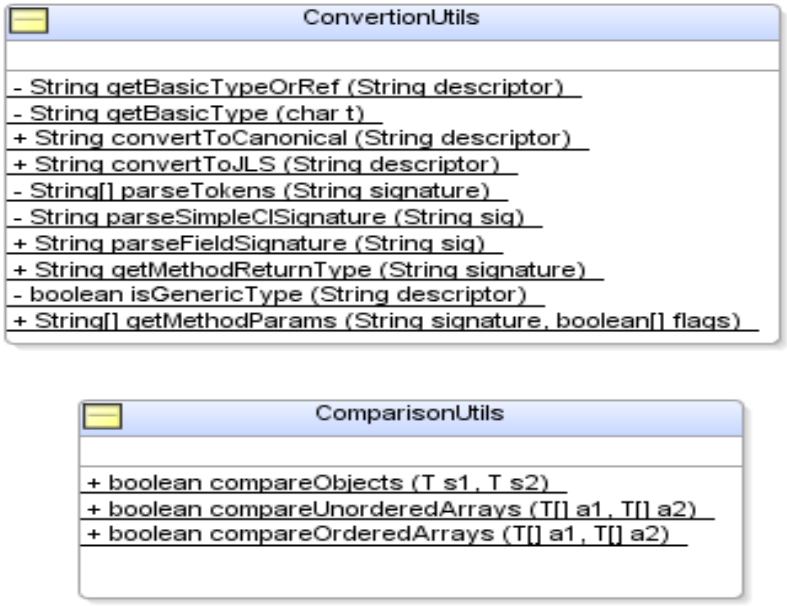
- void traverseWarFile (File file, ReadResultImpl result, String relPath, FileInfo parent) – Обхожда списъка от файловете, които се съдържат в подадения war файл (file). За всеки файл проверява дали е клас файл, директория или jar файл и извиква в косвена рекурсия съответно traverseClassFile, traverseFolder и traverseZipFile.

Изпълнението на тези операции е илюстрирано с диаграма на работния процес (UML Activity Diagram) на модула за обхождане на входните файлове (Фиг. 6).

*5.5. Пакет com.fmi.bytecode.annotations.util.**

В този пакет се намират класовете ComparisonUtils и ConvertionUtils. В тези класове са реализирани статични операции с публичен достъп, които се използват на различни места в разработената система. Тези операции са капсулирани с помощта на горните два класа, за да се избегне дублирането на код.

В тази точка са представени класовете, чрез UML клас диаграма (Фиг. 10) и е направено описание на всеки от тях.



Фиг. 10: Клас диаграма представяща класовете в пакета com.fmi.bytecode.annotations.util.*

5.5.1. Клас ComparisonUtils

Класа ComparisonUtils съдържа операции за сравнение на обекти и масиви. Тези операции се използват в реализацията за сравняване на:

- Атрибути на анотации.
- Реализирани интерфейси на клас.
- Полета на клас.
- Методи на клас.
- Конструктори на клас.
- Изключения на методи.
- Параметри на методи.
- Параметри на конструктори.

Операциите се дефинират като се използват Generic типове, специфицирани в JLS:

- public static <T> boolean compareObjects(T s1, T s2) – Сравнява два обекта, като допуска подадените референции към обектите да са null (Приложение 9.5.1).
- public static <T> boolean compareUnorderedArrays(T[] a1, T[] a2) – Сравнява два масива от обекти, без да взима под внимание реда, в който се срещат елементите им. Допуска подадените като аргументи референции към масивите да са null (Приложение 9.5.2).
- public static <T> boolean compareUnorderedArrays(T[] a1, T[] a2) – Сравнява два масива от обекти, като взима под внимание реда, в който се срещат елементите им. Допуска подадените като аргументи референции към масивите да са null (Приложение 9.5.3).

5.5.2. Клас ConvertionUtils

Класа ConvertionUtils съдържа различни операции свързани с конвертиране на типове и синтактичен анализ на сигнатури на методи и полета прочетени от байткода. Реализацията на методите по-долу е представена в Приложение 9.6.

- private static String getBasicType(char t) – По подаден символ като аргумент, представящ вграден прост JVM тип, връща съответния тип в езика Java (дефиниран в JLS).
- private static String getBasicTypeOrRef(String desc) – По подаден символ като аргумент, представящ вграден прост JVM тип или клас, връща съответния тип в езика Java (дефиниран в JLS).
- public static final String convertToCanonical(String desc) – По подаден символ като аргумент, представящ вграден прост JVM тип, клас или масив от произволен тип, връща съответния тип в езика Java (дефиниран в JLS).
- public static String parseFieldSignature(String sig) – По подадена сигнатура на поле, прочетена от байткода, връща съответния тип на полето в езика Java (дефиниран в JLS).
- private static boolean isGenericType(String descriptor) – По подаден JVM тип връща true, ако типа представя Generic тип.
- public static String getMethodReturnType(String signature) – По подадена сигнатура на метод, прочетена от байткода, връща типа на връщаната от метода стойност в езика Java (дефиниран в JLS).
- public static String[] getMethodParams(String signature, boolean[] flags) – По подадена сигнатура на метод, прочетена от байткода, връща масив съдържащ типовете на аргументите му, в реда, в който са декларирани. Типовете са форматирани съгласно JLS. В масива flags се записва стойност true за тези аргументи, които имат Generic типове.

6. Тестване

Процесът 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 [6]:

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) и Тестове базирани на сценариите за използване на системата (Scenario-based testing или Scenario-oriented testing).

6.1. Тестове на части от кода (Unit tests)

При разработването на всяка отделна единица, обособяваща самостоятелен модул на системата се разработват тестове, които имат за цел да верифицират резултата от обработката на дадено множество от входни данни.

В частност, отделните модули на разработената система са нейните Java класове. 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 (Debug перспективата на Eclipse [10]), 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)

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

Например, подава се като вход на системата един клас файл, който съдържа една анотация на ниво клас – “@javax.ejb.EJB”. Това става чрез извикването на:

```
ClassInfoReader cif = AnnotationsReaderFactory.getReader(bcReader);
ReadResult readResult = cif.read(new File[]{<пълен път до клас файла>});
```

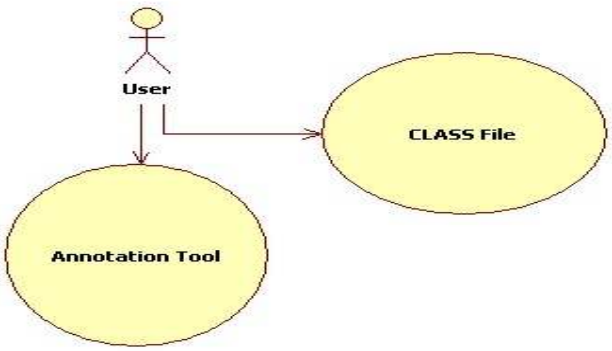
В следствие на което, когато потребителят извика:

```
readResult.getClass(<името на класа>).getAnnotation(“javax.ejb.EJB”);
```

трябва като резултат да се върне обект, реализация на интерфейса AnnotationRecord представящ тази анотация.

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 конкретен клас файл, който е подаден на системата, в някои от допустимите форми – директно подаден клас, файл в директория, в war файл, в jar файл или в допустима комбинация от тях. Тези сценарии са илюстрирани със следните диаграми представящи различните начини за използване на системата:

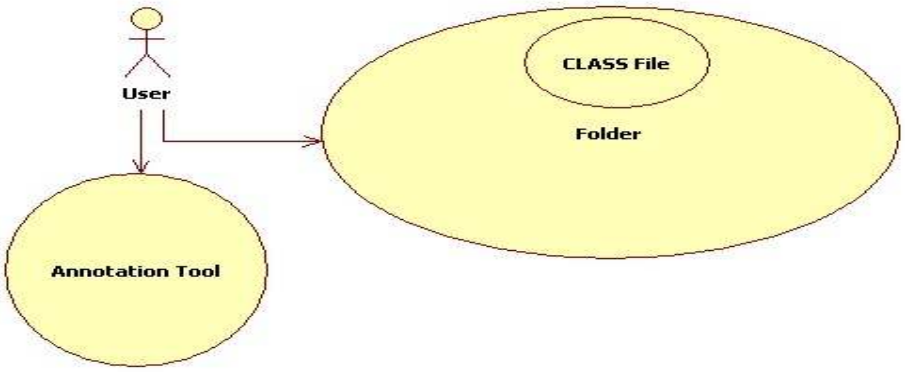
- Като вход на системата се подава клас файл:



Фиг. 11: Сценарий за използване на системата, в който като вход се подава клас файл

Модулът за анализ на входните файлове трябва да определи, че подаденията входен файл е клас файл. След това трябва да го подаде към следващия модул и т.н. Като резултат изходящия модел, върнат на клиента, трябва да съдържа мета-информацията на входния клас файл.

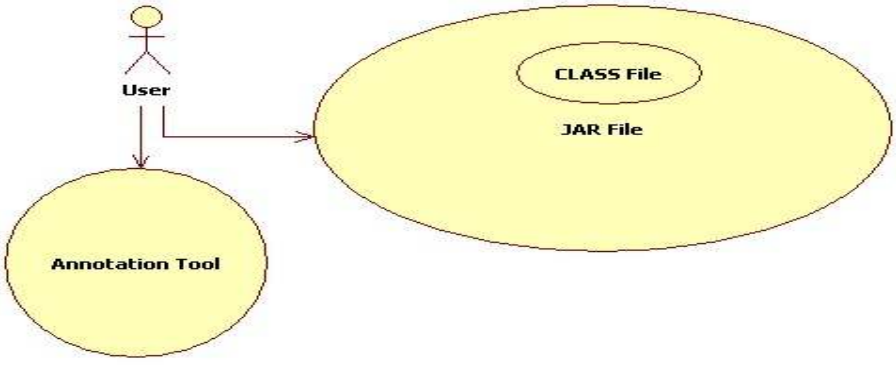
- Като вход на системата се подава директория, в която има клас файл:



Фиг. 12: Сценарий за използване на системата, в който като вход се подава директория, съдържаща клас файл

Модулът за анализ на входните файлове трябва да определи, че подаденията входен файл е директория. Трябва да отркие клас файла в нея и да го подаде към следващия модул и т.н. Като резултат изходящия модел, върнат на клиента, трябва да съдържа мета-информацията на клас файла.

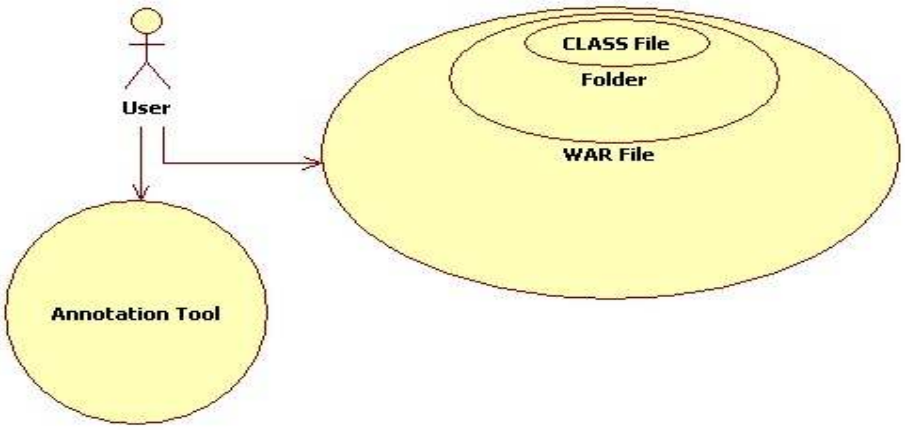
- Като вход на системата се подава jar файл, в който е пакетиран клас файл:



Фиг. 13: Сценарий за използване на системата, в който като вход се подава jar архив, съдържащ клас файл

Модулът за анализ на входните файлове трябва да определи, че подаденията входен файл е jar архив. Трябва да отркие клас файла в него и да го подаде към следващия модул и т.н. Като резултат изходящия модел, върнат на клиента, трябва да съдържа мета-информацията на клас файла.

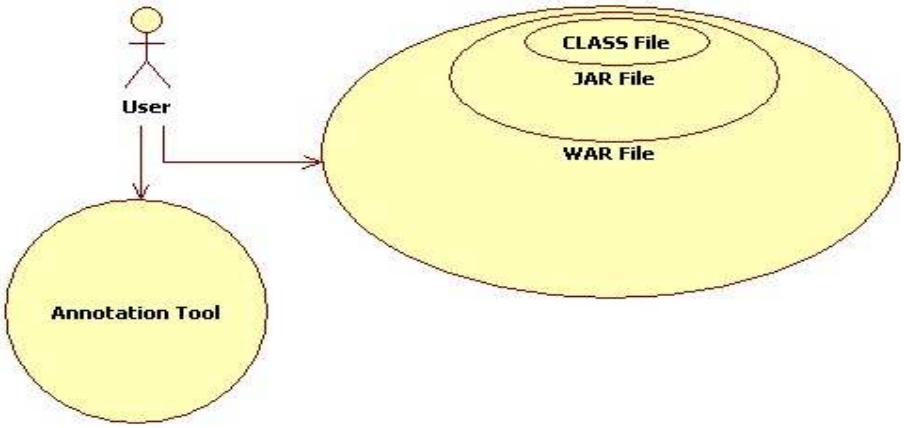
- Като вход на системата се подава war файл. В него е пакетирана директория, в която има клас файл:



Фиг. 14: Сценарий за използване на системата, в който като вход се подава war архив, съдържащ директория с клас файл

Модулът за анализ на входните файлове трябва да определи, че подаденията входен файл е war архив. След което, да отркие клас файла, който се съдържа в пакетирана в него директория и да го подаде към следващия модул и т.н. Като резултат изходящия модел, върнат на клиента, трябва да съдържа мета-информацията на клас файла.

- Като вход на системата се подава war файл. В него е пакетиран jar файл, който от своя страна съдържа клас файл:



Фиг. 15: Сценарий за използване на системата, в който като вход се подава war архив, съдържащ jar архив с клас файл

Модулът за анализ на входните файлове трябва да определи, че подаденият входен файл е war архив. След което, да отркие клас файла, който се съдържа в пакетиран в него jar архив и да го подаде към следващия модул и т.н. Като резултат изходящия модел, върнат на клиента, трябва да съдържа метаинформацията на клас файла.

След като клас файла е намерен, обработен и върнат на клиента, се верифицират неговите метаданни. Това се изразява със следните проверки:

- Проверява се дали името и пакета на класа отговарят на обработения файл.
- Проверяват се имената и стойностите на атрибутите на анотациите на класа.
- Проверяват се имената и стойностите на атрибутите на анотациите на полетата на класа.

- Проверяват се имената и стойностите на атрибутите на анотациите на методите на класа.
- Проверяват се имената и стойностите на атрибутите на анотациите на конструкторите на класа.
- Проверяват се имената и стойностите на атрибутите на анотациите на параметрите на методите и конструкторите на класа.
- Правят се необходимите проверки за коректност на вложените анотации.
- Правят се необходимите проверки за коректност на анотациите, чиито атрибути са масиви.
- Правят се необходимите проверки за коректност на анотациите, чиито атрибути са от тип enum.

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.
- Подробно описание на структурата на клас файла. Определяне на местата, където се пазят анотациите в компилирания Bytecode.
- Дефиниране на изискванията към разработената система. Определени са всички сценарии за използване на системата, като употребата й не се ограничи само до инструмент за обработка 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 мета-информацията 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 интерфейси, които са разработени въз основа на изискванията и сценариите за използване на системата. Изготвени са необходимите UML диаграми за илюстриране на дефинираните модели за работа.
- Реализация на решението, която е свързана с написването на програмен код, реализиращ интерфейсите, 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 система за извличане на мета-информация от Java Bytecode. Обектно ориентираният й модел и реализация позволяват изключително лесна и интуитивна работа с нея. С програмният й интерфейс могат да си служат успешно потребители, които имат бегла представа от структурата на

байткода. Функционалността, която предлага е сравнима с тази на вградения в Java, програмен интерфейс Reflection API.

Насоките за възможно развитие на дипломната работата са ориентирани към алтернативен начин за работа с нея, освен чрез програмният й интерфейс. А именно, реализацията на графичен интерфейс, с чиято помощ потребителите ще могат да работят интерактивно със системата, без да се налага да пишат програмен код. 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 записват извлечените метаданни в независим формат (например XML) и т.н.

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.
JLS	– Спецификация на езика Java (Java Language Specification), съгласно документацията J2SE(TM) Development Kit Documentation 5.0.
Bytecode / Байткод	– 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.
Generic	– Тип в езика Java въведен с JLS.
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 JavaBeans

8. Използвана литература

1. http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html
2. http://java.sun.com/docs/books/vmspec/2nd-edition/ClassFileFormat-Java5.pdf
3. http://java.sun.com/javaee/5/docs/tutorial/doc/
4. "Thinking in Java, 4rd Edition", Bruce Eckel
5. "UML destilled, 3rd edition", Martin Fowler
6. "Managing the Testing Process, Second Edition", John Wiley
7. "Object-Oriented Software Construction, Second Edition", Bertrand Meyer
8. "Data Structures and Algorithms in Java, 4th edition", Michael T. Goodrich, Roberto Tamassia
9. "Java 1.5 Tiger: A Developer's Notebook", Brett McLaughlin, David Flanagan
10. http://www.eclipse.org/
11. "Java(TM) Language Specification, 3rd Edition", James Gosling, Bill Joy, Guy Steele, Gilad Bracha
12. http://jakarta.apache.org/bcel/
13. http://asm.objectweb.org/
14. http://serp.sourceforge.net/

9. Приложение

9.1. Клас JavaClassFileImpl – метод fillAnnotationsMap()

```
void fillAnnotationsMap(){
  classLevelAnnotations = new HashMap<String,List<AnnotationRecord>>();
  Map<String, AnnotationRecord> anns = classInf.getAnnotations();
  for(AnnotationRecord ar : anns.values()) {
    ArrayList<AnnotationRecord> lst = new ArrayList<AnnotationRecord>();
    lst.add(ar);
    classLevelAnnotations.put(ar.getTypeName(), lst);
  }
}
```

9.2. Клас JARFileImpl – метод fillAnnotationsMap()

```
void fillAnnotationsMap(){
  classLevelAnnotations = new HashMap<String,List<AnnotationRecord>>();
  for(ClassInfo c : classes.values()) {
     Map<String, AnnotationRecord> anns = c.getAnnotations();
     for(AnnotationRecord ar : anns.values()) {
       String annName = ar.getTypeName();
       List<AnnotationRecord> lst = classLevelAnnotations.get(annName);
       If (lst == null) {
         lst = new ArrayList<AnnotationRecord>();
         classLevelAnnotations.put(annName, lst);
       }
       lst.add(ar);
     }
  }
}
```

9.3. Клас WARFileImpl – метод fillAnnotationsMap()

```
void fillAnnotationsMap(){
  classLevelAnnotations = new HashMap<String,List<AnnotationRecord>>();
  for (FileInfo f : files) {
    Map<String,List<AnnotationRecord>> anns=f.getClassLevelAnnotations();
    for (String key : anns.keySet()) {
      List<AnnotationRecord> lst = classLevelAnnotations.get(key);
      if (lst == null) {
        lst = new ArrayList<AnnotationRecord>();
        classLevelAnnotations.put(key, lst);
      }
      lst.addAll(anns.get(key));
    }
  }
}
```

9.4. Клас ReadResultImpl – метод addClass(ClassInfo ci)

```
public void addClass(ClassInfo ci) throws ReadingException {
  ClassInfoImpl old = (ClassInfoImpl)classes.get(ci.getName());
   if (old != null) {
     FileInfo[] fi = old.getContainingFiles();
     String files = "";
     if (fi != null) {
       for (FileInfo f: fi) {
         files += f.getName() + " , ";
       }
     }
     String newFile = "class stream";
     FileInfo[] cfs = ci.getContainingFiles();
     if ((cfs != null) && (cfs.length > 0)) {
       old.addContainingFile(cfs[0]);
```

```
      newFile = cfs[0].getName();
    }
    if (!old.equals(ci)) {
      problems.add( "There are different versions of the class [" +
                    ci.getName() + "]! " +
                    " One version can be found in: " + files +
                    " And the other in: " + newFile);
    }
  } else {
    classes.put(ci.getName(), ci);
  }
}
```

9.5. Клас ComparisonUtils

9.5.1. Метод boolean compareObjects(T s1, T s2)

```
   public static <T> boolean compareObjects(T s1, T s2) {
     if ((s1 == null) && (s2 == null)) {
       return true;
     }
     if((s1 == null) || (s2 == null)){
       return false;
     }
     return s1.equals(s2);
    }
```

9.5.2. Метод boolean compareUnorderedArrays(T[] a1, T[] a2)

```
   public static <T> boolean compareUnorderedArrays(T[] a1, T[] a2) {
     if ((a1 == null) && (a2 == null)){
       return true;
     }
     if ((a1 == null) || (a2 == null)){
       return false;
```

```
    }

    if (a1.length != a2.length) {
      return false;
    }

    List<T> l1 = Arrays.asList(a1);
    List<T> l2 = Arrays.asList(a2);
    return l1.containsAll(l2);
  }
```

9.5.3. Метод boolean compareOrderedArrays(T[] a1, T[] a2)

```
  public static <T> boolean compareOrderedArrays(T[] a1, T[] a2) {
    if ((a1 == null) && (a2 == null)){
      return true;
    }
    if ((a1 == null) || (a2 == null)){
      return false;
    }

    if (a1.length != a2.length) {
      return false;
    }

    for(int i=0; i<a1.length; i++){
      if (!a1[i].equals(a2[i])) {
        return false;
      }
    }
    return true;
}
```

9.6. Клас ConvertionUtils

9.6.1. Метод String getBasicType(char t)

```
private static String getBasicType(char t){
     switch(t){
       case 'B': return "byte";
       case 'C': return "char";
       case 'D': return "double";
       case 'F': return "float";
       case 'I': return "int";
       case 'J': return "long";
       case 'S': return "short";
       case 'Z': return "boolean";
       case 'V': return "void";
       default: return null;
     }
   }
```

9.6.2. Метод String getBasicTypeOrRef(String desc)

```
   private static String getBasicTypeOrRef(String desc){
     String bt = getBasicType(desc.charAt(0));
     if (bt == null){
       if(desc.charAt(0)=='L') {
         String result = desc.substring(1, desc.length() - 1);
         return result.replace('/', '.');
       } else {
         return null;
       }
     } else {
       return bt;
     }
   }
```

9.6.3. Метод String convertToCanonical(String desc)

```
public static final String convertToCanonical(String desc) {
  int dimension = 0;
  while (desc.startsWith("[")) {
     dimension++;
     desc = desc.substring(1);
  }
  StringBuilder buffer = new StringBuilder(desc.length());
  String bt = getBasicTypeOrRef(desc);
  if (bt == null){
    throw new IllegalArgumentException("Invalid type definition " +
                                       "detected in bytecode: " + desc);
  } else {
    buffer.append(bt);
  }
  for (int i = 0; i < dimension; i++) {
    buffer.append("[]");
  }
  return buffer.toString();
}
```

9.6.4. Метод String parseFieldSignature(String sig)

```
public static String parseFieldSignature(String sig){
  //determine array size
  int dimension = 0;
  while (sig.startsWith("[")) {
     dimension++;
     sig = sig.substring(1);
  }



  //remove the terminator symbol
```

```
if (sig.charAt(sig.length()-1) == ';') {
  sig = sig.substring(0, sig.length()-1);
}

char c = sig.charAt(0);
sig = sig.substring(1);

//check for basic type
String bType = getBasicType(c);
StringBuilder buffer = new StringBuilder(sig.length());
if (bType != null) {
  buffer.append(bType);
} else {
  //check for reference or param type
  if ((c == 'T') || (c == 'L')) {
    //check for inner class
    int in = sig.indexOf('.');
    if (in != -1) {
      String main = sig.substring(0, in);
      buffer.append(parseSimpleClSignature(main));
      String str1 = sig.substring(in, sig.length())
      StringTokenizer st = new StringTokenizer(str1, ".");
      while (st.hasMoreTokens()) {
        buffer.append(".");
        buffer.append(parseSimpleClSignature(st.nextToken()));
      }
    } else {
      buffer.append(parseSimpleClSignature(sig));
    }
  }
```

```
    }

    //append array demension
    for (int i = 0; i < dimension; i++) {
      buffer.append("[]");
    }
    return buffer.toString();
  }
```

9.6.5. Метод boolean isGenericType(String descriptor)

```
  private static boolean isGenericType(String descriptor){
    if (descriptor.indexOf('<') != -1) {
      return true;
    } else {
      while (descriptor.startsWith("[")) {
        descriptor = descriptor.substring(1);
      }
      if (descriptor.startsWith("T")) {
        return true;
      }
    }
    return false;
  }
```

9.6.6. Метод String getMethodReturnType(String signature)

```
  public static String getMethodReturnType(String signature){
    int si = signature.lastIndexOf(')');
    int ei = signature.indexOf('^');
    String rt = null;
    if (ei != -1) {
      rt = signature.substring(si+1, ei);
    } else {
```

```
      rt = signature.substring(si+1);
    }
    if (isGenericType(rt)) {
      return parseFieldSignature(rt);
    } else {
      return null;
    }
  }
```

9.6.7. Метод String[] getMethodParams(String sig, boolean[] flags)

```
  static String[] getMethodParams(String sig, boolean[] flags) {
    int si = sig.indexOf('(');
    int ei = sig.indexOf(')');
    String[] params = parseTokens(sig.substring(si+1, ei));

    ArrayList<String> p = new ArrayList<String>();
    for (int i = 0; i < params.length; i++) {
      String t = params[i];
      if (isGenericType(t)) {
        flags[i] = true;
      }
      p.add(parseFieldSignature(t));
    }
    return p.toArray(new String[0]);
  }
```