МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

ЧЕРНИГОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

КАФЕДРА ИНФОРМАЦИОННЫХ И КОМПЬЮТЕРНЫХ СИСТЕМ

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

Специальность: 8.05010202 – Системное программирование

Комплексная курсовая работа

Руководитель:

д.т.н., профессор В.В Литвинов

Исполнитель

ст. гр. МП-081 Д.Н. Уланович

Чернигов 2013

РЕФЕРАТ

Комплексная курсовая работа, 47 страниц, 30 рисунков, 1 таблицы, 13 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 разработчикам программного обеспечения ускорить процесс перехода от UML модели к реализации. В отличие от существующих, данная система позволяет в комплексе несколькими методами оценивать корректность диаграмм классов. Методы оценивают как разные части модели, так и позволяет оценить одну часть 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 – разработка 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

Комплексна курсова робота, 47 сторінок, 30 рисунків, 1 таблиця, 13 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 розробникам програмного забезпечення прискорити процес переходу від UML моделі до реалізації. На відміну від існуючих, дана система дозволяє в комплексі кількома методами оцінювати коректність діаграм класів. Методи оцінюють як різні частини моделі, так і дозволяють оцінити одну частину 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 - розробка 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

A comprehensive course work, 47 pages, 30 figures, 1 table, 13 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 allow software developers to accelerate the transition from UML models to implementation. Unlike existing, the current system allows several methods to evaluate complex correct class diagrams. Methods could estimate different parts of the model and to estimate one piece with 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 - the development 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 6](#_Toc359058391)

[1 АНАЛИЗ СУЩЕСТВУЮЩИХ СИСТЕМ ВЕРИФИКАЦИИ   
 UML-ДИАГРАММ 7](#_Toc359058392)

[1.1 Существующие системы верификации UML-диграмм 7](#_Toc359058393)

[1.1.1 ArgoUML 7](#_Toc359058394)

[1.1.2 Telelogic TAU G2 9](#_Toc359058395)

[1.1.3 Visual Paradigm for UML 11](#_Toc359058396)

[1.1.4 Enterprise Architect 15](#_Toc359058397)

[1.2 Сравнительная характеристика существующих систем   
 верификации диаграмм последовательности 20](#_Toc359058398)

[2 ОБЩИЕ МОДЕЛИ И МЕТОДОЛОГИИ РАЗРАБОТКИ   
 ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ПРОГРАММНЫХ СИСТЕМ 21](#_Toc359058399)

[2.1 Модели разработки 21](#_Toc359058400)

[2.1.1 Каскадная модель 22](#_Toc359058401)

[2.1.2 Инкрементальная модель 23](#_Toc359058402)

[2.1.3 Спиральная модель 27](#_Toc359058403)

[2.1.4 Модель прототипирования 29](#_Toc359058404)

[3 РАЗРАБОТКА СТРУКТУРЫ ДЛЯ ПОДСИСТЕМЫ ВЕРИФИКАЦИИ  
 ДИАГРАММЫ КЛАССОВ 31](#_Toc359058405)

[3.1 Задачи, выполняемые подсистемой   
 верификации диаграммы классов 31](#_Toc359058406)

[3.2 Архитектура системы верификации UML-диаграмм «UML Tester» 31](#_Toc359058407)

[3.3 Структура подсистемы верификации диаграммы классов 32](#_Toc359058408)

[3.3.1 Модуль обработки XML данных 33](#_Toc359058409)

[3.3.2 Модуль верификации диаграммы классов 36](#_Toc359058410)

[3.4 Используемые методы верификации диаграмм классов 36](#_Toc359058411)

[3.4.1 Метод шаблонов 36](#_Toc359058413)

[3.4.2 Метод идентификационного графа 39](#_Toc359058414)

[3.4.3 Метод множеств 41](#_Toc359058415)

[3.4.4 Метод тестового драйвера 43](#_Toc359058412)

[ВЫВОДЫ 45](#_Toc359058416)

[ПЕРЕЧЕНЬ ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ 46](#_Toc359058417)

ВВЕДЕНИЕ

Диаграмма классов является одной из основных составляющих модели любого программного обеспечения [9]. 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, и позволит разработчикам программного обеспечения ускорить процесс перехода от UML модели к реализации. 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 оценивают как разные части модели, так и позволяет оценить одну часть 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 разработки программной модели и на этапе перехода от модели к 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

На данный момент существует достаточно много инструментальных средств, позволяющих выполнять как различные виды тестирования, так и несколько видов тестирования программного обеспечения в комплексе.

Однако, существует небольшое количество инструментальных средств, которые выполняют верификацию программного обеспечения и, в частности, верификацию моделей программного обеспечения, где 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 [1].

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 Telelogic TAU G2

IBM Rational Tau обеспечивает стандартизованную разработку сложных систем и программного обеспечения на основе моделей [2]. Это решение используется при разработке информационных систем и корпоративных ИТ-приложений, в том числе архитектуры SOA.

- поддерживает разработку сервис-ориентированной архитектуры (SOA) на основе моделей;

- обеспечивает автоматическую проверку на наличие ошибок и проверку моделей на основе правил, а также предоставляет анализатор на базе моделей;

- объединяет процессы моделирования требований и разработки встроенных приложений, что позволяет группам разработчиков сосредоточиться на создании необходимых компонентов проекта или программы;

- обеспечивает возможность совместной работы, что позволяет всем участникам даже самых масштабных проектов легко и быстро связываться друг с другом и обмениваться данными;

- автоматически создает пользовательские отчеты и документацию на любой стадии цикла разработки;

- обеспечивает автоматическое генерирование и выполнение тестов.

- имеет открытую архитектуру, основанную на UML 2.1, что позволяет дополнять и настраивать Tau в соответствии с потребностями организации.

- поддерживаемые операционные системы: Sun Solaris, Windows.

Существует опция Verifier, которая в режиме "реального" времени (с возможность задания масштаба условной единицы времени) позволяет моделировать поведение всей системы или ее отдельных частей, с использованием MSС-диаграмм (Message Sequence Chart).

Пользователю предоставляется возможность задавать любые входные воздействия, что позволяет пользователю исследовать, как модель реагирует и отрабатывает тот функционал, который в нее закладывался. При этом в любом месте любой диаграммы пользователю предоставляется возможгность поставить знак остановки (break-point) и в специальных окнах (watch window) контролировать значения любых переменных. Наличие опции Verifier позволяет находить ошибки на самых ранних стадиях дизайна, что сокращает время на поиск ошибок и значительно снижает затраты на тестирование.

Существует возможность генерации кода, который автоматически создается встроенным кодо-генератором. В настоящее время Tau имеет 4 независимых кодогенератора: С, С++, C#, Java.

При таком подходе - если речь, например, идет о real-time или embedded systems - пользователь имеет возможность создавать модель (систему), не привязываясь к конкретной операционной системе или типу процессора. Такой подход дает возможность сосредоточить усилия разработчиков именно на создании модели и ее функционала.

Tau имеет 3 составляющих - собственно Tau, Tau/Architect и Tau/Developer [2] - каждая из которых имеет свое предназначение:

а) Tau:

1) построение моделей с использованием диаграмм, имеющихся в стандарте UML2.0;

2) проверка диаграмм на соответствие синтаксису и семантики (нотации) языка UML2.0;

3) автоматический поиск и локализация ошибок в диаграммах.

б) Tau/Architect:

1) построение моделей с использованием диаграмм, имеющихся в стандарте UML2.0;

2) проверка диаграмм на соответствие синтаксису и семантики (нотации) языка UML2.0;

3) автоматический поиск и локализация ошибок в диаграммах;

4) динамическая проверка поведения модели;

5) отработка и корректировка функционала, закладываемого в модель.

в) Tau/Developer:

1) построение моделей с использованием диаграмм, имеющихся в стандарте UML2.0;

2) проверка диаграмм на соответствие синтаксису и семантики (нотации) языка UML2.0;

3) автоматический поиск и локализация ошибок в диаграммах;

4) динамическая проверка поведения модели;

5) отработка и корректировка функционала, закладываемого в модель;

6) автоматическая генерация до исходного кода на языке используемого кодогенератора.

К недостаткам Telelogic можно отнести отсутствие каких-либо демоверсий или пробных версий с ограниченной функциональностью. Так же данное ПО является проприетарным, что ограничивает возможность его использования и распространения.

1.1.3 Visual Paradigm for UML

Программное обеспечение Visual Paradigm for UML является средством моделирования систем на языке UML и с помощью UML-диаграмм всех типов [3]. 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 1.3), а при генерации кода просто игнорируется факт ассоциации.



Рисунок 1.3 – 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 1.4).



Рисунок 1.4 – Диаграмма с неверным наследованием в Visual Paradigm

Но в отличии от ArgoUML Visual Paradigm правильно отреагировал на допущенную ошибку наследования, выдав предупредительное сообщение (рисунок 1.5), сообщающее пользователю что класс Child не может наследоваться более чем от одного суперкласса.



Рисунок 1.5 – Сообщение об ошибке наследование

Так же была создана диаграмма классов, содержащая неабстрактный класс в котором присутствует абстрактный метод (рисунок 1.6).



Рисунок 1.6 – Неабстрактный класс с абстрактным методом

Visual Paradigm сообщает пользователю, что абстрактная операция abstractMethod() может быть создана только в абстрактном классе (рисунок 1.7).



Рисунок 1.7 – 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 1.8.



Рисунок 1.8 – 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 1.9 представлена диаграмма, при верификации которой должно выдаваться сообщение об ошибке.



Рисунок 1.9 – 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.

1.1.4 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 [4].

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 1.10), выводится сообщение о наличии нуля ошибок и предупреждений (рисунок 1.11).



Рисунок 1.10 – Отсутствие верификации ассоциации в Enterprise Architect



Рисунок 1.11 – 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 1.12). На этой диаграмме содержатся как множественное наследование так и циклическое наследование, что запрещено во всех языках программирования.



Рисунок 1.12 – 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 1.13).



Рисунок 1.13 – 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 1.14.



Рисунок 1.14 – 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 1.15 представлена диаграмма, при верификации которой должно выдаваться сообщение об ошибке.



Рисунок 1.15 –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 | TelelogicTAUG2 | Visual Paradigm | Enterprise Architect |
| Поддержка UML 2.0 и выше | Нет | Да | Да | Да |
| Открытость кода, бесплатное распространение | Да | Нет | Нет | Нет |
| Возможность создания диаграммы классов | Да | Да | Да | Да |
| Возможность верификации диаграмм классов | Да | Да | Да | Да |
| Наличие информации о используемых методах верификации диаграммы классов | Нет | Нет | Нет | Нет |
| Наличие рекомендаций по исправлению ошибок верификации | 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 [7]. 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 более старый подход 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, идя навстречу изменяющимся требованиям пользователей. Внедрение ускоренного процесса явилось 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, koskmudel, waterfall model)

- спиральная модель (spiraalmudel, spiral model)

- инкрементная модель (inkrementaalmudel, incremental model)

- прототипирование (prototüüpimine, prototyping).

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 [7].

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 и т.д.

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 [7]. 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 и т.д.), чем полагаются на предварительное тщательное планирование технологии. Основное внимание уделяется людям, в том числе пользователям и постоянному тестированию. Говорят, что 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. Течение процесс изображает спираль. Первый виток (повторение) может быть, например, связан с исследованиями осуществимости системы, второй - с описанием требований, следующий - с планированием и т.д. 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 [7]. 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 и т.д.), 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 (ühekordne prototüüpimine, Throwaway prototyping), эволюционное прототипирование (evolutsiooniline prototüüpimine, Evolutionary prototyping), инкрементное прототипирование (lisanduv prototüüpimine, 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

3.1 Задачи, выполняемые подсистемой верификации диаграммы классов

Формальная верификация программного обеспечения – это приемы и методы формального доказательства (или опровержения) того, что модель программного обеспечения удовлетворяет заданной формальной спецификации [8]. 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.

3.2 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.

На рисунке 3.2 приведен обобщенный участок кода для парсера этих XML структур данных в динамическую модель:

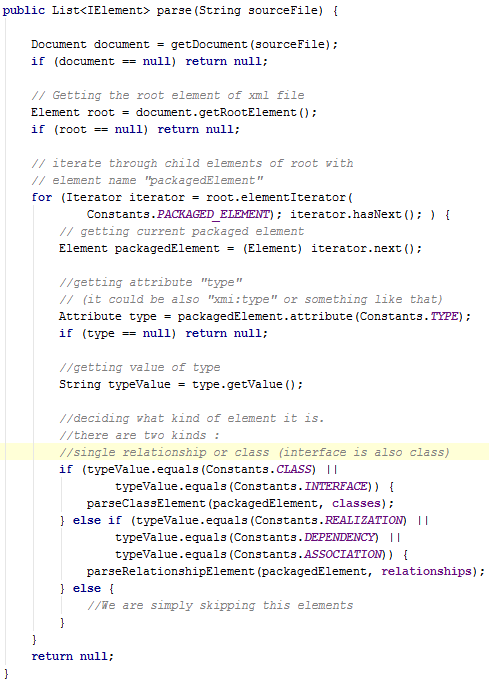


Рисунок 3.2 – Обобщенный код парсера

После отработки парсера, строится динамическая структура данных, с которой и работает верификатор.

3.3.2 Модуль верификации диаграммы классов

Модуль верификации диаграммы классов состоит в произведении обработки данных полученных с XML файла. Проводится верификация одних и тех же данных четырьмя методами верификации (рисунок 3.3).



Рисунок 3.3 – Работа модуля верификации

3.4 Используемые методы верификации диаграмм классов

3.4.1 Метод шаблонов

Еще одним подходом к верификации диаграммы классов является метод шаблонов. Данный метод предложила 2004 году Мира Балабан [11]. 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 3.4 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 3.4 - 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 3.5 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 3.5- 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 3.6 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 3.6 - 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».

3.4.2 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 [12].

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 3.7 и 3.8.



Рисунок 3.7 - Простейшая диаграмма классов



Рисунок 3.8 - Идентификационный граф для диаграммы классов с рисунка 3.7

На рисунке 3.7 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 3.8. 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 3.9 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 3.10.



Рисунок 3.9 - Пример использования метода идентификационного графа



Рисунок 3.10 - 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.

3.4.3 Метод множеств

Еще один подход к верификации диаграммы классов основан на представлении класса в качестве множества. Данный метод был предложен Calvanese и Lenzerini [13] 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 3.11 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 3.11 - 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.

3.4.4 Метод тестового драйвера

Данный подход позволяет формально оценить правильность создания диаграммы классов и выявить наиболее характерные ошибки [10]. 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.

ВЫВОДЫ

В работе описано несколько 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.

ПЕРЕЧЕНЬ ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ

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

2 IBM Rational Document Download Information Center [Электронный ресурс] : справка – Режим доступа : [http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/index.jsp](http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/index.jsp?topic=/com.ibm.help.download.tau.doc/topics/taudownload42.html)

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

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

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

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

7 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)

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

9 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.

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

11 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.

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

13 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.