Министерство образования и науки Украины

Черниговский НАЦИОНАЛЬНЫЙ технологический университет

Кафедра информационных и компьютерных систем

допустить к защите

Заведующий кафедрой ИКС

д-р техн. наук, проф.

В.В. Казимир

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_2014г.

**Уланович Дмитрий Николаевич**

**автоматизированная Система верификации UML-диграмм. Подсистема верификации диаграмм КЛАССОВ**

Специальность: 8.05010202 – Системное программирование

Квалификационная работа магистра

Научный руководитель:

д-р техн. наук, проф.

В.В. Литвинов

Консультант с охраны труда:

к.х.н., доцент

О.Л. Гуменюк

Чернигов – 2014

РЕФЕРАТ

Квалификационная работа магистра, 87 страниц, 47 рисунков, 4 таблицы, 19 источников.

Объект исследования – верификация диаграмм классов.

Целью работы является:

- разработка архитектуры автоматизированной системы верификации диаграмм классов;

- разработка и реализация собственного метода верификации на основе метода тестового драйвера.

При проектировании использовался смешанный метод, который объединяет такие методы как восходящее и нисходящее проектирование.

Результат разработки – архитектура автоматизированной системы верификации диаграмм классов, которая позволит устранить множество ошибок с программным обеспечением еще на этапе проектирования. В отличие от существующих, данная система позволяет в комплексе несколькими методами оценивать корректность диаграмм классов. Методы позволяют оценивать корректность, как различных аспектов диаграммы классов, так и позволяют оценить один и тот же аспект с точки зрения различных подходов верификации. Новизна заключается в разработке собственного метода верификации на основе тестового драйвера.

Результаты работы можно использовать в системах построения и верификации UML диаграмм.

Расчет экономической эффективности не проводился.

Работа имеет практическую ценность.

Прогноз развития данного проекта – расширение верификационных возможностей для различных языков программирования, возможность использовать за основу различные форматы данных, отличные от XML, а так же развитие проекта в широко используемых коммерческих и некоммерческих системах разработки UML диаграмм. Как возможное направление – превращение проекта в отдельную утилиту для верификации UML диаграмм классов.

ВЕРИФИКАЦИЯ, ТЕСТИРОВАНИЕ, UML, ДИАГРАММА КЛАССОВ, ТЕСТОВЫГО ДРАЙВЕР, ИДЕНТИФИКАЦИОННЫЙ ГРАФ, МНОЖЕСТВА, ШАБЛОНЫ, JAVA, ENTERPRISE ARCHITECT, VISUAL PARADIGM.

РЕФЕРАТ

Кваліфікаційна робота магістра, 87 сторінок, 47 рисунків, 4 таблиця, 19 джерел.

Об'єкт дослідження - верифікація діаграм класів.

Метою роботи є:

- розробка архітектури автоматизованої системи верифікації діаграм класів;

- розробка та реалізація власного методу верифікації на основі методу тестового драйвера.

При проектуванні використовувався змішаний підхід, який об'єднує такі методи як висхідний і низхідний.

Результат розробки - архітектура автоматизованої системи верифікації діаграм класів, яка дозволить усунути безліч помилок з програмним забезпеченням ще на етапі проектування. На відміну від існуючих, дана система дозволяє в комплексі декількома методами оцінювати коректність діаграм класів. Методи дозволяють оцінювати коректність, як окремих аспектів діаграми класів, так і дозволяють оцінити один і той же аспект з точки зору різних підходів верифікації. Новизна полягає в розробці власного методу верифікації на основі тестового драйвера.

Результати роботи можна використовувати в системах побудови та верифікації UML діаграм.

Розрахунок економічної ефективності не проводився.

Робота має практичну цінність.

Прогнозування розвитку даного проекту - розширення верифікаційних можливостей для різних мов програмування, можливість використовувати за основу різні формати даних, відмінні від XML, а також розвиток проекту в широко використовуваних комерційних і некомерційних системах розробки UML діаграм. Як можливий напрямок - перетворення проекту в окрему утиліту для верифікації UML діаграм класів.

ВЕРИФІКАЦІЯ, ТЕСТУВАННЯ, UML, ДІАГРАМА КЛАСІВ, ТЕСТОВИГО ДРАЙВЕР, ІДЕНТИФІКАЦІЙНИЙ ГРАФ, МНОЖИНИ, ШАБЛОНИ, JAVA, ENTERPRISE ARCHITECT, VISUAL PARADIGM.

THE ABSTRACT

The master work, 87 pages, 47 figures, 4 table, 19 sources.

The object of study - verification of class diagrams.

The aims are:

- the development of an automated verification system architecture class diagrams;

- to develop and implement of my own verification method based on the method of test drivers.

In the design is used a mixed method that combines techniques such as ascending and descending projection.

The result of the development - the architecture of the automated verification system of class diagrams, which will eliminate many mistakes with the software at the design stage. Unlike existing, this system allows the complex of several methods to evaluate the correctness of class diagrams. Methods allow to evaluate the correctness of different aspects of the class diagram and correctness of the same aspect from the point of view of the different approaches of verification. The novelty lies in the development of its own verification method based on the test driver.

The results can be used in systems of construction and verification of UML diagrams.

Cost-effectiveness was not performed.

The work has the practical value.

Forecast of development of the project – extending of verification capabilities for a variety of programming languages to use as a basis for a variety of data formats different than XML, as well as the development of the project are widely used in commercial and non-commercial systems design UML diagrams. As a possible direction - transform the project to a separate utility for verifying of UML class diagrams.

VERIFICATION, TESTING, UML, CLASS DIAGRAMMS, TESTING DRIVER, IDENTIFICATION GRAPH, SETS, TEMPLATES, JAVA, ENTERPRISE ARCHITECT, VISUAL PARADIGM.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 7](#_Toc376531556)

[1 АНАЛИЗ СУЩЕСТВУЮЩИХ СИСТЕМ ВЕРИФИКАЦИИ   
 UML- ДИАГРАММ 8](#_Toc376531557)

[1.1 Существующие системы верификации UML-диграмм 8](#_Toc376531558)

[1.1.1 ArgoUML 8](#_Toc376531559)

[1.1.2 IBM Rational Rose 11](#_Toc376531560)

[1.1.3 UModel 14](#_Toc376531561)

[1.1.4 Visual Paradigm for UML 17](#_Toc376531562)

[1.1.5 Enterprise Architect 21](#_Toc376531563)

[1.2 Сравнительная характеристика существующих систем   
 верификации диаграмм последовательности 26](#_Toc376531564)

[2 ОБЩИЕ МОДЕЛИ И МЕТОДОЛОГИИ РАЗРАБОТКИ   
 ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ПРОГРАММНЫХ СИСТЕМ 27](#_Toc376531565)

[2.1 Модели разработки 27](#_Toc376531566)

[2.1.1 Каскадная модель 28](#_Toc376531567)

[2.1.2 Инкрементальная модель 29](#_Toc376531568)

[2.1.3 Спиральная модель 33](#_Toc376531569)

[2.1.4 Модель прототипирования 35](#_Toc376531570)

[3 РАЗРАБОТКА СТРУКТУРЫ ДЛЯ ПОДСИСТЕМЫ   
 ВЕРИФИКАЦИИ ДИАГРАММЫ КЛАССОВ 37](#_Toc376531571)

[3.1 Задачи, выполняемые подсистемой верификации   
 диаграммы классов 37](#_Toc376531572)

[3.2 Архитектура системы верификации UML-диаграмм «UML Tester» 37](#_Toc376531573)

[3.3 Структура подсистемы верификации диаграммы классов 39](#_Toc376531574)

[3.3.1 Модуль обработки XML данных 39](#_Toc376531575)

[3.3.2 Модуль верификации диаграммы классов 45](#_Toc376531576)

[3.4 Используемые методы верификации диаграмм классов  
 и их реализация 46](#_Toc376531577)

[3.4.1.1 Метод шаблонов 46](#_Toc376531578)

[3.4.1.2 Реализация метода шаблонов 49](#_Toc376531579)

[3.4.2.1 Метод идентификационного графа 51](#_Toc376531580)

[3.4.2.2 Реализация метода идентификационного графа 53](#_Toc376531581)

[3.4.3.1 Метод множеств 55](#_Toc376531582)

[3.4.3.2 Реализация метод множеств 56](#_Toc376531583)

[3.4.4.1 Метод тестового драйвера 58](#_Toc376531584)

[3.4.4.2 Реализация метод тестового драйвера 62](#_Toc376531585)

[4 ЭКСПЕРЕМЕНТАЛЬНАЯ ЧАСТЬ 64](#_Toc376531586)

[4.1 Результаты реализации верификатора «UML Tester» 64](#_Toc376531587)

[4.2 Результаты тестирования верификатора «UML Tester» 65](#_Toc376531588)

[4.2.1 Тестирование метода шаблонов 65](#_Toc376531589)

[4.2.2 Тестирование метода идентификационного графа 66](#_Toc376531590)

[4.2.3 Тестирование метода тестового драйвера 67](#_Toc376531591)

[4.2.4 Тестирование работы методов в комплексе 68](#_Toc376531592)

[4.3 Выводы 69](#_Toc376531593)

[5 ОХРАНА ТРУДА И БЕЗОПАСНОСТЬ В ЧРЕЗВЫЧАЙНЫХ СИТУАЦИЯХ 70](#_Toc376531594)

[5.1 Характеристика системы управления охраной труда в учреждении 70](#_Toc376531595)

[5.2 Анализ условий труда на рабочем месте 74](#_Toc376531596)

[5.3 Обеспечение производственной санитарии в условиях производства 79](#_Toc376531597)

[5.3.1 Предложения по улучшению условий труда 79](#_Toc376531598)

[5.3.2 Расчет требуемого воздухообмена в помещении за   
 опасными веществами 79](#_Toc376531599)

[5.4 Безопасность в чрезвычайных ситуациях 81](#_Toc376531600)

[ВЫВОДЫ 85](#_Toc376531601)

[ПЕРЕЧЕНЬ ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ 86](#_Toc376531602)

[ПРИЛОЖЕНИЯ 88](#_Toc376531603)

[ПРИЛОЖЕНИЕ А 89](#_Toc376531604)

ВВЕДЕНИЕ

Диаграмма классов является одной из основных составляющих модели любого программного обеспечения [1]. Именно поэтому особенное значение имеет верификация диаграмм классов. На данный момент существует достаточно большое количество различных подходов к верификации диаграмм классов. Большинство из них позволяют оценить корректность диаграммы классов с различной точки зрения.

Данная работа является частью научной работы профессора кафедры программной инженерии Литвинова В.В и аспиранта Богдан И.В. по верификации и тестированию программного обеспечения.

Целью работы является разработка архитектуры автоматизированной системы верификации UML диаграмм классов, которая позволит более корректно оценить модель, где под моделью подразумевается множество UML диаграмм. Эта система позволит разработчикам программного обеспечения устранить множество ошибок с программным обеспечением еще на этапе проектирования. Так же целью является разработка собственного метода верификации основанного на построении тестового драйвера.

В отличие от существующих, данная система позволяет в комплексе несколькими методами оценивать корректность диаграмм классов. Методы позволяют оценивать корректность, как различных аспектов диаграммы классов, так и позволяют оценить один и тот же аспект с точки зрения различных подходов верификации.

Новизна заключается в разработке собственного метода верификации на основе тестового драйвера. Драйвер позволит более корректно выполнить верификацию диаграммы классов.

Практическую ценность имеет разработанный комплексный подход верификации диаграмм классов, а так же конечный продукт на основе этого подхода. Это позволить экономить ресурсы и время на этапе перехода от программной модели к ее конечной реализации.

Апробация результатов работы заключается в выступлении, с данной темой, на университетской научной конференции в апреле 2013 года.

1 АНАЛИЗ СУЩЕСТВУЮЩИХ СИСТЕМ ВЕРИФИКАЦИИ UML-ДИАГРАММ

Понятие CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное его значение было ограниченно только задачами автоматизации разработки программного обеспечения. Современные же CASE-средства охватывают обширную область поддержки многочисленных технологий проектирования информационных систем: от простых средств анализа и документирования до полномасштабных средств автоматизации, покрывающих весь жизненный цикл программного обеспечения.

Наиболее трудоемкими этапами разработки информационных систем являются этапы анализа и проектирования, в процессе которых CASE- средства обеспечивают качество принимаемых технических решений и подготовку проектной документации. При этом большую роль играют методы визуального представления информации. Это предполагает построение структурных или иных диаграмм, использование многообразной цветовой палитры, сквозную проверку синтаксических правил. Графические средства моделирования предметной области позволяют разработчикам в наглядном виде изучать существующую информационную систему, перестраивать ее в соответствии с поставленными целями и имеющимися ограничениями.

В разряд CASE- средств попадают как относительно дешевые системы для персональных компьютеров с весьма ограниченными возможностями, так и дорогостоящие системы для неоднородных вычислительных платформ и операционных сред. Современный рынок программных средств насчитывает более 300 различных CASE-средств.

Не смотря на большое количество CASE-средств - только некоторые из них выполняют верификацию программного обеспечения и, в частности, верификацию моделей программного обеспечения. Под моделью чаще всего подразумевается множество UML-диаграмм (диаграмма вариантов использования, диаграмма классов, диаграмма последовательности, диаграмма состояний и другие диаграммы из базового набора и не только).

1.1 Существующие системы верификации UML-диграмм

1.1.1 ArgoUML

ArgoUML — средство UML моделирования. ArgoUML является открытым программным обеспечением и распространяется под лицензией EPL. ArgoUML полностью написан на языке Java и для работы ему подходит любая операционная система с установленной Java 2 JRE или JDK версии 1.4 или выше [2].

Функциональность ArgoUML включает в себя:

* поддержку спецификаций UML 1.3, 1.4, XMI 1.0, 1.1, 1.2;
* 9 видов диаграмм UML (диаграммы классов, состояний, кооперации, последовательности, деятельности, прецедентов, объектов, компонентов, развёртывания);
* поддержку OCL для классов;
* генерацию исходного кода Java, C++, C# и PHP;
* обратный инжиниринг из исходного кода и байткода Java;
* автоматическую верификацию модели UML (design critics);
* возможность экспорта диаграмм в GIF, PNG, PS, EPS, SVG и PGML;
* возможность работы на десяти языках: EN, EN-GB, DE, ES, IT, RU, FR, PT NB ZH.

Начиная с Java 1.5 приложение являет платформенно-независимым, так же существует возможность работы с приложением через WEB интерфейс, таким образом, отсутствует необходимость установки приложения локально. Интерфейс ArgoUML не является перегруженным пользовательскими компонентами, поэтому он интуитивно понятен и позволяет быстро к нему адаптироваться неопытному пользователю.

К недостаткам данного приложения можно отнести нереализованность части функционала, а также отсутствие поддержки UML 2. ArgoUML позволяет строить диаграмму классов, однако, что касается её верификации, то пакет имеет как недостатки, так и некоторые преимущества.

Так в ArgoUML отсутствует верификация ассоциаций. Среда позволяет строить не корректную диаграмму классов (рисунок 1.1).



Рисунок 1.1 – Отсутствие верификации ассоциации в ArgoUML

Данный пример иллюстрирует отсутствие верификации ассоциаций в ArgoUML.

Так же данный пакет не выдает никаких сообщений касательно ошибок наследования, так множественное наследование является допустимым в ArgoUML (рисунок 1.2).



Рисунок 1.2 – Отсутствие верификации правил наследования в ArgoUML

Верификация правил наследования в ArgoUML отсутствует. Более того генератор кода Java для дочернего класса “Class A” выдает следующее:

public class Class A extends Parent 2, Parent 1 {

private Integer attribute1;

public String method1() {

return null;

}

}

Данный код не подлежит компиляции, т.к. компилятор выдаст информацию об ошибочном синтаксисе, содержащимся в описании класса.

Таким образом, после ряда тестов по построению диаграмм классов в среде ArgoUML, следует отметить что кроме определенного набора шаблонов верификации всех типов диаграмм, которые описаны в разделе Critics данного пакета, он не позволяет надежно верифицировать диаграммы классов.

Этот пакет актуален лишь в том случае, если нужно построить диаграмму классов небольшого размера не вникая в детали верификации, например, схематически изобразить ту или иную модель. Так же большим плюсом является открытость кода данной среды, следовательно любой программист может помочь улучшить инструментарий верификации в этом пакете.

1.1.2 IBM Rational Rose

Rational Rose - CASE-средство фирмы IBM (раньше Rational Software Corporation) - предназначено для автоматизации этапов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Rational Rose использует синтез-методологию объектно-ориентированного анализа и проектирования, основанную на подходах трех ведущих специалистов в данной области: Буча, Рамбо и Джекобсона [3]. Разработанная ими универсальная нотация для моделирования объектов (UML - Unified Modeling Language) является стандартом в области объектно-ориентированного анализа и проектирования. Именно в IBM Rational Rose язык UML стал базовой технологией визуализации и разработки программных систем, что определило популярность и стратегическую перспективность этого инструментария.

Конкретный вариант Rational Rose определяется языком, на котором генерируются коды программ:

- Java;

- C++;

- Smalltalk;

- PowerBuilder;

- Ada;

- SQLWindows;

- ObjectPro.

В рамках общего продукта IBM Rational Rose существуют различные варианты этого средства, отличающиеся между собой диапазоном предоставляемых возможностей. Базовым средством в настоящее время является IBM Rational Rose Enterprise Edition, которое обладает наиболее полными возможностями. Последней версией этого CASE-средства на данный момент является программа IBM Rational Rose 7.0.0, возможности которой аккумулируют практически все современные достижения в области информационных технологий. Наиболее характерные функциональные особенности этой программы заключаются в следующем [4]:

- интеграция с MS Visual Studio, которая включает поддержку на уровне прямой и обратной генерации кодов и диаграмм Visual Basic и Visual С++ с использованием ATL (Microsoft Active Template Library), Web-Классов, DHTML и протоколов доступа к различным базам данных;

- непосредственная работа (инжиниринг и реинжиниринг) с исполняемыми модулями и библиотеками форматов EXE, DLL, TLB, OCX.

- поддержка технологий MTS (Microsoft Transaction Server) и ADO (ActiveX Data Objects) на уровне шаблонов и исходного кода, а также элементов технологии Microsoft - COM+ (DCOM);

- полная поддержка компонентов CORBA и J2EE, включая реализацию технологии компонентной разработки приложений CBD (Component-Based Development), языка определения интерфейса IDL (Interface Definition Language) и языка определения данных DDL (Data Definition Language);

- полная поддержка среды разработки Java-приложений, включая прямую и обратную генерацию классов Java формата JAR, а также работу с файлами формата CAB и ZIP.

В основе работы Rational Rose лежит построение различного рода диаграмм и спецификаций, определяющих логическую и физическую структуры модели, ее статические и динамические аспекты. В их число входят диаграммы классов, состояний, сценариев, модулей, процессов.

Для организации групповой работы в Rational Rose возможно разбиение модели на управляемые подмодели. Каждая из них независимо сохраняется на диске или загружается в модель. В качестве подмодели может выступать категория классов или подсистема.

Верификации диаграмм классов, в Rational Rose присутствует, но не в полной мере. Стоит отметить, что в этой среде присутствует верификация множественного наследования (рисунок 1.3).

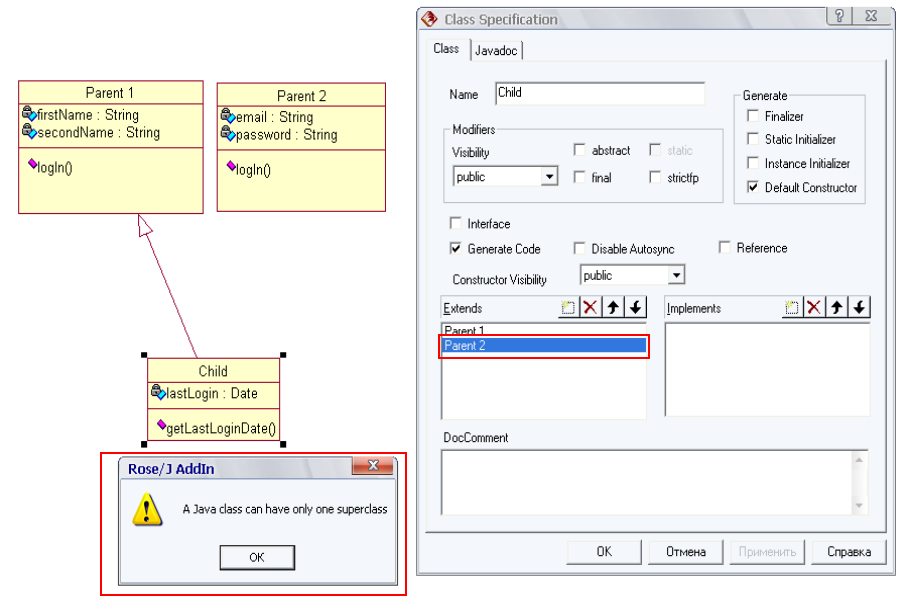


Рисунок 1.3 – Сообщение об ошибке, при попытке создания диаграммы с множественным наследованием

Как видно из рисунка 1.3, при попытке добавления второго суперкласса для класса Child возникает сообщение об ошибке, которое сигнализирует о не корректности данного действия.

Проверка корректности ассоциаций частично реализована, и проявляется на этапе построения диаграммы. Если пользователь пытается определить мощность ассоциации как не корректную (max < min), то Rational Rose выдает сообщение об ошибке (рисунок 1.4).

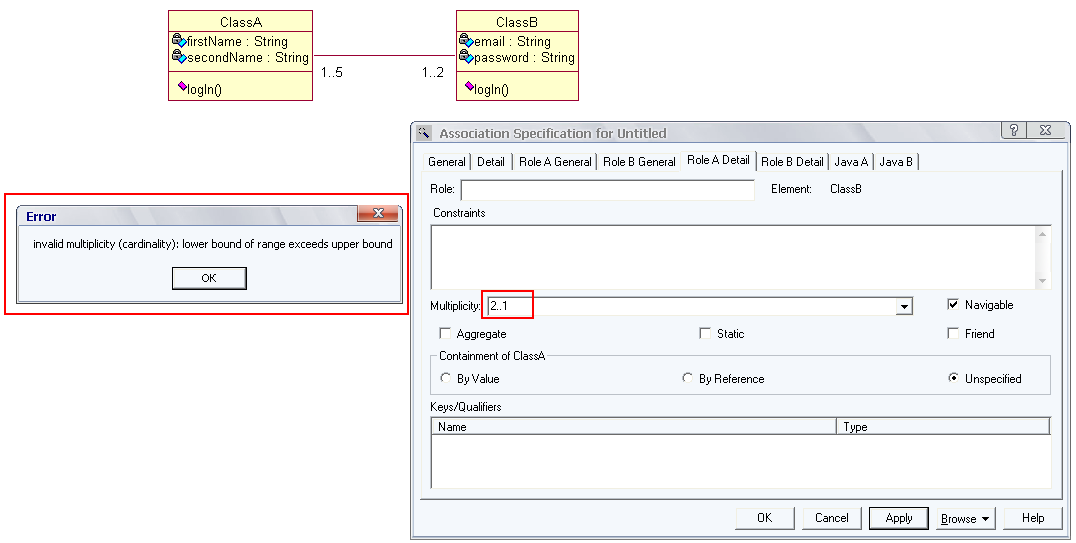


Рисунок 1.4 – Выявление некорректности в мощности ассоциации

Исходя из сообщения, полученного при построении некорректной диаграммы, следует отметить, что необходимости в отдельном модуле верификации ассоциаций нет.

Следующее что следует проверить в данной среде, - это наличие верификации для сложно зависимых диаграмм. На рисунке 1.5 представлена диаграмма с противоречивыми связями ассоциации и наследования.

Rational Rose сообщения о не корректности такого типа не выводит. Исходя из этого, делаем вывод, что верификация такого рода отсутствует.

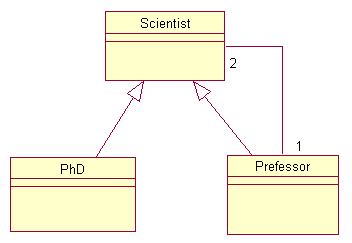


Рисунок 1.5 – Диаграмма классов с противоречивыми связями ассоциации и наследования

Подводя итог по Rational Rose необходимо выделить очень удобный пользовательский режим. В Rational Rose предусмотрено предотвращение некоторых ошибок проектирования непосредственно на этапе построения модели, но при этом отсутствует полноценная верификация диаграммы классов.

1.1.3 UModel

Программный продукт UModel, разработанный компанией Altova, - это UML-редактор с функцией замкнутого инжиниринга. Он поддерживает UML 2.1.1 и реализует кодогенерацию и обратный реинжиниринг для языков программирования C#, VB.NET и Java. Дополнительно UModel реализует диаграммы для работы с XML-схемами и нотацией BPMN и реализует плагины для Microsoft Visual Studio и Eclipse.

UModel поддерживает все типы диаграмм, входящих в стандарт UML 2.1.1 [5]. Это структурные диаграммы:

- диаграммы классов и объектов;

- диаграммы с композитной структурой, описывающие взаимодействие объектов при решении конкретных задач, внутреннюю структуру объектов, стратегии;

- архитектурные диаграммы - пакеты, диаграммы пакетов, диаграммы компонентов и развертывания;

- диаграммы поведения - деятельности, конечные автоматы, прецедентов, коммуникации, последовательности, обзора взаимодействия и синхронизации.

UModel реализует функции замкнутого инжиниринга - позволяет генерировать код на основе UML-диаграмм, создавать UML-диаграммы на основе имеющегося кода и выполнять автоматическую синхронизацию кода и модели. Поддерживается кодогенерация для Java 1.4, Java 5.0, Java 6.0, C# 1.2, C# 2.0, C# 3.0, VB 7.1, VB8.0 и VB 9.0.

Для того чтобы в UModel создать новый класс и сгенерировать для него рабочий код, достаточно выполнить всего несколько шагов. UModel проверяет синтаксис, сообщает обо всех потенциальных проблемах (например, тип возвращаемых методом данных не указан) и генерирует код. Код генерируется на основе шаблонов кодогенерации, заданных в отдельных файлах. Шаблоны можно отредактировать - учесть необходимые соглашения об именовании, добавить поддержку использования сторонних библиотек. Шаблоны реализованы на языке Spy programming language (SPL), активно применяемом в продуктах компании Altova.

Помимо стандартных UML-диаграмм, UModel реализует две дополнительные - диаграммы для XML-схем и BPMN-диаграммы (Business Process Modeling Notation).

Другое дополнение - диаграммы BPMN. BPMN-диаграммы являются [стандартом OMG](http://www.bpmn.org/) и активно применяются при моделировании бизнес-процессов бизнес-аналитиками, техническими разработчиками, бизнес-менеджерами. UModel не только полностью реализует возможности диаграмм BPMN, но и дополняет их полезными возможностями.

UModel реализует полноценную поддержку импорта и экспорта данных в формате [XMI версии 2.1](http://ru.wikipedia.org/wiki/XMI), соответствующей стандарту UML 2.1.1. Такая поддержка делает возможным обмен данными между UModel и другими UML-инструментами, разработанными, в частности, сторонними разработчиками. Для удобства в UModel реализована обратная совместимость с UML 2.1 и UML 2.0 - на случай, если потребуется открыть созданные диаграммы в UML-редакторе, не поддерживающем UML 2.1.1. Поддержка XMI позволяет создавать "стандартные" архивные копии проектов UModel, которые можно открывать с помощью любых современных UML-редакторов.

Несмотря на популярность и постоянное развитие данного программного продукта верификация модели реализована лишь частично. Отсутствует верификация корректности ассоциаций (рисунок 1.6), а так же проверка корректности множественного наследования (рисунок 1.7). Процесс проверки модели в этих случаях не дает сообщения об ошибках (рисунок 1.8).

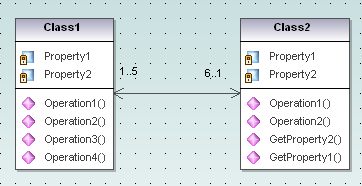


Рисунок 1.6 – Отсутствие верификации ассоциаций в UModel

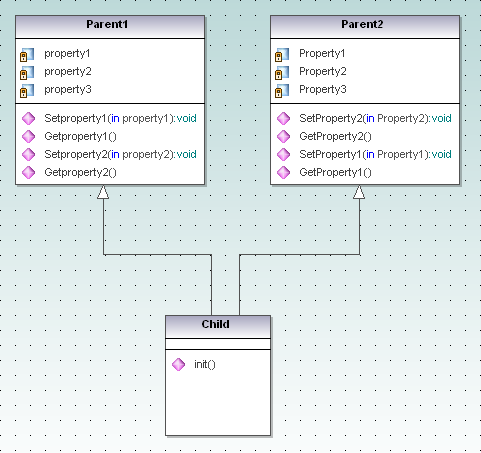


Рисунок 1.7 – Отсутствие верификации множественного наследования в UModel

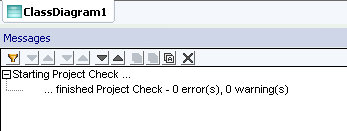


Рисунок 1.8 – Сообщение об отсутствие некорректностей

В пакете UModel присутствуют сообщения, которые предотвращают возможность построения диаграммы с циклической иерархией (рисунок 1.9).

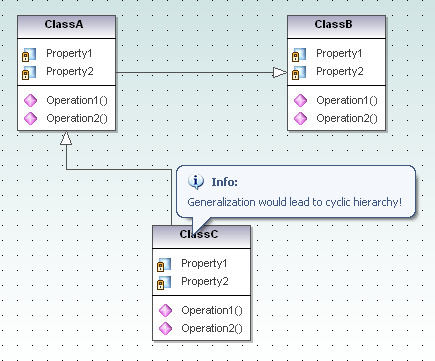


Рисунок 1.9 – Сообщение о циклической иерархии в UModel

При попытке добавления связи наследования классом ClassC от класса ClassB Umodel выдает сообщение о том, что данная связь будет некорректной.

1.1.4 Visual Paradigm for UML

Программное обеспечение Visual Paradigm for UML является средством моделирования систем на языке UML и с помощью UML-диаграмм всех типов [6]. Кроме UML, решение Visual Paradigm for UML работает с такими ключевыми в индустрии стандартами, как язык моделирования систем (SysML), графическая нотация моделирования бизнес-процессов (BPMN), стандарт XMI и т. п. Visual Paradigm for UML поддерживает весь рабочий цикл разработки – объектно-ориентированный анализ, фиксацию требований, планирование, моделирование, тестирование и многое другое. С помощью инструментов данного приложения можно вычерчивать все типы классовых диаграмм, просматривать в обратном порядке код, генерировать документацию и код с диаграмм. Продукт Visual Paradigm for UML рассчитан на широкий круг пользователей, включая системных и бизнес-аналитиков, системных инженеров и архитекторов, других специалистов, занятых в написании программ.

Возможности Visual Paradigm for UML:

- проектирование программного обеспечения с помощью UML – создание диаграмм прецедентов, документирование потока событий, генерация диаграмм активности и организация прецедентов с помощью грида;

- поддержка профиля UML – диаграмма профиля позволяет визуально проектировать стереотипы и тегированные значения;

- проектирование образцов – назначение диаграммы или ее частей в качестве образца для последующего использования в других проектах;

- создание диаграмм вариантов использования – визуализация системных функций или требований с помощью построения диаграмм прецедентов, которые содержат главных действующих лиц и основные варианты использования;

- запись потока событий – документирование взаимосвязей между действующими лицами и функциями с помощью редактора потока событий;

- настройка вариантов тестирования – тестирование событий на предмет приемлемости для пользователей;

- поддержка различных форматов – создание полностью настраиваемых отчетов с диаграммами бизнес-процессов и прецедентов в форматах PDF, HTML и Microsoft Word;

- публикация проекта на web-сервере – сохранение моделей в web-формате для обмена наиболее актуальными и точными бизнес-процессами и проектами системы с клиентами, членами рабочей группы и другими сотрудниками отдела;

- замкнутый (round-trip) инжиниринг из исходного кода Java – визуализация взаимосвязей между Java-объектами с помощью замкнутой разработки исходного кода Java в стандартные UML-диаграммы последовательностей;

- генерация исходного кода Java из UML-модели классов или обратный инжиниринг кода в UML-модель, синхронизация модели с исходным кодом.

- замкнутая разработка C++ – генерация, циклический инжиниринг и синхронизация между ANSI C++ и UML-диаграммами классов.

В целом средство является достаточно удобным и интуитивно понятным, но имеет ряд недостатков. Основным минусом является коммерческое использование, а также необходимость дополнительно скачивать документацию.

Что касается верификации диаграммы классов, то стоит выделить, что как и в ArgoUML VisualParadigm не поддерживает верификации ассоциаций (рисунок 1.10), а при генерации кода просто игнорируется факт ассоциации.



Рисунок 1.10 – Отсутствие верификации ассоциации в Visual Paradigm

Следующим шагом проверки уровня верификации в Visual Paradigm, является тест с неправильным наследованием. Построить такую диаграмму удалось без проблем (рисунок 1.11).



Рисунок 1.11 – Диаграмма с неверным наследованием в Visual Paradigm

Но в отличии от ArgoUML Visual Paradigm правильно отреагировал на допущенную ошибку наследования, выдав предупредительное сообщение (рисунок 1.12), сообщающее пользователю что класс Child не может наследоваться более чем от одного суперкласса.



Рисунок 1.12 – Сообщение об ошибке наследование

Так же была создана диаграмма классов, содержащая неабстрактный класс в котором присутствует абстрактный метод (рисунок 1.13).



Рисунок 1.13– Неабстрактный класс с абстрактным методом

Visual Paradigm сообщает пользователю, что абстрактная операция abstractMethod() может быть создана только в абстрактном классе (рисунок 1.14).



Рисунок 1.14 – Сообщение об ошибочном объявлении абстрактного метода в неабстрактном классе

Следующее что следует проверить в данной среде, - это наличие верификации для более сложных диаграмм. Одина из таких диаграмм представлена на рисунке 1.15.



Рисунок 1.15 – Диаграмма классов с противоречивыми связями наследования и ассоциации в Visual Paradigm

Данная диаграмма является противоречивой и не должна проходить верификацию, но Visual Paradigm не выдал сообщения об ошибке.

Последним шагом в определении качества верификации в данной среде является проверка на выявления ошибок мультизависимых классов. На рисунке 1.16 представлена диаграмма, при верификации которой должно выдаваться сообщение об ошибке.



Рисунок 1.16 – Не корректная диаграмма с противоречивыми зависимостями

В данном случае сообщения об ошибке тоже выведено не было, это свидетельствует о том, что верификация сложнозависымых диаграмм в Visual Paradigm не заложено.

Таким образом, необходимо отметить, что Visual Paradigm предоставляет лишь частичную верификацию диаграмм классов.

1.1.5 Enterprise Architect

[Enterprise Architect](http://www.sparxsystems.com.au/products/ea.html) (EA) – CASE-инструмент для проектирования и конструирования программного обеспечения. EA поддерживает спецификацию UML2.0+, описывающую визуальный язык, которым могут быть определены модели проекта [7].

Некоторые из ключевых функций ЕА:

- создание элементов UML-моделей широкого круга назначения;

- размещение этих элементов в диаграммах и пакетах;

- создание коннекторов между элементами;

- документирование созданных элементов;

- генерация кода для конструируемого ПО;

- реверс-инжиниринг имеющегося кода на некоторых языках.

Используя EA, можно выполнять форвард и реверс-инжиниринг ActionScript, C++, C#, Delphi, Java, Python, PHP, VB.NET and Visual Basic классов, синхронизировать код и элементы моделей, проектировать и генерировать элементы баз данных. Из моделей может быть быстро создана документация в стандартном rtf-формате и импортирована в Word для финального редактирования, так же доступна генерация HTML-документов.

EA поддерживает все модели/диаграммы UML 2.0. С его помощью можно моделировать бизнес-процессы, веб-сайты, пользовательские интерфейсы, сети, конфигурации аппаратного обеспечения, сообщения и т.д., оценивать размер трудозатрат проектных работ в часах, фиксировать и трассировать требования, ресурсы, тест-планы, дефекты и запросы на изменения.

Таким образом, EA – современный инструмент, который поддерживает все аспекты цикла разработки, обеспечивая полную трассировку от начала проектирования до размещения и поддержки. Также он обеспечивает поддержку тестирования, управления сопровождением и изменениями.

Enterprise Architect является проприетарным программным обеспечением, соответственно, доступен не всем, не позволяет проследить алгоритм работы и увидеть исходный код. Также существует мало документации по верификации для данного пакета, что также может вызвать некоторые затруднения при работе документацию.

Для оценки корректности верификации в пакете Enterprise Architect было проведено тестирование на наличие верификации ассоциаций, наследования, сложнозависымых диаграмм, так же как и предыдущие среды не поддерживает верификации ассоциаций (рисунок 1.17), выводится сообщение о наличии нуля ошибок и предупреждений (рисунок 1.18).



Рисунок 1.17 – Отсутствие верификации ассоциации в Enterprise Architect



Рисунок 1.18 – Отсутствие сообщений об ошибках при верификации не правильной модели.

Преимущество над предыдущими средами заключается в присутствии ассоциации в сгенерированном коде.

Следующим шагом проверки уровня верификации в Enterprise Architect, является тест с неправильным наследованием. Построить такую диаграмму удалось без проблем (рисунок 1.19). На этой диаграмме содержатся как множественное наследование так и циклическое наследование, что запрещено во всех языках программирования.



Рисунок 1.19 – Диаграмма с недопустимым наследованием в Enterprise Architect

В результате верификации оказалось, что EA не обнаружил ни одной ошибки, и не выдал никаких предупреждений. Более того EA сгенерировал код, который является ошибочным, и не подлежит компиляции. К примеру, для класса ClassD было сгенерировано следующее:

public class ClassD extends ClassA ClassB ClassC {

public ClassD(){

}

public void finalize() throws Throwable {

super.finalize();

}

}//end ClassD

Так же была создана диаграмма классов, содержащая неабстрактный класс, в котором присутствует абстрактный метод (рисунок 1.20).



Рисунок 1.20 – Неабстрактный класс с абстрактным методом

Как и в предыдущих случаях, диаграмма успешно прошла верификацию, не выдав ни одного предупреждения. При генерации кода, так же не возникло никаких предупреждений о том, что диаграмма является не корректной, просто сгенерировав код, который не скомпилируется.

Таким образом, следует подчеркнуть, что проверка корректности ассоциаций и наследования в ЕА не осуществляется.

Как и в предыдущих средах проведено тестирование на наличие верификации противоречивых зависимостей. Одина из таких представлена на рисунке 1.21.



Рисунок 1.21 – Диаграмма с противоречивыми зависимостями в Enterprise Architect

Как и предыдущие среды, которые были рассмотрены, Enterprise Architect не выдал сообщения об ошибке.

Последним шагом в определении качества верификации в данной среде является проверка на выявления ошибок мультизависимых классов. На рисунке 1.22 представлена диаграмма, при верификации которой должно выдаваться сообщение об ошибке.



Рисунок 1.22 –Диаграмма с противоречивыми зависимостями

В данном случае сообщения об ошибке тоже выведено не было, это свидетельствует о том, что верификация противоречивых зависимостей в EA отсутствует.

ЕА является очень мощным инструментом для построения UML диаграмм, но для верификации диаграмм не обладает необходимым инструментарием.

1.2 Сравнительная характеристика существующих систем верификации диаграмм последовательности

В результате проведенных исследований была составлена сравнительная характеристика основных средств построения UML диаграмм (таблица 1.1).

Таблица 1.1 – Сравнительная характеристика основных сред построения диаграмм класс

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Свойство | AgroUML | Rational Rose | UModel | Visual Paradigm | Enterprise Architect |
| Поддержка UML 2.0 и выше | Нет | Да | Да | Да | Да |
| Открытость кода, бесплатное распространение | Да | Нет | Нет | Нет | Нет |
| Возможность создания диаграммы классов | Да | Да | Да | Да | Да |
| Возможность верификации диаграмм классов | Да | Да | Да | Да | Да |
| Наличие информации о используемых методах верификации диаграммы классов | Нет | Нет | Нет | Нет | Нет |
| Наличие рекомендаций по исправлению ошибок верификации | Да | Да | Да | Да | Да |

2 ОБЩИЕ МОДЕЛИ И МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ПРОГРАММНЫХ СИСТЕМ

2.1 Модели разработки

Модель жизненного цикла разработки системы - это обобщенное (абстрактное) описание процесса развития [8]. Это описание процесса, исходя из какой-то определенной точки зрения. Описывая модели процессов, как правило, говорят о таких действиях, как планирование модели данных, дизайн пользовательского интерфейса и т.д., однако они могут также включать документацию и описание ролей.

В моделях процесса можно встретить два принципиально разных подхода:

- интенсивное планирование;

- ускоренная разработка.

Интенсивное планирование (tugev planeerimine). Этот подход, который состоит в детализированном планировании видов деятельности и программного обеспечения и точном следовании плана на следующем витке разработки. Прогресс деятельности разработки измеряется с помощью той же схемы.

Ускоренная (agile), т.е. гибкая разработка, при которой планирование происходит по частям (постепенно), и благодаря чему возможно изменить ход процесса разработки при изменении требований заказчика. Внедрение agile процесса стало результатом быстро меняющихся потребностей клиентов. Процесс должен быть гибким и способным реагировать на изменение, расширение и адаптацию к новым требованиям продукта.

Если прежде происходило отнесение моделей разработки системы строго к той или иной категории, то теперь как считает Ян Соммервилль, на этом уровне жесткое деление нецелесообразно, так согласно довольно многим моделям можно запустить как ускоренный (гибкий), так и планируемый метод разработки.

На протяжении всей истории было предложены несколько общих моделей разработки систем и наиболее важные из них следующие:

- водопадная модель (каскадная);

- спиральная модель;

- инкрементная модель;

- прототипирование.

2.1.1 Каскадная модель

Водопадная (каскадная) модель (или классическая модель) является первой описанной моделью жизненного цикла программной системы, которая проистекала из обычных процессов производства, имевших место в строительстве и механике и пр. Модель описал Уинстон В. Ройс (Winston W. Royce) в 1970 году. Водопадная модель - наиболее старая и наиболее подвергнувшаяся критике модель процессов [9].

Согласно основной идее виды деятельности распределяют таким образом, чтобы каждое действие проходило последовательно отдельным этапом. Ройс разделил процесс на следующие основные этапы (стоит отметить, что перечень этапов варьируется в изложении различных авторов):

1. Определение требований. Этот этап можно разделить на две категории - системный анализ (все то, что окружает конкретное программное обеспечение), и анализ требований. Документируют поведение системы, производительность, интерфейс.

2. Планирование системного и программного обеспечения. Фокусируется на основных свойствах программы, таких как структуры данных, архитектура программного обеспечения, функции интерфейса, алгоритмические и процедурные детали. Качество проекта возможно оценить. Результат документируется.

3. Реализация и тестирование модулей. В проекте Описанная система в проекте программируется в виде модулей и комплектом программ, которые тестируются отдельно. Чем детальнее проект, тем проще и более механическим может быть этап реализации.

4. Интеграция и тестирование системы. Объединяют программы и модули, тестируют всю систему, после тестирования продукт передается заказчику. Во время тестирования сосредотачиваются как на логических деталях, так и на том, отвечает ли система требованиям в отношении функциональности (проверка достоверности).

5. Эксплуатация и сопровождение - это обычно самая длинная фаза. Систему изменяют, если пользователи находят ошибки, либо окружение и рабочая среда изменяются или клиент нуждается в новой функциональности. Фаза повторяет все предыдущие этапы в рамках изменения существующей системы.

Результатом каждой фазы является один или несколько документов, которые утверждаются. Следующая фаза не должна начинаться до того, пока предыдущая не завершена. У фаз имеются определенные совпадения, и информация передается от одной фазы к другой (рисунок 2.1).



Рисунок 2.1 - Водопадная модель

Логика водопадной (каскадной) модели часто изображается таким образом, что стрелки направлены только сверху вниз (от предыдущей фазы к следующей). К предыдущим шагам не возвратиться, так же, как и в жилищном строительстве жилого дома нельзя вернуться к фундаменту после установки крыши. В этом случае по модели было бы невозможно вообще ничего позже изменить в системе. Ройс тоже признал этот вариант в своей статье невозможным. Фактически Ройс добавил уже вначале для каждого этапа изображающую обратную связь стрелку. Тем не менее, возвращение из каждого этапа на предыдущий - невозможно. Если во время тестирования была обнаружена ошибка, чьи корни лежат в архитектуре, то согласно модели можно вернуться к изменениям в архитектуре только тогда, когда программное обеспечение уже используется (см. направления стрелок на рисунке).

Решения в отношении системы принимаются на ранней стадии и, в конечном счете, система не может быть такой, как этого хочет пользователь (требования рано замораживаются). К тому же заказчик сам не знает вначале, что он хочет и его неопределенность вполне естественна.

Водопадная модель подходит для больших систем, которые разрабатываются в нескольких местах сразу. В этом случае предварительное и корректное планирование позволит лучше координировать работы.

2.1.2 Инкрементальная модель

Один из ключевых вопросов разработки программного обеспечения - как справиться с изменениями, поскольку в случае крупномасштабных программных проектов наличие изменений неизбежно [10]. Предпринимательство изменяется, что ведет к изменению требований, создаются новые технологии, которые целесообразно применять для усовершенствования программных систем, и изменяются сами платформы, для которых создавались системы. Указанные изменения требуют переделки и стоят, как повторного анализа требований вместе с внедрением, так и с реализацией новых функциональных возможностей. Стоимость изменений должна быть настолько низкой, насколько это возможно. Таким образом, в процесс разработки необходимо внести те виды деятельности, которые помогут предвидеть изменения до того момента, когда их внесение потребует уже существенный объем работ. Например, создание прототипа поможет заранее показать клиенту основные характеристики системы. Изменения лучше всего делать тогда, когда их внесение по возможности дешево. На основании этого, имеют резон постепенные (инкрементальные) разработка и передача продукта. Таким образом, изменения можно вносить и в тех частях, которые еще не начали разрабатывать.

Смысловую путаницу вызывают понятия итеративной и инкрементальной разработки. Согласно Алистеру Кокбернц (Alistair Cockburn) здесь имеем дело с двумя разными моделями разработки:

- инкрементальная разработка;

- итеративная разработка.

Инкрементальная разработка - это поэтапная и следующая временным графикам стратегия, в которой разные части системы разрабатываются в разное время и разными темпами, и если одна часть готова, тогда ее интегрируют в систему. Альтернативной стратегией было бы решение кодировать все части системы, а затем интегрировать весь код сразу.

Итеративная разработка - это так называемая стратегия изменений, где предусматриваются переделка и исправление существующих компонентов системы. Альтернативная стратегия заключалась бы в планировании деятельности таким образом, чтобы всё делалось бы с первой попытки.

Согласно Яну Соммервиллю (Ian Sommerville) «итеративная модель разработки» скорее общее название для ряда так называемых гибридных моделей (в том числе, и инкрементальная и спиральная модели). Слово «итеративный» подчеркивает то, что действия в этой модели повторяются.

Независимо от того смысла, который стоит за итеративной разработкой, инкрементальная (поэтапная) разработка довольно однозначно описана в различных источниках.

Инкрементальная разработка может быть как плановая, так и гибкая. Модель обеспечивает построение вначале небольшой части системы с дальнейшим ее расширением в несколько этапов. Поэтапный подход позволяет разработчикам и будущим пользователям системы учиться, начиная с ранних итераций, получать обратную связь еще тогда, когда возможно сделать изменения, например, в архитектуре системы, без переписи всего кода. Инкрементальная модель представлена на рисунке 2.2.



Рисунок 2.2 - Инкрементальная разработка

Спецификация программного обеспечения, проект и реализация делятся на части (increment), которые разрабатываются одна за другой. Таким образом, уменьшается количество нуждающихся в переделке частей системы, и клиенты получают возможность в течение более длительного времени обдумать свои желания. К типичным действиям надо отнести еще то, что компоненты (части) изготавливаемой системы находятся в использовании, что помогает клиенту получить большую ясность в своих дальнейших требованиях, предъявляемых к продукту.

Ход действий заключается в следующем: прежде всего, определяются требования в более общей форме и подразделяются на более и менее важные. Затем определяются части поставки, то есть, в каком количестве и из чего состоящие поставки начнет получать клиент в качестве своего программного обеспечения. Под поставкой понимаются части системы, то есть инкремент (increment). Каждая поставка должна добавлять к системе определенную функциональность. При этом выпуск начинают с компонентов (частей) с наивысшим приоритетом. Когда части системы определены, берут первую часть и начинают ее детализировать, используя для этого наиболее подходящий процесс (и почему бы не водопадную модель). В то же время можно уточнять требования и для других частей, которые в текущей совокупности требований данной работы были заморожены. Если очень необходимо, можно позже возвратиться к этой части. Если часть готова, то она поставляется клиенту (заказчику), который может использовать ее в работе (или, по крайней мере, серьезно испытать). Это позволит клиенту уточнить требования для следующих компонентов (частей) (или для последующих версий того же самого компонента). Затем занимаются со следующей частью системы. Новые части состыковываются с существующей системой. Все части не должны разрабатываться с использованием одного и того же процесса.

Преимущества инкрементальной (поэтапной) разработки:

1. Затраты, которые получаются в связи с изменением требований пользователей, уменьшаются, повторный анализ и совокупность документации значительно сокращаются по сравнению с водопадной (каскадной) моделью.

2. Легче получить отзывы от клиента о проделанной работе - клиенты могут озвучить свои комментарии в отношении готовых частей и тоже могут видеть, как много сделано. Таким образом, первые части системы являются как бы прототипом всей системы в целом.

3. У клиента имеется возможность быстро получить и освоить созданное программное обеспечение - клиенты могут получить реальные преимущества от системы раньше, чем это было бы возможно с водопадной моделью.

Проблемы инкрементальной (поэтапной) разработки:

1. Прогресс не очень хорошо прослеживаемый - менеджеры должны постоянно измерять прогресс процесса. В случае быстрой разработки не стоит создавать документы для каждого маленького изменения версии.

2. Структура системы имеет тенденцию к ухудшению при добавлении новых компонентов (частей) - постоянные изменения нарушают структуру системы. Чтобы избежать этого и повысить качество программного обеспечения требуется дополнительное время и деньги на рефакторинг. Плохая структура делает программное обеспечение сложным и дорогостоящим для последующих изменений.

Гибкие (ускоренные, agile) методы разработки

Для гибких методов разработки пригодно использование инкрементальной модели. Начало распространения гибкого метода разработки программного обеспечения уходит в 2001 год, когда те, кто отвечали за тогдашнее перепланирование методов разработки, подписали «Манифест гибкой разработки» (или просто «Гибкий манифест», «The Agile Manifesto»), в котором в наиболее важные пунктах внимание акцентируется на человеке и на взаимодействии между людьми:

- люди, и общение важнее, чем процессы и рабочие инструменты;

- работающая программа важнее, чем документация;

- сотрудничество с клиентами является более важным, чем переговоры по контракту;

- идти навстречу пожеланиям об изменениях более важно, чем следование плану.

Больше считаются с информацией, полученной в процессе обратной связи (нагрузочное тестирование, мнение пользователей), чем полагаются на предварительное тщательное планирование технологии. Основное внимание уделяется людям, в том числе пользователям и постоянному тестированию. С гибким методом достигается лучший результат за те же деньги, однако при использовании гибкого метода труднее заранее запланировать, когда какая-то функция программного обеспечения будет готова - «Agile process will provide the most bang for the buck, but won't say exactly when that bang will be». («Agile-процесс обеспечит максимальную отдачу, но не скажет точно, когда это произойдет».)

Наиболее известными и наиболее распространенными гибкими методами являются экстремальное программирование (XP), Scrum, Feature Driven Development (FDD), Open Unified Process (OpenUP) и др.

Экстремальное программирование, или XP, является наиболее известным гибким методом. В XP шаги делаются крайне (экстремально - отсюда и наименование метода) недолгими по сравнению с традиционными моделями разработки - первый законченный этап шагов может быть длительностью в несколько дней или неделю, тогда, как в классических моделях он продолжается месяцы и годы. Перед кодированием пишутся автоматизированные тесты, которые должно пройти программное обеспечение, затем программируют парами (то есть, два программиста за одним компьютером кодируют один программный отрывок - так называемое «парное программирование»). Если готовый код проходит тесты (испытания), то шаг программирования в данной итерации закончен.

2.1.3 Спиральная модель

Спиральная модель также является одной из итеративных моделей разработки.

Впервые спиральную модель описал Барри Боэм (Barry Boehm) в своей статье 1986 года [11]. Течение процесс изображает спираль. Первый виток (повторение) может быть, например, связан с исследованиями осуществимости системы, второй - с описанием требований, следующий - с планированием. Несколько витков в основном связаны с реализацией программного обеспечения, причем его создание происходит поэтапно (инкрементально). Однако, конечно, не следует уравнивать витки спирали с обычными фазами процесса разработки. Каждый виток разделен от 3 до 6 секторов (разные авторы делят по-разному). Каждый виток начинается планированием ближайших задач и оценкой рисков и заканчивается так называемым клиентом, то есть задача должна быть выполнена и протестирована. Трудоемкость секторов не должна быть одинаковой. Поэтому имеется 4 сектора (см. также рисунок 2.3):

1. Постановка задач (Objective setting) - определяются цели этой фазы, то есть витка, ограничения процесса, результаты, план управления, потенциальные риски и альтернативные стратегии, исходя из рисков.

2. Оценивание и сокращение рисков (Risk assessment and reduction) - для каждого найденного риска делается анализ, предпринимаются некоторые действия для сокращения рисков (например, риск, чьи требования не являются адекватными: изготавливается прототип).

3. Разработка и проверка достоверности (Development and validation) - выбирается модель разработки, исходящая из оцененных рисков (модель должна быть такой, чтобы помочь снизить риски). Например, если в пользовательском интерфейсе имеется самый большой риск, то тогда может помочь прототипирование.

4. Планирования (Planning) - проект рассматривается и делается решение о том, переходить ли на следующий виток, если решают продолжить, делается план для следующей фазы.

Пример спиральной модели изображен на рисунке 2.3. Реальный процесс разработки может видоизменяться в числе вариаций, так и в части расположения видов деятельностей.



Рисунок 2.3 - Спиральная модель

Наиболее важным отличием этой модели от других является учет рисков. Риск - это вероятность того, что что-то может пойти не так. Из-за материализации рисков превышаются сроки, и происходит перерасход средств, поэтому необходимо учитывать риски и принимать меры по их смягчению. По Соммервиллю точно такую модель используют редко, но она помогла понять природу итеративной разработки, и направила внимание на необходимость учета рисков.

2.1.4 Модель прототипирования

Прототип - это первоначальная версия системы, которая используются для апробирования возможностей дизайна и демонстрирования идей [12]. Прототипы можно использовать на различных фазах разработки. Например, на этапе анализа требований при их нахождении и проверке; на этапе дизайна при исследовании выбора возможностей и планировании пользовательского интерфейса.

Преимущества прототипов: лучшее удобство при использовании системы, более точная совместимость с реальными потребностями пользователей; более высокое качество и более лучшее удобство сопровождения и меньше трудностей при разработке. Процесс создания прототипа представлен на рисунке 2.4.



Рисунок 2.4 - Процесс создания прототипа

Этапы прототипирования являются следующими:

- накопление требований - это делают на более общем уровне и там же фиксируют то, что в дальнейшем необходимо начать обязательно уточнять;

- интенсивное планирование - сосредоточение внимания на видимой части (ввод, вывод, формы) в результате чего и создается прототип. Клиент имеет возможность оценить прототип и на этой основе уточнить свои желания;

- затем следует итерация улучшения прототипа до тех пор, пока это не удовлетворит пользователя. В то же время разработчик получает новые идеи исходя из пожеланий клиента.

При разработке прототипа важно, чтобы он мог бы быть быстро создан, используя вспомогательные инструменты (средства) (язык быстрого прототипирования и рабочие инструменты). Прототип не должен содержать в себе всей функциональности. Он должен сосредотачиваться на том, что не очень хорошо понятно. В прототипе не должно быть контроля ошибок, и прототип направлен на функциональные требования (например, на проблемы, связанные с безопасностью)

Прототипирование можно делать на различной основе - например, быстрое прототипирование (Throwaway prototyping), эволюционное прототипирование (Evolutionary prototyping), инкрементное прототипирование (Incremental prototyping).

Принципы быстрого прототипирования заключаются в следующем:

Такие прототипы должны быть уничтожены после создания, поскольку они не могут служить хорошей базой для реальной системы - например, не стоит на их основе выполнять нефункциональные требования, структура прототипа не подходит для дальнейшей разработки и также не отвечает требованиям качества модели.

Вкратце, в отличие от водопадной модели согласно итеративной модели разработки документ всестороннего анализа, который содержит неизменные требования пользователей и «подписывается на крови» между заказчиком системы и исполнителем, не составляется с самого начала. Итеративные модели позволяют проще вносить изменения в систему, получать от пользователей ранние отзывы, тестировать на ранней фазе пригодность проекта разработки архитектурного решения системы и т.п.

Не существует единственной и наилучшей модели разработки системы. Решение, какую модель выбрать, необходимо вынести исходя из конкретного программного проекта: результата, навыков и знаний команды, временных графиков, выяснения и стабильности потребностей клиентов.

3 РАЗРАБОТКА СТРУКТУРЫ ДЛЯ ПОДСИСТЕМЫ ВЕРИФИКАЦИИ ДИАГРАММЫ КЛАССОВ

Формальная верификация программного обеспечения – это приемы и методы формального доказательства (или опровержения) того, что модель программного обеспечения удовлетворяет заданной формальной спецификации [13]. Для того, чтобы доказать формально какое-либо утверждение относительно работы реальной программы, анализируемое программное обеспечение должно быть представлено формальной моделью. Формальная модель обычно проще самой проверяемой программы, это абстракция, в которой отражены наиболее существенные характеристики программного обеспечения. Таким образом, процесс создания любого программного обеспечения начинается с создания его модели, которой для эффективного дальнейшего использования необходимо пройти формальную верификацию.

3.1 Задачи, выполняемые подсистемой верификации диаграммы классов

Основными задачами, которые положены на подсистему верификации диаграмм классов, являются:

- верификация UML диаграмм классов;

- оценка корректности несколькими методами в комплексе;

- указание на место возникновения некорректности.

3.2 Архитектура системы верификации UML-диаграмм «UML Tester»

На рисунке 3.1 приведена архитектура инструментального средства UML Tester.

Утилита UML2 modeling tools

Модуль обработки XML данных

Обработка диаграммы классов

Обработка диаграммы последовательности

Модуль верификации

Верификация диаграммы последовательности

Верификация диаграммы

классов

Модуль обработки данных

Пользовательский интерфейс

Рисунок 3.1 – Архитектура инструментального средства «UML Tester»

Далее приводится последовательность разработки данного верификатора:

1. Выбор языка программирования и оптимальной утилиты для создания UML – диаграмм.

Главными критериями выбора утилиты были её открытость, а также возможность создания максимального количества существующих UML – диаграмм. Таким образом, выбор остановился на утилите uml2 modelingtools, которая является бесплатной, обеспечивает создание всех базовых диаграмм, а также интегрируется с открытой средой разработки Eclipse. Исходя из этого, Java был выбран как базовый язык для создания автоматизированной системы верификации.

2. Представление UML-диаграмм в виде XML-файлов.

В ходе исследований было обнаружено, что при создании диаграммы классов с помощью выбранной утилиты, появляются два файла: непосредственно сама диаграмма, а также файл \*.uml, в котором диаграмма представляется в виде XML-файла. Именно этот файл будет использоваться далее для парсинга диаграммы, так как формат XML является одним из простейших и существенно упрощает работу разработчика.

3. Реализация парсеров XML-файлов, представляющих диаграммы (получение необходимых ключевых данных).

После получения нужных диаграмм в формате XML, данные XML необходимо обработать и представить в виде java объектов. Получение этих объектов происходит при помощи вручную написанных парсеров c использованием библиотеки dom4j, которая позволяет парсить DOM-дерево для Java.

Парсер диаграммы классов позволяет получить названия классов, их идентификаторы, названия методов и атрибутов. Что касается парсера диаграммы последовательности, он позволяет получить линии жизни, сообщения, их идентификаторы, фрагменты, показывающие процесс передачи сообщений и их последовательность.

4. Реализация методов верификации. Идея, используемая в предлагаемом средстве верификации, состоит в том, что существующие и предложенные ранее похдоды верификации диаграмм используются в комплексе.

3.3 Структура подсистемы верификации диаграммы классов

Структурно подсистема верификации UML диаграмм состоит из двух основных модулей:

- модуль обработки XML данных (парсер);

- модуль верификации диаграммы классов.

3.3.1 Модуль обработки XML данных

Поскольку каждая UML диаграмма имеет свое представление в виде XML данных, то на первом этапе разработки верификатора диаграмм классов необходимо преобразовать XML данные в структуру данных в динамической памяти, с которой удобно работать при верификации.

Примером описания класса является XML структура данных следующего вида:

<packagedElement xmi:type="uml:Class" xmi:id="\_0MUd4P\_dEeGbE8gz2iVRag" name="XMLElement" isAbstract="true">

<ownedAttribute xmi:id="\_AZBEQP\_eEeGbE8gz2iVRag" name="name" visibility="private" aggregation="composite">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

<defaultValue xmi:type="uml:LiteralString" xmi:id="\_0eFdIP\_6EeGbE8gz2iVRag" value="ZIM"/>

</ownedAttribute>

<ownedAttribute xmi:id="\_AfzowP\_eEeGbE8gz2iVRag" name="id" visibility="public" aggregation="composite">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

</ownedAttribute>

<ownedOperation xmi:id="\_\_D8uwP\_dEeGbE8gz2iVRag" name="getId">

<ownedParameter xmi:id="\_LYtsAP\_eEeGbE8gz2iVRag" direction="return">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

</ownedParameter>

</ownedOperation>

<ownedOperation xmi:id="\_\_rUQwP\_dEeGbE8gz2iVRag" name="setId">

<ownedParameter xmi:id="\_XwhWMP\_eEeGbE8gz2iVRag" name="id">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

<defaultValue xmi:type="uml:LiteralString" xmi:id="\_XwhWMf\_eEeGbE8gz2iVRag"/>

</ownedParameter>

</ownedOperation>

<ownedOperation xmi:id="\_\_7ezsP\_dEeGbE8gz2iVRag" name="getName">

<ownedParameter xmi:id="\_OW1qAP\_eEeGbE8gz2iVRag" direction="return">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

</ownedParameter>

</ownedOperation>

<ownedOperation xmi:id="\_AFPzsP\_eEeGbE8gz2iVRag" name="setName">

<ownedParameter xmi:id="\_ZXlaoP\_eEeGbE8gz2iVRag" name="name">

<type xmi:type="uml:PrimitiveType" href="pathmap://UML\_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>

<defaultValue xmi:type="uml:LiteralString" xmi:id="\_ZXlaof\_eEeGbE8gz2iVRag"/>

</ownedParameter>

</ownedOperation>

</packagedElement>

В XML описании класса ключевыми являются такие элементы как: packagedElement – описывается любой элемент диаграммы, в зависимости от атрибутов, можно различать классы от интерфейсов и связей, ownedOperation – метод класса, атрибуты характеризуют метод, ownedAttribute – описывает поля класса.

Пример описания связи между классами (ассоциация):

<packagedElement xmi:type="uml:Association" xmi:id="\_U7BaoOnjEeGnKNahlmSO\_Q" name="agregacia"

memberEnd="\_U7BaoenjEeGnKNahlmSO\_Q \_U7CBsunjEeGnKNahlmSO\_Q">

<ownedEnd xmi:id="\_U7BaoenjEeGnKNahlmSO\_Q" name="src" type="\_5TlkkOniEeGnKNahlmSO\_Q"

association="\_U7BaoOnjEeGnKNahlmSO\_Q">

<upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="\_U7CBsenjEeGnKNahlmSO\_Q" value="1"/>

<lowerValue xmi:type="uml:LiteralInteger" xmi:id="\_U7CBsOnjEeGnKNahlmSO\_Q" value="1"/>

</ownedEnd>

<ownedEnd xmi:id="\_U7CBsunjEeGnKNahlmSO\_Q" name="dst" type="\_2H82YOnfEeGnKNahlmSO\_Q" aggregation="shared"

association="\_U7BaoOnjEeGnKNahlmSO\_Q">

<upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="\_U7CowenjEeGnKNahlmSO\_Q" value="1"/>

<lowerValue xmi:type="uml:LiteralInteger" xmi:id="\_U7CowOnjEeGnKNahlmSO\_Q" value="1"/>

</ownedEnd>

</packagedElement>

Здесь ключевым элементом является ownedEnd, - описывающий источник, или конец ассоциации.

Чтобы реализовать переход от XML представления диаграммы классов к динамической структуре данных, которую можно верифицировать, реализован XML-парсер. Диаграмма классов парсера, представлена на рисунке 3.2.

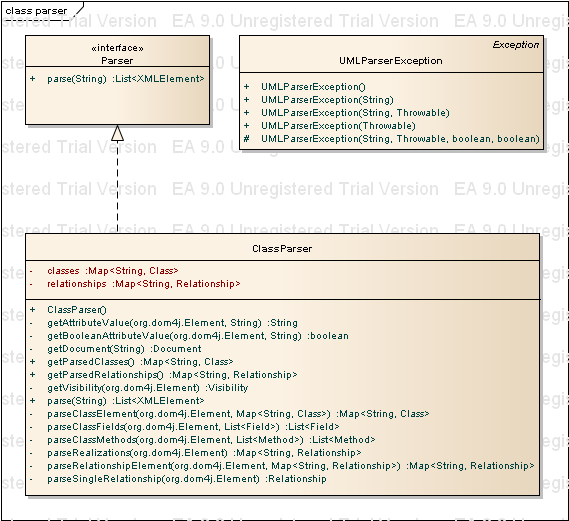


Рисунок 3.2 – Диаграмма классов парсера

На рисунке 3.3 приведен обобщенный участок кода Java для парсера XML структуры данных в динамическую модель.

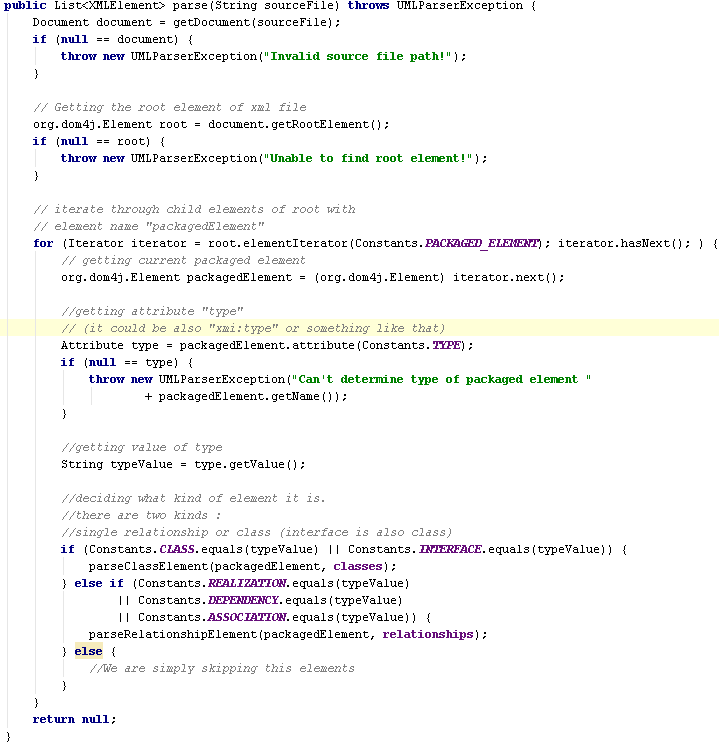


Рисунок 3.3 – Обобщенный код парсера

Принцип работы заключается в последовательном проходе по XML фалу, а именно по элементам типа “packagedElement”. Затем определяется тип элемента, к которому он принадлежит:

- класс (может содержать элементы наследования);

- интерфейс;

- имплементация;

- зависимость (агрегация или композиция);

- ассоциация.

Если текущий элемент является классом или интерфейсом – считываются все атрибуты для них и, если есть, все элементы наследования, которые являются вложенными. В других случаях парсер обрабатывает один из видов отношений (ассоциация, композиция, агрегация).

После отработки парсера, мы имеем в динамической памяти данные, с которыми может работать верификатор. Данными являются все необходимые для верификации составляющие. Диаграмма классов данных, извлеченных из XML файла представлена на рисунках 3.4 и 3.5.

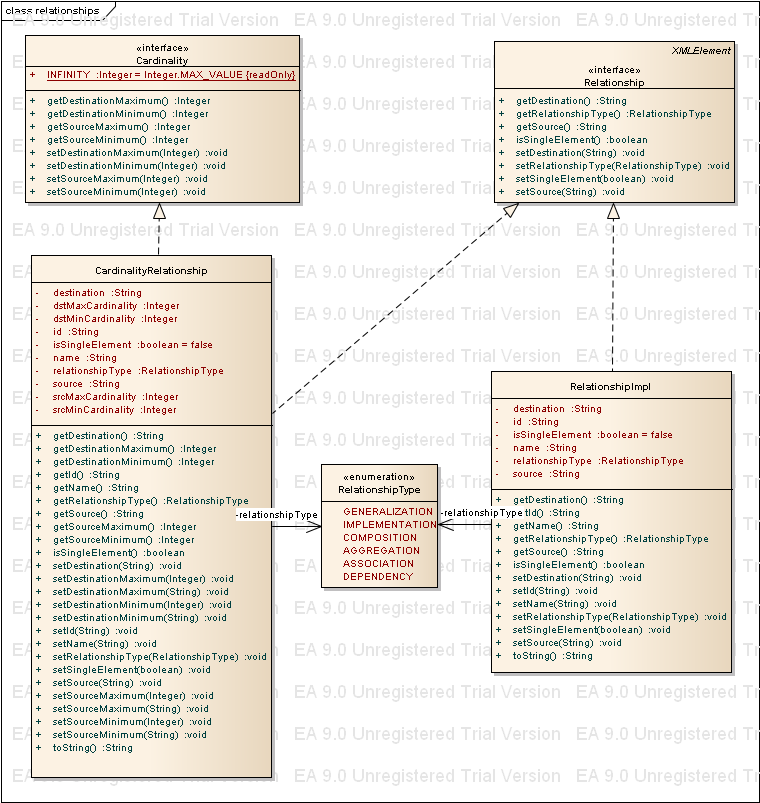


Рисунок 3.4 – Диаграмма классов модели для отношений

На диаграмме классов с рисунка 3.4 необходимо выделить два интерфейса: Relationship - который используется для отношений любого типа с целью обобщенной разработки, а так же интерфейс Cardinality, который используется с той же целью, но для мощностных отношений.

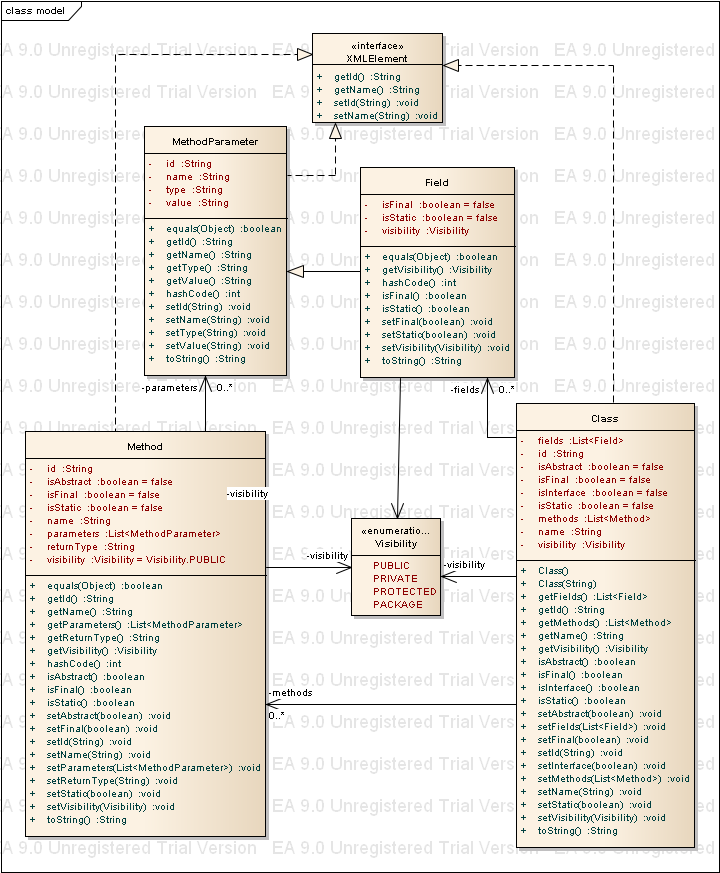


Рисунок 3.5 – Диаграмма классов модели для классов и интерфейсов

На диаграмме классов (рисунок 3.5) так же изображены все структурные элементы, присутствующие при объектно-ориентированном подходе разработки.

3.3.2 Модуль верификации диаграммы классов

Модуль верификации диаграммы классов предназначен для оценки корректности построения диаграммы классов, представленной на данном этапе в виде XML файла. Преимущество реализованной модели, в частности, заключается и в возможности проведения как последовательной, так и распараллеленной верификации. В данном проекте реализована верификация одних и тех же данных четырьмя методами верификации (рисунок 3.6):

- метод тестового драйвера;

- метод идентификационного графа;

- метод множеств;

- метод шаблонов.



Рисунок 3.6 – Принцип работы модуля верификации

Диаграмма классов модуля верификации представлена на рисунке 3.7. Как видно с диаграммы, модуль верификации является легко масштабируемым, и без внесения изменений в архитектуру проекта, представляется возможным добавить любое количество верификационных методов.

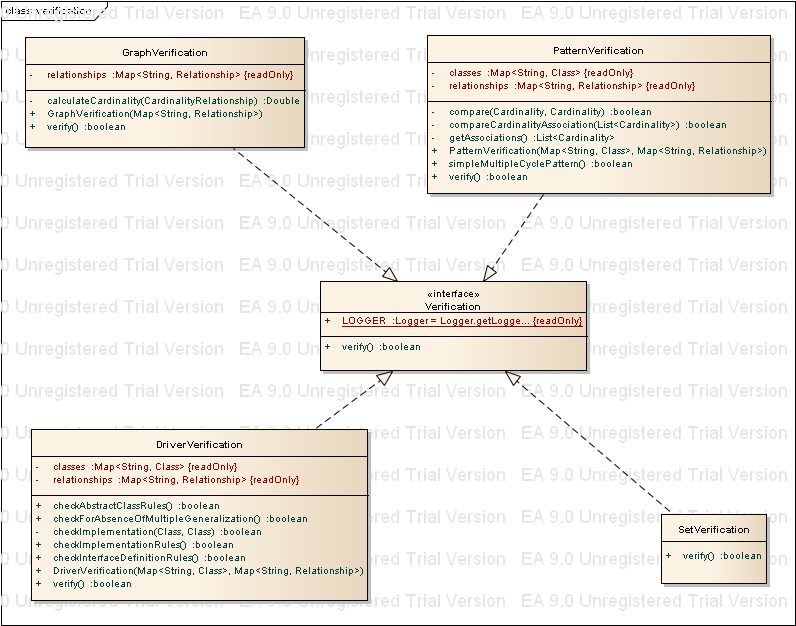


Рисунок 3.7 – Диаграмма классов модуля верификации

3.4 Используемые методы верификации диаграмм классов и их реализация

3.4.1.1 Метод шаблонов

Данный метод предложила 2004 году Мира Балабан [14]. Понятие шаблона в данном подходе отличается от хорошо известного и понятного понятия шаблона проектирования, который предоставляет собой формализованное описание часто встречающейся задачи проектирования, удачное решение данной задачи, а также рекомендации по применению этого решения в различных ситуациях. Шаблон согласно методу Мира Балабан напротив является примером ошибок при построении диаграммы классов.

Mira Balaban предложила несколько шаблонов. Один из самых популярных шаблонов носит название Цикл множественной иерархии (Multiplicity Hierarchy Cycle).

Согласно данному шаблону если между двумя классами имеется связь «наследование» (обобщение), которая означает наличие между ними отношения один-к-одному или же несколько дочерних классов к одному родительскому классу, то и любая другая связь между данными классами также должна указывать на наличие отношения один-к-одному или же несколько дочерних классов к одному родительскому классу.

На рисунке 3.8 представлен пример данного шаблона - диаграмма классов, на которой имеется класс «Ученый» и классы «Кандидат наук» и «Профессор», которые наследуются от данного класса. Таким образом, класс «Кандидат наук» связан с классом «Ученый» отношением один-к-одному или же несколько объектов класса «Кандидат наук» к одному классу объекта «Ученый», точно также класс «Профессор» связан с классом «Ученый отношением один-к-одному или же несколько объектов класса «Профессор» к одному классу объекта «Ученый». Но при этом класс «Ученый» связан с классом «Профессор» ассоциативной связью в соотношении: два объекта класса «Ученый» к одному классу объекта «Профессор». Таким образом, между двумя классами имеются две различные связи, одна из которых указывает на наличие отношения: два объекта класса «Ученый» к одному классу объекта «Профессор». А вторая на наличие отношения: один-к-одному или же несколько объектов класса «Профессор» к одному классу объекта «Ученый», что противоречит друг другу и шаблону «Цикл множественной иерархии».



Рисунок 3.8 - Пример шаблона «Цикл множественной иерархии»

Подвидом шаблона «Цикл множественной иерархии» является шаблон «Взаимодействие циклов множественной иерархии (Interaction of multiplicity and inter-association hierarchy constraints).

Согласно данному шаблону если между двумя классами имеется связь «наследование» (обобщение), которая означает наличие между ними отношения один-к-одному или же несколько дочерних классов к одному родительскому классу, и каждый из них ассоциативно связан с третьим классом, то данные связи должны указывать на наличие одинаковых отношений.

На рисунке 3.9 представлен пример данного шаблона - диаграмма классов, на которой представлен класс «Ученый» и классы «Кандидат наук» и «Профессор», которые наследуются от данного класса. А также класс «Читатель», который ассоциативно связан с классом «Ученый» и классом «Профессор».



Рисунок 3.9- Пример шаблона «Взаимодействие циклов множественной иерархии»

Таким образом, класс «Профессор» связан с классом «Ученый» отношением один-к-одному, но при этом он может быть связан с количеством от 2 до бесконечности объектами класса «Читатель». В тоже время класс «Ученый» может быть связан с количеством от 1 до 3 объектов класса «Читатель». Так как отношения 1 и 1 со стороны классов «Ученый» и «Профессор» удовлетворяют условию 1=1, но отношения 1...3 и 2..\* со стороны класса «Читатель» не удовлетворяют условию (1..3)=(2..\*), то связи противоречат друг другу и шаблону «Простой множественный цикл».

Не менее популярный и часто встречаемый шаблон носит название Простой множественный цикл (Pure Multiplicity Cycle).

Согласно данному шаблону, если между двумя классами имеется несколько ассоциативных связей и каждая связь имеет свое отношение, то все отношения со стороны одного и того же класса должны либо быть равными, либо же включать друг друга.

На рисунке 3.10 представлен пример данного шаблона - диаграмма классов, на которой представлены три класса: класс «Деканат», класс «Преподаватель» и класс «Студент». Класс «Деканат» связан двумя ассоциациями с классом «Преподаватель», причем в одной ассоциации от 0 до бесконечности объектов класса «Деканат» связаны с любым числом от 0 до бесконечности объектов класса «Преподаватель», а во второй – от 1 до бесконечности объектов класса «Деканат» связаны с 0 или 1 объектом класса «Преподаватель». Так как отношения 0...\* и 1..\* со стороны класса «Деканат» удовлетворяют условию (1..\*) Є (0..\*), и отношения 0...\* и 0..1 со стороны класса «Преподаватель» удовлетворяют условию (0..1) Є (0..\*), то связи не противоречат друг другу и шаблону «Простой множественный цикл».



Рисунок 3.10- Пример шаблона «Простой множественный цикл»

В то же время класс «Преподаватель» связан двумя ассоциациями с классом «Студент», причем в одной ассоциации 1 объект класса «Преподаватель» связан с 1 объектом класса «Студент», а во второй – 1 объект класса «Преподаватель» связан с 3 объектами класса «Студент». Так как отношения 1 и 1 со стороны класса «Преподаватель» удовлетворяют условию 1=1, но отношения 1 и 3 со стороны класса «Студент» не удовлетворяют ни условию 1=3, ни условию 1 Є 3, то связи противоречат друг другу и шаблону «Простой множественный цикл».

3.4.1.2 Реализация метода шаблонов

Поскольку библиотека шаблонов, используемых в данном методе, может расширяться, оптимальным решением для его реализации является использование паттерна проектирования «Observer» («Наблюдатель»), который представлен на рисунке 3.11. Использование данного шаблона позволяет регистрировать в системе любое количество шаблонов, без изменения архитектуры проекта.



Рисунок 3.11 – Паттерн «Observer» адаптированный для регистрации шаблонов

При такой реализации класс PatternManager позволяет регистрировать любое количество верификационных шаблонов. Когда происходит событие верификации, PatternManager инициирует оповещение всех зарегистрированных паттернов о том, что необходимо провести проверку на наличие в диаграмме анти паттерна. Все классы, реализующие интерфейс Pattern могут быть добавлены в качестве слушателя событий. Чтобы добавить новый анти паттерн достаточно реализовать интерфейс Pattern и, при помощи PatternManager’a, добавить его в список слушателей.

Обобщенный код метода поиска паттерна «Простой множественный цикл» представлен на рисунке 3.12.

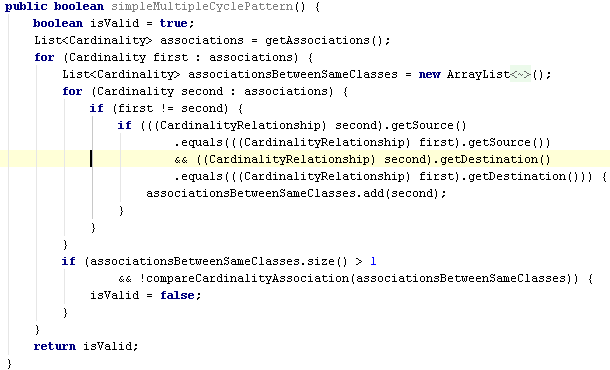


Рисунок 3.12 – Обобщенный код метода поиска паттерна  
 «Простой множественный цикл»

Принцип работы заключается в проходе по всем ассоциациям диаграммы классов и выявлении пар ассоциаций между двумя классами. Затем происходит проверка каждой пары ассоциаций на не противоречие. Если найдено, хотя бы одно противоречие в паре ассоциаций диаграмма является некорректной.

3.4.2.1 Метод идентификационного графа

Еще один подход к верификации диаграммы классов основан на построении идентификационного графа. Разработкой данного метода в период с 1990 по 2001 года занимались американские ученые Hartmann, Lenzerini, Nobili и Thalheim [15].

Суть данного метода состоит в построении идентификационного графа, предоставляющего собой ориентированный граф. Узлами данного графа являются классы, а также ассоциативные связи между ними, а дуги связывают ассоциации с теми классами, между которыми на диаграмме классов и указаны соответствующие ассоциативные связи. В качестве весов дуг используются отношения, которыми ассоциативно связаны классы. Как и в обычном графе, в идентификационном графе вес пути вычисляется как произведение весов всех дуг, входящих в состав данного пути.

Основным предназначением идентификационного графа является определение причины нарушения конечной выполнимости диаграммы классов. Также как и метод шаблонов, данный метод позволяет выявить противоречивые связи – так называемые критические циклы.

Суть данного метода представлена на рисунках 3.13 и 3.14.



Рисунок 3.13 - Простейшая диаграмма классов



Рисунок 3.14 - Идентификационный граф для диаграммы классов с рисунка 3.13

На рисунке 3.13 представлена простейшая диаграмма классов с двумя классами: Class1 и Class2, между которыми имеется ассоциативная связь С в отношении: от min1 до max1 объектов класса Class1 связаны с количеством от min2 до max2 объектов класса Class2. Идентификационный граф для этой простейшей диаграммы классов, построенный согласно данному методу, представлен на рисунке 3.14. Граф имеет три узла Class1, Class2 и С, а также четыре дуги, которые их связывают. Вес дуг слева на право и сверху вниз составляет соответственно max2, 1/min1, max1, 1/min2. Согласно данному методу, критическим, указывающим на нарушение конечной выполнимости, является цикл, вес которого меньше 1. Таким образом, если

 (3.1)

то «ассоциация» содержит нарушение конечной выполнимости.

На рисунке 3.15 представлен пример использования метода идентификационного графа для определения причины нарушения конечной выполнимости диаграммы классов. На данной диаграмме представлено два класса: класс «Преподаватель» и класс «Студент», причем от 1 до 5 объектов класса «Преподаватель» ассоциацией «Информация» об успешности связаны с количеством от 1 до 30 объектов класса «Студент». Идентификационный граф для данной диаграммы классов представлен на рисунке 3.16.



Рисунок 3.15 - Пример использования метода идентификационного графа



Рисунок 3.16 - Идентификационный граф для диаграммы классов с рисунка 3.9

Тогда согласно данному методу критический цикл для этого идентификационного графа выглядит следующим образом:

 (3.2)

поэтому ассоциация не содержит нарушения конечной выполнимости.

3.4.2.2 Реализация метода идентификационного графа

Реализация метода идентификационного графа является самой простой из всех методов. Это обусловлено самим методом, который сосредоточен только на выявление некорректностей в ассоциациях. Блок-схема алгоритма представлена на рисунке 3.17.

В самом начале устанавливается флаг «isValid» в значение «true», поскольку до отработки метода ошибок не выявлено и диаграмма на этом этапе корректна. Если алгоритм находит хоть одну ошибку - «isValid» устанавливается в значение «false», что свидетельствует о некорректности диаграммы, но для выявления всех некорректностей алгоритм не прерывается и отрабатывает до конца.

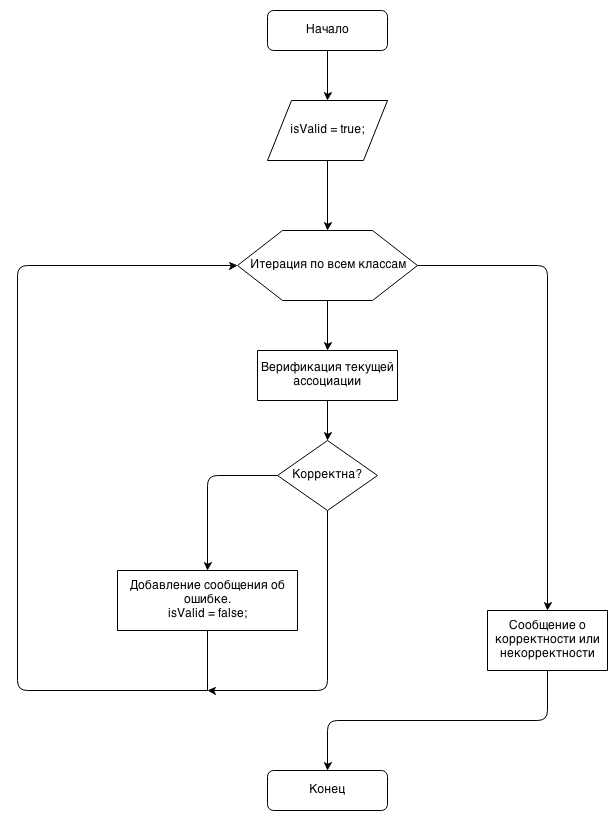


Рисунок 3.17 – Блок-схема реализации метода идентификационного графа

3.4.3.1 Метод множеств

Еще один подход к верификации диаграммы классов основан на представлении класса в качестве множества. Данный метод был предложен Calvanese и Lenzerini [16] и позволяет оценить корректность, прежде всего иерархий классов.

Согласно данному методу каждый класс, входящий в состав иерархии, представляется в виде множества. Далее создается система неравенств, в которую включаются все возможные неравенства и равенства, если такие имеются, между классами-множествами. Затем необходимо решить систему неравенств и если результатом станет не пустое подмножество, то данная иерархия классов построена корректно, если же пустое множество – иерархия классов не корректна.

На рисунке 3.18 представлена диаграмма классов, которая является иерархией классов. В этой иерархии классы «Роза», «Орхидея» и «Кактус» наследуются от класса «Растение». В тоже время классы «Роза» и «Орхидея» наследуются от класса «Цветок», а класс «Цветок» ассоциативно связан с классом «Орхидея» отношением один-к-одному.



Рисунок 3.18 - Диаграмма классов, предоставляющей собой иерархию классов

Таким образом, к данной диаграмме может быть применен метод множеств. Система неравенств для данной иерархии выглядит следующим образом:

 (3.3)

Последние два неравенства в данной системе противоречат друг другу, поэтому решением данной системы неравенств является пустое множество и согласно данному методу иерархия классов не корректна.

3.4.3.2 Реализация метод множеств

Реализация метод множеств является не детерминированным в общем случае и не позволяет составить полную систему неравенств, которые могут определить некорректности. Тем не менее, легко определяются неравенства, которыми можно описать связи наследования и имплементации классов. Блок-схема алгоритма представлена на рисунке 3.19.

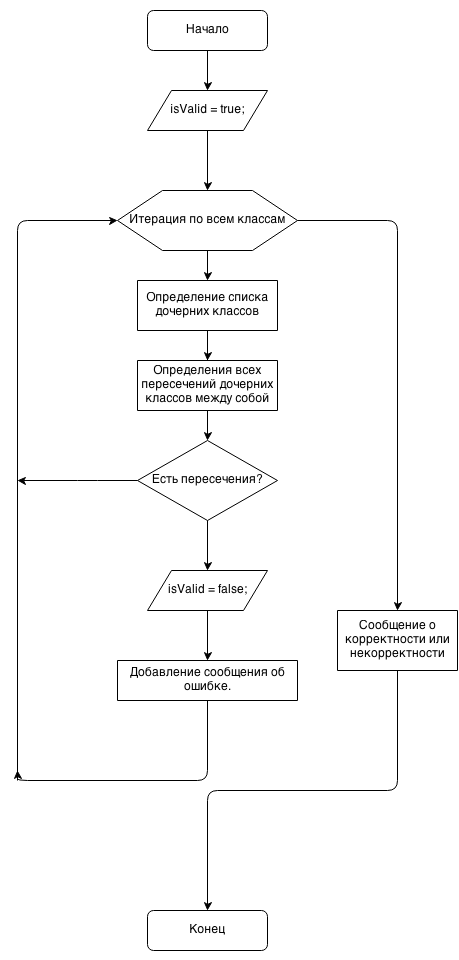


Рисунок 3.19 - Блок-схема метода множеств

3.4.4.1 Метод тестового драйвера

Данный подход позволяет формально оценить правильность создания диаграммы классов и выявить наиболее характерные ошибки [17]. Прежде, чем приступать к созданию драйвера, необходимо определить, следует ли выполнять верификацию каждого конкретного класса в автономном режиме, представляя его как отдельный модуль, или же необходимо воспринимать класс как более крупный компонент системы. Решение принимается на основании следующих факторов:

- роль данного класса в системе, в частности, степень связанного с ним риска;

- сложность класса, измеряемая количеством состояний, операций и связей с другими классами;

- объем трудозатрат, связанных с разработкой драйвера для верификации класса.

Если какой-либо класс должен стать частью некоторой библиотеки классов, то наиболее оптимальной является всесторонняя верификация классов, причем, даже в том случае, если затраты на разработку драйвера окажутся высокими, поскольку очень важным является его корректное функционирование.

Верификация классов из диаграммы классов чаще всего выполняется путем разработки драйвера, который создает экземпляры каждого из классов и окружает эти классы соответствующей средой. Таким образом, становиться возможным выполнение драйвера. Драйвер посылает одно или большее количество сообщений экземпляру класса в соответствии со спецификацией тестового случая, затем проверяет исход этих сообщений на основании значений ответа, изменения экземпляра и (или) один или большее число параметров сообщения. В обязанности драйвера чаще всего входит удаление любого созданного им экземпляра в том случае, если в языке программирования, таком как С++, имеет место управляемое программистом распределение памяти.

Если для конкретного класса характерны статические элементы данных и (или) операции, то для них также необходимо выполнять верификацию. Такие элементы данных и методы принадлежат самому классу, но не каждому экземпляру этого класса. Класс можно рассматривать как объект – например, в Java это экземпляр класса Class.

Если поведение экземпляров класса базируется на значениях атрибутов уровня класса, то все случаи, предназначенные для верификации этих атрибутов уровня класса, должны рассматриваться как расширение состояний этих экземпляров.

Если между двумя и более классами на диаграмме классов присутствует связь «наследование» (обобщение), то драйверу необходимо проверить отсутствие множественного наследования для языка Java, а также отсутствие двунаправленного наследования, при котором каждый из двух классов является и родительским, и дочерним одновременно – для всех объектно-ориентированных языков программирования.

Если между классом и интерфейсом на диаграмме классов присутствует связь «реализация», то драйверу необходимо проверить реализует ли класс все те методы, сигнатуры которых указаны в интерфейсе.

Если в классе присутствует хотя бы один абстрактный метод, то драйверу необходимо проверить, чтоб и класс был абстрактным.

Тестовый драйвер может выявлять огромное количество ошибок проектирования. Так на рисунке 3.20 и 3.21 изображены две диаграммы классов, первая из которых является корректной, а вторя – нет.



Рисунок 3.20 – Корректная имплементация интерфейса

В данном случае класс RelationshipImpl реализует все методы интерфейса Relationship.



Рисунок 3.21 – Некорректная имплементация интерфейса

Некорректность второй диаграммы заключается в не соблюдении правил имплементации интерфейсов. Каждый класс, реализующий интерфейс обязан реализовать все методы интерфейса, а класс RelationshipImpl не реализует метод setSource(), описанный в интерфейсе Relationship.

Тестовый драйвер позволяет выявлять некорректности в описании интерфейсов. Пример интерфейса, который является некорректным и будет выявлен при помощи тестового драйвера, представлен на рисунке 3.22.



Рисунок 3.22 – Некорректный интерфейс на диаграмме классов

Все интерфейсы могут иметь только публичные поля и методы, при этом все поля должны быть предопределены некоторым значением, которое в дальнейшем используется как константа. В нашем же случае поля CONST1 и CONST2 не имеют значения, об этом и сообщит тестовый драйвер.

Еще одной некорректностью, которая выявляется тестовым драйвером, является описание абстрактных классов. Пример некорректного описания абстрактного класса представлен на рисунке 3.23.



Рисунок 3.23 – Некорректное описание абстрактного класса

Если класс имеет хотя бы один абстрактный метод, то класс должен быть объявлен как абстрактный. В этом случае с диаграммы классов видно, что методы isAbstract() и isInterface() объявлены как абстрактные, а класс не является абстрактным – такая диаграмма является некорректной.

Метод тестового драйвера так же используется и для выявления некорректностей наследования. На рисунке 3.24 представлена, некорректна диаграмма классов с множественным и циклическим наследованием.



Рисунок 3.24 – Диаграмма классов с циклическим и множественным наследованием

Циклическое наследование присутствует между классами ClassB и ClassD, а так же ClassD наследуется одновременно от трех классов, это свидетельствует о некорректности диаграммы.

3.4.4.2 Реализация метод тестового драйвера

Как и для метода шаблонов рациональным является использование паттерна проектирования «Observer», который представлен на рисунке 3.25. Поскольку тестовый драйвер может расширяться, в виду добавления частных методов верификации – это решение является оптимальным.



Рисунок 3.11 – Паттерн «Observer» адаптированный для тестового драйвера

DriverManager позволяет регистрировать любое количество драйверов. Когда происходит событие верификации, DriverManager инициирует оповещение всех зарегистрированных драйверов о том, что необходимо провести проверку диаграммы. Все классы, реализующие интерфейс Method могут быть добавлены в качестве слушателя событий.

4 ЭКСПЕРЕМЕНТАЛЬНАЯ ЧАСТЬ

В данном разделе представлены результаты собственных исследований. Приводятся результаты реализации верификатора «UML Tester», результаты тестирования верификатора, а так же проводится сравнение полученных результатов верификации с результатами, полученными при тестировании современных CASE-средств.

4.1 Результаты реализации верификатора «UML Tester»

Верификатор «UML Tester» имеет простой графический интерфейс (рисунок 4.1), который позволяет выбирать конкретный метод верификации, или же выбирать комплексное тестирование. Так же интерфейс имеет окно для вывода сообщений, сформированных во время верификации, и элементы управления, позволяющие выбирать диаграмму для верификации и начало тестирования.

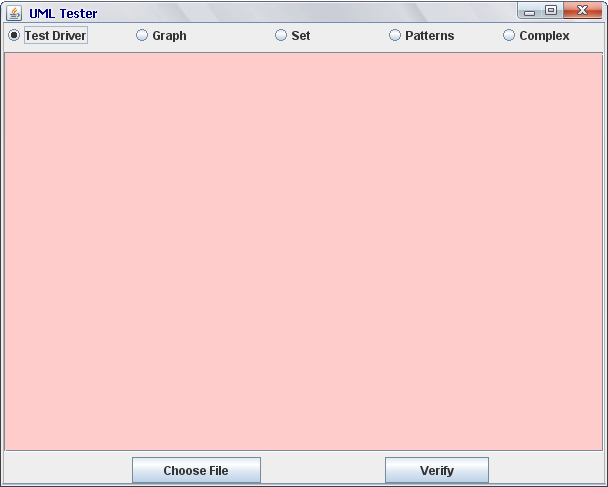


Рисунок 4.1 – Интерфейс верификатора «UML Tester»

Вверху находятся элементы управления, позволяющие определить метод верификации:

- «Test Driver» - метод тестового драйвера;

- «Graph» - метод идентификационного графа;

- «Set» - метод множеств;

- «Patterns» - метод шаблонов;

- «Complex»- комплексное тестирование всеми вышеперечисленными методами.

По средине находится окно для вывода сообщений. Внизу находятся кнопка выбора диаграммы («Choose File») и начала верификации («Verify»).

4.2 Результаты тестирования верификатора «UML Tester»

4.2.1 Тестирование метода шаблонов

В качестве тестируемого шаблона выбран «Простой множественный цикл». Для того чтобы проверить работоспособность верификатора, в диаграмму классов добавляем некорректное отношение между двумя классами, которое должно быть выявлено методом шаблонов. На рисунке 4.2 представлена некорректная диаграмма. Ассоциации «AssociationC» и «AssociationD» противоречивы, так как не имеют области пересечения.

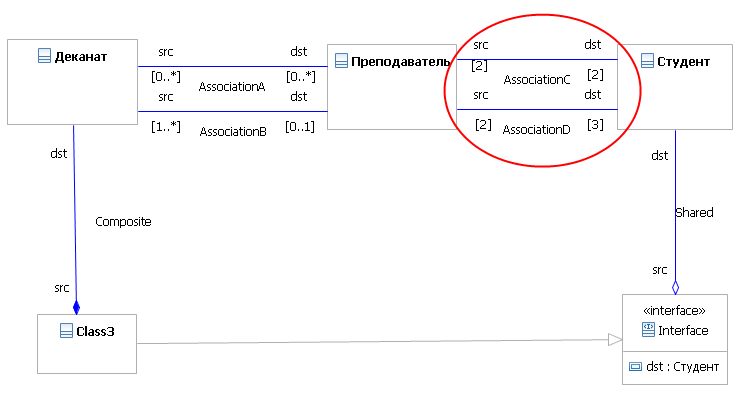


Рисунок 4.2 – Некорректная диаграмма классов, выявляемая методом шаблонов

Ассоциации «AssociationC» и «AssociationD» противоречивы, так как не имеют области пересечения. На рисунке 4.3 представлен результат работы верификатора.

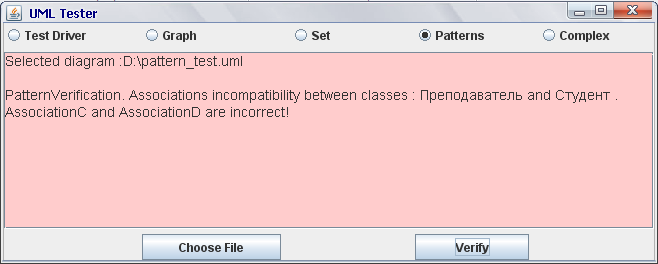


Рисунок 4.3 – Результат верификации некорректной диаграммы методом шаблонов

Как видно из рисунка, была обнаружена некорректность ассоциаций между классами «Преподаватель» и «Студент», а так же указано, какие именно ассоциации противоречивы.

4.2.2 Тестирование метода идентификационного графа

Для того чтобы протестировать данный метод целесообразно создать диаграмму классов, содержащую большое количество связей, в частности ассоциаций (рисунок 4.4). При этом одна из них должна быть некорректной.

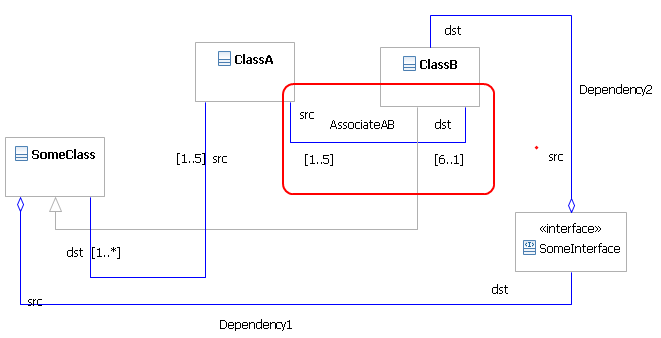


Рисунок 4.4 – Некорректная диаграмма классов, выявляемая методом идентификационного графа

Некорректной ассоциацией на этой диаграмме является «AssociateAB», которая соединяет классы ClassA и ClassB. Результат верификицииэтой модели представлен на рисунке 4.5.

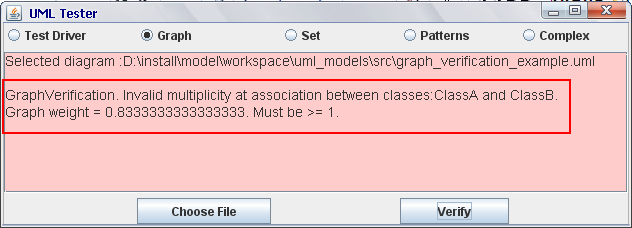


Рисунок 4.5 – Результат верификации некорректной диаграмм методом идентификационного графа

«UML Tester» сообщает, ассоциация «AssociateAB», которая соединяет классы ClassA и ClassB действительно является некорректно, так как вес идентификационного графа меньше единицы.

4.2.3 Тестирование метода тестового драйвера

Тестовый драйвер позволяет выявлять достаточно большое количество ошибок проектирования, для его проверки была создана диаграмма классов с тремя различными типами ошибок (рисунок 4.6).

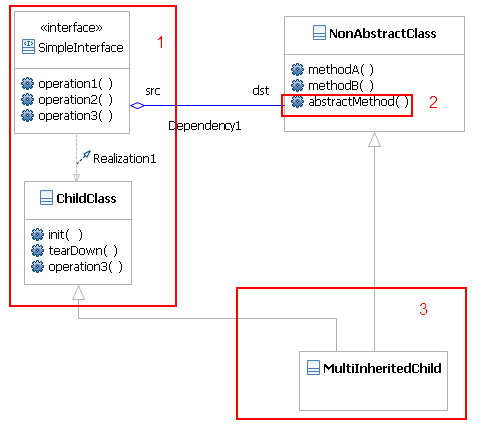


Рисунок 4.6 – Диаграмма классов с несколькими некорректностями, выявляемыми методом тестового драйвера

Первая ошибка заключается в некорректной реализации интерфейса SimpleInterface, методы operation1 и operation2 которого не были реализованы. Вторая ошибка в том, что неабстрактный класс NonAbstractClass содержит абстрактный метод и третья - класс MultiInheritedChild одновременно реализует 2 суперкласса. «UML Tester» выявил все три ошибки, показав соответствующие предупреждения (рисунок 4.7).

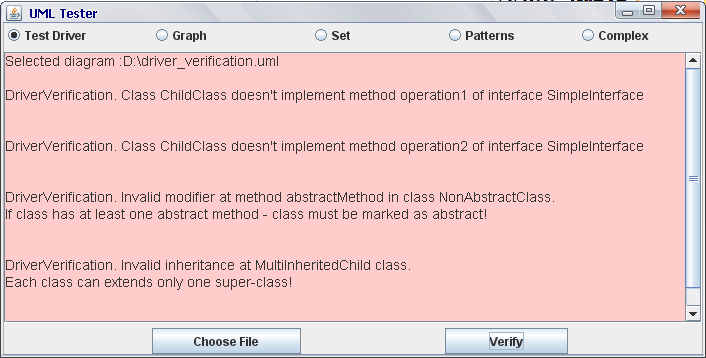


Рисунок 4.7 – Результат верификации методом тестового драйвера

4.2.4 Тестирование работы методов в комплексе

Для комплексного тестирования модели взята диаграмма, которая содержит некорректности, выявляемые различными методами верификации (рисунок 4.8).

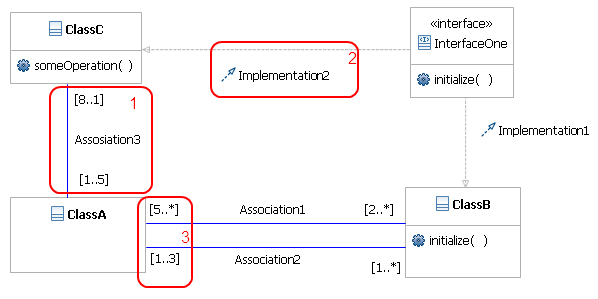


Рисунок 4.8 – Диаграмма классов с тремя различными некорректностями

Первая ошибка на диаграмме выявлена при помощи метода идентификационного графа (ассоциация “Assosiation3”), вторая при помощи тестового драйвера (не реализованный метод интерфейса), третья методом шаблонов (рисунок 4.9).

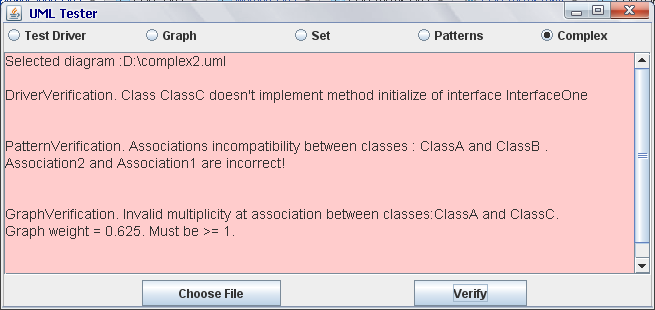


Рисунок 4.9 – Результат комплексной верификации модели

Как видно с рисунка, «UML Tester» выявил все допущенные некорректности на диаграмме классов, сообщил о них и указал причину некорректности.

4.3 Выводы

Тестирование верификатора «UML Tester» показало его эффективность в поиске ошибок проектирования. Каждый из методов верифицирует отдельный аспект модели и позволяет выявлять некорректности различного рода. Комплексное использование методов позволяет достичь максимально высокой производительности верификации. Все CASE-средства, рассмотренные в первом разделе позволяют оценить модель на корректность лишь частично, чего является не достаточно при проектировании больших систем. «UML Tester» наглядно демонстрирует высокий процент выявления некорректностей модели.

5 ОХРАНА ТРУДА И БЕЗОПАСНОСТЬ   
В ЧРЕЗВЫЧАЙНЫХ СИТУАЦИЯХ

Охрана труда – это система правовых, социально-экономических, организационно-технических, санитарно-гигиенических и лечебно-профилактических мероприятий и средств, направленных на сохранение здоровья и работоспособности человека в процессе труда.

Статья 43 Конституции Украины закрепляет право граждан на труд и одновременно устанавливает, что данное право должно осуществляться в условиях, которые отвечают требованиям безопасности и гигиены.

Государственная политика в области охраны труда базируется на принципах статьи 4 Закона «Об охране труда», который был принят 14 октября 1992 года:

- приоритета жизни и здоровья работников по отношению к результатам производственной деятельности предприятия, учреждения, организации;

- полной ответственности собственника за создание безопасных и безвредных условий труда;

- комплексного решения задач охраны труда на основе национальных программ по этим вопросам;

- социальной защиты работников, полного возмещения ущерба лицам, пострадавшим от несчастных случаев на производстве и профессиональных заболеваний;

- установления единых нормативов по охране труда для всех предприятий, независимо от форм собственности и видов их деятельности;

- использования экономических методов управления охраной труда;

- осуществления обучения населения, профессиональной подготовки и повышения квалификации работников по вопросам охраны труда.

В ходе разработки данного раздела определены цели по соблюдению норм и правил охраны труда в заведении, проанализированы факторы, влияющие на человека. Предложены меры, необходимые для нормальной и стабильной работы персонала, как с экономичной, так и с социальной стороны.

5.1 Характеристика системы управления охраной труда в учреждении

В данном разделе проведен анализ рабочего места в черниговском отделении компании SysIQ inc. Работников в офисе насчитывается 33. Отделение находится в отдельно построенном четырехэтажном здании. Недалеко от офиса находится рекреационная зона – лесной заповедник Яловщина. Большие автомобильные магистрали отсутствуют .

Так как на предприятии работает 33 человека, то работодателем, согласно статьи 15 Закона Украины “Об охране труда”, был назначен по совместительству сотрудник, имеющий соответствующую подготовку.

Был проведен анализ системы управления охраной труда данного отделения компании. Анализ включает характеристику законодательного и нормативно-правового регулирования прав, обязанностей и ответственности работодателей за соблюдение законодательства об охране труда. Результаты анализа документации предприятия приведены в таблице 5.1.

Таблица 5.1 – Нормативно-законодательная база по охране труда на предприятии

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | | Название документа | Отметка об исполнении (+/-) |
| **Документы внешнего происхождения** | | | |
| 1 | | Закон Украины «Об охране труда» | + |
| 2 | | Типовое положение о службе охраны труда | + |
| 3 | | Положение о порядке расследования несчастных случаев. | - |
| 4 | | Порядок расследования и ведения учета несчастных случаев, профессиональных заболеваний и аварий на производстве. | - |
| 5 | | Типичное положение об учениях по вопросам охраны труда. | - |
| 6 | | Положение о разработке инструкций по охране труда. | - |
| 7 | | Перечень работ с повышенной опасностью. | - |
| 8 | | Перечень работ, где необходим профессиональный отбор. | - |
| 9 | | Предельные нормы поднятия и перемещения тяжестей женщинами. | - |
| 10 | | Предельные нормы поднятия и перемещения тяжестей несовершеннолетними. | - |
| 11 | | Положение о медицинском осмотре работников отдельных категорий. | + |
| 12 | | Перечень должностей должностных лиц, которые обязаны проходить предварительную и периодическую проверку знаний по охране труда. | + |
| 13 | | Порядок разработки и утверждения собственником нормативных актов об охране труда, действующих на предприятии. | + |
| 14 | | Положение о порядке обеспечения работников специальной одеждой, специальной обувью и другими средствами индивидуальной защиты. | - |
| № з/п | | Название документа | Отметка об исполнении (+/-) |
| 15 | | Порядок проведения аттестации рабочих мест по условиям. | + |
| 16 | | Типовое положение о комиссии по вопросам охраны труда. | - |
| 17 | | Типовое положение «О кабинете охраны труда» | - |
| **Нормативно-правовые акты и проекты приказов по охране труда, которые должны быть на предприятии** | | | |
| 1 | | Программа (текст) вводного инструктажа по охране труда и тексты инструктажа на рабочих местах. | + |
| 2 | | Перечень профессий работников, освобожденных от проведения первичного, повторного и внепланового инструктажей (по согласованию с государственным инспектором по надзору за охраной труда). | - |
| 3 | | Перечень работ с повышенной опасностью. | - |
| 4 | | Положение о порядке выполнения работ с повышенной опасностью | - |
| 5 | Перечень профессий, исполнители которых имеют право на обеспечение средствами индивидуальной защиты с определением конкретных видов средств для них. | | - |
| 6 | Положение об обучении по вопросам охраны труда. | | - |
| 7 | Перечень работ, для выполнения которых необходим профессиональный отбор. | | - |
| 8 | Перечень профессий работников, которые должны проходить предварительный и периодические медицинские осмотры. | | - |
| 9 | Перечень должностных лиц, которые обязаны проходить предварительную и периодическую проверку знаний по вопросам охраны труда. | | + |
| 10 | Перечень работ, на которых запрещается применение труда женщин. | | - |
| 11 | Перечень работ, на которых запрещается применение труда несовершеннолетних | | - |
| 12 | Перечень инструкций по охране труда, которые должны действовать на данном предприятии. | | + |
| 13 | Перечень существующих профессий и должностей, работники которых имеют право на получение молока или других равноценных пищевых продуктов | | - |
| 15 | Состав постоянно действующей комиссии по проверке знаний по вопросам охраны труда. | | - |
| № з/п | Название документа | | Отметка об исполнении (+/-) |
| **Приказы** | | | |
| 1 | | Об утверждении Положения о системе управления охраной труда СУОТ. | + |
| 2 | | Об утверждении Положения о проведении трехступенчатого административно-общественного контроля в системе управления охраной труда заведения (учреждения). | - |
| 3 | | Об утверждении графика проведения периодических медосмотров работников. | - |
| 4 | | Об утверждении комиссии по проверке знаний по вопросам охраны труда должностных лиц | - |
| 5 | | Об утверждении членов добровольной противопожарной дружины и льготы для них. | - |
| 6 | | План локализации и ликвидации аварийных ситуаций и аварий. | + |
| **Журналы, которые должна иметь служба охраны труда** | | | |
| 1 | | Регистрации несчастных случаев, которые произошли с воспитанниками, учащимися,  студентами, курсантами, слушателями, аспирантами (форма Н-Н); | - |
| 2 | | Регистрации несчастных случаев, которые произошли с работающими на производстве (Форма Н-1, Н-5); | - |
| 3 | | Учета сообщений о несчастном случае; | - |
| 4 | | Учета профессиональных заболеваний (отравлений); | - |
| 5 | | Учета объектов повышенной опасности; | + |
| 6 | | Учета выдачи инструкций по охране труда в учебном заведении; | - |
| 7 | | Регистрации вводного инструктажа по вопросам охраны труда; | + |
| 8 | | Регистрации протоколов лабораторных исследований условий труда (при необходимости). | - |
| **Документы, которые должны храниться в службе охраны труда** | | | |
| 1 | | Инструкции по охране труда (первые экземпляры). | + |
| 2 | | Акты регистрации несчастных случаев, которые произошли с воспитанниками, учащимися, студентами, курсантами, слушателями, аспирантами по форме Н-Н. | - |
| 3 | | Акты расследования несчастных случаев, профессиональных заболеваний и аварий с работающими на производстве по форме Н-1, Н-5. | - |
| № з/п | | Название документа | Отметка об исполнении (+/-) |
| 4 | | Акты расследования профессиональных заболеваний по форме П-4. | - |
| 5 | | Карта учета профессионального заболевания (отравления). | - |
| 6 | | Материалы расследования несчастных случаев (сообщения), профессиональных заболеваний (первые экземпляры). | - |
| 7 | | Предписания органов государственного надзора, приказы и мероприятия по их выполнению. | - |
| 8 | | Предписание специалистов службы охраны труда. | - |
| 9 | | План работы службы охраны труда. | + |
| 10 | | Протоколы проверки знаний по вопросам охраны труда должностных лиц предприятия. | + |
| 11 | | Протоколы лабораторных исследований аттестации рабочих мест по условиям труда. | + |
| 12 | | Материалы проверки состояния условий и безопасности труда комиссией учебного заведения и утверждены мероприятия по устранению выявленных нарушений. | - |
| 13 | | Комплексные мероприятия для достижения установленных нормативов и повышения существующего уровня охраны труда, предусмотренных коллективным договором. | - |
| 14 | | Материалы аудита охраны труда, в том числе оценки технического состояния производственного оборудования, аттестации рабочих мест по условиям труда на соответствие нормативно-правовым актам по охране труда. | + |

5.2 Анализ условий труда на рабочем месте

Здание имеет как природную, так и искусственную систему вентиляции. Большие оконные проемы позволяют получать днём достаточно освещения, а светодиодные лампы позволяют получить необходимый свет в вечернее время.

Освещенность рабочего места была измерена люксметром Ю-117 ПК. Уровень природной освещенности поверхности, где расположен ПК составляет 200 лк при освещенности той же поверхности открытым горизонтом в 20000 лк, т.е. КПО = 1%, что не соответствует нормативному КПО.

Для искусственного освещения в помещении используются люминесцентные лампы.

Расчет искусственного освещения проведем для комнаты площадью 50 м, ширина которой составляет 5м, длина - 10м, высота - 3м. Воспользуемся методом использования светового потока. Для определения необходимого числа светильников, которые должны обеспечить нормированный уровень освещенности, определим световой поток, падающий на рабочую поверхность по формуле:

 (5.1)

где F – рассчитываемый световой поток, лм;

E - нормированная минимальная освещенность, лк; Е = 300 лк;

S - площадь освещаемого помещения (в нашем случае S = 50м);

Z - отношение средней освещенности к минимальной (обычно принимается равным Z = 1,1);

K - коэффициент запаса, учитывающий уменьшение светового потока лампы в результате загрязнения светильников в процессе эксплуатации (его значение зависит от типа помещения и характера проводимых работ в нем, в нашем случае К = 1,5);

η - коэффициент использования светового потока, коэффициент отражения потолка – 80%, стен –30%,пола–10%.

Вычислим индекс помещения по формуле:

 (5.2)

где S - площадь помещения, S = 50м;

h - расчетная высота подвеса, h = 3 м;

A - ширина помещения, А = 5 м;

B - длина помещения, В = 10 м.

Подставив значения в формулу 4.2, получим:



Зная индекс помещения I и учитывая коэффициенты отражения потолка, стен и поля, по таблице 4 [ДБН В.2.5-28-2006] находим η = 0,49.  
Подставим все значения в формулу 5.1 для определения светового потока F:

 лм

Для освещения используются люминесцентные лампы типа ЛБ 40-1, световой поток которых F = 4320 лм. Рассчитаем необходимое количество ламп в светильниках по формуле 5.3:

 (5.3)

где N – необходимое количество ламп в светильниках ламп;

F – световой поток, F = 54551 лм;

Fл – световой поток лампы, Fл = 4320 лм.

****

В помещении используются светильники типа ОД. Каждый светильник комплектуется двумя лампами. То есть необходимо использовать 6 светильников с 2 работающими лампами в них. На момент аттестации рабочего места оператора работали все лампы, поэтому уровень искусственного освещения удовлетворял санитарным нормам.

Рабочее пространство каждого сотрудник удовлетворяет требования нормы в 6 м2.

В комнате находится 8 сотрудников, у каждого из которых в наличии имеется ноутбук Dell Vostro (40дБ) и монитор ViewSony(15 дб).

Рассчитаем суммарный шум за формулой:

 (5.4)

где, Li - уровень звукового давления i-го источника шума;

n - количество источников шума.



В таблице 5.2 представлено состояние условий труда на рабочем месте.

Таблица 5.2 - Фактическое состояние условий труда на рабочем месте

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Значение фактора | | | | Условия и характер труда | | |
| Параметры | Допустимые нормы | | Фактические нормы | |
| 1 класс | 2 класс | 3 класс |
| Санитарно-гигиенические показатели | | | | | | | |
| Температура воздуха | 23-25 °С | | | 24 °С | - | - | - |
| Относительная влажность | 40-60 % | | | 60 % | - | - | - |
|  | | | Значение фактора | | | Условия и характер труда | | |
| Параметры | | | Допустимые нормы | | Фактические нормы | 1 класс | 2 класс | 3 класс |
| Скорость движения воздуха | | | 0,1-0,2 м/с | | 0,1 м/с | - | - | - |
| Природное освещение | | | 1,275% | | 1% | - | - | - |
| Искусственное освещение | | | 300 лк | | 336 лк | - | - | - |
| Шум | | | 65 дБ | | 49 дБ | - | - | - |
| Показатели напряженности труда | | | | | | | | |
| Продолжительность сосредоточения (%) | | | < 75 | | 76 | + | - | - |
| Плотность сигналов, в среднем за час | | | < 300 | | 400 | + | - | - |
| Напряженность анализаторных функций:  - зрение  - слух | | | < Точная  90-100% | | Грубая и мало точная  90-100% | -  - | -  - | -  - |
| Эмоциональное и интеллектуальное напряжение | | | Свободный график | | Свободный график | - | - | - |
| Число элементов в многократно повторяющейся операции | | | 10-4 | | 10-4 | - | - | - |
| Продолжительность выполнения повторяющихся, с | | | 0-20 | | 0-20 | - | - | - |
| Время наблюдения за ходом производственного процесса без активных действий, % | | | < 95 | | 20 | - | - | - |
| Сменность | | | Односменная | | Односменная | - | - | - |
| Показатели тяжести труда | | | | | | | | |
| Энергозатраты, ккал | | | 145-160 | 165 | | + | - | - |
| Статическое напряжение, кс\*с | | | 4400-90000 | 7000 | | - | - | - |
|  | | | Значение фактора | | | Условия и характер труда | | |
| Параметры | | | Допустимые нормы | Фактические нормы | | 1 класс | 2 класс | 3 класс |
| Физическое напряжение, кг\*м/мин | | | < 220 | 120 | | - | - | - |
| Мелкие стереотипные движения кистей и пальцев рук | | | < 40000 | 61000 | | - | + | - |
| Перемещение в пространстве | | | < 4 км | < 4 км | | - | - | - |

В результате анализа условий труда определено, что самыми неблагоприятно влияющими факторами производственной среды являются сложность и напряженность работы, непрерывные умственные и физические нагрузки. Предложены методы по устранению настолько тяжелого и напряженного режима работы и обеспечению нормального режима работы:

- метод отвлечения состоит в произвольном переключении внимания с одного объекта на другой. Овладение им необходимо личностям, у которых условия напряженной работы вызывают навязчивые мысли, отрицательные эмоции или эмоциональное чрезмерное возбуждение. Каждый работник должен воспитать у себя способность на некоторое время произвольно отвлекаться от процесса работы и связанных с ним отрицательных эмоций и сосредотачивать мысли на какой-либо посторонней теме;

- метод самоприказа состоит в сознательном побуждении себя к выполнению очень трудных и сложных действий, в создании внутреннего состояния уверенности в том, что эти действия будут выполнены. Если вам предстоит важное событие, начните работать с формулой за несколько дней до него, а не прямо в день этого события;

- метод регуляции дыхания состоит в повторении сочетаний глубокого вдоха с последующим продолжительным выдохом. Правильно поставленное брюшное дыхание вовлекает в дыхательный акт все легкие, увеличивая емкость, обычно задействованную в дыхании; улучшает насыщение крови кислородом. Двигающаяся диафрагма массирует органы брюшной полости, в первую очередь, печень, оживляя ее кровоснабжение. Спокойное упорядоченное дыхание способствует эмоциональному выравниванию;

- метод мышечного расслабления способствует снятию не только психического, но и физического напряжения;

- выполнение гимнастических упражнений, позволяет снять умственную нагрузку, расслабить кисти рук, снять напряжение в позвоночнике, возникающего в результате постоянного сидячего образа работы.

5.3 Обеспечение производственной санитарии в условиях производства

5.3.1 Предложения по улучшению условий труда

Конструкция рабочего стола должна отвечать современным  
требованиям эргономики и обеспечивать оптимальное размещение на  
рабочей поверхности используемого оборудования (дисплея,  
клавиатуры, принтера) и документов.

Рабочий стул должен быть подъемно-поворотным, регулируемым по высоте, с углом наклона сиденья и спинки, поверхность сиденья должна быть плоской, передний край - закругленным. Регулировка по каждому из параметров должно осуществляться независимо, легко и надежно фиксироваться. Шаг регулирования элементов стула должен составлять: для линейных размеров -15 ... 20 мм, для угловых 2 ... 5 °. Усилия регулирование должно превышать 20 Н.

Высота поверхности сидения должна регулироваться в пределах 400 ... 500 мм, а ширина и глубина составлять не менее 400 мм.

Угол наклона сиденья - до 15 град. вперед и до 5 град. назад.

Высота спинки стула должна составлять (300 - 320) мм, ширина - не менее 380 мм, радиус кривизны горизонтальной плоскости - 400 мм. Угол наклона спинки должен регулироваться в пределах 5 ... 15 град. от вертикального положения. Расстояние от спинки до переднего края сиденья в пределах 260 ... 400 мм.

Для снижения статического напряжения мышц верхних конечностей следует использовать стационарные или съемные подлокотники длиной не менее 250 мм, шириной 50 ... 70 мм, регулируются по высоте над сиденьем в пределах 230 ... 260 мм и расстояния между подлокотниками в пределах 350 ... 500 мм.

5.3.2 Расчет требуемого воздухообмена в помещении за опасными веществами

В данном разделе производится расчет инженерной задачи, в которой необходимо вычислить требуемый воздухообмен в помещении за опасными веществами.

Исходные данные:

- помещение объемом 100м3;

- 3 вещества однонаправленного действия (таблица 5.3);

- вредные вещества не отсасываются;

- объем вредных веществ, которые поступают в помещение, не превышает 0,3ПДК.

Таблица 3.3 – Исходные данные

|  |  |  |
| --- | --- | --- |
| Вредные вещества | ПДК, мг/м3 | Количество вредных выделений, г/час |
| Ацетон | 200 | 0,60 |
| Бутилацетат | 200 | 0,15 |
| Ксилол | 50 | 0,02 |

Если в помещение выделяются вредные вещества в виде пара, газа, пыли, то расчет воздухообмена L м3/час, выполняют по формуле 5.5:

 (5.5)

где Lр.з – объем воздуха удаляемого из рабочей зоны местными отсосами, обще-обменной вентиляцией или на технологические нужды м3/час, при плотности воздуха ρ = 1,2 кг/м3;

M - количество вредных веществ, поступаемых в помещение, мг / ч;

Ср.з., Свид, Сп - соответствующие концентрации вредных веществ в воздухе (р.з. - рабочей зоны, уд. - в удаляемом; п. - в подаваемом), мг/м3.

Поскольку Lрз = 0, т.е. с рабочей зоны не отсасывается воздух, то приведенная формула 5.5 упрощается до вида 5.6:

 (5.6)

Содержание вредных веществ в воздухе, которое поступает в производственное помещение не должно превышать 0,3 ПДК, тогда формулу 5.6 можно упростить, и получить формулу 5.7:

 (5.7)

где ПДК – предельно допустимая концентрация вещества.

При одновременном выделении в воздух рабочей зоны помещения нескольких вредных веществ не однонаправленного действия воздухообмен принимают по той вредным веществом, для которой по расчету, необходим больший воздухообмен.

В случае мы имеем дело с одновременным выделением нескольких вредных веществ однонаправленного действия. В этом случае необходимо суммировать значения L, рассчитанные для каждой отдельной вещества и кратность воздухообмена рассчитывать по суммарному значению L.

Проводим расчет объема L:



По полученным данным производится расчет кратности воздухообмена, час-1 формула 5.8:

 (5.8)

где  - воздухообмен, м3/час;

Vв - свободный внутренний объем помещения, м3.

В свою очередь свободный внутренний объем подсчитывается по формуле 5.9:

 (5.9)

где V - объем помещения, м3.

Вычисляем коэффициент K:



Кратность воздухообмена показывает, сколько раз в течение часа обменивается воздуха в помещении. В данном случае достаточно проветривать раз в 12 часов, но для обеспечения комфортных условий труда лучше обменивать воздух раз в час.

5.4 Безопасность в чрезвычайных ситуациях

1. Принадлежность помещения или рабочей зоны к соответствующей категории или классу пожарной опасности определяется на основе свойств веществ и материалов, применяемых в технологическом процессе или в оформлении помещений, количества этих веществ и материалов, особенностями производства. Для рабочих мест пользователей ПК по требованиям НПАОП 0.00-1.28-10, помещения с ПК имеют II степень огнестойкости.

Категория взрывопожарной и пожарной опасности помещения, а также класс его взрывоопасности по ПУЭ должны быть обозначены в соответствии табличкой по установленным нормам на входных дверях производственных и складских помещений. Но при исследовании помещений данных пометок на входных дверях не было обнаружено. Это является нарушением правил безопасности и следует принять меры по обозначению на входных дверях класса по взрывоопасности помещений этажа, а особенно нашего помещения, в котором будут выполняться работы.

По классу взрыво- и пожаро- опасности помещение, в котором проводятся работы (аудитория №92), относится к классу В (пожароопасные помещения). Помещение может содержать горючие и тяжело горючие вещества и материалы, вещества и материалы которые могут взаимодействовать с водой, кислородом воздуха или один с другим.

Площадь этажа с учетом четырех аудиторий площадью по 77 м2 каждая и коридора площадью 88 м2 составляет 550 м2. Исходя из этих данных на этаж необходимо не менее 3 переносных пенных и водных огнетушителя объемом 8,5 кг. Равноценная защита обеспечивается также 3 порошковыми огнетушителями объемом 4,2 кг.

Расстояние от возможного очага пожара до места расположения огнетушителя не должно превышать 30 м.

2. Потенциальными причинами возникновения пожаров при эксплуатации оборудования являются возникновение токов короткого замыкания, искрение, перегрев проводников при длительной эксплуатации ПК.

В соответствии с ГОСТ 12.1.004-91 3.1. противопожарная защита должна достигаться применением одного из следующих способов или их комбинацией:

- применением средств пожаротушения и соответствующих видов пожарной техники;

- применением автоматических установок пожарной сигнализации и пожаротушения;

- применением основных строительных конструкций и материалов, в том числе используемых для облицовок конструкций, с нормированными показателями пожарной опасности;

- устройствами, обеспечивающими ограничение распространения пожара;

- организацией с помощью технических средств, включая автоматические, своевременного оповещения и эвакуации людей;

- применением средств коллективной и индивидуальной защиты людей от опасных факторов пожара;

- применением средств против-дымной защиты.

Средства коллективной и индивидуальной защиты должны обеспечивать безопасность людей в течение всего времени действия опасных факторов пожара. Коллективную защиту следует обеспечивать с помощью пожаробезопасных зон и других конструктивных решений. Средства индивидуальной защиты следует применять также для пожарных, участвующих в тушении пожара.

Система противодымной защиты объектов должна обеспечивать отсутствие задымленности, снижение температуры и удаление продуктов горения и термического разложения на путях эвакуации в течение времени, достаточного для эвакуации людей и (или) коллективную защиту людей в соответствии с требованиями и (или) защиту материальных ценностей.

На каждом объекте народного хозяйства должно быть обеспечено своевременное оповещение людей и (или) сигнализация о пожаре в его начальной стадии техническими или организационными средствами.

Но параметры аварийного освещения и аварийной вентиляции не предусмотрены на данном этаже. Также отсутствуют средства автоматического пожаротушения, средства сообщения и сигнализации о пожаре и нет возможности автоматического отключения вентиляции в случае пожара

3. В данном помещении присутствует медицинская аптечка первой помощи в доступном месте. Система автоматической пожарной сигнализации с дымовыми пожарными вещателями и переносными углекислотными огнетушителями (из расчета 2 шт. на каждые 20 м ² площади помещения) не соответствует требованиям. Количество огнетушителей не соответствуют установленному необходимому количеству. Но, тем не менее, подходы к средствам пожаротушения и быстрого отключения электросхем оборудования свободны и не загромождены для доступа к ним.

Для помещения, в котором проводятся работы по дипломному проектированию, а также целого этажа разработан план эвакуации, представленный на рисунке 5.1.

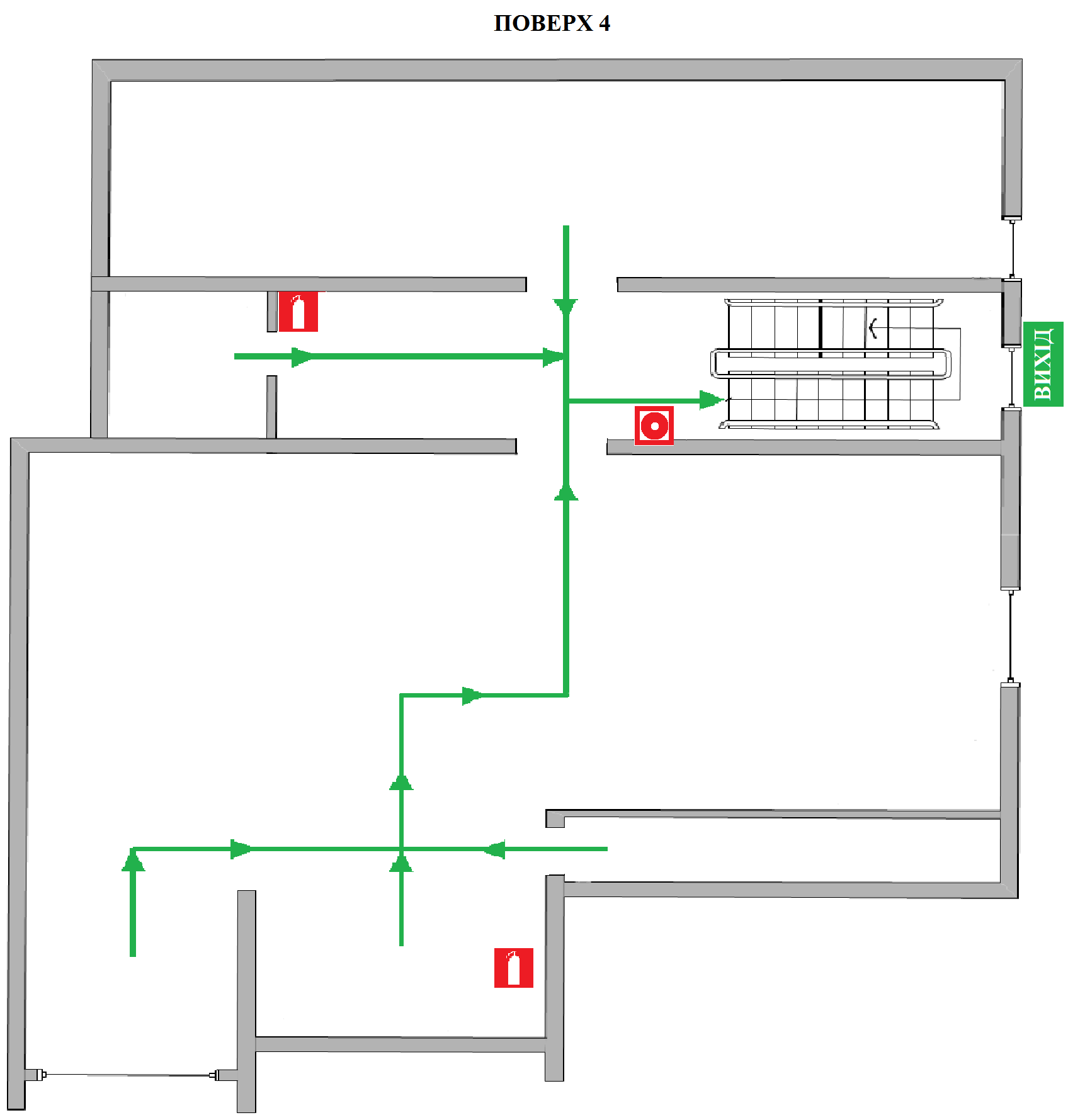


Рисунок 5.1 – План эвакуации этажа

Количество эвакуационных выходов с помещений и с этажа, ширина дверей и коридоров, открывание дверей по ходу эвакуационного пути (согласно ГОСТ12.1.004–91) позволяет вовремя эвакуироваться персоналу с помещений и этажа.

С данным планом должны быть ознакомлены все работающие на этом этаже. Для своевременной эвакуации и во избежание возникновения хаотичности при эвакуации указаны направления движения людей. Соблюдение инструктажа и выполнение всех описанных и утвержденных заранее правил поведения при аварийной ситуации помогут избежать травм и своевременно эвакуироваться людям.

ВЫВОДЫ

В работе описано наиболее часто используемых к верификации диаграмм классов. Все они позволяют оценить корректность диаграммы классов с разной точки зрения. Так, метод, основанный на построении драйвера, позволяет выявить только наиболее характерные ошибки на диаграмме. Метод шаблонов позволяет оценить корректность ассоциативных связей и связей типа «обобщение». Метод, связанный с построением идентификационного графа, позволяет оценить корректность исключительно ассоциативных связей. Метод, основанный на представлении классов в качестве множеств, позволяет оценить корректность исключительно иерархий классов. Таким образом, наиболее эффективным является комплексное использование данных методов.

Тестирование верификатора «UML Tester» показало его эффективность в поиске ошибок проектирования. Каждый из методов верифицирует отдельный аспект модели и позволяет выявлять некорректности различного рода. Комплексное использование методов позволяет достичь максимально высокой производительности верификации. Все существующие CASE-средства, позволяют оценить модель на корректность лишь частично, чего является не достаточно при проектировании больших систем.

Таким образом, разработанная система верификации является шагом вперед в области верификации программного обеспечения. Она позволяет экономить ресурсы и время на этапе перехода от программной модели к ее конечной реализации.

ПЕРЕЧЕНЬ ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ

1 Буч Г., Якобсон А., Рамбо Дж. UML. Классика CS. 2-е изд. / Пер. с англ.; Под общей редакцией проф. С. Орлова — СПб.: Питер, 2006. — 736 с.

2 Open Source Software Engineering Tools [Электронный ресурс] : сайт продукта — Режим доступа : http://argouml.tigris.org.

3 Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. Проектирование информационных систем. – М.: ИНТУИТ.ру, 2005 . - 296 с.

4 Леоненков А.В. Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose. – М.: ИНТУИТ.ру, БИНОМ. Лаборатория знаний, 2006 . - 320 с.

5 Altova UModel 2014 [Электронный ресурс] : техническая документация – Режим доступа : http://manual.altova.com/UModel/umodelenterprise/

6UML, BPMN and Enterprise Architecture Tool for Software Development [Электронный ресурс] : сайт продукта – Режим доступа : <http://www.visual-paradigm.com>

7 UML tools for software development and modelling - Enterprise Architect UML modeling tool [Электронный ресурс] : сайт продукта – Режим доступа : <http://www.sparxsystems.com.au>

8 12 European Certificate of Informatics Professionals [Электронный ресурс]: лекции – Режим доступа : [http://www.e-uni.ee/e-kursused/eucip/ arendus\_vk/122\_\_\_\_.html](http://www.e-uni.ee/e-kursused/eucip/arendus_vk/122____.html)

9 Dr. Winston, W.Royce «Managing The Development of Large Software Systems», EE WESCON, August 1970, pages 1-9.

10 Ч. Хоар. Взаимодействующие последовательные процессы / Ч. Хоар; пер. с англ. – М.: Мир, 1989. – 264 с.

11 A Spiral Model of Software Development and Enhancement [Электронный ресурс] : статья – Режим доступа : http://csse.usc.edu/csse/TECHRPTS/1988/usccse88-500/usccse88-500.pdf

12 European Certificate of Informatics Professionals [Электронный ресурс]: лекции – Режим доступа : http://www.e-uni.ee/e-kursused/eucip/arendus\_vk/1224.html

13 OMG, Architecture Board ORMSC «Model Driven Architecture (MDA)», ormsc/2001-07-01, July 9, 2001.

14 Balaban М. Management of Correctness Problems in UML Class Diagrams – Towards a Pattern-based Approach / Balaban М., Maraee А., Stur А. – Beer Sheva: Department of Computer Science, Ben-Gurion University of the Negev, 2002. – 33 р.

15 Thalheim, B. Fundamentals of Entity-Relationship Modeling / Hartmann, Lenzerini, Nobili, Thalheim. - Annals Mathematics and Artificial Intelligence, №7, 1993. – pp. 197-256.

16 Calvanese, D., Lenzerini, M. On the Interaction between ISA and Cardinality Constraints. In Proceedings of the 10th IEEE International Conference on Data Engineering, Houston, Texas, USA. IEEE Computer Society, Washington, DC, USA, 1994. – pp. 204-213.

17 Макгрегор Дж. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Дж. Макгрегор, Д. Сайкс; пер. с англ. – К.: ООО «ТИД «ДС»», 2002. – 432 с.

18 Охорона праці. Методичні вказівки до виконання розрахунково-графічної роботи для студентів напрямів підготовки 6.050102 – “Комп’ютерна інженерія”; 6.050802 – “Електроніка” / Укл.: Гуменюк О.Л., Бивойно Т.П. – Чернігів: ЧДТУ, 2011. – 66 с.

19 Охорона праці. Методичні вказівки до виконання розрахунково-графічної роботи для студентів напряму підготовки 6050701 – “Електротехніка” / Укл.: Гуменюк О.Л., Челябієва В.М., – Чернігів: ЧДТУ, 2010. – 49 с.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Программный проект  
Подсистема верификации UML диаграмм классов «UML Tester»

СОДЕРЖАНИЕ

[1 МОДУЛЬ ОБРАБОТКИ XML ДАННЫХ 3](#_Toc376535658)

[2 МОДУЛЬ ПРЕДСТАВЛЕНИЯ ДИАГРАММЫ КЛАССОВ В   
 ВИДЕ ДИНАМИЧЕСКОЙ СТРУКТУРЫ ДАННЫХ 18](#_Toc376535659)

[3 МОДУЛЬ ВЕРИФИКАЦИИ 39](#_Toc376535660)

[4 МОДУЛЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА 49](#_Toc376535661)

1 Модуль обработки XML данных

package com.diploma.classdiagram.parser;

import com.diploma.classdiagram.model.XMLElement;

import java.util.List;

public interface Parser {

/\*\*

\* This method should parse well-formed XML file

\* and return list of xml elements.

\*

\* @param sourceFile - The XML file path.

\* @return - List of XML elements.

\*/

public List<XMLElement> parse(String sourceFile) throws UMLParserException;

}

package com.diploma.classdiagram;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 26.08.12

\* Time: 11:06

\*/

public final class Constants {

// top level elements

public static final String PACKAGED\_ELEMENT = "packagedElement";

public static final String ELEMENT\_IMPORT = "elementImport";

// packaged element's xmi:type

public static final String CLASS = "uml:Class";

public static final String INTERFACE = "uml:Interface";

public static final String REALIZATION = "uml:Realization";

public static final String DEPENDENCY = "uml:Dependency";

public static final String ASSOCIATION = "uml:Association";

public static final String GENERALIZATION = "generalization";

public static final String AGGREGATION\_TYPE = "aggregation";

public static final String COMPOSITION = "composite";

public static final String AGGREGATION = "shared";

public static final String GENERAL = "general";

public static final String TYPE = "type";

public static final String NAME = "name";

public static final String COVERED\_BY = "coveredBy";

public static final String RECEIVE\_EVENT = "receiveEvent";

public static final String COVERED = "covered";

public static final String ID = "id";

public static final String SUPPLIER = "supplier";

public static final String CLIENT = "client";

public static final String OWNED\_END = "ownedEnd";

public static final String SRC = "src";

public static final String DST = "dst";

public static final String EMPTY\_STRING = "";

public static final String TRUE = "true";

public static final String FALSE = "false";

public static final String METHOD = "ownedOperation";

public static final String ATTRIBUTE = "ownedAttribute";

public static final String METHOD\_PARAMETER = "ownedParameter";

public static final String VISIBILITY = "visibility";

public static final String PRIVATE = "private";

public static final String PUBLIC = "public";

public static final String PROTECTED = "protected";

public static final String PACKAGED = "package";

public static final String IS\_LEAF = "isLeaf";

public static final String IS\_STATIC = "isStatic";

public static final String IS\_READ\_ONLY = "isReadOnly";

public static final String IS\_ABSTRACT = "isAbstract";

public static final String DIRECTION = "direction";

public static final String DEFAULT\_VALUE = "defaultValue";

public static final String VALUE = "value";

public static final String UPPER\_VALUE = "upperValue";

public static final String LOWER\_VALUE = "lowerValue";

public static final String TYPE\_HREF = "href";

public static final String RETURN = "return";

}

package com.diploma.classdiagram.parser;

import com.diploma.classdiagram.model.\*;

import com.diploma.classdiagram.model.relationships.RelationshipType;

import com.diploma.classdiagram.model.Visibility;

import com.diploma.classdiagram.model.Class;

import com.diploma.classdiagram.model.relationships.CardinalityRelationship;

import com.diploma.classdiagram.model.relationships.Relationship;

import com.diploma.classdiagram.model.relationships.RelationshipImpl;

import com.diploma.classdiagram.Constants;

import com.diploma.classdiagram.model.XMLElement;

import org.dom4j.Attribute;

import org.dom4j.Document;

import org.dom4j.DocumentException;

import org.dom4j.io.SAXReader;

import java.util.\*;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 25.08.12

\* Time: 22:45

\*/

public class ClassParser implements Parser {

private Map<String, Class> classes;

private Map<String, Relationship> relationships;

public ClassParser() {

classes = new HashMap<String, Class>();

relationships = new HashMap<String, Relationship>();

}

public Map<String, Class> getParsedClasses() {

return classes;

}

public Map<String, Relationship> getParsedRelationships() {

return relationships;

}

public List<XMLElement> parse(String sourceFile) throws UMLParserException {

Document document = getDocument(sourceFile);

if (null == document) {

throw new UMLParserException("Invalid source file path!");

}

// Getting the root element of xml file

org.dom4j.Element root = document.getRootElement();

if (null == root) {

throw new UMLParserException("Unable to find root element!");

}

// iterate through child elements of root with

// element name "packagedElement"

for (Iterator iterator = root.elementIterator(Constants.PACKAGED\_ELEMENT); iterator.hasNext(); ) {

// getting current packaged element

org.dom4j.Element packagedElement = (org.dom4j.Element) iterator.next();

//getting attribute "type"

// (it could be also "xmi:type" or something like that)

Attribute type = packagedElement.attribute(Constants.TYPE);

if (null == type) {

throw new UMLParserException("Can't determine type of packaged element " + packagedElement.getName());

}

//getting value of type

String typeValue = type.getValue();

//deciding what kind of element it is.

//there are two kinds :

//single relationship or class (interface is also class)

if (Constants.CLASS.equals(typeValue) || Constants.INTERFACE.equals(typeValue)) {

parseClassElement(packagedElement, classes);

} else if (Constants.REALIZATION.equals(typeValue) || Constants.DEPENDENCY.equals(typeValue) || Constants.ASSOCIATION.equals(typeValue)) {

parseRelationshipElement(packagedElement, relationships);

} else {

//We are simply skipping this elements

}

}

return null;

}

/\*\*

\* This method should parse class elements (also interfaces elements)

\* and return the map of current UML diagram.

\* Parsed element is adding to current class map.

\*

\* @param classElement - Current XML class element.

\* @param classes - Map of classes where result storing.

\* @return The same list of classes with new parsed class element.

\*/

private Map<String, Class> parseClassElement(org.dom4j.Element classElement, Map<String, Class> classes) {

//create new instance of class

Class newClass = new Class();

//set main attributes

newClass.setName(getAttributeValue(classElement, Constants.NAME));

newClass.setId(getAttributeValue(classElement, Constants.ID));

//set visibility type

newClass.setVisibility(getVisibility(classElement));

//check if final, static or abstract

newClass.setFinal(getBooleanAttributeValue(classElement, Constants.IS\_LEAF));

newClass.setStatic(getBooleanAttributeValue(classElement, Constants.IS\_STATIC));

newClass.setAbstract(getBooleanAttributeValue(classElement, Constants.IS\_ABSTRACT));

//check if it class or interface

newClass.setInterface(classElement.attribute(Constants.TYPE).getValue().equals(Constants.INTERFACE));

List<Field> fields = new ArrayList<Field>();

List<Method> methods = new ArrayList<Method>();

parseClassFields(classElement, fields);

parseClassMethods(classElement, methods);

// try to find generalizations elements

if (classElement.element(Constants.GENERALIZATION) != null) {

parseRealizations(classElement);

}

newClass.setFields(fields);

newClass.setMethods(methods);

classes.put(newClass.getId(), newClass);

return classes;

}

/\*\*

\* This method should parse class field element

\* and return the list of fields for current class.

\* Parsed element is adding to current fields list.

\*

\* @param element - Current XML class element.

\* @param fields - List of fields of current class where result storing.

\* @return The same list of fields with new parsed field.

\*/

private List<Field> parseClassFields(org.dom4j.Element element, List<Field> fields) {

// iterate through class attributes

for (Iterator iterator = element.elementIterator(Constants.ATTRIBUTE); iterator.hasNext(); ) {

// getting current attribute element

org.dom4j.Element attributeElement = (org.dom4j.Element) iterator.next();

Field field = new Field();

field.setId(getAttributeValue(attributeElement, Constants.ID));

field.setName(getAttributeValue(attributeElement, Constants.NAME));

field.setVisibility(getVisibility(attributeElement));

field.setFinal(getBooleanAttributeValue(attributeElement, Constants.IS\_LEAF));

field.setStatic(getBooleanAttributeValue(attributeElement, Constants.IS\_STATIC));

field.setType(getAttributeValue(attributeElement, Constants.TYPE));

// check for default value

org.dom4j.Element defaultValueElement = attributeElement.element(Constants.DEFAULT\_VALUE);

if (defaultValueElement != null) {

field.setValue(getAttributeValue(defaultValueElement, Constants.VALUE));

}

// if not user defined data type

if (field.getType().equals(Constants.EMPTY\_STRING)) {

org.dom4j.Element typeElement = attributeElement.element(Constants.TYPE);

if (typeElement != null) {

field.setType(getAttributeValue(typeElement, Constants.TYPE\_HREF));

}

}

// adding parsed attribute

fields.add(field);

}

return fields;

}

/\*\*

\* This method should parse class method element

\* and return the list of methods for current class.

\* Parsed element is adding to current methods list.

\*

\* @param element - Current XML class element.

\* @param methods - List of methods of current class where result storing.

\* @return The same list of methods with new parsed method.

\*/

private List<Method> parseClassMethods(org.dom4j.Element element, List<Method> methods) {

// iterate through class methods

for (Iterator iterator = element.elementIterator(Constants.METHOD); iterator.hasNext(); ) {

// getting current method element

org.dom4j.Element methodElement = (org.dom4j.Element) iterator.next();

Method method = new Method();

method.setId(getAttributeValue(methodElement, Constants.ID));

method.setName(getAttributeValue(methodElement, Constants.NAME));

method.setVisibility(getVisibility(methodElement));

method.setFinal(getBooleanAttributeValue(methodElement, Constants.IS\_LEAF));

method.setStatic(getBooleanAttributeValue(methodElement, Constants.IS\_STATIC));

method.setAbstract(getBooleanAttributeValue(methodElement, Constants.IS\_ABSTRACT));

List<MethodParameter> methodParameters = new ArrayList<MethodParameter>();

for (Iterator it = methodElement.elementIterator(Constants.METHOD\_PARAMETER); it.hasNext(); ) {

org.dom4j.Element parameterElement = (org.dom4j.Element) it.next();

MethodParameter methodParameter = new MethodParameter();

methodParameter.setId(getAttributeValue(parameterElement, Constants.ID));

methodParameter.setName(getAttributeValue(parameterElement, Constants.NAME));

methodParameter.setType(getAttributeValue(parameterElement, Constants.TYPE));

// check if it's return parameter

String direction = getAttributeValue(parameterElement, Constants.DIRECTION);

boolean isRetVal = !direction.equals(Constants.EMPTY\_STRING) && direction.equals(Constants.RETURN);

org.dom4j.Element typeElement = parameterElement.element(Constants.TYPE);

if (typeElement != null) {

methodParameter.setType(getAttributeValue(typeElement, Constants.TYPE\_HREF));

}

if (isRetVal) {

method.setReturnType(methodParameter.getType());

} else {

methodParameters.add(methodParameter);

}

method.setParameters(methodParameters);

}

methods.add(method);

}

return methods;

}

/\*\*

\* This method parse current "packagedElement" that is relationship.

\* It is parse all kinds of relationships except Generalization.

\*

\* @param relationshipElement - Current "packagedElement" in the tree.

\* @param relationships - List of relationships where would be

\* stored current parsed relationship.

\* @return List with parsed relationship.

\*/

private Map<String, Relationship> parseRelationshipElement(org.dom4j.Element relationshipElement, Map<String, Relationship> relationships) {

//getting name of element

String typeName = getAttributeValue(relationshipElement, Constants.TYPE);

//deciding what kind of association it is.

Relationship relationship = new RelationshipImpl();

relationship.setSingleElement(true);

//simple alone (dependency and realization)

if (typeName.equals(Constants.DEPENDENCY) || typeName.equals(Constants.REALIZATION)) {

relationship.setName(getAttributeValue(relationshipElement, Constants.NAME));

relationship.setId(getAttributeValue(relationshipElement, Constants.ID));

relationship.setSource(getAttributeValue(relationshipElement, Constants.SUPPLIER));

relationship.setDestination(getAttributeValue(relationshipElement, Constants.CLIENT));

relationship.setRelationshipType(typeName.equals(Constants.DEPENDENCY) ? RelationshipType.DEPENDENCY : RelationshipType.IMPLEMENTATION);

}

//structured alone relationship (composition, aggregation and association)

else {

relationship = parseSingleRelationship(relationshipElement);

}

// add parsed relationship to list

relationships.put(relationship.getId(), relationship);

return relationships;

}

/\*\*

\* This method parse current DOM element that is

\* "packagedElement" for relationship and have 3-level structure.

\*

\* @param relationshipElement - DOM element to parse.

\* @return Parsed Relationship element.

\* It could be Aggregation, Composition or Association.

\*/

private Relationship parseSingleRelationship(org.dom4j.Element relationshipElement) {

Relationship relationship = new CardinalityRelationship();

// set the name and identifier of element

relationship.setName(getAttributeValue(relationshipElement, Constants.NAME));

relationship.setId(getAttributeValue(relationshipElement, Constants.ID));

relationship.setSingleElement(true);

// get the source and destination describe elements

for (Iterator iterator = relationshipElement.elementIterator(Constants.OWNED\_END); iterator.hasNext(); ) {

// getting current owned element

org.dom4j.Element element = (org.dom4j.Element) iterator.next();

if (getAttributeValue(element, Constants.NAME).equals(Constants.SRC)) {

//set the identifier of source class

relationship.setSource(getAttributeValue(element, Constants.TYPE));

org.dom4j.Element upperValue = element.element(Constants.UPPER\_VALUE);

org.dom4j.Element lowerValue = element.element(Constants.LOWER\_VALUE);

((CardinalityRelationship) relationship).setSourceMinimum(getAttributeValue(lowerValue, Constants.VALUE));

((CardinalityRelationship) relationship).setSourceMaximum(getAttributeValue(upperValue, Constants.VALUE));

} else {

//set the identifier of destination class

relationship.setDestination(getAttributeValue(element, Constants.TYPE));

org.dom4j.Element upperValue = element.element(Constants.UPPER\_VALUE);

org.dom4j.Element lowerValue = element.element(Constants.LOWER\_VALUE);

((CardinalityRelationship) relationship).setDestinationMinimum(getAttributeValue(lowerValue, Constants.VALUE));

((CardinalityRelationship) relationship).setDestinationMaximum(getAttributeValue(upperValue, Constants.VALUE));

//this attribute describes relationship type.

Attribute attribute = element.attribute(Constants.AGGREGATION\_TYPE);

if (attribute == null) {

relationship.setRelationshipType(RelationshipType.ASSOCIATION);

} else if (attribute.getValue().equals(Constants.AGGREGATION)) {

relationship.setRelationshipType(RelationshipType.AGGREGATION);

} else {

relationship.setRelationshipType(RelationshipType.COMPOSITION);

}

}

}

return relationship;

}

/\*\*

\* This method parse class element and find all realizations

\*

\* @param classElement - Current class element.

\* @return List of realizations.

\*/

private Map<String, Relationship> parseRealizations(org.dom4j.Element classElement) {

Map<String, Relationship> generalizations = new HashMap<String, Relationship>();

for (Iterator iterator = classElement.elementIterator(Constants.GENERALIZATION); iterator.hasNext(); ) {

org.dom4j.Element generalization = (org.dom4j.Element) iterator.next();

Relationship relationship = new RelationshipImpl();

relationship.setId(getAttributeValue(generalization, Constants.ID));

relationship.setName(getAttributeValue(generalization, Constants.NAME));

relationship.setSingleElement(false);

relationship.setRelationshipType(RelationshipType.GENERALIZATION);

relationship.setSource(getAttributeValue(generalization, Constants.GENERAL));

relationship.setDestination(getAttributeValue(generalization.getParent(), Constants.ID));

generalizations.put(relationship.getId(), relationship);

}

relationships.putAll(generalizations);

return generalizations;

}

/\*\*

\* This method return the value of chosen attribute of element.

\* If element have no such attribute the result would be empty string.

\*

\* @param element - Current element to parse.

\* @param attributeName - Name of attribute

\* @return Value of current element

\*/

private String getAttributeValue(org.dom4j.Element element, String attributeName) {

if (element == null) throw new NullPointerException();

Attribute attribute = element.attribute(attributeName);

if (null == attribute) {

return Constants.EMPTY\_STRING;

}

String value = attribute.getValue();

if (null == value) {

return Constants.EMPTY\_STRING;

}

return value;

}

/\*\*

\* This method returns true or false value for attribute of element

\*

\* @param element - Current element

\* @param attributeName - Name of attribute

\* @return true or false

\*/

private boolean getBooleanAttributeValue(org.dom4j.Element element, String attributeName) {

if (element == null) throw new NullPointerException();

Attribute attribute = element.attribute(attributeName);

if (attribute == null) {

return false;

}

String value = attribute.getValue();

if (value == null || value.equals(Constants.FALSE)) {

return false;

}

return true;

}

/\*\*

\* This method should return the visibility type for current element

\*

\* @param element - Current element

\* @return Visibility type

\* @see Visibility

\*/

private Visibility getVisibility(org.dom4j.Element element) {

Visibility visibility = Visibility.PUBLIC;

String visibilityValue = getAttributeValue(element, Constants.VISIBILITY);

if (visibilityValue.equals(Constants.EMPTY\_STRING) || visibilityValue.equals(Constants.PUBLIC)) {

visibility = Visibility.PUBLIC;

} else if (visibilityValue.equals(Constants.PRIVATE)) {

visibility = Visibility.PRIVATE;

} else if (visibilityValue.equals(Constants.PACKAGED)) {

visibility = Visibility.PACKAGE;

} else if (visibilityValue.equals(Constants.PROTECTED)) {

visibility = Visibility.PROTECTED;

}

return visibility;

}

/\*\*

\* This method parsing the XML file

\* and return DOM object.

\*

\* @param filePath - Path to XML file that would be parsed.

\* @return - The Document object.

\*/

private Document getDocument(String filePath) {

SAXReader reader = new SAXReader();

Document document = null;

try {

document = reader.read(filePath);

} catch (DocumentException e) {

e.printStackTrace();

}

return document;

}

}

package com.diploma.classdiagram.parser;

public class UMLParserException extends Exception {

public UMLParserException() {

super();

}

public UMLParserException(String message) {

super(message);

}

public UMLParserException(String message, Throwable cause) {

super(message, cause);

}

public UMLParserException(Throwable cause) {

super(cause);

}

protected UMLParserException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {

super(message, cause, enableSuppression, writableStackTrace);

}

}

2 Модуль представления ДИАГРАММЫ КЛАССОВ В ВИДЕ ДИНАМИЧЕСКОЙ СТРУКТУРЫ ДАННЫХ

package com.diploma.classdiagram.model.relationships;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 23.09.12

\* Time: 12:53

\*/

public interface Cardinality {

/\*\*

\* Value to describe the infinity value

\*/

public static final Integer INFINITY = Integer.MAX\_VALUE;

/\*\*

\* This method should return minimum value of source cardinality.

\*

\* @return minimum value of source cardinality

\*/

public Integer getSourceMinimum();

/\*\*

\* This method should return maximum value of source cardinality.

\*

\* @return maximum value of source cardinality

\*/

public Integer getSourceMaximum();

/\*\*

\* This method should return minimum value of destination cardinality.

\*

\* @return minimum value of destination cardinality

\*/

public Integer getDestinationMinimum();

/\*\*

\* This method should return minimum value of source cardinality.

\*

\* @return minimum value of source cardinality

\*/

public Integer getDestinationMaximum();

/\*\*

\* @param value - Minimum value of source cardinality.

\*/

public void setSourceMinimum(Integer value);

/\*\*

\* @param value - Maximum value of source cardinality.

\*/

public void setSourceMaximum(Integer value);

/\*\*

\* @param value - Minimum value of destination cardinality.

\*/

public void setDestinationMinimum(Integer value);

/\*\*

\* @param value - Maximum value of destination cardinality.

\*/

public void setDestinationMaximum(Integer value);

}

package com.diploma.classdiagram.model.relationships;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 23.09.12

\* Time: 13:04

\*/

public class CardinalityRelationship implements Relationship, Cardinality {

private String id;

private String name;

private String source;

private String destination;

private boolean isSingleElement = false;

private RelationshipType relationshipType;

private Integer srcMinCardinality;

private Integer srcMaxCardinality;

private Integer dstMinCardinality;

private Integer dstMaxCardinality;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getSource() {

return source;

}

public void setSource(String source) {

this.source = source;

}

public String getDestination() {

return destination;

}

public void setDestination(String destination) {

this.destination = destination;

}

public boolean isSingleElement() {

return isSingleElement;

}

public void setSingleElement(boolean isSingleElement) {

this.isSingleElement = isSingleElement;

}

public RelationshipType getRelationshipType() {

return relationshipType;

}

public void setRelationshipType(RelationshipType type) {

this.relationshipType = type;

}

public Integer getSourceMinimum() {

return srcMinCardinality;

}

public Integer getSourceMaximum() {

return srcMaxCardinality;

}

public Integer getDestinationMinimum() {

return dstMinCardinality;

}

public Integer getDestinationMaximum() {

return dstMaxCardinality;

}

public void setSourceMinimum(Integer value) {

if (value == -1) {

srcMinCardinality = INFINITY;

return;

}

srcMinCardinality = value;

}

public void setSourceMaximum(Integer value) {

if (value == -1) {

srcMaxCardinality = INFINITY;

return;

}

srcMaxCardinality = value;

}

public void setDestinationMinimum(Integer value) {

if (value == -1) {

dstMinCardinality = INFINITY;

return;

}

dstMinCardinality = value;

}

public void setDestinationMaximum(Integer value) {

if (value == -1) {

dstMaxCardinality = INFINITY;

return;

}

dstMaxCardinality = value;

}

public void setSourceMinimum(String value) {

if (value.equals("\*")) {

srcMinCardinality = INFINITY;

return;

} else if ("".equals(value)) {

srcMinCardinality = 0;

return;

}

srcMinCardinality = Integer.valueOf(value);

}

public void setSourceMaximum(String value) {

if (value.equals("\*")) {

srcMaxCardinality = INFINITY;

return;

} else if ("".equals(value)) {

srcMaxCardinality = 0;

return;

}

srcMaxCardinality = Integer.valueOf(value);

}

public void setDestinationMinimum(String value) {

if (value.equals("\*")) {

dstMinCardinality = INFINITY;

return;

} else if ("".equals(value)) {

dstMinCardinality = 0;

return;

}

dstMinCardinality = Integer.valueOf(value);

}

public void setDestinationMaximum(String value) {

if (value.equals("\*")) {

dstMaxCardinality = INFINITY;

return;

} else if ("".equals(value)) {

dstMaxCardinality = 0;

return;

}

dstMaxCardinality = Integer.valueOf(value);

}

@Override

public String toString() {

String relationshipInfo =

"RelationshipImpl name : " + getName() + "\n" +

"RelationshipImpl type : " + getRelationshipType() + "\n" +

"source : " + getSource() + "\n" +

"destination : " + getDestination() + "\n" +

"isSingle : " + isSingleElement() + "\n" +

"srcMin = " + getSourceMinimum() + "\n" +

"srcMax = " + getSourceMaximum() + "\n" +

"dstMin = " + getDestinationMinimum() + "\n" +

"dstMax = " + getDestinationMaximum() + "\n" +

"ID : " + getId() + "\n";

return relationshipInfo;

}

}

package com.diploma.classdiagram.model.relationships;

import com.diploma.classdiagram.model.XMLElement;

/\*\*

\* User: ZIM

\* Date: 25.08.12

\* Time: 16:19

\*/

public interface Relationship extends XMLElement {

/\*\*

\* This method should return identifier

\* of source (also known as parent or supplier)

\* element of relationships.

\* <p/>

\* For example, if class A generalizing class B

\* method should return identifier of class B.

\*

\* @return - Source (also known as parent or supplier) identifier of relationship.

\*/

public String getSource();

/\*\*

\* @param source - Set the source (also known as parent or supplier) of relationship

\* @see #setSource(String) for detail information.

\*/

public void setSource(String source);

/\*\*

\* This method should return identifier

\* of destination (also known as child or client)

\* element of relationship.

\* <p/>

\* For example, if class A generalizing class B

\* method should return identifier of class A.

\*

\* @return - Destination (also known as child or client) identifier of relationship.

\*/

public String getDestination();

/\*\*

\* @param destination - Set the destination (also known as child or client) identifier of relationship.

\* @see #setDestination(String) for detail information.

\*/

public void setDestination(String destination);

/\*\*

\* This method should return

\* <b>true</b> if element is single in xml file and

\* <b>false</b> if element if a part(child) of owner packaged element.

\*/

public boolean isSingleElement();

/\*\*

\* @param isSingleElement - Set <b>true</b> if it is single and <b>false</b> if not.

\* @see #isSingleElement() for detail information.

\*/

public void setSingleElement(boolean isSingleElement);

/\*\*

\* Method should return relationship type

\* witch is one of:

\*

\* @return - RelationshipImpl type.

\* @see RelationshipType

\*/

public RelationshipType getRelationshipType();

/\*\*

\* Method set the relationship type.

\*

\* @param type - RelationshipImpl type.

\*/

public void setRelationshipType(RelationshipType type);

}

package com.diploma.classdiagram.model.relationships;

import com.diploma.classdiagram.model.XMLElement;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 25.08.12

\* Time: 17:42

\*/

public class RelationshipImpl implements Relationship {

private String id;

private String name;

private String source;

private String destination;

private boolean isSingleElement = false;

private RelationshipType relationshipType;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getSource() {

return source;

}

public void setSource(String source) {

this.source = source;

}

public String getDestination() {

return destination;

}

public void setDestination(String destination) {

this.destination = destination;

}

public boolean isSingleElement() {

return isSingleElement;

}

public void setSingleElement(boolean isSingleElement) {

this.isSingleElement = isSingleElement;

}

public RelationshipType getRelationshipType() {

return relationshipType;

}

public void setRelationshipType(RelationshipType type) {

this.relationshipType = type;

}

@Override

public String toString() {

String relationshipInfo =

"RelationshipImpl name : " + getName() + "\n" +

"RelationshipImpl type : " + getRelationshipType() + "\n" +

"source : " + getSource() + "\n" +

"destination : " + getDestination() + "\n" +

"isSingle : " + isSingleElement() + "\n" +

"ID : " + getId() + "\n";

return relationshipInfo;

}

}

package com.diploma.classdiagram.model.relationships;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 19.08.12

\* Time: 21:26

\*/

public enum RelationshipType {

// Наследование

GENERALIZATION,

// Реализация интерфейса

IMPLEMENTATION,

// Композиция

COMPOSITION,

// Агрегация

AGGREGATION,

// Ассоциация

ASSOCIATION,

// Зависимость

DEPENDENCY

}

package com.diploma.classdiagram.model;

import java.util.Iterator;

import java.util.List;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 18.08.12

\* Time: 17:29

\*/

public class Class implements XMLElement {

private String id;

private String name;

private Visibility visibility;

private boolean isFinal = false;

private boolean isStatic = false;

private boolean isAbstract = false;

private boolean isInterface = false;

private List<Field> fields;

private List<Method> methods;

public Class() {

}

public Class(String className) {

this.setName(className);

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public boolean isInterface() {

return isInterface;

}

public void setInterface(boolean anInterface) {

isInterface = anInterface;

}

public Visibility getVisibility() {

return visibility;

}

public void setVisibility(Visibility visibility) {

this.visibility = visibility;

}

public List<Field> getFields() {

return fields;

}

public void setFields(List<Field> fields) {

this.fields = fields;

}

public List<Method> getMethods() {

return methods;

}

public void setMethods(List<Method> methods) {

this.methods = methods;

}

public boolean isFinal() {

return isFinal;

}

public void setFinal(boolean aFinal) {

isFinal = aFinal;

}

public boolean isStatic() {

return isStatic;

}

public void setStatic(boolean aStatic) {

isStatic = aStatic;

}

public boolean isAbstract() {

return isAbstract;

}

public void setAbstract(boolean anAbstract) {

isAbstract = anAbstract;

}

@Override

public String toString() {

String classInfo =

"Class name : " + getName() + "\n" +

"isInterface : " + isInterface() + "\n" +

"Visibility : " + getVisibility() + "\n" +

"isStatic : " + isStatic() + "\n" +

"isFinal : " + isFinal() + "\n" +

"isAbstract : " + isAbstract() + "\n" +

"ID : " + getId() + "\n" +

"Fields list : " + "\n";

int count = 0;

for (Iterator iterator = fields.iterator(); iterator.hasNext(); ) {

count++;

classInfo += count + ". " + ((Field) iterator.next()).toString() + "\n";

}

count = 0;

classInfo += "Methods List : \n";

for (Iterator iterator = methods.iterator(); iterator.hasNext(); ) {

count++;

classInfo += count + ". " + ((Method) iterator.next()).toString() + "\n";

}

return classInfo;

}

}

package com.diploma.classdiagram.model;

/\*\*

\* User: ZIM

\* Date: 18.08.12

\* Time: 17:33

\*/

public class Field extends MethodParameter {

private Visibility visibility;

private boolean isFinal = false;

private boolean isStatic = false;

public Visibility getVisibility() {

return visibility;

}

public void setVisibility(Visibility visibility) {

this.visibility = visibility;

}

public boolean isFinal() {

return isFinal;

}

public void setFinal(boolean aFinal) {

isFinal = aFinal;

}

public boolean isStatic() {

return isStatic;

}

public void setStatic(boolean aStatic) {

isStatic = aStatic;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

if (!super.equals(o)) return false;

Field field = (Field) o;

if (isFinal != field.isFinal) return false;

if (isStatic != field.isStatic) return false;

if (visibility != field.visibility) return false;

if (getValue() != field.getValue()) return false;

return true;

}

@Override

public int hashCode() {

int result = super.hashCode();

result = 31 \* result + visibility.hashCode();

result = 31 \* result + (isFinal ? 1 : 0);

result = 31 \* result + (isStatic ? 1 : 0);

return result;

}

@Override

public String toString() {

String fieldInfo =

"Attribute name : " + getName() + "\n" +

"Visibility : " + getVisibility().toString() + "\n" +

"isStatic : " + isStatic() + "\n" +

"isFinal : " + isFinal() + "\n" +

"Type : " + getType() + "\n" +

"ID : " + getId() + "\n" +

"Value : " + getValue() + "\n";

return fieldInfo;

}

}

package com.diploma.classdiagram.model;

import java.util.Iterator;

import java.util.List;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 18.08.12

\* Time: 21:57

\*/

public class Method implements XMLElement {

private String id;

private String name;

private List<MethodParameter> parameters;

private String returnType;

private Visibility visibility = Visibility.PUBLIC;

private boolean isStatic = false;

private boolean isFinal = false;

private boolean isAbstract = false;

public List<MethodParameter> getParameters() {

return parameters;

}

public void setParameters(List<MethodParameter> parameters) {

this.parameters = parameters;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

/\*\*

\* @return - Return type of method.

\*/

public String getReturnType() {

return returnType;

}

/\*\*

\* @param returnType - Set the return type of method.

\*/

public void setReturnType(String returnType) {

this.returnType = returnType;

}

public Visibility getVisibility() {

return visibility;

}

public void setVisibility(Visibility visibility) {

this.visibility = visibility;

}

public boolean isFinal() {

return isFinal;

}

public void setFinal(boolean aFinal) {

isFinal = aFinal;

}

public boolean isStatic() {

return isStatic;

}

public void setStatic(boolean aStatic) {

isStatic = aStatic;

}

public boolean isAbstract() {

return isAbstract;

}

public void setAbstract(boolean anAbstract) {

isAbstract = anAbstract;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Method method = (Method) o;

if (!name.equals(method.getName())) return false;

if (isFinal != method.isFinal) return false;

if (isStatic != method.isStatic) return false;

if (!parameters.equals(method.parameters)) return false;

if (!returnType.equals(method.returnType)) return false;

//if (visibility != method.visibility) return false;

return true;

}

@Override

public int hashCode() {

int result = parameters.hashCode();

result = 31 \* result + returnType.hashCode();

result = 31 \* result + visibility.hashCode();

result = 31 \* result + (isStatic ? 1 : 0);

result = 31 \* result + (isFinal ? 1 : 0);

return result;

}

@Override

public String toString() {

String methodInfo =

"Method name : " + getName() + "\n" +

"Visibility : " + getVisibility() + "\n" +

"isStatic : " + isStatic() + "\n" +

"isFinal : " + isFinal() + "\n" +

"isAbstract : " + isAbstract() + "\n" +

"return type : " + getReturnType() + "\n" +

"ID : " + getId() + "\n";

methodInfo += "Parameters list : \n";

int count = 0;

if (parameters != null) {

for (Iterator iterator = parameters.iterator(); iterator.hasNext(); ) {

count++;

methodInfo += count + ". " + ((MethodParameter) iterator.next()).toString() + "\n";

}

}

return methodInfo;

}

}

package com.diploma.classdiagram.model;

/\*\*

\* User: ZIM

\* Date: 18.08.12

\* Time: 21:55

\*/

public class MethodParameter implements XMLElement {

private String id;

private String name;

private String type;

private String value;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

/\*\*

\* @return - Data type of parameter.

\*/

public String getType() {

return type;

}

/\*\*

\* @param type - Data type of parameter.

\*/

public void setType(String type) {

this.type = type;

}

/\*\*

\* @return - Value of parameter.

\*/

public String getValue() {

return value;

}

/\*\*

\* @param value - Value of parameter

\*/

public void setValue(String value) {

this.value = value;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

MethodParameter that = (MethodParameter) o;

if (!id.equals(that.id)) return false;

if (!name.equals(that.name)) return false;

if (!type.equals(that.type)) return false;

return true;

}

@Override

public int hashCode() {

int result = type.hashCode();

result = 31 \* result + value.hashCode();

return result;

}

@Override

public String toString() {

String paramInfo =

"Parameter name : " + getName() + "\n" +

"Type : " + getType() + "\n" +

"ID : " + getId() + "\n" +

"Value : " + getValue() + "\n";

return paramInfo;

}

}

package com.diploma.classdiagram.model;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 18.08.12

\* Time: 18:14

\*/

public enum Visibility {

PUBLIC,

PRIVATE,

PROTECTED,

PACKAGE

}

package com.diploma.classdiagram.model;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 18.08.12

\* Time: 16:54

\*/

public interface XMLElement {

/\*\*

\* @param id - Unique identifier of element.

\*/

public void setId(String id);

/\*\*

\* This method should return unique identifier of element.

\*

\* @return - Unique identifier.

\*/

public String getId();

/\*\*

\* This method set the name of element.

\*

\* @param name - Name of element.

\*/

public void setName(String name);

/\*\*

\* This method should return name of current element.

\*

\* @return - Name of element.

\*/

public String getName();

}

3 МОДУЛЬ ВЕРИФИКАЦИИ

package com.diploma.classdiagram.verification;

import org.apache.log4j.Logger;

/\*\*

\* User: ZIM

\* Date: 26.01.13

\* Time: 15:26

\*/

public interface Verification {

public static final Logger LOGGER = Logger.getLogger(Verification.class);

public boolean verify();

}

package com.diploma.classdiagram.verification;

/\*\*

\* User: d.ulanovych

\* Date: 11/19/13

\*/

public class SetVerification implements Verification {

@Override

public boolean verify() {

return false;

}

}

package com.diploma.classdiagram.verification;

import com.diploma.classdiagram.model.Class;

import com.diploma.classdiagram.model.relationships.RelationshipType;

import com.diploma.classdiagram.model.relationships.Cardinality;

import com.diploma.classdiagram.model.relationships.CardinalityRelationship;

import com.diploma.classdiagram.model.relationships.Relationship;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

/\*\*

\* User: d.ulanovych

\* Date: 04.03.13

\* Time: 10:06

\*/

/\*\*

\* Простой множественный цикл

\*/

public class PatternVerification implements Verification {

private final Map<String, Relationship> relationships;

private final Map<String, Class> classes;

public PatternVerification(Map<String, Class> classes, Map<String, Relationship> relationships) {

this.relationships = relationships;

this.classes = classes;

}

@Override

public boolean verify() {

boolean isValid = true;

isValid &= simpleMultipleCyclePattern();

return isValid;

}

/\*\*

\* Паттерн №1 : Простой множественный цикл.

\* TODO: should be formed as pattern library.

\*/

public boolean simpleMultipleCyclePattern() {

boolean isValid = true;

List<Cardinality> associations = getAssociations();

List<Cardinality> associationsBetweenSameClasses = new ArrayList<Cardinality>();

for (int i = 0; i < associations.size(); i++) {

CardinalityRelationship first = (CardinalityRelationship) associations.get(i);

associationsBetweenSameClasses.add(first);

for (int j = i + 1; j < associations.size(); j++) {

CardinalityRelationship second = (CardinalityRelationship) associations.get(j);

if ((first.getSource().equals(second.getSource()) || first.getSource().equals(second.getDestination()))

&& (first.getDestination().equals(second.getDestination()) || first.getDestination().equals(second.getSource()))

&& !first.getSource().equals(first.getDestination()) && !second.getSource().equals(second.getDestination())) {

associationsBetweenSameClasses.add(second);

}

}

if (associationsBetweenSameClasses.size() > 1 && !compareCardinalityAssociation(associationsBetweenSameClasses)) {

isValid = false;

}

associationsBetweenSameClasses.clear();

}

return isValid;

}

private List<Cardinality> getAssociations() {

List<Cardinality> associations = new ArrayList<Cardinality>();

for (Relationship relationship : relationships.values()) {

if (RelationshipType.ASSOCIATION.equals(relationship.getRelationshipType())) {

associations.add((CardinalityRelationship) relationship);

}

}

return associations;

}

private boolean compareCardinalityAssociation(List<Cardinality> associations) {

boolean isValid = true;

for (int i = 0; i < associations.size(); i++) {

Cardinality first = associations.get(i);

for (int j = i + 1; j < associations.size(); j++) {

Cardinality second = associations.get(j);

if (!compare(first, second)) {

LOGGER.error(PatternVerification.class.getSimpleName() +

". Associations incompatibility between classes : " +

classes.get(((Relationship) first).getSource()).getName() + " and " +

classes.get(((Relationship) first).getDestination()).getName() + " .\n" +

((Relationship) first).getName() + " and " + ((Relationship) second).getName() + " are incorrect!\n\n");

isValid = false;

}

}

}

return isValid;

}

private boolean compare(Cardinality first, Cardinality second) {

if (first.getSourceMinimum() > second.getSourceMaximum()

|| second.getSourceMinimum() > first.getSourceMaximum()

|| first.getDestinationMinimum() > second.getDestinationMaximum()

|| second.getDestinationMinimum() > first.getDestinationMaximum()) {

return false;

}

return true;

}

}

package com.diploma.classdiagram.verification;

import com.diploma.classdiagram.model.\*;

import com.diploma.classdiagram.model.Class;

import com.diploma.classdiagram.model.relationships.RelationshipType;

import com.diploma.classdiagram.model.relationships.CardinalityRelationship;

import com.diploma.classdiagram.model.relationships.Relationship;

import java.util.Iterator;

import java.util.Map;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 26.01.13

\* Time: 15:10

\*/

public class GraphVerification implements Verification {

private final Map<String, Class> classes;

private final Map<String, Relationship> relationships;

public GraphVerification(Map<String, Class> classes, Map<String, Relationship> relationships) {

this.relationships = relationships;

this.classes = classes;

}

@Override

public boolean verify() {

boolean isValid = true;

Iterator iterator = relationships.entrySet().iterator();

while (iterator.hasNext()) {

Map.Entry pair = (Map.Entry) iterator.next();

Relationship relationship = (Relationship) pair.getValue();

if (RelationshipType.ASSOCIATION.equals(relationship.getRelationshipType())) {

final CardinalityRelationship rel = (CardinalityRelationship) relationship;

final Double cardinality = calculateCardinality(rel);

if (cardinality < 1) {

isValid = false;

LOGGER.error(GraphVerification.class.getSimpleName()

+ ". Invalid multiplicity at association between classes:"

+ classes.get(rel.getSource()).getName() + " and "

+ classes.get(rel.getDestination()).getName() +

".\nGraph weight = " + cardinality + ". Must be >= 1. \n\n");

}

}

}

return isValid;

}

private Double calculateCardinality(final CardinalityRelationship relationship) {

Double sourceMinimum = Double.valueOf(relationship.getSourceMinimum());

Double sourceMaximum = Double.valueOf(relationship.getSourceMaximum());

Double destinationMinimum = Double.valueOf(relationship.getDestinationMinimum());

Double destinationMaximum = Double.valueOf(relationship.getDestinationMaximum());

Double cardinality = destinationMaximum / (1.0d / sourceMinimum) \* sourceMaximum \* (1.0d / destinationMinimum);

return cardinality;

}

}

package com.diploma.classdiagram.verification;

import com.diploma.classdiagram.model.Class;

import com.diploma.classdiagram.model.Field;

import com.diploma.classdiagram.model.Method;

import com.diploma.classdiagram.model.relationships.RelationshipType;

import com.diploma.classdiagram.model.Visibility;

import com.diploma.classdiagram.model.relationships.Relationship;

import java.util.\*;

/\*\*

\* Created with IntelliJ IDEA.

\* User: ZIM

\* Date: 26.01.13

\* Time: 11:20

\*/

public class DriverVerification implements Verification {

private final Map<String, Class> classes;

private final Map<String, Relationship> relationships;

public DriverVerification(Map<String, Class> classes, Map<String, Relationship> relationships) {

this.classes = classes;

this.relationships = relationships;

}

public boolean verify() {

boolean isValid = true;

isValid &= checkImplementationRules();

isValid &= checkInterfaceDefinitionRules();

isValid &= checkAbstractClassRules();

isValid &= checkForAbsenceOfMultipleGeneralization();

return isValid;

}

/\*\*

\* This method is checking all implementation rules.

\*

\* @return <b>true</b> if all implementation rules are observed.

\*/

public boolean checkImplementationRules() {

boolean isValid = true;

for (Relationship relationship : relationships.values()) {

if (RelationshipType.IMPLEMENTATION.equals(relationship.getRelationshipType())) {

Class source = classes.get(relationship.getSource());

Class destination = classes.get(relationship.getDestination());

if (!checkImplementation(source, destination)) {

isValid = false;

}

}

}

return isValid;

}

/\*\*

\* This method is checking access level of each interface's attribute and method.

\* Also check for the initialization of attributes of interface.

\*

\* @return <b>true</b> if access level of each attribute and method are valid.

\*/

public boolean checkInterfaceDefinitionRules() {

boolean isValid = true;

final Collection<Class> classCollection = classes.values();

for (Class clazz : classCollection) {

if (clazz.isInterface()) {

final List<Method> methods = clazz.getMethods();

final List<Field> fields = clazz.getFields();

for (Method method : methods) {

if (!(Visibility.PUBLIC.equals(method.getVisibility()) || Visibility.PACKAGE.equals(method.getVisibility()))) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Invalid accessor of method " + method.getName() +

" in interface " + clazz.getName() + ".\nIt must be PUBLIC only\n\n!");

isValid = false;

}

}

for (Field field : fields) {

if (!(Visibility.PUBLIC.equals(field.getVisibility()) || Visibility.PACKAGE.equals(field.getVisibility()))) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Invalid accessor of field " + field.getName() +

" in interface " + clazz.getName() + ".\nIt must be PUBLIC only!\n\n");

isValid = false;

}

if (null == field.getValue()) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Error in field " + field.getName() +

" of interface " + clazz.getName() + ".\nFields in interfaces must has value.\n\n");

isValid = false;

}

}

}

}

return isValid;

}

/\*\*

\* This method is checking rules for using keyword abstract.

\* If at least one method is declared as an abstract then hole class must be declared as <b>abstract</b>.

\*

\* @return <b>true</b> if abstract rules are followed.

\*/

public boolean checkAbstractClassRules() {

boolean isValid = true;

final Collection<Class> classCollection = classes.values();

for (Class clazz : classCollection) {

final List<Method> methods = clazz.getMethods();

for (Method method : methods) {

if (method.isAbstract() && !clazz.isAbstract()) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Invalid modifier at method "

+ method.getName() + " in class " + clazz.getName() +

".\nIf class has at least one abstract method - class must be marked as abstract!\n\n");

isValid = false;

}

}

}

return isValid;

}

/\*\*

\* This method is checking for the absence of multiple-generalization.

\*

\* @return <b>true</b> if generalization rules are followed.

\*/

public boolean checkForAbsenceOfMultipleGeneralization() {

boolean isValid = true;

Set<Class> destinationClasses = new HashSet<Class>();

for (Relationship relationship : relationships.values()) {

if (RelationshipType.GENERALIZATION.equals(relationship.getRelationshipType())) {

Class currentClass = classes.get(relationship.getDestination());

if (destinationClasses.contains(currentClass)) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Invalid inheritance at " + currentClass.getName() +

" class.\nEach class can extends only one super-class!\n\n");

isValid = false;

}

destinationClasses.add(currentClass);

}

}

return isValid;

}

/\*\*

\* This method is checking implementation rules between two classes.

\*

\* @param source Source class (parent), must be an interface.

\* @param destination Destination class (child) of implementation, must be a class.

\* @return Result of checking of implementation rules between two classes.

\* Return <b>true</b> when source class is an interface and destination class

\* is a class with all implemented methods or destination class is abstract class.

\*/

private boolean checkImplementation(Class source, Class destination) {

boolean isValid = true;

// Abstract class mustn't implement interface's methods.

if (destination.isAbstract()) {

return isValid;

}

if (!source.isInterface()) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Error at class " + source.getName() +

".\n Implementation relationship is using only for interfaces!\n\n");

return false;

}

for (Method method : source.getMethods()) {

if (!destination.getMethods().contains(method)) {

LOGGER.error(DriverVerification.class.getSimpleName() + ". Class " + destination.getName()

+ " doesn't implement method " + method.getName() + " of interface " + source.getName() + "\n\n");

isValid = false;

}

}

return isValid;

}

}

4 МОДУЛЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

package com.diploma.classdiagram.gui;

import org.apache.log4j.AppenderSkeleton;

import org.apache.log4j.spi.LoggingEvent;

import javax.swing.\*;

public class GuiAppender extends AppenderSkeleton {

private JTextArea log;

public GuiAppender(JTextArea log){

this.log = log;

}

@Override

protected void append(LoggingEvent event) {

log.append(event.getMessage().toString());

log.append("\n");

}

@Override

public void close() {

}

@Override

public boolean requiresLayout() {

return false;

}

}

package com.diploma.classdiagram.gui;

import com.diploma.classdiagram.parser.ClassParser;

import com.diploma.classdiagram.parser.UMLParserException;

import com.diploma.classdiagram.verification.\*;

import javax.swing.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.io.File;

public class ClassTester {

private JPanel panel1;

private JRadioButton graphRadioButton;

private JRadioButton testDriverRadioButton;

private JRadioButton setRadioButton;

private JRadioButton patternsRadioButton;

private JRadioButton complexRadioButton;

private JButton verifyButton;

private JButton fileSelector;

private JTextArea log;

private File currentDiagram;

final JFileChooser fileChooser = new JFileChooser();

public ClassTester() {

testDriverRadioButton.setSelected(true);

Verification.LOGGER.addAppender(new GuiAppender(log));

testDriverRadioButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

testDriverRadioButton.setSelected(true);

graphRadioButton.setSelected(false);

setRadioButton.setSelected(false);

patternsRadioButton.setSelected(false);

complexRadioButton.setSelected(false);

}

});

graphRadioButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

graphRadioButton.setSelected(true);

testDriverRadioButton.setSelected(false);

setRadioButton.setSelected(false);

patternsRadioButton.setSelected(false);

complexRadioButton.setSelected(false);

}

});

setRadioButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

setRadioButton.setSelected(true);

graphRadioButton.setSelected(false);

testDriverRadioButton.setSelected(false);

patternsRadioButton.setSelected(false);

complexRadioButton.setSelected(false);

}

});

patternsRadioButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

patternsRadioButton.setSelected(true);

graphRadioButton.setSelected(false);

setRadioButton.setSelected(false);

testDriverRadioButton.setSelected(false);

complexRadioButton.setSelected(false);

}

});

complexRadioButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

complexRadioButton.setSelected(true);

graphRadioButton.setSelected(false);

setRadioButton.setSelected(false);

patternsRadioButton.setSelected(false);

testDriverRadioButton.setSelected(false);

}

});

fileSelector.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

fileChooser.setCurrentDirectory(currentDiagram);

int returnVal = fileChooser.showOpenDialog(null);

if (returnVal == JFileChooser.APPROVE\_OPTION) {

currentDiagram = fileChooser.getSelectedFile();

log.append("Selected diagram :"+currentDiagram.getAbsolutePath() + "\n\n");

}

}

});

verifyButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

verify();

}

});

}

public static void main(String[] args) {

JFrame frame = new JFrame("UML Tester");

frame.setContentPane(new ClassTester().panel1);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.pack();

frame.setVisible(true);

}

private void verify() {

if (null != currentDiagram) {

ClassParser parser = new ClassParser();

Verification verification = null;

try {

parser.parse(currentDiagram.getAbsolutePath());

} catch (UMLParserException e) {

for (StackTraceElement element : e.getStackTrace()) {

log.append(element.toString());

log.append("\n");

}

return;

}

if (testDriverRadioButton.isSelected()) {

verification = new DriverVerification(parser.getParsedClasses(), parser.getParsedRelationships());

} else if (setRadioButton.isSelected()) {

verification = new SetVerification();

} else if (patternsRadioButton.isSelected()) {

verification = new PatternVerification(parser.getParsedClasses(), parser.getParsedRelationships());

} else if (graphRadioButton.isSelected()) {

verification = new GraphVerification(parser.getParsedClasses(), parser.getParsedRelationships());

}

if (null != verification) {

verification.verify();

} else if (complexRadioButton.isSelected()) {

verification = new DriverVerification(parser.getParsedClasses(), parser.getParsedRelationships());

verification.verify();

verification = new SetVerification();

verification.verify();

verification = new PatternVerification(parser.getParsedClasses(), parser.getParsedRelationships());

verification.verify();

verification = new GraphVerification(parser.getParsedClasses(), parser.getParsedRelationships());

verification.verify();

}

}

}}