|  |  |  |
| --- | --- | --- |
|  | **Софийски университет „Св. Кл. Охридски”**  Факултет по математика и информатика  *Катедра „Софтуерни технологии”* |  |

**ДИПЛОМНА РАБОТА**

на тема

„Mоделно базирана разработка на софтуер за вградена софтуерна система чрез автоматично извличане на архитектурна информация”

Дипломант: **Димитър Делянов Манев**

Специалност: **Софтуерни технологии**

Факултетен номер: **M22499**

Научен ръководител:

**доц. д-р Александър Димов**

София, 2014 г.

**Съдържание**

[1. Увод 5](#_Toc410247677)

[1.1 Състояние на индустрията 5](#_Toc410247678)

[1.2 Софтуер за вградени системи 5](#_Toc410247679)

[1.3 Софтуерна архитектура 6](#_Toc410247680)

[1.4 Мотивация за решението 7](#_Toc410247681)

[1.5 Цел и задачи на дипломната работа 7](#_Toc410247682)

[1.6 Очаквани ползи от реализацията 8](#_Toc410247683)

[1.7 Структура на дипломната работа (0,5-1стр.) 8](#_Toc410247684)

[2. Реверсивен инжинеринг и възстановяване на дизайн 9](#_Toc410247685)

[2.1 Основни дефиниции 9](#_Toc410247686)

[2.1.1 Терминология на реинженеринга и софтуерната подръжка 9](#_Toc410247687)

[2.1.2 Терминология в софтуерната архитектура 10](#_Toc410247688)

[2.2 Реинженеринг на компонентно-базиран софтуер и среда за изпълнението му 13](#_Toc410247689)

[2.2.1 Реинженеринг и компонентно-базиран софтуер 13](#_Toc410247690)

[2.2.2 Среда за архитектурна реконструкция 14](#_Toc410247691)

[2.3 Съществуващи инструменти за реинженеринг 16](#_Toc410247692)

[2.3.1 Оркестрирана много-изгледна среда за софтуерно архитектурна реконструкция [R14] 17](#_Toc410247693)

[2.3.2 Компонентният модел на Dassault Systèmes (DS) [R14] 25](#_Toc410247694)

[2.4 Сравнителен анализ на изложените решения 29](#_Toc410247695)

[2.4.1 Критерии 29](#_Toc410247696)

[2.4.2 Сравнителен анализ 30](#_Toc410247697)

[2.5 Изводи 31](#_Toc410247698)

[3. Анализ 32](#_Toc410247699)

[3.1 Концептуален модел 32](#_Toc410247700)

[3.2 Работни процеси 34](#_Toc410247701)

[3.2.1 Подготовка на критерии за анализ 34](#_Toc410247702)

[3.2.2 Анализиране на проект и сериализация на хранилището 35](#_Toc410247703)

[3.2.3 Генерация на базов код 36](#_Toc410247704)

[3.3 Потребителски (функционални) изисквания 36](#_Toc410247705)

[3.3.1 Типични случаи на употреба 36](#_Toc410247706)

[3.3.2 Мета-модел на архитектурното хранилище 44](#_Toc410247707)

[3.3.3 Формат на генерирания базов код 49](#_Toc410247708)

[3.3.4 Група от критерии за стандартна архитектура 51](#_Toc410247709)

[3.4 Качествени (нефункционални) изисквания 57](#_Toc410247710)

[3.4.1 Скалируемост 57](#_Toc410247711)

[3.4.2 Модифицируемост и документация 57](#_Toc410247712)

[3.4.3 Поддръжка и възможност за разширение 57](#_Toc410247713)

[3.4.4 Потребителски интерфейс 57](#_Toc410247714)

[3.4.5 Тестваемост 58](#_Toc410247715)

[3.5 Изводи 58](#_Toc410247716)

[4. Използвани технологии, платформи и методологии 59](#_Toc410247717)

[4.1 Изисквания към средствата (технологии, платформи и методологии) 59](#_Toc410247718)

[4.1.1 Език за програмиране 59](#_Toc410247719)

[4.1.2 Модел на софтуерната система 59](#_Toc410247720)

[4.1.3 UML и формати за представянето му 60](#_Toc410247721)

[4.1.4 Генератор на базовия код 61](#_Toc410247722)

[4.2 Видове средства за разработване на решението – оценка спрямо критериите 61](#_Toc410247723)

[4.2.1 Език за програмиране: 61](#_Toc410247724)

[4.2.2 Среда за разработване на UML модел 62](#_Toc410247725)

[4.2.3 Код генератор 62](#_Toc410247726)

[4.3 Избор на средствата (технологии, платформи и методологии) 63](#_Toc410247727)

[4.3.1 Език за програмиране 63](#_Toc410247728)

[4.3.2 Среда за разработване на UML модел 64](#_Toc410247729)

[4.3.3 Генератор на базов код 64](#_Toc410247730)

[4.4 Изводи 65](#_Toc410247731)

[5. Проектиране 67](#_Toc410247732)

[5.1 Обща архитектура 67](#_Toc410247733)

[5.1.1 Слоеве 67](#_Toc410247734)

[5.1.2 Пакетна диаграма (основен изглед) 68](#_Toc410247735)

[5.2 Модел на данните (Мета-Модел) 69](#_Toc410247736)

[5.2.1 Инфраструктурни 70](#_Toc410247737)

[5.2.2 Софтуерен компонент 72](#_Toc410247738)

[5.3 Диаграми (на структура и поведение - по слоеве) 75](#_Toc410247739)

[5.3.1 Слой Анализатор 75](#_Toc410247740)

[5.3.2 Слой Скенер 78](#_Toc410247741)

[5.3.3 Слой Мета-Модел 80](#_Toc410247742)

[5.3.4 Слой Сериализатор 81](#_Toc410247743)

[5.3.5 Слой Генерация на базов код 83](#_Toc410247744)

[5.4 Ресурсни и спомагателни модули 84](#_Toc410247745)

[5.4.1 Пакетна диаграма 85](#_Toc410247746)

[6. Реализация, тестване/експерименти и (евентуално) внедряване 86](#_Toc410247747)

[6.1 Реализация на модулите 86](#_Toc410247748)

[6.1.1 Анализатор (диаграма на внедряване) 86](#_Toc410247749)

[6.1.2 Скенер 86](#_Toc410247750)

[6.1.3 Мета-модел 87](#_Toc410247751)

[6.1.4 Сериализатор (диаграма на внедряване) 88](#_Toc410247752)

[6.1.5 Спомагателни модули (диаграма на внедряване) 88](#_Toc410247753)

[6.1.6 Специфични критерии за стандартна архитектура (диаграма на внедряване) 88](#_Toc410247754)

[6.1.7 Генерация на базов код (реализация) 89](#_Toc410247755)

[6.2 Системна интеграция (опционално) 89](#_Toc410247756)

[6.3 Планиране на тестването - тестови сценарии, процедури, ... 89](#_Toc410247757)

[6.3.1 Цели 89](#_Toc410247758)

[6.3.2 Модулни тестове 90](#_Toc410247759)

[6.3.3 Функционални тестове 90](#_Toc410247760)

[6.3.4 Нефункционални тестове 90](#_Toc410247761)

[6.4 Модулно и системно тестване 90](#_Toc410247762)

[6.5 Анализ на резултатите от тестването и начин на отразяването им 90](#_Toc410247763)

[6.6 Експериментално внедряване (технологични изисквания, инсталиране, условия, използване, ...) 90](#_Toc410247764)

[7. Заключение 91](#_Toc410247765)

[7.1 Обобщение на изпълнението на началните цели 91](#_Toc410247766)

[7.2 Насоки за бъдещо развитие и усъвършенстване 91](#_Toc410247767)

[7.3 Използвана литература 91](#_Toc410247768)

[8. Приложения 94](#_Toc410247769)

[Приложение 1 Терминологичен речник 94](#_Toc410247770)

[Приложение 2 Реализирани документи 95](#_Toc410247771)

[Приложение 3 Степен на изразителност на езиците за програмиране 96](#_Toc410247772)

[Приложение 4 “Ръчно” написан код генератор 96](#_Toc410247773)

[Приложение 5 Шаблони за генериране на базов код 96](#_Toc410247774)

[Приложение 6 Легенда на диаграмите за работни процеси 101](#_Toc410247775)

[Приложение 7 Карта на работните процеси 102](#_Toc410247776)

# Увод

**“(1) Закон за непрекъснатост на промяната:** Програма, която се използва в естествена среда задължително трябва да се променя или прогресивно ще става все повече и повече неизползваема.

**(2) Закон за нарастващата сложност:** С развитието на една програма нейната структура става все по-сложна. Все повече ресурси са необходими за да се спазва и опростява нейната структура.”

Първите два от осемте закона на Лехман и Белади, 1985

**Абстракт:**

Разработката на софтуер заема все по-голям дял във все повече и повече индустрии. Съответно софтуерът за вградени системи навлиза все повече и повече в живота ни, като растежа на разпространението му е огромен. Тъй като най-често езика за разработка на такива системи е “C”, възможностите да се използват готови инструменти за моделиране и компоненти модели са минимални. Използвайки основни дефиниции от софтуерната архитектура, предложеното решение се фокусира върху: 1) възможността да се извлича архитектурна информция в стандартен унифициран модел (UML) от вече съществуваща вградена система писана на езика “C” и 2) възможността от вече извлечения модел да се генерира базов код за разработване на подобна система. Описани са основните цели на решението както и очакваните ползи от него.

## Състояние на индустрията

Софтуера е фактор с нарастваща важност за разходите и печалбите на пазарните продукти, не само в рамките на традиционните „софтуерно доминирани” домейни като телекомуникации и информационни системи, но също така и в други технологично ориентирани отрасли като механика, авиация, астронавтика или развлекателна индустрия, чиито дял на разходи за разработка на софтуер е от 30-50 процента.

Изучаването на различни случаи показва, че 60-80% разходите по софтуерен продукт произлизат от еволюции на програмите [R20, стр. 157-174]. Повече от 50% от времето за еволюция на програма се изкарва в разучаване на програмата, преди въпросната промяна да бъде проектирана и реализирана, което е показано в няколко различни случая [R21, ръководство 48]. Това се налага тъй като необходимата информация за задачата е често непълна и некоректно документирана и за това трябва да се извлече от изходния код. Отговорните за поддръжката, зле информирани и притиснати от сроковете на проекта, често коригират проблема локално, предимно в под-системите, с които са запознати най-добре. Тези локални промени често пренебрегват оригиналния дизайн и тъй като не са истински решения, а само третират проблема симптоматично, предизвикват проблеми в други части на системата и усложняват бъдещата работа по системата. Това е един порочен кръг, който завършва с докарването на една система до състояние, в което тя вече не може да се поддържа освен ако не се взимат превантивни мерки.

## Софтуер за вградени системи

*Софтуерът за вградени системи* или накратко казано *вграден софтуер*, е компютърен софтуер, който управлява машини или устройства, които обикновенно не се считат за компютри. Специализиран е за конкретен хардуер, върху който се изпълнява. Въведение за вградения софтуер може да се намери в [R22].

Вградените системи продължават да навлизат все повече и повече в нашия живот, като растежа на разпространението им е огромен. Тези системи трябва да отговарят на нарастващ брой изисквания за функционалност, време за реагиране, ограничения към процесорно време и памет, консумация на енергия, цена и т.н. Същевременно стандартните подходи за компонентно базирана разработка на софтуерното инженерство като CORBA, DCOM/COM, Enterprise JavaBeans и т.н. не са подходящи за вградените системи, тъй като компонентите създадени от тях, не отговарят на голяма част от гореспоменатите изисквания. Oсвен това тъй като най-често езика за разработка е „C” става почти невъзможно да се прилагат и способностите на инструментите за моделиране, особено **извличане на моделна информация** и **генериране на код**, тъй като езика не разполага със силни механизми за изграждане на значими абстракции. За да могат да бъдат конкурентно способни тези системи трябва да могат лесно да се променят. Това налага разширяване на средата за дизайн и имплементация на тези системи с нови автоматизирани инструменти даващи възможност голям брой от изискванията към системата да бъдат проверени в ранен етап на разработката. Решение за това е използването на специфични компоненти за областта на приложение в съчетание със специфичен модел на разработваната система.

## Софтуерна архитектура

В този документ ще се придържаме към дефиницията на Бас, Клементс и Казман [R23, глава “*Архитектурни структури и изгледи*”] за архитектурни структури описващи софтуерна система със следните 3 групи, зависещи от типа на елементите, които представят:

* *Модулни структури* – Тук елементите са модули, които представляват единици за имплементация. Те разглеждат системата от гледна точка на формиране на изходния код. Отговарят за различни функционалности. С по-малко описание за това как въпросния код се държи по време на изпълнение.
* *Структури от компоненти-и-конектори* – Тук елементите са компоненти действащи в реално време (единици за изчисление) и конектори (средствата за комуникация между компонентите).
* *Структури на разпределение* – Показват връзката между софтуерните елементи и елементи в една или повече външни среди, върху които софтуера се създава и изпълнява. Т.е. дава отговор на въпроси като: На кой процесор се изпълнява даден софтуерен елемент? Къде се съхранява даден елемент по време на разработка, тестване и изграждане на системата? Какво е разпределението на софтуерните компоненти по различните екипи.

Предложеното решение в този документ се концентрира в автоматизирано извличане на изглед, който предлага *структура от компоненти-и-конектори* отговарящи на горното описание.

## Мотивация за решението

Възможността за извличането на архитектурна информация от изходния код на вградени системи ще даде възможност за автоматизиран анализ на свойствата на тази система, нещо което най-често се прави на ръка и дава възможност за пропуски, съответно и до дефекти на самата система. От друга страна реконструирането на архитектура от модел извлечен от вече валидирана и успешно посрещнала качествените си изисквания система, допринася до значително намаляване на дефектите на бъдещата система. Удобно би било да се създаде приложение за анализиране на вече съществуващи системи, което изготвя модел отговарящ на компонентите и интерфейса, по който си комуникират един с друг както и информация за системата в динамичен вид. По този начин се извлича информация за архитектурата на вече създадени системи, което значително улеснява дизайна на нови системи с подобни изисквания. Тъй като в момента UML (Unified Modeling Language) е стандартния за индустрията език за моделиране, за който съществуват множество инструменти и формати за представяне, удачно е той да бъде избран за представяне на архитектурните модели. Както и възползвайки се от предимствата на UML, ще се ускори разработката на нови системи и ще се подобри документацията и поддръжката както на системите в разработка, така и на тези, които са вече в продукция. Предизвикателството на създаването на такъв инструмент, е че повечето вградени приложения са разработени и продължават да се разработват на езикa за програмиране „C”. Това налага нестандартен подход към анализа на тези системи, тъй като за разлика от съвременните обектни езици за програмиране (като C++, C#, JAVA и т.н), езика „C” е слабо поддържан от инструментите за UML обработка и дизайн.

## Цел и задачи на дипломната работа

**Цел на дипломната работа:**

Да се разработи архитектурен инструмент, който извлича информация за интерфейсите на компонентите от техния код и представя еквивалентен UML модел със следните артефакти: класове, компоненти(зависимости м/у компоненти). Инструментът трябва да може да генерира базов код за нова система по даден модел.

**Задачи, произтичащи от целта:**

1. Да се направи проучване и анализ на други подобни инструменти и подходи
2. Да се изследват отворени формати за представяне на UML модели и да се избере най-подходящия за извършване на задачата.
3. Да се създаде проект на софтуерния инструмент за генерация на базов код на системата, които съдържа три основни модула: модул за генериране на UML, анализатор на софтуерен код, генератор на код.
4. Да се разработи модул за генериране на UML модели.
5. Да се разработи анализатор на софтуерен код.
6. Да се разработи генератор на базов код на системата по даден модел.

Горе-описаните задачи са развити посредством концептуален модел в точка *3.1* и в последствие са декомпозирани на потребителски изисквания в точка *3.3*.

## Очаквани ползи от реализацията

Целта на този документ е да предостави метод и решение за придържане към оригиналната структура на дадена система и опростяването **ѝ**, което допринася възможността за **еволюирането на програмата** с помощта на по-малко ресурси. Еволюция на програмата може да бъде корекция на грешки, подобряване на производителността или други атрибути, адаптиране на продукта към нова среда или добавяне на функционалност.

Предложеното решение дава възможности за:

* Поддържане на компонентен модел на софтуерна система (вкл. Вграден софтуер) писана на езика за програмиране “C”. Т.е. позволява внедряването на готови инструменти за моделиране и същевременно:
  + създава условия за създаване на верига от стандартни инструменти работещи с компонентия модел като например: *код генерация* (част от заданието), *инструменти за анализ* (спазване на времеви изисквания, използвана памет), *инструменти за преструктуриране, инструменти за симулация (стимулиране на входовете на системата с тестова платформа, както и софтуерна симулация на средата)*
* Бързо навлизане в софтуерната система посредством модел представящ я с високо ниво на абстракция
  + Чрез визуализация на модела разработчикът може лесно да проследи веригата на възникване на проблема и да се насочи към проблемния компонент и интерфейс.
  + Дава възможност за автоматизирано, бързо и структурирано търсене в модела.
* Възможност за модификация в самия модел и отразяване на промяната директно в кода чрез генерация на изходен код. Като както модификацията може да бъде отстраняване на даден дефект, така тя може да бъде добавяне или премахване на дадена функционалност.
* В комбинация горните 2 точки дават възможност за бърза, точна и сигурна промяна на софтуерната система с минимален риск.
* Възможност за лесна миграция на бизнес логиката, чрез трансформация на модела и код генерация към по съвременна (или различна) платформа.

## Структура на дипломната работа (0,5-1стр.)

# Реверсивен инжинеринг и възстановяване на дизайн

**Абстракт:**

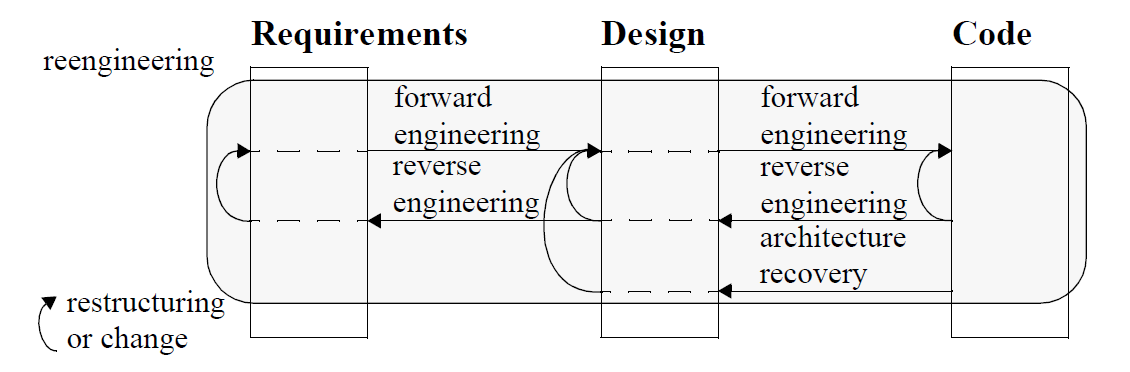
В тази глава задаваме основните предпоставки за възникване и дефиниция на реверсивния инжинеринг. Следва представяне на терминология от софтуерната архитектура във връзка с възстановяването **ѝ**. След като сме изложили предходните две можем да продължим с комбинирането им и прилагането им в реинженеринг на компонентно-базиран софтуер, който се крие в основата на заданието на дипломната работа, и как този подход подпомага съзряването и еволюцията на разработваните софтуерни системи. За да прилагаме този подход обаче ни е необходима среда, която да спомага извършването му, затова формализираме вече изложената теория във изграждащи блокове на система, която извършва архитектурна реконструкция. Всички горе-описани са изграждащи блокове за обособяване концепцията (*3.1*) на решението в тази дипломна работа, като допълнително съм използвал насоки от вече съществуващи подобни решения, които също са описани и анализирани в тази глава.

## Основни дефиниции

Секцията дава дефиниция на реинженеринга и дава връзката му с традиционното софтуерно инженерство. Излага също необходима терминология от софтуерните архитектири, която е необходима за прилагането реинженеринга.

### Терминология на реинженеринга и софтуерната подръжка

Следващите четери раздела съдържат стандартна терминология на реинженеринга. Определенията за реверсивен инженеринг, преструктуриране и реинженеринг са предложени от [R19]. *Фигура 1* изобразява връзките между тези термини.



Фигура (Връзка между термините)

#### Традиционен инженеринг (forward engineering)

Софтуерното инженерство първоначално е смятано за стремящо се към развиването на нови системи, въпреки че включва реверсивният инженеринг и реинженеринга. За да се избегнат подзначенията на термина софтуерен инженеринг, се въвежда терминът *традиционен инженеринг*. Това е процесът на преминаване от абстракции на високо ниво и логически, неимплементирани проекти към физическата имплементация на една система.

#### Реверсивен инженеринг

Реверсивният инженеринг има точно обратната цел на традиционния инженеринг. *Реверсивният инженеринг* е процес на анализ на една субектна система за

* идентификация на системните компоненти и техните взаимовръзки и
* създаване на репрезентации на системата под друга форма или на по-високо ниво на абстракция.

Важно е да се отбележи, че реверсивният инженеринг не включва промяна на субектната система, или създаване на нова, базирана на реверсивно-инженерната система. Това е процес на изследване, а не на промяна или копиране.

#### Преструктуриране (моделна трансформация)

Това е трансформацията от една репрезентативна форма към друга на същото ниво на абстракция, запазвайки външното поведение на системата (фунционалност и семантика). *Преструктурирането* често се използва като форма на превантивна мярка за подобряване на състоянието на системата следвайки даден предпочитан стандарт.

#### Реинженеринг

*Реинженеринг*, познат още като реновация и регенериране, е изследването и промяната на една система, за да се реконструира в нова форма и в следващата имплементация на новата форма. Реинженерингът по принцип включва някаква форма на реверсивен инженеринг (за да постигне по-абстрактно описание), следвана от форма на традиционен инженеринг или преструктуриране.

### Терминология в софтуерната архитектура

Все още има спорове относно определението за софтуерна архитектура, но повечето специалисти са на едно мнение, че тя трябва да включва поне *компоненти* и *съединителни звена*, както и тяхната йерархична декомпозиция [R24]. Компонентите са изчислителните части и съединители, които описват връзките между тези компоненти ([R17] стр. 5; [R18] стр. 5). Основни примери за компоненти са абстрактни типове данни, задачи за „клиент-сървър“ или интерфейси и услуги; примери за конектори са процедурните извиквания, споделени глобални променливи, канали или Unix контакти (sockets).

#### Възстановяване (recovery) на архитектурата

Това е дисциплина от реверсивния инженеринг, която има за цел възстановяване на софтуерната архитектура на дадена система.

#### Компоненти

Големите системи са съставени от подсистеми, които могат да се управляват поотделно. Тези подсистеми от своя страна са разделени на по-малки подсистеми. Най-малката декомпозиция е **модул**, който може да съдържа само функции, подпрограми и типови декларации, докато една **подсистема** е групиране на модули или подсистеми на по-ниско ниво. Подсистемите и модулите са **статични архитектурни компоненти**, които се различават по състав. **Динамичните архитектурни компоненти** са примери за изчислителни единици, създадени по време на работа; напр. съгласуващи се задачи или опашки. Тази теза се отнася единствено за статичните компоненти. И все пак разпознаването на статични компоненти често е предпоставка за откриване на динамични, тъй като последните често са само примери на статичните компоненти, като опашка Х, създадена в работен режим, която е пример за абстрактна *Опашка*, имплементирана от статичен компонент.

Добрият дизайн стига до декомпозиция, в която модулите, както и подсистемите получават висока свързаност на отговорностите (*cohesion*) и ниско функционално обвързване (*coupling*). **Свързаността на отговорностите** на модул е степента, която трябва да достигнат индивидуалните му компоненти, за да извършат дадена задача (*2.1.2.3* и [R15], стр.325). **Функционалното обвързване** (***coupling***)е степента на взаимна зависимост между модулите ([R16], стр.85).

Сзиперски ([R25], стр.548) например, предлага следната дефиниция за компонент:

***Софтуерен компонент*** *е композираща се единица договорено специфичен интерфейс и експлицитни зависимости към средата. Софтуерен компонент може да се разположи независимо и би могъл да очаства в композиция по инициатива от трета страна.*

Модулът в съвременните програмни езици е синтактична единица, която поддържа капсулирането. Съставена е от интерфейс на предоставените части и незадължителна скрита имплементация. Предоставените елементи са общи константи, променливи, подпрограми, потребителски типове и понякога – вложени модули.

При първия си дизайн е възможно модулите на една системна декомпозиция да покажат ниско ниво на свързване и висока кохезия, но по време на продължителна поддръжка, първоначалната декомпозиция може да се влоши. Например, функция F, която може да принадлежи на модул А, се слага в модул В. Така кохезията на модул В намалява, а свързването между A и В се повишава, тъй като F има нужда от подробности за имплементацията на А. Също така концепцията на А е делокализирана, защото отчасти е реализирана от В. Високото свързване и ниската кохезия затрудняват осъществяването на промени. Реинженерингът трябва да преструктурира системата така че съществената концепция на А да се имплементира единствено от модул А, за да се улесни бъдещата поддръжка.

Дискусията показва, че един реален модул невинаги съвпада с основната си концепция. Съществува отклонение между синтактичната и логическата единица. За да разграничим тези два типа единици, ще наречем последната **атомарен компонент**. Под „модул“ разбираме единствено синтактичната единица и така следваме типичната терминология на програмния език.

**Модулът** е синтактична единица, която се използва за групиране на обекти. Той е съставен от интерфейс и незадължителна имплементация. Обектите в интерфейса са достъпни за други модули; имплементацията е тайната на модула.

**Компонентът** е група свързани елементи с една обща цел, или концепция, важащи на архитектурно ниво. **Атомарен компонент** наричаме не-йерархически компонент, съставен от свързани общи константи, променливи, подпрограми и/или потребителски типове. За разлика от атомарния компонент, една **подсистема** е йерархичен компонент, съставен от свързани атомарни компоненти и/или подсистеми от по-ниско ниво.

Идеята за преструктуриране(*2.1.1.3*) на една система е да се реализира атомарен компонент от един модул. Един модул имплементира само един атомарен компонент за постигане на максимална кохезия и минимално функционално обвързване. На практика, степента на свързаност на отговорностите в един модул е различна. Следващият раздел описва това с подробности.

#### Свързаност на отговорностите (Кохезия) на модули

Йордън и Константин ([R16], стр. 108) изброяват следните степени на свързаност на отговорностите:

* **Функционални:** модулът извършва само една определена функция.
* **Последователни:** модулът извършва повече от една функция, но те се появяват в ред, определен от спецификацията.
* **Комуникационни:** модулът извършва няколко функции, но върху едно изложение на данни (неорганизирано като един тип или структура).
* **Процедурни:** модулът извършва няколко функции и те са свързани само с основна процедура, засегната от софтуера.
* **Времеви:** модулът извършва повече от една функции и те са свързани само от факта, че трябва да се извършат в един период от време.
* **Логически:** модулът извършва повече от една функции и те са свързани само логически.
* **Случайни:** модулът извършва няколко функции, които не са свързани помежду си.

Тези категории са изброени от най-желателни (функционални) до най-малко желателни. Класификацията е установена в края на 70-те, когато преобладава функционалната парадигма, а структурният дизайн е най-честият такъв метод. Скорошните тенденции към обектно-ориентираната парадигма – и оттук към езиците и методите, които подкрепят абстрактни данни от модули – на пръв поглед сякаш си противоречи с традиционните представи. Един модул, базиран на абстрактни данни може да има няколко различни функции; но всички те са свързани в смисъла, че характеризират типа абстрактни данни, или по-общо казано: атомарният компонент.

## Реинженеринг на компонентно-базиран софтуер и среда за изпълнението му

След изложените терминология на реинженеринга и софтуерните архитектури е време да съчетаем двете и да разгледаме възможността на прилагането на реинженеринг в компонентно-базиран софтуер. Също така представяме ползите от съчетанието на двата подхода, както и възможността за прилагане на реинженеринг за трансформиране (миграция) на традиционно разработен софтуер към копмонентно-базиран такъв. Съчетаването на всички тези подходи обаче е най-добре да стане в среда, която да спомага (автоматизира в голяма степен) извършването му, затова предлагаме необходимите изграждащи блокове на система, която извършва архитектурна реконструкция.

### Реинженеринг и компонентно-базиран софтуер

Компонентната технология осигурява известна гъвкавост, която не може да бъде постигната от която и да е друга традиционна технология. Някои големи компании вече са разработили големи и успешни компонентно-базирани софтуерни продукти. Много други са в процес на усвояване на тази технология.

#### RE като помощник в съзряването и еволюцията на CB софтуера

Приемаме, че вече е достъпно едно прилично количество компонентно-базиран (CB) софтуер. Съответните RE трансформации са вътрешни за CB вълната. Например, реверсивният инженеринг започва с имплементацията на компонентно-базиран софтуерен продукт и опитва да възстанови компонентно-базираните концепции. Приложимостта и сложността на всички реверсивно-инженерни трансофрмации до голяма степен зависят от (1) въпросния компонентен модел и (2) информацията, която ще бъде извлечена.

Някои факти са тривиални за извличане, когато компонентната инфраструктура осигурява съоръжения за самонаблюдение. Други факти са много по-сложни и изискват RE техники. Нашият опит говори, че възстановяването на информация за компонентната вътрешна структура обикновено е възможно.

Възстановяването на връзки и следователно на цялостната топология, би могло да бъде тривиална задача, ако връзките се материализират (externalized), или сложна задача, ако връзките се поставят дълбоко в кода (напр. COM, JavaBeans, EJB). Заради полиморфизма и други късни свързващи механизми, това не може да бъде постигнато само с анализ на изходния код. Често архитектурата става известна едва след като се зареди. Още повече, че архитектурата може да се развие динамично по време на изпълнението. Във всички случаи е ясно, че за да са полезни RE техниките за CB софтуера, трябва да се базират и на статична, и на динамична информация. Статичната информация трябва да бъде извлечена от широк кръг източници, включително изходен код, но и конфигурационни файлове, дескриптори и т.н. Добрата новина е, че получаването на информация по време на работа на компонентно ниво може да е лесна задача, когато се осигурят съоръжения за наблюдение на компонентната активност.

Когато проблемът с извличането на информация се реши, почти всички техники в RE могат да бъдат разгледани и адаптирани към контекста на компонентно-базирания софтуер. Преструктурираните компоненти скоро ще се превърнат в проблем, тъй като днес няма ясна представа за това какво е „добър“ компонент. Компонентно-базираните метрики трябва да бъдат определени, оценени и т.н.

Друга интересна гледна точка е, че не е нужен само заради еволюцията на софтуера, но и за еволюцията на „езика“ (компонентния модел), използван за написването му. След известно време един голям компонентно-базиран софтуер може да съдържа части от код, написан с различни версии на компонентния модел. Подръжката и локализацията на остарялите конструкции и смяната им с по-нови, може да се окаже много важно, особено когато големите издания покажат несъвместимост.

#### RE като помощник за миграцията към компонентно-базирана технология и интеграция с традиционните технологии

Трансформациите, описани по-горе, имат смисъл само за компании, които вече разполагат с компонентно-базиран софтуер. Други възможни RE трансформации започват от традиционен софтуер и създават компонентно-базирани такъв. В този случай, голяма част от извършената работа по традиционния софтуер може да бъде използвана отново след известни настройки, тъй като се променя единствено целта на трансформацията. Например в последните години се работи много усилено в посока откриване/възстановяване на „компоненти“. Терминът „компонент“ тук се използва в широкия си смисъл, но тези функционални единици могат да се застъпят от компоненти, както е определено от компонентните модели.

Добрата новина е, че някои компонентни технологии като CCM директно осигурява съоръжения за обвиване на наследен код. Например един CCM компонент може да бъде разделен на много различни „сегменти“ и „изпълнители“, които могат да бъдат приложени в използваните езици. Всъщност не става ясно дали компонентът, открит по този начин ще съставлява „добър“ компонент от гледна точка на компонентно базирания модел. Във всеки случай, ще бъдат нужни допълнителни усилия, за да се оползотвори пълния потенциал на компонентната технология. От една страна, компонентният модел осигурява по-богати комуникационни протоколи, например събитийна комуникация. От друга страна, подобни компонентни инфраструктури от EJB и CCM осигуряват услуги като управление на транзакции, постоянство и др.

### Среда за архитектурна реконструкция

Описанието на софтуерна архитектура трябва да предаде основни решения, взети по време на разработването на дизайна на системата. Ран **[R9]** ни насочва че основните решения за дизайна са тези, които които ще е скъпо да променим, следователно и най-критични за разработката и поддръжката на системата. Има четери категории дизайнерски решения: *концепции, архитектурно значими изисквания, структура и текстура*. Концепциите се отнасят до начина, по който мислим за системата (напр. В операционна система могат да се използват концепции като задачи , процеси, опашки и т.н.). Решенията за системаните концепции са вероятно най-важните и трудно могат да се променят в следващите стадии на разработка. Архитектурно важни изисквания са основните проблеми, които трябва да се адресират с подходяща софтуерната архитектура. Те трябва да се фокусират върху критичните характеристики, които искаме да постигнем със системата. Структурата описва декомпозицията на системата в взаимозависими компоненти и техните зависимости на правилно ниво на абстракция. Текстурата се отнася до дизайнерските решения, които са влиаят имплементационно ниво и са архитектурно зависими (дизайнерски шаблони, политики). Според **[R9]** дефинираме софтуерната архитектура като *„множество от концепции и диазйнерски решения за структурата и текстурата на софтуера, които трябва да направим преди съответната разработка за да позволим ефективно задоволяване на архитектурно значими, експлицитни функционални и качествени изисквания, както и имплицитни такива на даден проблем и конкретните домейни на приложение“.*

Архитектурната реконструкция (или реверсивна архитектура) засяга задачата за възстановяване на дизайнерски решения, които са били взети по време на разработване на системата. Те се състоят от решения, които са били изгубени (тъй като не са били документирани или разработчика е напуснал) или са непознати (например, допускания, които не са били взети в предвид първоначално). Целта е да се хвърли светлина във всички категории на дизайнерските решения, които са свързани с описанието на софтуерната архитектура. Реконструкцията се изпълнява на базата на изучаване на наличните артефакти (документация, изходен код, експерти) и посредством извличане нова архитектурна информация, която не е била очевидна първоначално. Подхода може да се обобщи със следния четери-стъпков итеративен процес **[R10]**:

#### Дефиниране на архитектурните концепции:

Целта на тази фаза е да се възстанови и изясни архитектурно значимите концепции, които изграждат системата. Тези концепци представят начина, по който разработчиците мислят за системата и те трябва да станат терминология на процеса по реконструкция. Те представляват изграждащите блокове на системата и комуникационната инфраструктура, която позволява на компонентите да комуникират по време на изпълнение. Тези концепции би трябвало да са видими в референтия архитектурен документ в противен случай трябва да се извлекат посредством реверсивен инженеринг. Стъпката също трябва да идентифицира начина, по който архитектурните концепции са свързани с имплементацията. В разпределените системи, архитектурните концепции могат да бъдат апликации, сървъри, софтуерни транспортни среди, докато в операционните системи могат да бъдат задачи, процеси, опашки, споделени памети и т.н. Текстурите също биха могли да се причислят към тази фаза. Например дизайнерски шаблон може да крие модел на взаимодействие, който е архитектурно значим.

#### Събиране на данни

Тази фаза събира информация описваща софтуерната архитектура на системата. Създаваме модел на системата, чиито части са инстанции на концепциите идентифицирани във фаза 1. Правилен избор на концепции ще подсигури, че моделът e съставен от единици на правилното ниво на абстракция. Тази фаза основно се грижи за събирането на информация от колкото за обосноваване на архитектурата. Така, че тази задача може лесно да се автоматизира с подходящи иснтрументи

Различни източници на информация са въвлечени в този процес. Изходния код от една страна за статичен анализ и симулация за динамичен анализ. Освен отва документация, софтуерни диаграми (например съхранени в CASE инструменти), експерти също могат да допринесат за създаване и допълване на модела.

#### Абстракция

Модела от предходната фаза е обикновенно с много ниско ниво на абстракция. Целта на тази фаза е да обогати модела с абстракция подхождаща на дадения домейн, което ще допринесе за по високо ниво на изглед на системата. Познати абстракции могат лесно да се добавят към системата. Непознатите абстракции трябва да бъдат идентифицирани от архитект, категоризирани, наименовани и след това нанесени в модела. Тази дейност се извършва ръчно от архитект и след това се фиксира в правила за абстракция. Процеса на абстракция трябва също така да произведе архитектурни изгледи, които ще се представят в последната фаза.

#### Презентация

Архитектите трябва да презентират реконструираната архитектура в различни изгледи. Тези изгледи могатда бъдат: логически, процесни, физически и свързани с разработката, като форматите могат да бъдат различни.

## Съществуващи инструменти за реинженеринг

Следват две предложения за инструменти, които изпълняват повечето от изискванията на заданието на тази дипломна работа. Първото предложение си поставя за цел да извелче архитектурна информация за анализираното приложение на базата на домейна на употреба и съществуваща документация, включително тестови сценарии. Докато втория инструмент е базиран на използването на мета-модел, описващ компонентен модел за разработка на софтуер. Въпросният инструмент може реверсивно да анализира единствено и само софтуер създаден с въпросния компонентен модел, но пък с много голяма точност, освен това използването на мета-модела дава възможност за лесно и евтино разработване редица инструменти за визуализация, анализ и преструктуриране.

### Оркестрирана много-изгледна среда за софтуерно архитектурна реконструкция [R14]

#### Предложения за много-изгледен анализ

Предложеното решение се състои от много-изгледен модел и много-изгледен процес (*Фигура 2* и съответно *Фигура 3*). Много-изгледният модел представя връзките между трите изгледа *дизайн, поведение* и *структура (Design view, Behavior view, Structure View:* Фигура 2*)* в една диаграма на класовете, където сценариите са основните елементи за извличане и съвместна работа на трите изгледа. Много-изгледният процес на *Фигура 3* показва цялостния механизъм за извличане на три изгледа на софтуерната система. По време на процеса се генерират сценарии с помощта на доказателства, изведени от знанията на потребителя относно домейна на приложението, взаимодействието между система и потребител, системните документи от високо ниво (които са налице) и потребителските ръководства. Структурата на сценариите трябва да съответства на обикновен синтаксис на израз. Структурираните сценарии се анализират, за да се генерира изглед за проектиране на софтуерната система, представлявана от 2 типа диаграми: *диаграма на същност-връзка* (E-R) и *диаграма на дейностите*. Тези диаграми представляват имплементираната функционалност и главната системна информация, които се манипулират от дейностите. За възстановяване на изглед на режима на работа, потребителят изследва изгледа за проектиране и избира определени свойства, които да се използват от възстановителния процес. За всяко специфично свойство се определят сценарии, всеки от които притежава въпросното свойство. Изпълнението на тези сценарии на инструментирана софтуерна система генерира отпечатъци, които след обработка ще представляват *шаблони на изпълнение*. Всеки такъв шаблон е поредица от изходен код извиквания на функции, които са често срещани във всички сценарии.



Фигура 2 (Примерен много-изгледен модел [диаграма на класовете], представящ връзките между три изгледа [R14, стр. 3])



*Фигура 3 (Примерен много-изгледен процес за извличане на 3 изгледа от една софтуерна система [R14, стр. 3])*

Повтарянето на този процес за колекция от свойства ни позволява да определим реализацията на софтуерните свойства в изходния код, а именно основните функции. Накрая всяка група основни функции, имплементиращи дадено свойство ще бъдат използвани като ядро на клъстер при възстановяването на структурния изглед, за да може да се създаде по-голяма група свързани функции. Структурният изглед използва връзките между функциите, за да определи близостта на други функции с ядрото на всeки клъстер, което в последствие генерира групи, представляващи софтуерните компоненти.

Цялостният процес в много-изгледна рамката ни позволява да свързваме диаграми с абстрактен дизайн към конкретната имплементация на функционалните елементи на изгледа за проектиране.

#### Генериране на изглед за проектиране (Design view, *Фигура 2*)

В този раздел се обсъждат стъпките за трансформиране на знанията в текста на сценариите в информация, свързана с проектирането и представлявана от два типа диаграми: *диаграма „същност-връзка*“ и *диаграма на дейностите*, използвайки процеса, изобразен на *Фигура 5*. Този подход генерира и структурира редица сценарии, след което използва сценариен домейн модел, за да превърне изградените сценарии в съставки на усвоените проектни диаграми. Този процес се състои от три стъпки:

**Стъпка 1 (генериране на сценарий)**

Сценариите са основните елементи, които организират предложената много-изгледна възстановителна рамка на архитектурата. Приемаме структурирана текстова репрезентация на сценарии, която съответства с обикновения синтаксис на *Фигура 4*. В този синтаксис всеки сценарий се състои от поредица от един или повече *участници (actors), действия* и *работна информация*, всеки от които могат да имат нула или повече ограничения, които ще бъдат определени в *стъпка 2*. Можем да генерираме синтактично правилни сценарии, които ще бъдат декомпозирани, с помощта на домейн моделът, за да се запълни базата знания на сценарийни шаблони и да се използват отново бизнес правилата в подобен случай. Източниците за генериране на сценарий са доказателства като: потребителски интерфейс, ръководство за потребители и знанията на потребител-специалист за системата.

**Стъпка 2 (декомпозиция на сценарий)** Диаграмата на класовете на предложения сценариен модел е представена на *Фигура 6*. Този модел трябва да осигури класова информация в сценарии от различни области на приложение. След прилагане този модел над три системи, включително система обслужваща ресторанти за бързо хранене, инструмент за рисуване Xfig и софтуер на банкомат. Текстът на структурираните сценарии се анализира с помощта на този домейн модел и получените инстанции на класове в модела се записват в обектната база. Схемата на тази база има елемент за всеки клас на домейн модела, както и индексен запис като основен ключ.

Както показва *Фигура 6*, в този модел всеки пример за сценариен клас се състои от един или повече примери за *участник, работна информация* и *класове на изпълнението,* както и нула или повече примера за подчинени класове и ограничителни класове.



Фигура (Синтаксис на регулярен израз за генериране на сценарий, където “+” и “0..М” [“1..N”] представляват композиция [R14, стр. 4])



Фигура (Генерация на изглед на дизайна базиран на задачния сценарий и сценарий базиран на домейна на модела [R14, стр. 4])



Фигура (Сценарий за домейн базиран модел, който се използва за сканиране на сценарий и попълни обектната база [R14, стр. 4])

Отдолу са представени класовете на предложения сценариен модел:

* *Тип потребител/участник (актьор):* това е „човек“, „система“ или „системен компонент“, който общува с другите потребители по време на изпълнение на сценарии.
* *Действие:* действието, извършвано от потребителя по време на изпълнението на сценариите. По принцип действието манипулира пример от Работната информация. Действията могат да бъдат 3 типа: *въвеждане (input), вътрешно действие (internal), извеждане (output),* базирани върху обхвата на работната информация.
* *Работна информация:* това е информацията, която се обработва (разменя, пренася, съобщава, запазва в системата и т.н.) от потребителя по време на изпълнение на сценария.
* *Зависимост:* това е връзката между два примерни класа *Потребител, Действие, Работна информация.* По време на реализиране на сценария, зависимостите се установяват между новогенерираните примери на домейн модел класове (отговарящи на въпросния сценарий) и между новогенерираните и старите примери в базата. Зависимостта може да е *информационна зависимост* или *завимисост на действието.*

Предложеният домейн модел на *Фигура 6* включва *класово ограничение.* То се състои от информация за ограниченията, които могат да се свържат със случаи от всеки подклас на *Информация, Действие, Зависимост.* Примери за тези ограничения включват: *капацитет, обхват на стойност, поредно число, време, привилегия* и др.

**Стъпка 3 (генериране на дизайн)** В тази стъпка диаграмите „същност-връзка“ и диаграмите на дейностите се генерират със случаите на класовете на домейн модела, запазени в обектната база.

* *Диаграма „Същност-връзка“.* Примерите на класове *Потребител* и *Работна информация* в базата са кандидат-същности в отговарящите им атрибути в диаграмата „Същност-връзка“. А примерите на различни подкласове на зависимост на данните са кандидат-връзки, свързващи различни същности и определящи атрибути на тези същности.
* *Диаграма на дейностите.* Примерите за класове *действие* са кандидат-дейности в диаграмата на дейностите. А примерите за различни подкласове на *Action dependency* свързват други елементи от диаграмата на дейностите като: периферия, ромб, съединения и разклонения.



Фигура (Възтановка на поведенчески изглед базиран на шаблони на изпълнение използван да идентифицира функционалности на дадена част от кода [R14, стр. 5])



Фигура (Колекция от 7 опитини изпълненя. Различните типове оцветени полета отговарят на три различни шаблона на изпълнение [R14, стр. 5])

Предложеното генериране на изгледа за проектиране от сценарии е систематичен подход за превръщане на неформалната информация от сценарии в добре оформени диаграм и. В следващия раздел ще е показано как диаграмите за генериране на изгледа за проектиране се прилагат за осигуряване на свойства и сценарии, използвани за възстановяване на работния режим в система та.

#### Възстановяване на поведението (Behavior view, *Фигура 2*)

*Фигура 7* показва стъпките за възстановяване поведението на една софтуерна система като начин за идентифициране реализацията на софтуерните свойства от функциите на изходния код. Стъпките са:

**Стъпка 1**  Както се споменава в точка 2.3.1.1, важните свойства на една софтуерна система се идентифицират като резултат от процеса на генериране на дизайна. Диаграмата на дейностите ни помага да определим редица сценарии, коит о изследват едно софтуерно свойство. Наричаме ги поредица от сценарии със специфични свойства. Например, в случая на софтуерна система за рисуване, група сценарии, които споделят операцията „премести“, за да преместят начертана фигура на компютърния екран ще създаде подобна сценарийна система със специфични свойства. Софтуерната система трябва да изработи *следи на изпълнение*, при задействане на сценарий в системата. Основна пречка в анализа на една система е големият размер на следите на изпълнение, които затрудняват анализа. За да може да се работи с големия размер на следите, филтрираме всички излишни функции.

В останалата част на раздела, ще опишем прилагането на техниката на извличане на последователни шаблони, за да намерим групи функции в *следи на изпълнение*, отговарящи на определени системни черти. Въпросната техника се използва за извличане на често срещани шаблони сред поредиците транзакции. В тази книга използваме гореспоменатата техника, за да извлечем често появяващи се функции сред следите на изпълнение на програмата. Извлеченият шаблон се състои единствено от прилежащи части на следите на изпълнение. Тази характеристика създава смислени шаблони на изпълнение, които отговарят на основните функции, имплементиращи специфични функционални свойства на системата. Чрез предложения подход различаваме два типа шаблони:

* *Общи:* Общият шаблон съществува в по-голямата част от отличителните сценарии, изпълнявани в системата. За да извлечем такъв шаблон, трябва да използваме филтриращия механизъм, за да изключим отличителните шаблони от тази група. Пример е шаблонът на функционалните случаи, създаден от инициализиращия компонент от всяко изпълнение на програма.
* *Отличителни (feature-specific):* Всеки шаблон в тази категория отговаря на основните функции, които имплементират дадена функционалност от отличителния сценарий. Подобен шаблон съществува в повечето части на един отличителен сценарий. Както вече споменахме общите шаблони се извличат заедно с отличителните шаблони. Отделянето на тези два типа шаблони се обсъжда по-нататък в раздела.

*Фигура 8* показва колекция от седем опитини изпълненя и отговарящите извлечени изпълнителни шаблони, включващи и отличителни, и общи шаблони.



Фигура (Извличане на структурен изглед базиран на максимална асоциативност и клъстерна оптимизация [R14, стр. 6])

**Стъпка 2** (анализ на шаблони на изпълнение). Фокусираме се върху шаблоните на изпълнение, които отговарят на определени черти на сценария. В този контекст, разработваме структурните характеристики на концептуалната мрежа, за да отделим функциите, характерни за една специфична функционалност от групата функции, които имплементират базови функционалности. В контекста на концептуалната мрежа, атрибутите, които се споделят сред повечето обекти, се появяват в горната част на мрежата и обратно. В нашата концепция за анализ на мрежата обектът е черта от отличителния сценарий, а атрибутът е фунция, участваща в изпълнителните шаблони на същия сценарий. Тъй като общите функции се изпълняват почти във всички отличителни сценарии, те се появяват в горната част на мрежата. От друга страна, функциите, които са характерни само за определени черти/свойства на софтуера, се намират в долната част на мрежата. В резултат на това се идентифицират основните функции, които имплементират само някои свойства на системата.

В допълнение към основното приложение на примерното възстановяване на поведението, като идентифициране на софтуерната имплементация в изходния код, този подход е използван за: 1) измерване на свързаността на функционалните свойства сред структурните модули; 2) оценяване на структурната свързаност на софтуерните модули и 3) визуализиране на функционалното разпределение на специфични черти в графична решетка. В следващия раздел ще обсъдим как да използваме резултата от изгледа на поведението, за да осигурим семантика на възстановяването на структурния изглед на системата.

#### Възстановяване на структурен изглед (Structure view, *Фигура 2*)

*Фигура 9* показва стъпките за възстановяване на структурния изглед на софтуерната система, който генерира свързани софтуерни модули от функции на изходния код. Процесът се състои от 2 основни стъпки: *извличане на факти* и *реконструкция на модула.*

**Стъпка 1** (*извличане на факти)* В тази стъпка софтуерната система генерира абстрактно синтактично дърво (AST), което съдържа всички структури, отговарящи на програмния език на софтуерната система. Например използвайки графична схема, която дефинира критерии на архитектурното ниво за възстановяване на софтуерни модули и техните връзки, създаваме графична репрезентация на софтуерната система и я запазваме във базата от факти. Имайки предвид броя критерии в една средно голяма софтуерна система (обикновено повече от 1000), търсенето на източник на графиката е сериозен проблем. Оттук трябва да ограничиим търсещия домейн за всеки модул към група приемливи критерии. Прилагаме алгоритми за разработване на данни, базирани на асоциации върху графиката в по-малки области ( домейни за изследване), където всеки такъв домейн се състои от няколко критерия, които са свързани със критерий от този домейн, така нареченото *основно звено*. След това ограничаваме търсенето на всеки модул към един или повече от тези области.

**Стъпка 2** *(реконструкция на модула)* Извършваме контролирана оптимизация на групирането ( клъстерна техника), която итеративно генерира софтуерни клъстери, като свързани модули от функции, свързани помежду си чрез импортирани и експортирани функции. Всеки модул се състои от едно или повече *основни* *звена*, докато основните функции на модула и под- оптимална версия на A\* търсещия алгоритъм се използват за събиране на групата от високо асоциирани функции в един модул. Пространството на търсене з а модула е ограничено до функциите в домейните на търсене на съответното основно звено. Определяме метрическа система на сходства, базираща се на група същности с максимална асоциация на свойствата. Тя се определя във фор мата на максимална поредица от критерии, които имат еднакви връзки с всеки член на друга максимална поредица от критерии. Оттук тези групи притежават по-голяма свързаност и са подходящи за създаване на клъстери, базирани на метриката за подобие на свързани критерии. Тя кодира структурното свойство на групите на максимално асоциираните същности.

Структурното възстановяване без съдействието на предложените много-изгледи, разчита на улеснения чрез инструменти, които генерират списък с най-високо квалифицираните *основни звена* за следващото възстановяване. Докато този метод създава свързани модули от функции, не може да създаде смислени такива, тъй като основните функции се избират за основата на статични структурни свойства, а не на функционалността на основните звена. Така предложеният метод осигурява основни функции като модулни основни звена, които имплементират смислени софтуерни черти. Тези софтуерни черти се извличат от диаграми на дизайна, произлизащи от функционалните изисквания на софтуера. На *Фигура 9* основните функции от изгледа на поведение се използват, за да се създадат семантично смислени групи като системни компоненти.

Клъстерните техники могат да се класифицират като „автоматични или контролирани“ и „ориентирани към потребителя“. Те се опитват да преструктурират оригиналната система в нова, като така тя бива съставена от висококачествени модули от гледна точка на високата свързаност и ниското функционално обвързване. Дадена клъстерна техника трябва да отгоавря на определени изисквания в новите среди за софтуерно архитектурно възстановяване, включително: участие на потребителя, за да направлява процеса, според знания за домейна/документа; постепенно възстановяване, позволяващо частично системно възстановяване; декомпозиция на йерархията, която да се справи със сложните проблеми, присъстващи в анализа на големи системи. Клъстерната техника използвана в тази точка служи като ефективен помощник на потребителя, осигурявайки адекватно ниво на автоматизация и полезна информация, за да може потребителят да изследва и контролира възстановителния процес.

### Компонентният модел на Dassault Systèmes (DS) [R14]

Този компонентен модел накратко отговаря на почти всички условия от заданието. Тези, които не покрива са: че езика, който се анализира и генерира не е “C”, a “C++”, както и това, че е предназначен за разработването на мултиплатформена 3-измерна CAD/CAM/CAE система (CATIA), а не вграден софтуер. Съответно запознаването с използваните концепции, подходи и инструменти, както и анализирането им би било от ползва при създаването на подобна система със сходни изисквания.

Разработването на CATIA V5 започва в средата на 90-те. DS установяват, че обектно ориентираната технологията има сериозни ограничения и че C++ не отговаря на всички изисквания за изграждане на тяхната система. Двата най-важни аспекта са следните:

* **паралелно проектиране.** С++ обектите са прекалено близко свързани: дори една минимална промяна може да предизвика голям брой прекомпилирания. За продуктите на големите компании и ограниченията при паралелното проектиране, това е голям проблем.
* **възможности за разширяване.** Основните клиенти и партньори на CATIA трябва да могат да разширят DS компонентите със собствен код, дори без наличието на изходния код.

За да се решат тези (и други) проблеми, DS създават компонентен модел, заемайки идеи от COM, Corba и Java. Следва кратко и неформално описание на „Обектен редактор“ (ОР). Въпреки името си, ОР се разглежда най-успешно като компонентен модел.

В следващите подточки ще обърнем внимание на:

* **Концептуално ниво** – обяснява от какви части (*интерфейс, имплементация, разширения*) е съставен един компонент от компонентния модел
* **Ниво на реализация** – начинът, по който DS са избрали да осъществяват имплементационната част от компонентния модел
* **Често срещани проблеми** – основните проблеми, с които са се сблъскали DS, изграждайки големи софтуерни продукти използвайки този техния компонентен модел. Също така и метода, който са използвали за справяне с тези проблеми.
* **Създаване на инструменти за реверсивен инженеринг** – въвеждането на мета-модел описващ компонентния им модел. Подобен модел е дефиниран (*3.3.2*) и използван (*6.1.3*) в текущата дипломна работа.

#### Концептуално ниво

ОР компонентите са части от код, които могат да бъдат манипулирани чрез използването на интерфейс. Интерфейсите могат да бъдат разгледани като абстрактни проксита за реални обекти, които получават клиентски заявки и ги препращат към компонента, реализиращ интерфейса. Концепцията за интерфейса помага в подхода към проблема на паралелното проектиране, тъй като отделя интерфейсните клиенти от промените на компонентната имплементация.

По-точно казано, един компонент е съставен от установени елементарни частици код, наречени *имплементации* (реализирни от С++ клас). Една от тези имплементации е основата (на компонента). Други имплементации, наречени „разширения“, могат по-късно да бъдат добавени към основата, за да разширят компонента. Важно е да се отбележи, че разширенията се отнасят към основата, но основата игнорира факта, че се разширява. Това позволява добавянето на ново разширение в по-късен етап, без да има нужда от прекомпилиране на основата, или на някое от останалите разширения.

#### Ниво на реализация

Всички концепции на ОР се имплементират посредством С++ единици. Например, и интерфейсите, и реализациите се представляват от С++ класове. Всъщност нивото на реализация е много по-сложно, тъй като съответсвието не е едно към едно, т.е.: реализацията на една ОМ единица може да създаде много С++ единици.

За да се контролира софтуерът, ОР концепциите се превеждат в С++ код, с помощта на шаблони. Този подход много наподобява подходите на други компонентни модели (пример [R4]). В случая на ОР се добавя допълнителна информация в изходния код чрез макрота (macros). Това облекчава постоянното писане на различни кодове. Някои от тях също се създават автоматично.

#### Често срещани проблеми

ОР успешно се използва за изграждането на големи софтуерни продукти (стотици приложения, съставени от хиляди компоненти, разработени от стотици софтуерни инженери). Появяват се няколко проблема:

* **Нужда от концептуален изглед.** Софтуерните инженери описват компоненти, използващи нискокачествени механизми на реализационно ниво (например макро). ОР концептуалните единици са смесени с огромно количество С++ код.
* **Нужда от централизирано описание.** Информацията за една ОР единица често се разпределя сред много различни файлове, включително и изходен код и речници.
* **Нужда от формализация.** ОР компонентният модел неформално се дефинира чрез огромна документация. Независимо от това, че е много ценна, тази документация често се оказва неточна, а много от реализационните ограничения са зле документирани. Освен това, тъй като реализационните техники се развиват с времето, за да се осигури постоянно подобрение, най-точната информация идва само от опитни софтуерни инженери.
* **Нужда от специализирани инструменти.** Софтуерните инженери разработват и поддържат компоненти, използвайки традиционни С++ инструменти. Въпреки че са подходящи за изпълнение на повечето задачи, тези инструменти не могат например да разберат софтуерния режим на работа на концептуално ниво. DS също разработват различни инструменти, които да се справят с определени проблеми, но възможностите им са ограничени.

И наистина ОР моделът, както други компонентни модели (COM, CCM и др.), е труден за научаване и разбиране. Опитните софтуерни инженери се учат как да изграждат компоненти, но често се затрудняват при установяване на проблема, когато софтуерът, който са разработили, не демонстрира очаквания режим на работа.

Липсва ясна картина на цялостната компонентна структура на концептуално ниво. Реализационното ниво е налице, но съдържа прекалено много технически детайли. Реверсивният инженеринг осигурява логически подход към тези проблеми, тъй като целта му е да *„създаде репрезентации на системата под друга форма, или на по-високо абстрактно ниво“*. И все пак, докато повечето техники на реверсивното инженерство се занимават с традиционни и ясни концепции, проблемът тук е да се справим с реверсивното инженерство на компонентно-базирани софтуерни системи, което е сравнително нов проблем.

#### Мета-Моделът

Едно от средствата за решаване на гореописаните проблеми е дефинирането на мета-модел, който да може да описва всяка концепция на OM като обектно-ориентиран елемент с UML нотация.

Основни характеристики на дефинирания мета-модел са:

* Описва компонентите като “черни кутии”
* Описва елементите на компонентите по отделно
* Свързва елементите на компонентите
* Дава възможност за откриване на потенциални несъответствия

#### Създаване на инструменти за реверсивен инженеринг

Създаването на мета-модел, не само улеснява разбирането на компонентния модел, но той дава и добра основа за разработването на платформа за реверсивен инженеринг, върху която могат да се изградят и кооперират голяма верига от инструменти вариращи от визуализиращи такива до инструменти за анализ и преструктуриране (виж *2.1.1.3* ). Разработката на тези инструменти от нулата определено не е рентабилна. За щастие, от мета-модела може да бъде генерирана обща платформа.

##### Платформа за реверсивен инженеринг

*Фигура 10* показва опростен изглед на цялостната архитектура на платформата. Тази традиционна архитектура за подобна среда на реверсивен инженеринг е съставена от следните части:

* **Екстрактори** (*Extractors*)**:** Първата стъпка е да се извлече информация от конкретни софтуерни артефакти. В нашия случай се анализират изходният код и речниците.
* **Хранилище** (*Repository*): Хранилището играе централна роля в средата. Една от важните черти на нашия подход е че структурата на това хранилище се извлича директно от мета-модела.
* **Инструменти** (*Tools*)**:** Инструментите генерират различни изгледи на хранлището. Докато някои инструменти генерират определени такива, общите инструменти използват спецификация на изгледа, който ще бъде генериран. Както ще видим, мета-моделът може да бъде използван директно, за да изрази информацията, която ще се покаже.



Фигура (DS платформа за реверсивен инженеринг)

Подобна е и концепцията на решението от дипломната работа, която може да се разгледа подробно в точка *3.2.*

##### Пример за инструменти за визуализация

Показването на компоненти бе първото приложение на нашата платформа за реверсивен инженеринг. Беше доста интересен експеримент, тъй като компонентите са изградени чрез използването на макрота и други механизми от ниско ниво, разгърнати над голям брой файлове. Софтуерните инженери така и не са „виждали“ тези компоненти.

##### Визуализиране на компоненти с универсален инструмент.

*Фигура 11* показва вътрешният изглед на компонент, показан чрез **GSEE** (Среда за изследване на универсален софтуер). Компонентът е представен като дърво от лявата част на прозореца и като диаграма отдясно.

Самата GSEE е универсална среда. Всички елементи в мета-модела могат да бъдат използвани. Може да се извлече и допълнителна информация, благодарение на команден език, който е близък по функционалност до OCL. Основният плюс на инструмента GSEE е, че прави възможно показването на каквато и да е информация, присъстваща в repository-то почти без усилия.



Фигура (основен изглед на GSEE)

## Сравнителен анализ на изложените решения

В тази точка ще бъдат предложени критерии за анализ на изложените съществуващи подобни решения на заданието. Както и сравнителен анализ на базата на предложените критерии.

### Критерии

#### Функционалност

Това е основния критерий, по който ще сравняваме изложените решения. Начина по който те възстановяват архитектурна информация на базата на изходен код се различава. Основно инструментите за анализ изпълняват следните основни функционалности:

* **Структурен анализ** – възможността на иснтрумента да извлича компоненти и връзки между компоненти (конектори), както и статични свойства на самите елементи (локални типове данни, променливи, константи и т.н.)
* **Динамичен анализ** – способността на инструмента да извлича динамичен изглед, интеракция между статичните елементи. Това включва детайлно описание изпълнението на отделните задачи и/или процеси заложени в анализирания код. Възможни са два подхода при извършването на този анализ:
  + *Статичен анализ на кода* – посредством обхождане на всички артефакти на изходния код, представяме извикванията на функции в граф (или обектен модел), на базата на който извличаме представителна динамична архитектурна информация.
  + *Анализ при изпълнение на кода* – посредством предварително подготвени тестови сценарии и предварително инструментиран код получаваме дървета на извикване на функции, които след обработка представяме като динамична архитектурна информация.
* **Мета-модел** – представянето на всички архитектурни елементи посредством единен модел, на базата на който могат да се изграждат модели представящи архитектурата на множество софтуерни системи. Въвеждането му дава възможност за стандартизация на комуникация на множество инструменти вариращи от инструменти за визуализация до такива за комплексен анализ и преструктуриране.

#### Гъвкавост

* **Степен на покритие на потенциални архитектури** – възможността на инструмента да се използва за широка гама от архитектури. Существуват два варианта:
  + *без предварителна подготовка* – инструмента е достатъчно гъвкав за да няма нужда да му се задават специфични критерии за конкретния изходен код под анализ
  + *с предварителна подготовка* – необходимо е предварително залагане на критерии преди анализ на даден изходен код.
* **Оперативна съвместимост** - способността на инструмента да оперира свъместно с външни системи и стандарти.

#### Лекота на употреба

* **Потребителски интерфейс**
  + инструмента позволява лесно задаване и изпълнение на анализа над даден изходен код, посредством например интуитивен графичен интерфейс.
  + възможност за наблюдаване на резултата от анализа в графичен вид.

### Сравнителен анализ

И двете предложени решения извършват *структурен анализ*, като много-изгледната среда за архитектурна реконструкция (*2.3.1*) прави този анализ без предварително дефиниране на критерии за компоненти и конектори. За разлика от предложението на DS, което има предварително дефинирани екстрактори (*Фигура 10*) подготвени за спецялно за компонентия модел. Съответно първия инструмент може да се класифицира като по-гъвкав и универсален, от друга страна инструмента за анализ на DS е настроен точно за техния компонентен модел, съответно ще дава много точни резултати когато е приложен над техния код.

Що се отнася до *динамичен анализ*, имаме ясно изразена стратегия в първия предложен инструмент за анализ, като метода по който се извършва е от типа *анализ при изпълнение на кода*. За решението на DS нямаме доказателство, че такава функционалност съществува.

От двата инструмента решението на DS залага използването на мета-модел за техния компонентен модел, което автоматично открива възможност за създаването на много спомагателни инструменти работещи на базата на мета-модела, което подобрява и неговата *оперативна съвместимост*.

Инструмента за анализ на DS показва силен акцент на графичното представяне на резултата от анализа, докато при много-изгледния среда това не е толкова силно изразено.

## Изводи

След като бяха изложени основните методи в реверсивния инжинеринг и връзките между тях, както и необходимите термини и дефиниции от софтуерните архитектури, които са изграждащи блокове при решение на проблема от заданието, беше представено и концептуално описание на среда за архитектурна реконструкция. Предложени бяха и две съществуващи решения на зададения проблем както и критерии за анализ на такъв тип система. Съответно беше направен и анализ на двете решения. На кратко първата предложена съществуваща система за анализ (много-изгледна среда за архитектурна реконструкция, виж *2.3.1*) предлага по-голяма гъвкавост що се отнася да типа изследвана архитектура. Докато втората предложена (платформа за реверсивен инженеринг и компонентен модел на DS, виж *2.3.2*) е силно стандартизирана и подготвена за конкретен компонентен модел, описан с предварително дефиниран мета-модел (*2.3.2.4*), благодарение на който се създава възможност за лесно изграждане и кооперация на голяма гама от инструменти базирани на същия мета-модел.

# Анализ

**Абстракт:**

Главата започва с развиване на концептуален модел на базата на заданието (точка *1.5*) като последователно се добавят необходими елементи (процеси и структури от данни), които първоначално не са видими от заданието. Този процес продължава до достигане на завършен концептуален модел. След това разширяваме концептуалния модел и го представяме посредством диаграми на процесите. На базата на този концептуален модел описваме: *типични случаи на употреба*, *мета-модел на архитектурното хранилище*, в което ще складираме анализираната архитектурна информация, *формат на генерирания базов код* и *групата от критерии* на базата, които ще изпълняваме над проекта под анализ.

## Концептуален модел

Следвайки основните изисквания от точка *1.5*, можем да извлечем две основни функционалности на системата:

* *Изпълнение на автоматичен анализ върху даден код*
* *Генерация на базов код*

Първата функционалност ще обхожда структурата и елементите на кода от проекта под анализ и ще търси съвклупност от доказателства за наличието на основни архитектурни елементи. При наличието на достатъчно доказателства за съществуването на базов архитектурен елемент той ще бъде отразен в хранилището за архитектурни артефакти, като връзките му с други такива артефакти също ще бъдат отразени (ако има такива). След като всички елементи от проекта под анализ са обходени, ще разполагаме с хранилище отразяващо елементите на архитектурата и връзките между тях.

След като разполагаме с хранилище отговарящо на архитектурата на системата може да започне генерация на базов код. Т.е. с предварително заготвени файлови шаблони и структура, обхождайки хранилището генерираме файлове осигуряващи средата на комуникация и обвивка на основните архитектурни елементи.



Фигура (концептуален модел)

Този модел обаче е непълен, тъй като липсват критериите, по които решаваме, че даден елемент от кода и/или структурата на проекта отговарят на артефакт в архитектурата. Което добавя допълнително изискване към първоначалните условия. А именно *подготовка на критерии за анализ.* Подготовката на тези критерии ще доведе до *база от критерии,* които ще бъдат допълнителните входни данни на функционалността *изпълнение на анализ*, освен кода на проекта под анализ. Успешното изпълнение на условието в един критерий над даден елемент от кода под анализ, ще бъде доказателство за съществуването на архитектурен елемент. Следователно този архитектурен елемент ще бъде добавен към хранилището на архитектурния модел (*Фигура 13*). Подготвянето на критериите е посредством ръчен анализ и ръчно въвеждане в базата.



Фигура (Подготовка на критерии за анализ)

За описването на даден архитектурен стандарт се очаква да имаме група от такива критерии подбрани в база. Не изключваме варианта в даден момент тази база да съдържа богат асортимент от групи критерии за различни архитектрурни стандарти. Т.е. колкото повече архитектурни стандарти базата ни с критерии разпознава, толкова по ценнен става инструмента, който разработваме.

Като прибавим допълващата функционалност(*подготовка на критерии за анализ*) концептуалния модел започва да изглежда по следният начин:



Фигура (разширен концептуален модел на системата)

Остава още една стъпка, която не е изобразена и тя е възможността да съхраним *хранилището на архитектурни елементи* (извлечения архитектурен модел) във файлов формат. Т.е. трябва да сериализираме получения модел в XMI формат. Идеята на което ни дава възомжност за преработка, в случай че преизползваме модела и го пренастройваме на базата на нови изисквания, които сме получили. Това също ни позволява да комуникираме модела с инструменти работещи със същия стандарт на сериализация.



Фигура (разширен концептуален модел със сериализация)

На кратко, след ръчна подготовка на *критериите за архитектурни артефакти* за дадена стандартна архитектура, съхранени в *базата с критерии* изпълняваме анализ над проект отговарящ на същия стандарт. Този анализ представлява проверка на всички критерии от базата върху всички елементи от кода и структурата му, като в резултат получаваме *архитектурен модел*. След като анализът е преключил преминаваме към сериализация на въпрсония модел в стандартен файлов формат, от който можем да преминем към генерация на базов код, от която получаваме файлове осигуряващи средата на комуникация и обвивка на основните архитектурни елементи.

## Работни процеси

Тук на базата на концептуалния модел представяме детайлни процеси, които трябва да спазва приложението за да отговаря на условията. Първата подточка описва графично *подготовката на критерии за анализ,* след това описваме *изпълнението на анализ* и *сериализация на модел* и на последно място описваме *генерацията на базов код.*

Виж цяла карта на процесите в *Приложение 7*

### Подготовка на критерии за анализ



Фигура (Подготовка на критерии за анализ, легенда: Приложение 6)

Това е единствения ръчен процес, в който е необходимо да се дефинират критериите за архитектурни елементи по подобие на точка 3.3.4, да се приведат във формат удобен за четене от базовия анализатор и да се поместят в базата с критериите.

### Анализиране на проект и сериализация на хранилището



Фигура (Анализиране на проект и сериализация на хранилището, легенда: Приложение 6)

След като е стартиран процеса и групата от критерии, които ще се ползват, започва обхождане на зададения проект и всеки един от елементите му се подлага под анализ със съответната група от критерии. След като даден критерий е удовлетворен се създава съответни елемент в UML хранилището. Този процес се повтаря до изчерпване на всички елементи на проекта.

Следва конвертиране на UML хранилището (модела) в XMI формат.

### Генерация на базов код



Фигура ( Генерация на базов код, легенда: Приложение 6)

Процеса започва с избиране на XMI файл съдържащ UML модел. Следва десериализация на XMI файла или конвертирането му към обектен UML модел. След това елементите на въпросния UML модел се обхождат и се генерира базов код на системата на базата на предварително заготвени шаблони.

## Потребителски (функционални) изисквания

В тази точка представяме основна диаграма на случаите на употреба както и подробно описание на всеки един случай на употреба. Следва подробно описание на мета-модела, с който ще описваме извлечената архитектурна информация последвани от структура и формат на генерирания базов код (*3.2.3*). На последно място описваме в детайл критериите на базата, на които трябва да се *приготвят критерии за анализ* (*3.2.1*).

### Типични случаи на употреба

От концептуалния модел представен в точка *3.1* и работните процеси (*3.2*), можем да синтезираме следната диаграма на типични случаи на употреба:



Фигура ( Типични случаи на употреба)

Следва подробно описание на отделните случаи на употреба:

#### Подготовка на критериите за анализ (*Prepare analysis criteria*)

|  |  |
| --- | --- |
| ***Свързани изисквания*** | Случай на употреба 3.3.1.2  Случай на употреба 3.3.1.3  Случай на употреба 3.3.1.4  Изискване 3.2.1 |
| ***Цел в конктекста*** | Това е процеса на запълване на базата с критерии за всички възможни артефакти |
| ***Предусловия*** | Достъпът до системата се у становява с права на *Конфигуратор* |
| ***Условия за успешно изпълнение*** | Действията по създаване на критерии за всеки един вид архитектурен артефакт са завършили успешно |
| ***Условия за неуспешно изпълнение*** | Един или повече действия по създаване на критерий за вид архитектурен артефакт е завършил неуспешно |
| ***Основни участници*** | Конфигуратор (Configurator) |
| ***Основен поток на събития*** | Конфигуратора изпълнява всички действия по създаване на критерии за архитектурен артефакт |
| ***Разширения*** | Няма |

Таблица ( Подготовка на критериите за анализ)

#### Подготви критерий за файлов формат (*Prepare File criteria*)

Виж  *Фигура 19*

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.2.1 | |
| ***Цел в конктекста*** | Това е действието при което се въвежда критерий за разпознаване на даден файлов формат по задаване на файлов път | |
| ***Предусловия*** | Достъпът до системата се установява с права на *Конфигуратор* | |
| ***Условия за успешно изпълнение*** | Въвеждането на критерия в базата е преключило успешно | |
| ***Условия за неуспешно изпълнение*** | Въвеждането на критерия в базата е преключило неуспешно | |
| ***Основни участници*** | Конфигуратор (Configurator) | |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Конфигуратора създава критерии за файлов тип |
|  | 2 | Конфигураора добавя критерия в базата |
| ***Разширения*** | Няма | |

Таблица (Подготви критерий за файлов формат)

#### Подготви критерий за компонент (*Prepare component criteria*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.2.1 | |
| ***Цел в конктекста*** | Това е действието при което се въвежда критерий за разпознаване на компонент в структурата и елементите на кода от проекта под анализ | |
| ***Предусловия*** | Достъпът до системата се установява с права на *Конфигуратор* | |
| ***Условия за успешно изпълнение*** | Въвеждането на критерия в базата е преключило успешно | |
| ***Условия за неуспешно изпълнение*** | Въвеждането на критерия в базата е преключило неуспешно | |
| ***Основни участници*** | Конфигуратор (Configurator) | |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Конфигуратора създава критерии за компонент |
|  | 2 | Конфигураора добавя критерия в базата |
| ***Разширения*** | Няма | |

Таблица (Подготви критерий за компонент)

#### Подготви критерий за конектор (*Prepare connector criteria*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.2.1 | |
| ***Цел в конктекста*** | Това е действието при което се въвежда критерий за разпознаване на конектор в структурата и елементите на кода от проекта под анализ | |
| ***Предусловия*** | Достъпът до системата се установява с права на *Конфигуратор* | |
| ***Условия за успешно изпълнение*** | Въвеждането на критерия в базата е преключило успешно | |
| ***Условия за неуспешно изпълнение*** | Въвеждането на критерия в базата е преключило неуспешно | |
| ***Основни участници*** | Конфигуратор (Configurator) | |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Конфигуратора създава критерии за конектор |
|  | 2 | Конфигураора добавя критерия в базата |
| ***Разширения*** | Няма | |

Таблица (Подготви критерий за конектор)

#### Изпълни анализ (*Perform analysis*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.2.2 | |
| ***Цел в конктекста*** | проверка на всички критерии от базата върху всички елементи от проекта под анализ и структурата му, като в резултат получаваме *архитектурен модел* | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Достъпът до системата се установява с права на *Потребител* |
|  | 2 | Наличие на група от критерии съответстваща на стандарта на архитектурата на проетка под анализ |
| ***Условия за успешно изпълнение*** | UML хранилище отговарящо на архитектурата на проекта под анализ е създадено | |
| ***Условия за неуспешно изпълнение*** | Заявката за анализ на проекта е отказана | |
| ***Основни участници*** | Потребител (User) | |
|  | *събитие* | Събитие за анализ на проект е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Подбиране на групата от критерии отговарящи на стандарта на архитектурата на проекта под анализ |
|  | 2 | Задаване на път до проекта под анализ |
|  | 3 | Стартиране на анализа |
| ***Разширения*** | Няма | |

Таблица (Изпълни анализ)

#### Обхождане елементите на проекта (*Scan Project's source files*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Случай на употреба 3.3.1.7 | |
| ***Цел в конктекста*** | Обхожда всички елементи(директории, файлове) от съдържанието на проекта под анализ. Подготвя всеки от елементите в подходяща форма за анализ. | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Валиден път до проекта под анализ |
| ***Условия за успешно изпълнение*** | Обхождане на всички файлове от пътя на проекта без входно/изходни грешки | |
| ***Условия за неуспешно изпълнение*** | Обхождане на всички файлове от пътя на проекта с една или повече входно/изходни грешки | |
| ***Основни участници*** | *събитие* | събитие за сканиране на проекта е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | За всеки елемент от структурата на проекта премини към стъпка 2 |
|  | 2 | Приведи елемента в удобен вид за анализ |
|  | 3 | Извлечи архитектурна информация от елемента 3.3.1.7 |
| ***Разширения*** | Няма | |

Таблица (Обхождане на елементите на проекта)

#### Извлечи архитектурна информация (*Extract architecture data*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Случай на употреба 3.3.1.6  Случай на употреба 3.3.1.8 | |
| ***Цел в конктекста*** | По даден елемент(директория, файл) от проекта под анализ изпълнява група от критерии за наличието на архитектурни артефакти | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Валиден път до елемента на проекта под анализ |
| ***Условия за успешно изпълнение*** | Архитектурни артефакти са създадени в хранилището | |
| ***Условия за неуспешно изпълнение*** | Възникнала е грешка при създаване на архитектурни артефакти в хранилището | |
| ***Основни участници*** | *събитие* | Събитие за анализиране на елемент е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Над съдържанието на елемента изпълни подходяща група от критерии |
|  | 2 | В случай на разпознат архитектурен артефакт създай *архитектурен елемент* в базата  3.3.1.8 |
| ***Разширения*** | Няма | |

Таблица (Извлечи архитектурна информация)

#### Създай архитектурен модел (*Create Architecture model*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Случай на употреба 3.3.1.7  Изискване 3.3.2 | |
| ***Цел в конктекста*** | Създаване на архитектурен артефакт в хранилището (архитектурния модел). | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Въведен е валиден архитектурен артефакт |
| ***Условия за успешно изпълнение*** | Архитектурен артефакт е създадени в хранилището | |
| ***Условия за неуспешно изпълнение*** | Възникнала е грешка при създаване архитектурния артефакт в хранилището | |
| ***Основни участници*** | *събитие* | Събитие за въвеждане на архитектрурен елемент в хранлищитео е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Създай въведения архитектурен артефакт в хранилището |
| ***Разширения*** | Няма | |

Таблица (Създай архитектурен модел)

#### Сериализация на UML хранилището (*Serialize UML model*)

Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.3.2 | |
| ***Цел в конктекста*** | По зададено архитектурно хранилище (модел) създай съответстващ UML модел сериализиран в XMI файл . | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Валидно архитектурно хранилище |
| ***Условия за успешно изпълнение*** | Нов XMI файл съдържащ UML модел съответстващ на архитектурното хранилище е създаден | |
| ***Условия за неуспешно изпълнение*** | Възникнала е грешка при създаване на XMI файл | |
| ***Основни участници*** | Потребител (User) | |
|  | *събитие* | Събитие за конвериране на архитектурносто хранлище в XMI файл е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Задай валидно архитектурно хранилище |
|  | 2 | Създай XMI файл съдържащ UML модел съответстващ на архитектурното хранилище |
| ***Разширения*** | Няма | |

Таблица (Сериализация на UML хранилището)

#### Генериране на базов код (*Generate Base code*)

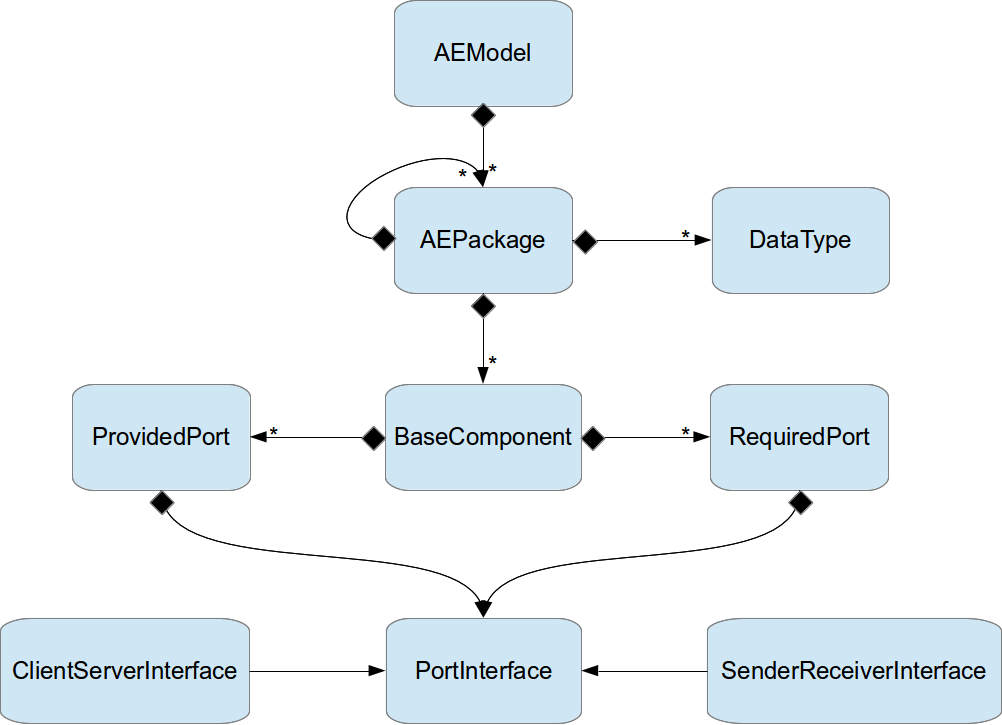
Виж  *Фигура* 19

|  |  |  |
| --- | --- | --- |
| ***Свързани изисквания*** | Изискване 3.3.3  Изискване 3.2.3 | |
| ***Цел в конктекста*** | По зададен XMI файл съответстващ на UML хранилище(модел) генерирай базов код по предварително зададен шаблон | |
| ***Предусловия*** | ***Номер*** | ***Условие*** |
|  | 1 | Валиден XMI файл съдържащ UML хранилище |
| ***Условия за успешно изпълнение*** | Генерирана е структура с базов код на базата на съдържащата се в XMI файла архитектура | |
| ***Условия за неуспешно изпълнение*** | Възникнала е грешка при генерирането на базов код | |
| ***Основни участници*** | Потребител (User) | |
|  | събитие | Събитие за генериране на базов код е настъпило |
| ***Основен поток на събития*** | ***Стъпка*** | ***Действие*** |
|  | 1 | Задай валиден XMI файл съдържащ UML хранилище |
|  | 2 | генерирай базов код по предварително зададен шаблон |
| ***Разширения*** | Няма | |

Таблица (Генериране на базов код)

### Мета-модел на архитектурното хранилище

Следва мета-модела необходим за описване на архитектурното хранилище (UML хранилище: *Фигура 17*). Важно е да се отбележи, че за направата на този мета-модел се използват идеи от виртуалната функционална шина/среда (Virtual Functional Bus [R7]) на AUTOSAR.



Фигура ( Мета-модел за описване на архитектурата)

#### AEModel ( Инстанция на модела)

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | AEModel | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | AEPackageList | AEPackage | *List* |
| **Описание** | Основен елемент на модела. Използва се като базова инстанция и главна точка за достъп до всички останали елементи от модела. | | |

Таблица ( Описание на AEModel)

#### AEPackage ( Пакет)

Виж  *Фигура 20*

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | AEPackage | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | DataTypeList | DataType | *List* |
|  | BaseComponentList | BaseComponent | *List* |
|  | AEPackageList | AEPackage | *List* |
| **Описание** | Основен контейнер на елементи в модела. Той групира елементи и също така може да съдържа други пакети. | | |

Таблица ( Описание на AEPackage)

#### BaseComponent ( Компонент)

Виж  *Фигура 20*

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | BaseComponent | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | PPort | ProvidedPort | *List* |
|  | RPort | RequiredPort | *List* |
| **Описание** | Представлява модулна част от системата, която енкапсулира своето съдържание. По същество е заменим в средата, която е използван. Поведението му се определя от предлагани и необходим интерфейс (*PPort*/*RPort*). | | |

Таблица (Описание на BaseComponent)

#### ProvidedPort (Предлаган интерфейс)

Виж  *Фигура* 20

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | ProvidedPort | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | Interface | PortInterface | 1 |
| **Описание** | Договорна точка на компонента съдържаща предлаган интерфейс на средата, в която компонента (*BaseComponent*) ще се интегрира. Описанието на интерфейса се съдържа в елемента *Interface*. | | |

Таблица (Описание на ProvidedPort)

#### RequiredPort (Необходим интерфейс)

Виж  *Фигура* 20

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | RequiredPort | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | Interface | PortInterface | 1 |
| **Описание** | Договорна точка на компонента съдържаща необходим интерфейс от средата, в която компонента (*BaseComponent*) ще се интегрира. Описанието на интерфейса се съдържа в елемента *Interface*. | | |

Таблица (Описание на RequiredPort)

#### PortInterface

Виж  *Фигура* 20

|  |  |
| --- | --- |
| **Наименование** | PortInterface |
| **Описание** | Базов класификатор представляващ група от съгласувани публични свойства и задължения. Тъй като представлява декларация, не може да бъде директно инстанциран. |

Таблица (Описание на PortInterface)

#### DataType (Тип данни)

Виж  *Фигура* 20

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | DataType | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | LowerLimit | Integer | 1 |
|  | UpperLimit | Integer | 1 |
| **Описание** | Елемент представляващ тип данни. Съдържа долна, горна граница и наименование. | | |

Таблица (Описание на DataType)

#### SenderReceiverInterface (Интерфейс за пренос на данни)

Класификатор, който реализира *PortInterface* (виж  *Фигура* 20)



Фигура (диаграма на SenderReceiverInterface)

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | SenderReceiverInterface | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | DataElementList | DataElement | *List* |
| **Описание** | Класификатор, който реализира *PortInterface*. Предназначен е за описване на интерфейс отговорен за пренос на различни типове данни. | | |

Таблица (Описание на SenderReceiverInterface)

#### DataElement (Елемент от данни)

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | DataElement | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | DT | DataType | 1 |
| **Описание** | Елемент от данни. Съдържа наименование и инстанция от тип данни. | | |

Таблица (Описание на DataElement)

#### ClientServerInterface (интерфейс клиент/сървър)



Фигура (диаграма на ClientServerInterface)

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | ClientServerInterface | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | OperationList | Operation | *List* |
| **Описание** | Класификатор, който реализира *PortInterface*. Предназначен е за описване на интерфейс отговорен изпълняване/заявяване на лист от операции на принципа на клиент-сървър. | | |

Таблица (Описание на ClientServerInterface)

#### Operation (Операция)

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | Operation | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | DT | DataType | 1 |
|  | ParamDataList | ParamData | *List* |
| **Описание** | Елемент деклариращ операция с определен прототип: наименование (*Name*), резултат (*DT*), списък с параметри (*ParamDataList*) | | |

Таблица (Описание на Operation)

#### ParamData (Параметър)

Виж *Фигура 22*

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование** | ParamData | | |
| **Атрибути** | **Наименование** | **Тип** | **Дименсии** |
|  | Name | String | 1 |
|  | DT | DataType | 1 |
| **Описание** | Елемент представляващ параметър на операция (*Operation*). Съдържа наименование и тип данни (*DT*) | | |

Таблица (Описание на ParamData)

### Формат на генерирания базов код

Генерирания код се разделя на две основни части:

* интерфейсна
* имплементационна

Интерфейсната част се грижи за комуникацията между компонентите. Тя имплементира конекторите между компонентите. Имплементационната част съдържа реализацията на функционалностите на компонентите.



Фигура (Структура на генерирания базов код)

На *Фигура 23* е показана структурата, която трябва да се спазва при генерация на базовия код.

#### Основна директория

Основната директория приема името на модела, от който се генерира.

#### Проектна директория

Проектната директория приема името на пакета наследник на моделния елемент.

#### Интерфейсна директория

Интерфейсната част е представена от папката *“Interface”*, чието съдържание е:

##### *rte.h*

Декларация на интерфейсите и конекторите между компонентите.

##### rte.c

Дефиниция на интерфейсите и конекторите между компонентите

##### rte\_<cmp>.h

Интерфейсен договор за всеки един компонент, предоставящ връзките му с останалите компоненти. *<cmp>* се замества с името на компонента.

Детайлен шаблон за файловете може да се намери в *Приложение 5.*

#### Директорийна структура на слоевете

Непосредствено под *Проектната директория* 3.3.3.2 следва да се генерират директори следващи пакетната структура в модела.

#### Компонентна имплементация

Компонентната имплементация се представлява от един файл за всеки компонент:

* *<cmp>.c* - файла се наименова с името на компонента. Генерирания файл ще съдържа празни функции за имплементиране на предоставените интерфейсните от тип *ClientServerInterface*. Т.е. след генерация се очаква разработчика да попълни имплементацията интерфейсните функции, като е допустимо да добави локални функции.

Детайлен шаблон може да се намери в *Приложение 5.*

### Група от критерии за стандартна архитектура

Следват изисквания за критерии отговарящи на примерна стандартна аркитектура на вградена система:

#### Критерий за компонент

Файловете на компонентите са представени с една и съща част преди разширението, изключение може да направи добавяне на един от следните символи: *[‘x’, ‘p’,’ j’]*

T.e. компонента се разпознава по следния начин:

**<име на компонент>**[‘x’|’p’|’j’|’’].<разширение>

#### Критерии за конектори

Критериите за конектори включват описания на конектор в различните видове файлове, в които той може да се срещне. Съответно в случай, че по време на анализ на файл попаднем на наличие на такъв конектор е показанао и като какъв елемент от мета-модела (3.3.2) да се опише.

##### “C” имплементационен файл

* mDATControl

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATControl* | | |
| Формат | *mDATControl(<име на контрола>, <аргумент>)* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *ClientServerInterface* | *DATCtrl\_<име на контрола>\_If* |  |
| Операция | *Operation* | *Invoke* | *Void* |
| Параметър | ParamData | *SubControlSelector* | *Integer* |

Таблица (Описание на mDATControl, \*.c файл)

* mDATRead

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATRead* | | |
| Формат | *mDATRead[Table](<тип данни>, <елемент>, [<индекс на таблицата>], <тип достъп>)* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *<тип данни>* |

Таблица (Описание на mDATRead, \*.c файл)

* mDATWrite

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATWrite* | | |
| Формат | *mDATWrite[Table](<тип данни>, <елемент>[, <индекс на таблицата>], <стойност>, <тип достъп>)* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *<тип данни>* |

Таблица (Описание на mDATWrite, \*.c файл)

* TOSReadSignal

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *TOSReadSignal* | | |
| Формат | *TOSReadSignal(cTOSSignal<име на сигнала>)* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *TOSSig\_<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *Integer* |

Таблица (Описание на TOSReadSignal, \*.c файл)

* TOSWriteSignal

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *TOSWriteSignal* | | |
| Формат | *TOSWriteSignal(cTOSSignal<име на сигнала>)* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *TOSSig\_<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *Integer* |

Таблица (Описание на TOSWriteSignal, \*.c файл)

##### “C” хедър файл

* mDATControl

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATControl* | | |
| Формат | *#define mDATControl<име на контрола>(<аргумент>)* | | |
| Посока на интерфейса | *Provided* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *ClientServerInterface* | *DATCtrl\_<име на контрола>\_If* |  |
| Операция | *Operation* | *Invoke* | *Void* |
| Параметър | ParamData | *SubControlSelector* | *Integer* |

Таблица (Описание на mDATControl, \*.h файл)

* mDATRead

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATRead* | | |
| Формат | *#define mDATRead[Table]<тип данни><елемент><тип достъп>* | | |
| Посока на интерфейса | *Provided* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *<тип данни>* |

Таблица (Описание на mDATRead, \*.h файл)

* mDATWrite

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *mDATRead* | | |
| Формат | *#define mDATWrite[Table]<тип данни><елемент><тип достъп>* | | |
| Посока на интерфейса | *Provided* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *<елемент>\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *<тип данни>* |

Таблица (Описание на mDATWrite, \*.h файл)

##### JIL файл (\*.jil)

* OnControl

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *OnControl* | | |
| Формат | *OnControl = <тип контрол>* *{* …  *Ctrl = <име на контрола>;* … *};* | | |
| Посока на интерфейса | *Required* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *ClientServerInterface* | *Ctrl\_<име на контрола>\_If* |  |
| Операция | *Operation* | *Invoke* | *Void* |

Таблица (Описание на OnControl, \*.jil файл)

* ProdControl

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *ProdControl* | | |
| Формат | *ProdControl**{* …  *Name = <име на контрола>;* … *};* | | |
| Посока на интерфейса | *Provided* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *ClientServerInterface* | *Ctrl\_<име на контрола>\_If* |  |
| Операция | *Operation* | *Invoke* | *Void* |

Таблица ( Описание на ProdControl, \*.jil файл)

* TOSSignal

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | *TOSSignal* | | |
| Формат | *Signal <име на сигнала>;* | | |
| Посока на интерфейса | *Provided* | | |
|  | Елемент | Наименование | Тип |
| Интерфейс | *SenderReceiverInterface* | *TOSSig\_< име на сигнала >\_If* |  |
| Елемент от данни | *DataElement* | *<елемент>* | *Integer* |

Таблица ( Описание на TOSSignal, \*.jil файл)

## Качествени (нефункционални) изисквания

### Скалируемост

Дизайна на системата трябва да позволява лесно разширяване. Най-вече възможността да се добавят критерии за конектори и компоненти както и добавянето на нови файлови анализатори. Също основния анализатор на проекти (*3.3.1.5*) трябва да може да бъде заместван с алтернативен такъв в случай, че за дадена стандартна архитектура, основния анализатор не е подходящ.

### Модифицируемост и документация

С цел лесна и ефикасна модификация и поддържане на точна документация на ниво дизайн, разработката на системата да се направи с UML2.x модел, от който да се генерира основната структура на кода.

### Поддръжка и възможност за разширение

След добавяне на нови изисквания към системата в периода на поддръжка на системата, биха могли да се отличат части от кода, които се изпълняват неефективно и би било изгодно те да се пренапишат на език, който се изпълнява по-ефективно от предложения. Поради това е добре езика, на който е имплементирана системата да е с възможност за разширяване с други езици за програмиране.

### Потребителски интерфейс

Системата трябва да представлява библиотека с класове представящи:

* Основен анализатор
* Анализатори на различни файлови формати
* Компоненти критерии
* Конекторни критерии
* Сериализатор на извлечения архитектурен модел към XMI (*3.3.1.9*)

Трябва да съдържа примерна имплементация с елементите на библиотеката.

### Тестваемост

Систематата трябва да се предостави със съответния тестов софтуер и тестови данни, които са достатъчни да изпълнят регресионни тестове на всичките ѝ функционалности.

## Изводи

След като развихме концептуалния модел (точка *3.1*) достигнахме до извода, че има изискване, което не е явно зададено в точка *1.5*, а именно *изготвяне на критерии за анализ*, в основата на които е заложено разпознаването на архитектурни елементи. След детайлен анализ на не-явното изискване изведохме три негови под случая:  *подготовка на критерии за тип файл; подготовка на критерии за компонент; подготовка на критерии за конектор.*

При наличие на база с критерии за дадена софтуерна архитектура, можем да анализираме проект следващ същия стандарт. С анализа на проекта, спазвайки изискванията за правилата за трансформация към мета-модела в точка *3.3.4* получаваме архитектурно хранилище, което от своя страна сериализираме в файл с разпознаваем XMI формат носещ UML модел отговарящ на хранилището.

След това на базата на този XMI файл можем да стартираме генерация на базов код на следствие от анализа, който представлява файлове осигуряващи средата на комуникация и обвивка на компонентите.

# Използвани технологии, платформи и методологии

**Абстракт:**

В тази глава ще изложим изискванията към технологиите и методологиите за разработка. Ще изброим възможни кандидати и как те се представят спрямо заложените изисквания. След това ще анализираме резултатите и ще изберем най-подходящите технологии, платформи и методологии.

## Изисквания към средствата (технологии, платформи и методологии)

### Език за програмиране

Трябва да отговаря на следните изисквания:

* + *от високо ниво* – тъй като за добър дизайн и реализация на инструмента ще ни е нужен език с висока степен на абстракция, а не лесен достъп до изграждащи елементи на компютърната система.
  + *обектно-ориентиран* – тъй като избрания подход за разработка на инструмента е чрез изработване и поддържане на обектен модел на дизайна и генериране на базов код (класове, методи, свойства и връзки по между им).
  + *силна подръжка на работа с регулярни изрази* – анализа над файловете на изследвания проект ще се изпълнява посредством регулярни изрази.
  + *лесно преносим* – за да може инструмента да се прилага лесно в различни организации използващи за разработка на проектите си различни видове платформи.
  + *висока степен на изразителност* – за пестене време на разработка
  + *с възможност за разширяване на части от кода с други езици за програмиране* - след добавяне на нови изисквания към системата, биха могли да се отличат части от кода, които се изпълняват неефективно и би било изгодно те да се пренапишат на език, с който се изпълняват по-добре от първоначално избрания.

### Модел на софтуерната система

* Разработката на проекта да се извършва посредством обектен модел UML2.x.
* Изисквания към среда за разработка на модела:
  + Да поддържа импорт и експорт на XMI 2.x файлове
  + Да може да генерира избрания език за програмиране
* Основната структура на кода да се генерира от обектния модел. Това включва:
  + Пакети
  + Модули
  + Класове
  + Връзки м/у класовете (асоциация, агрегация и композиция)
  + Атрибути и методи на класовете
  + Инициализации

Това ще гарантира автоматична проследимост м/у дизайна в модела и кода на системата.

* В разработката на модела да се използват софтуерни шаблони за дизайн (software desing patterns), които да позволяват гъвкавост и лесна скалируемост на модела.

### Генератор на базовия код

Генерирането на базовия код е изискване от заданието на дипломната работа (точка *1.5*). За целта е необходима платформа за генериране на код на базата на UML, която трябва да изпълнява изискване *3.2.3.* На кратко казано платформа за трансформация от тип: модел към текст (MOFM2T)

## Видове средства за разработване на решението – оценка спрямо критериите

Следва изброяване на средства в три направления: *език за програмиране на решението*, *среда за разработване на UML модел*, който ще се използва за дизайна на инструмента и *платформа за генериране на код* на базата на UML необходим за изпълняване на изискване *3.2.3.* Респективно с изброяване на средствата ще представим и сравнителен анализ на базата на критериите описани в предната подточка за всеки един кандидат.

### Език за програмиране:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Език** | **Java** | **Python** | **C#** | **C++** |
| **от високо ниво** | ДА | ДА | ДА | ДА |
| **обектно-ориентиран** | ДА | ДА | ДА | ДА |
| **силна подръжка на работа с регулярни изрази** | ДА | ДА | ДА  ([слабо документирана](http://www.regular-expressions.info/tools.html) **[R12]**) | НЕ  ([има популярна библиотека с отворен код](http://www.regular-expressions.info/tools.html)) |
| **лесно преносим** | ДА | ДА | ДА | ДА  (в зависимост от използваните библиотеките) |
| **Степен на изразителност (*Приложение 3*)** | 700 | 300 | 750 | 700 |
| **възможност за разширяване с други езици за програмиране** | ДА | ДА | ДА | ДА |

Таблица (Сравнение на потенциални езици за програмиране на софтуерната система)

### UML и формати за представянето му

На базата на изискването от заданието (точка *1.5*) да се избере най-подходящ формат за представяне на UML първо ще направим кратко представяне на унифицирания език за моделиране (UML), след което ще обобщим предложенията за представянето му.

#### UML

Основните цели на UML са да подобри индустрията посредством осъвместяването на инструменти за визуално обектно моделиране. За да е възможен смислен обмен на моделна информация между инструментите е необходимо да се установи стандартно формално представяне и нотацията на модела. UML изпълнява следните изисквания [R26, стр. 17]:

* Формална дефиниция на общ MOF-базиран мета-модел, който специфицира абстрактния синтаксис на UML.
* Детайлно описание на семантиката на всяка UML концепция.
* Спецификация на лесно четимa нотация за елементите представящи различни UML концепции за моделиране. Както и правила за комбинирането им в различни типове диаграми отговарящи на различни аспекти на моделираната система.
* Детайлна дефиниция на начините, по които UML инструментите могат да станат съвместими с тази спецификация. Това става посредством XML-базирана спецификация на форматa за обмен (XMI) на модел, която трябва да е реализиран от сътветните инструменти.

На базата на последната точка можем да заключим, че стандартния формат за представяне на UML е XMI. Съществуват различни затворени формати за съхранение на UML модели, но особено този на IBM Rational [R28] е набрал популярност в края на миналото десетилетие, същевременно все повече и повече набира популярност имплементацията на XMI на Eclipse UML2 компонента, част от EMF, който е отворен формат.

#### XMI

XMI е широко използван формат за обмяна на модели на основата на XML. XMI дефинира много от важните аспекти в описването на обекти с XML:

* Основата е в представянето на обекти посредством XML елементи и атрибути.
* Тъй като обектите са обикновенно свързани помежду си, XMI предлага стандартни механизми за свързване на обекти в даден файл или между файлове.
* Идентичността на обектите позволява те да могат да бъдат реферирани от други обекти посредством идентификатор (ID) и универсален уникален идентификатори (UUID).
* Валидация на XMI документи посредством XML Schema.

XMI описва решения на горните проблеми като специфицира разширена Бакус-Наур Форма (EBNF) на правила за създаване на XML документи и схеми, които представят обектите консистентно.

В официалната спецификация на XMI [R27, стр. 17] се твърди, че е възможно да се дефинира начин, по който да се генерира схема от MOF модел, която представя MOF-съвместим модел.

Това означава, че всеки модел описан с MOF мета-модела би могъл да се опише с XMI, което се отнася и за UML, както беше описано в предната точка.

### Среда за разработване на UML модел

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Среда за разработване** | [**Enterprise Architect**](http://www.sparxsystems.com.au/)  **(v10.0)** | [**BoUML**](http://www.bouml.fr/) **(v4.21)** | [**Visual Paradigm**](http://www.visual-paradigm.com/) | [**Eclipse Papyrus**](http://www.eclipse.org/papyrus/) |
| **UML 2.x** | ДА | ДА | ДА | ДА |
| **MDA** | ДА | ДА | ДА | ДА |
| **XMI (импорт/експорт)** | ДА | ДА | ДА | ДА |
| **Генерира следните езици** | ActionScript, C, C#, C++, Delphi, Java, PHP, Python, Visual Basic, Visual Basic .NET, DDL, EJB, XML Schema, Ada, VHDL, Verilog, WSDL, BPEL, Corba IDL | C++, Java, PHP, IDL, Python | Java, C#, C++, PHP, Ada, Action Script | Ada 2005, C/C++, Java |
| **Лиценз** | комерсиален | GPL преди v5.0 | комерсиален | EPL |
| **Коментар** | Въпреки, че Python e споменат като език, който се генерира от средата. Той далеч не е на необходимото ниво | - | - | - |

Таблица (Сравнение на потенциални среди за UML моделиране на софтуерната система)

### Код генератор

|  |  |  |
| --- | --- | --- |
| **Код генератор** | [**Acceleo**](http://en.wikipedia.org/wiki/Acceleo) | **“Ръчно” написан**  **код генератор (*Приложение 4*)** |
| **Платформа** | Преносим (Java/Eclipse базиран) | зависима от избрания език за програмиране |
| **Стандарт** | ***MOFM2T***/***EMF*** съвместим | няма |
| **Възможност за разширяване на генерирания код** | неограничена | ограничена |
| **Време за подготовка преди употреба** | 3 дена  (при наличен EMF базиран входен модел) | 2 седмици  (дизайн/имплементация/тест) |

Таблица (Сравнение на потенциални код генератори за софтуерната система)

## Избор на средствата (технологии, платформи и методологии)

### Език за програмиране

Както става ясно от заданието, основно изискване към системата е да прави текстови анализ на кода на съществуваща софтуерна система, от което произлиза и изискването използвания език да е от високо ниво и добра подръжка на работа с регулярни изрази. Всички от предложените езици(*Таблица 34* (Сравнение на потенциални езици за програмиране на софтуерната система)) са от високо ниво, но C++ и C# имат някой ограничения: при C++ има възможност за използване на библиотека с отворен код ([PCRE](http://www.pcre.org/)), която би донесла допълнителни зависимост на системата към външна библиотека, докато при C# има налична подръжка по подразбиране, но за сметка на това е слабо документирана според **[R12]**.

Всички предложени езици поддържат обектно ориентирано програмиране. Това е основно изискване, тъй като това ще допринесе за по-лесната подръжка на системата, разбира се, само по себе си не е достатъчно условие, на първо място е важно системата да има строен дизайн, за което са добавени допълнителни изисквания.

От гледна точка на преносимост, предложения C++ изостава от останалите, тъй като е компилиран за разлика от останалите, които са интерпретирани езици и позволяват много по-лесна преносимост.

На базата на изследването направено в *Приложение 3* можем да заключим, че от предложените езици, Python е с най-голяма степен на изразителност. Това е от голяма полза тъй като софтуерната система, ще се разработва от един човек.

Необходимостта за разширяване с други езици произлиза от факта, че след добавяне на нови изисквания към системата, биха могли да се отличат части от кода, които се изпълняват неефективно и би било изгодно те да се пренапишат на език, с който се изпълняват по-ефективно от първоначално избрания. Естествено по този начин оставяме възможността за достъп до библиотеки, които не са написани на избрания език за програмиране. Всички от предложените езици имат различни възможности за разширяване.

Като избор на език за програмиране се спираме на **Python**, тъй като той отговаря на всички изисквания и се представя най-добре в степента си на изразителност, тази характеристика ще намали времето за разработка на системата. Естествено тъй като софтуерната система ще се генерира от модел, в който ще бъде заложена голяма част от бизнес логиката за решаване на проблема, не е изключено на по-късен етап да се премине към друг по-изгоден език, разбира се, след внимателно планиране.

Версията, на която се спираме е **Python 2.7.3**

### Формат за представяне на UML

Тъй като нашата цел (произлизаща не директно от заданието, а по-скоро от очакваните ползи на решението, точка *1.6*) е създаването на модел и представянето му във формат позволяващ интероперативност между различни видове инструменти (без ограничение да са разработени от даден производител) за обектно ориентирано моделиране, избираме отворения, предвиден и специфициран от същата организация, която разработва UML, a именно XMI формат.

Тъй като съвременната и перпоръчвана версия на **UML** е версия **2.4.1**, същата е версията и на **XMI** формата, който ще използваме.

### Среда за разработване на UML модел

Предложените среди за разработване на UML модел (*Таблица 35* (Сравнение на потенциални среди за UML моделиране на софтуерната система)) поддържат работа с UML 2.x. Мотивацията да се използва UML 2.x се крие зад факта, че ще разработваме обектно ориентирана софтуерна система, а той се е доказал с времето като подходящ модел за такава разработка.

Тъй като имаме изискване за генериране на код от разработвания модел, е необходимо средата за разработка да подръжа моделно разработвана архитектура (MDA), което е и характеристика на всяка една от предложените среди.

Необходимостта да може средата за разработка да поддържа импорт и експорт на XMI файлове се поражда от потенциалната възможност в бъдеще да се премине към друга система за моделиране. Възможността да експортираме системата в отворен формат ще ни позволи едно такова начинание.

Както се вижда от таблицата само 2 от средите поддържат код генерация на избрания език (Python), което означава, че избора ни се свежда до тези 2. Освен това както е видно и от коментара в таблицата тестова генерация на Python код от Enterprise Architect (версия 10) показа незадоволителни резултати. За сметка на това кратък тест с примерен модел от BoUML показа, че генерирания Python код отговаря на всички изисквания от секция 4.1.2.

Следователно изборът на среда за моделиране остава **BoUML** версия **4.21**.

Изборът на софтуерни шаблони за дизайн ще бъде представен в Глава 5.

### Генератор на базов код

В предложените в секция 4.2.3 варианти за генератор на базовия код имаме представител на силно подкрепен от стандарт (***MOFM2T***) и такъв, който разчита на собствена разработка и не следва изисквания на даден стандарт. Ограничението на първия е, че той може да работи с модели съвместими с EMF (моделната среда на Eclipse), но добрата новина е, че UML2 е съвместим с EMF, благодарение на EMF имплементацията на UML2 част от MDT(Model Development Tools) на Eclipse. От другата страна, алтернативния вариант с собствено разработен генератор е гъвкав в това отношение и би могъл да работи с широка гама модели, стига те да са представени в обектен модел за конкретния език на реализация. При Acceleo основната цел е да предостави удобни инструменти за сглобяване на код генератор базиран на файлови шаблони, той дава много удобен достъп до елементите на модел, като освен това дава възможност за използване на OCL (Object Constraint Language) върху модела, което позволява още по-голяма гъвкавост. Добавянето на нов тип файлов шаблон или специфична файлова структура като освен това и възможността за надграждане на шаблон са основни характеристики на Acceleo. Докато при изцяло собствено разработен генератор, тези изисквания тепърва трябва да се заложат в дизайна. И финално разработването на собствен генератор би имало основание ако средата, в която ще се изпълнява генератора е силно ограничена като ресурси (налични библиотеки, възможност за добавяне на нови библиотеки), но в противен случай разработването на такъв е силно рисковано, от гледна точка на време за разработка и усъвършенстване.

Така че се спираме на стандартизираният вариант за генериране на код от модел **Acceleo**.

## Изводи

Що се отнася до езика за програмиране изборът, на който сме се спряли се отнася основно до фазата на проекта, а именно ранна разработка. Но си оставяме възможност да преминем изцяло или дори частично към друг език за програмиране в последваща фаза. Улеснение за това ни дава изискването разработването на софтуерната система да стане чрез модел, по този начин голяма част от бизнес логиката на системата остава в модела и това позволява сравнително лесно преминаване към друг език за програмиране. Същевременно възможността на предложения език за да бъде разширяван посредством други езици за програмиране ни позволява при необходимост някой части от системата да бъдат написани или пренаписани на други езици.

От разгледаните среди за разработка на UML2.x модели стигаме до извода, че много внимателно трябва да се подбират и тестват тези инструменти, тъй като повечето от тях са с комерсиален лиценз и въпреки това има риск някой от предложените им характеристики да не отговарят на нашите изискивания. В същото време инструмент със свободен лиценз за същата дейност би могъл да отговори на изискванията. Разбира се оставяме си възможност за преминаване към друг инструмент благодарение на изискването да експортира модела в отворен стандарт(XMI). В случай, че бъде взето решение да се премине към друг инструмент за разработване на софтуерния модел във следваща фаза е важно да се има предвид, че подръжката на XMI в различните инструменти не е еквивалентна**[R13, 9]** и е твърде вероятно да се наложат серия от трансформации на съответния сериализиран модел, които е препоръчително предварително да се планират.

След проучването за генератор на базов код можем да допуснем, че изискването за генерацията на Python в инструмента за разработване на модела би могло да отпадне, тъй като можем сами да изготвим шаблони за необходимите ни файлове и директно генерираме посредством Acceleo и експортиран XMI модел на софтуерната система. Или още по-добре ако можем да намерим готови шаблони за генериране на Python от UML2. Разбира се за момента използваме на готово предложената генерация от BoUML.

Що се отнася до избора на генератор за базов код, основната цел е да се ползва нещо стандартно, с което се работи лесно, бързо се конфигурира и позволява възможности за лесно добавяне на нови шаблони. Това всичко ни го предоставя Acceleo, като ни открива и възможност същевременно в бъдеща фаза на проекта да генерираме кода на системата за анализиране.

# Проектиране

**Абстракт:**

В тази глава представяме архитектурата на системата, която представлява директни извадки от разработения UML модел **[D1]** на решението с допълнително описание. Първо представяме общата архитектура на приложението под формата на слоеве и основна пакетна диаграма. Следва дизайн (клас диаграми включващи описание) на мета-модела (*3.3.2*) и на всеки един от останалите слоеве.

## Обща архитектура

Тук представяне архитектурните слоеве на приложението с кратко описание на всеки един от тях и обща диаграма. Следва основен изглед на пакетите (UML пакетна диаграма) имплементиращи словете и описание на всеки един основен пакет.

### Слоеве

Генералната архитектура се базира на пет слоя представляващи:

* *Анализатор* – модули грижещи се за стартиране на анализа и обхождане на проекта под анализ
* *Скенер* – модули съдържащи и изпълняващи критериите за анализ на отделните елементи на проекта под анализ
* *Мета-модел* – описание на мета-модела на хранилището на архитектурни елементи
* *Сериализатор* – модули грижещи се за сериализацията на хранилището на архитектурни елементи към файлов формат
* *Външни модули* – използвани библиотеки и външни модули
* *Генерация на базов код* – модули генериращи базовия код



Фигура (Слоеве на системата)

Като основно правило за слоевете, е че само два съседни слоя могат да комуникират по между си.

### Пакетна диаграма (основен изглед)



Фигура (Обща архитектура)

На фигурата горе е изобразена пакетната диаграма на системата, като тя следва подредбата на слоевете (*5.1.1*).

Основно системата се разделя на два основни пакета:

#### Архитектурен екстрактор (ArchExtractor)

Архитектурния екстрактор съдържа следните пакети:

* *Базов (Base)* – съдържа тази част на системата, която не се очаква да се разширява:
  + *пакет на Анализатора (Analyzer)* – съдържа базови класове изпълняващи анализ на проект (*3.3.1.5*)
  + *пакет на Скенера (Parser)* – съдържа базови класове изпълняващи обхождане елементите на проекта (*3.3.1.6*), извличане на архитектурна информация (*3.3.1.7*) и създаване архитектурен модел (*3.3.1.8*)
  + *пакет на инфраструктурата (Infrastructure)* – съдържа инфраструктурните елементи от мета-модела (*3.3.2*)
  + *пакет софтуерен компонент (SwComponent)* - съдържа елементи от мета-модела описващи софтуерния компонент (*3.3.2*)
  + *пакет сериализатор на модела (ModelConverter) -* сериализира архитектурното хранилище (*3.3.1.9*)
* *Специфичен (Specific) –* тази част от системата, която се очкава да се разширява. Т.е. очаква се по един пакет описващ критерии, специфични скенери и анализатори за дадена стандартна архитектура. Конкретно в случая съдържа:
  + *пакет с критерии за стандартна архитектура (STK)* – съдържа имплементация на групата от критерии в точка *3.3.4*
    - *пакет за специфични скенери (StkParser) –* съдържа както файлови скенери така и критерии за компонент и конектори в стандартната архитектура (*3.3.4*)
    - *пакет за специфични интерфейси (StkPortInterfaces) –* съдържа метод фабрика (*method factory pattern*) за бързо представяне на конектори от стандартната архитектура в елементи от мета-модела (*3.3.2*)
    - *пакет съдържащ типове данни на стандартната архитектура (StkDataTypes)* – съдържа метод фабрика (*method factory pattern*) за бързо представяне на типове данни на стандартната архитектура в елементи от мета-модела
* *Библиотека (Lib)* – спомагателни външни за системата модули

#### Генератор на базов код (Base code generation)

*Acceleo* базиран проект изпълняващ:

* Генериране на базов код (*3.3.1.10*)

## Модел на данните (Мета-Модел)

Текущата точка включва кратко описание на основните класове от имплементацията на мета-модела (*3.3.2*) както и клас диаграма.

Подробно описание (на методи, атрибути и връзки) може да се намери в генерираната документация на модела **[D2]**

### Инфраструктурни

Следващите подсекции представят елементите от пакет “*Infrastructure* ” на UML модела на системта **[D1]**.

#### Identifiable

* *Описание* – класа е основен за имплементацията на мета-модела. Наследяването му от даден елемент показва, че въпросния елемент има наименование.
* *Клас диаграма*:



Фигура ( клас диаграма на Identifiable)

#### AEPackage

* *Описание* – Основен контейнер на пакетируеми елементи(*5.2.1.3*) в имплементацията на мета-модела. Реализира изискване *3.3.2.2.*
* *Клас диаграма*:



Фигура (клас диаграма на AEPackage)

#### PackagableElement

* *Описание* – Основен клас за имплементацията мета-модела. Клас, който го наследява ще бъде третиран като “*пакетируем*”, т.е. директно ще може да бъде прибавен в пакет (*AEPackage*).
* *Клас диаграма*:



Фигура (клас диаграма на PackagableElement)

#### InfrastructureFactory

* *Описание* – Метод фабрика за елементи от имплементацията на мета-модела в пакета “*Infrastructure*” на UML модела на софтуерната система **[*Приложение 2*:D1]**. Т.е. следните два елемента:
  + *AEModel*
  + *AEPackage*
* *Клас диаграма*:



Фигура (клас диаграма на InfrastructureFactory)

#### FactoryProvider

* *Описание* – представлява имплементация на “Съдържател” (Handler) от шаблона за дизайн: “Верига отговорности” (Chain of responsibility). Отговорен е за предоставянето на фабрика за конкретни обекти от инфраструктурния пакет. В случай, че “родителя” (*itsSuccessor*) е инициализиран, ще предаде заявката към него.
* *Клас диаграма*:



Фигура (клас диаграма на FactoryProvider)

#### AEModel

* *Описание* – Елемент корен на модела. Съдържа всички останали елементи в себе си. Реализира изискване *3.3.2.1.*
* *Клас диаграма*:



Фигура (клас диаграма на AEModel)

### Софтуерен компонент

Следващите подсекции представят елементите от пакет “*SwComponent*” на UML модела на системта **[D1]**.

#### BaseComponent

* *Описание* – Представя базовя компонент от мета-модела. Реализира изискване *3.3.2.3.*
* *Клас диаграма*:



Фигура (клас диаграма на BaseComponent)

#### ComponentFactory

* *Описание* – Метод фабрика за елементи от имплементацията на мета-модела в пакета “*Component*” на UML модела на софтуерната система **[*Приложение 2*:D1]**. Т.е. следня елемент: *BaseComponent*
* *Клас диаграма*:



Фигура (клас диаграма на ComponentFactory)

#### ProvidedPort

* *Описание* – Представя “*ProvidedPort*” от мета-модела. Реализира изискване *3.3.2.4*
* *Клас диаграма*:



Фигура (клас диаграма на ProvidedPort)

#### RequiredPort

* *Описание* – Представя “*RequiredPort*” от мета-модела. Реализира изискване *3.3.2.5*
* *Клас диаграма*:



Фигура (клас диаграма на RequiredPort)

#### SenderReceiverInterface

* *Описание* – Представя “*SenderReceiverInterface*” от мета-модела. Реализира изискване *3.3.2.8*
* *Клас диаграма*:



Фигура (клас диаграма на SenderReceiverInterface)

#### ClientServerInterface

* *Описание* – Представя “*ClientServerInterface*” от мета-модела. Реализира изискване *3.3.2.10*
* *Клас диаграма*:



Фигура (клас диаграма на ClientServerInterface)

#### PortInterfaceFactory

* *Описание* – Метод фабрика за елементи от имплементацията на мета-модела в пакета “*PortInterface*” на UML модела на софтуерната система **[*Приложение 2*:D1]**. Т.е. следните два елемента:
  + *SenderReceiverInterface*
  + *ClientServerInterface*
* *Клас диаграма*:



Фигура (клас диаграма на PortInterfaceFactory)

## Диаграми (на структура и поведение - по слоеве)

Текущата точка включва кратко статично и динамично описание на съдържанието на отделните слоеве на системата.

Подробно описание (на методи, атрибути и връзки) както и диаграми които не са показани тук, може да се намери в генерираната документация на модела **[*Приложение 2*:D2]**.

### Слой Анализатор



Фигура (Слой Анализатор)

#### Пакетна диаграма



Фигура (съдържание на пакет Analyzer)

Пакета *Analyzer* представя слоя *Анализатор.* Мястото му в цялата система може да се види на *Фигура 25.*

Съдържа:

##### *IAnalyzer*

Aбстрактен клас (*интерфейс*) представящ всички възможни анализатори.

##### BaseAnalyzer

Реализира *IAnalyzer* и имплементира изискване *3.3.1.5* и *3.3.1.6*.

###### Клас диаграма



Фигура (клас диаграма на BaseAnalyzer)

Също така наследява и *FactoryProvider(5.2.1.4)*, основната цел на което е да зададе на листа от обекти *BaseFileParser* себе си като доставчик на фабрики за обекти по време на инициализацията си:

###### Инициализация



Фигура (последователност на инициализиране на BaseAnalyzer)

###### Изпълнение



Фигура (последователност на изпълнение на функционалността на BaseAnalyzer)

След извикване на метода *scan(…)* с аргументи *път към проекта под анализ(ProjectDir)* и *хранилище на архитектурни елементи (inOutAEModel)* метода обхожда зададената файловата структура от *ProjectDir*, като за всеки файл проверява дали има *Файлов Скенер(itsBaseFileParserList)*, който разпознава файловия формат. В случай, че има скенер, който разпознава формата, се проверява дали вече съществува, компонент който чиито файл се анализира или се регистрира нов компонент (*updateComponent(…)*). След това се изпълнява анализ на файла с *fulfillComponentData(…)*.

След обхождане на всички файлове получаваме архитектурно хранилище (модел) *inOutAEModel* с попълнена архитектурна информация отговаряща на анализирания проект.

### Слой Скенер



Фигура (слой Скенер)

#### Пакетна диаграма



Фигура (съдържание на пакет Parser)

Пакета *Parser* представя слоя *Скенер.* Мястото му в цялата система може да се види на *Фигура 25.*

Съдържа:

##### IFileParser

Aбстрактен клас (*интерфейс*) представящ всички възможни файлови скенери.

##### IPortCriteria

Aбстрактен клас (*интерфейс*) представящ всички възможни критерии за конектори между компоненти.

##### BaseFileParser

Абстрактен клас реализиращ *IFileParser* и отговорен за имплементацията на изискване *3.3.1.7.*

###### Клас диаграма



Фигура (клас диаграма на BaseFileParser)

Също така наследява и *FactoryProvider(5.2.1.4)*, основната цел на което е да зададе на листа от обекти *BasePortCriteria* себе си като доставчик на фабрики за обекти в своята инициализация:

###### Инициализация



Фигура (Инициализация на BaseFileParser)

###### Изпълнение



Фигура (последователност на изпълнение на функционалността на BaseFileParser)

Метода *fulfillComponentData(…)* се извиква от инстанция на *IAnalyzer* с аргументи *име на компонент*, *път до файла под анализ* и *инстанция на компонент(inOutIComponent).* След което се проверява валидността на съдържанието на *файла под анализ* и същевременно се подготвя за анализ(премахване на коментари, структурно форматиране и т.н.). Ако файла е валиден изпълнява всички критерии за конектор в списъка *itsBasePortCriteriaList*. В инстанцията на компонента (*inOutIComponent*) се попълва допълнително детайли при изпълнението на критериите *execute(…)*. След като всички критерии от списъка са тесвани предаваме компонента като изход от метода.

##### BasePortCriteria

Абстрактен клас реализиращ *IPortCriteria* и отговорен за имплементацията на изискване *3.3.1.7* и *3.3.1.8.*

За повече информация виж *BaseFileParser(5.3.2.1.3)*

### Слой Мета-Модел



Фигура (Слой Мета-Модел)

Виж точка *5.2*

### Слой Сериализатор



Фигура (слой Сериализатор)

#### Пакетна диаграма



Фигура (съдържаниен на пакет ModelConverter)

Пакета *ModelConverter* представя слоя *Сериализатор.* Мястото му в цялата система може да се види на *Фигура 25.*

Съдържа:

##### XMIConverter

Клас отговорен за конвертирането на архитектурното хранилище на системата към UML модел съхранен в XMI формат. Имплементира изискване *3.3.1.9*

###### Клас диаграма



Фигура (клас диаграма на XMIConverter)

###### Изпълнение



Фигура (последователност на изпълнение на функционалността на XMIConverter)

По заявка на *потребителя*: *convert(…)* със аргументи *входен модел* отговарящ на мета-модела на системата (*5.2*) и *изходен XMI модел*, обхождаме елементите на входния модел и ги конвертираме в съответстващ *XMI модел.*

### Слой Генерация на базов код



Фигура (слой Генерация на базов код)

Този слой е представен от *Acceleo* проект поради изискване *4.3.3*. Проекта изпълнява изискване *3.3.1.10*. Като входни данни получава сериализирания UML модел в XMI формат, след което се генерира базов код на базата на UML модела посредством следните файлови шаблони:

#### Файлова структура



Фигура (файлова структура на Генератор на базов код)

##### RteImpl.mtl

Файлов шаблон с *Acceleo* формат изпълняващ изискване *3.3.3.3.2*

##### RteHeader.mtl

Файлов шаблон с *Acceleo* формат изпълняващ изискване *3.3.3.3.1*

##### RteComponentImpl.mtl

Файлов шаблон с *Acceleo* формат изпълняващ изискване *3.3.3.5*

##### RteComponentHeader.mtl

Файлов шаблон с *Acceleo* формат изпълняващ изискване *3.3.3.3.3*

##### main.mtl

Кумулативен файлов шаблон с *Accelеo* формат стартиращ генерацията на всеки един шаблон от гореописаните (виж *Фигура 55*).

## Ресурсни и спомагателни модули



Фигура (слой Външни модули)

### Пакетна диаграма



Фигура (пакетна диаграма Lib)

Външни модули използвани от системата са следните:

* os – стандартен Python модул с интерфейси на операционната система
* re – стандартен Python модул за операции с регулярни изрази
* xmi – библиотека имплементация на XMI мета-модела, базирана на *PyEMOF*

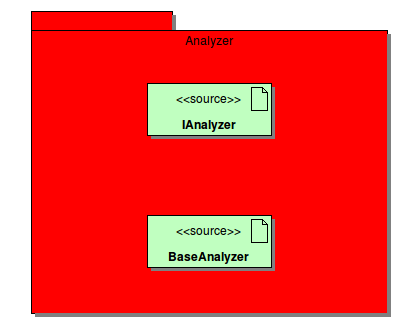
# Реализация, тестване/експерименти и (евентуално) внедряване

(10-15стр.)

## Реализация на модулите

Модулите дефинирани в Глава *5*, са генерирани посредством BoUML по указание на точки *4.1.2* и *4.3.2* следвайки структурата изобразена на *Фигура 25*, посредством следните диаграми на внедряване:

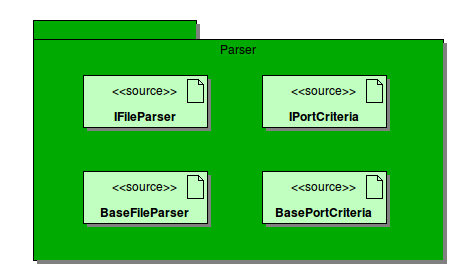
### Анализатор (диаграма на внедряване)



Фигура (Анализатор – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.3.1* в *Python* модули.

### Скенер

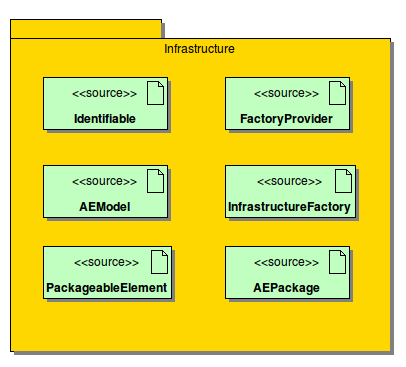


Фигура (Скенер – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.3.2* в *Python* модули.

### Мета-модел

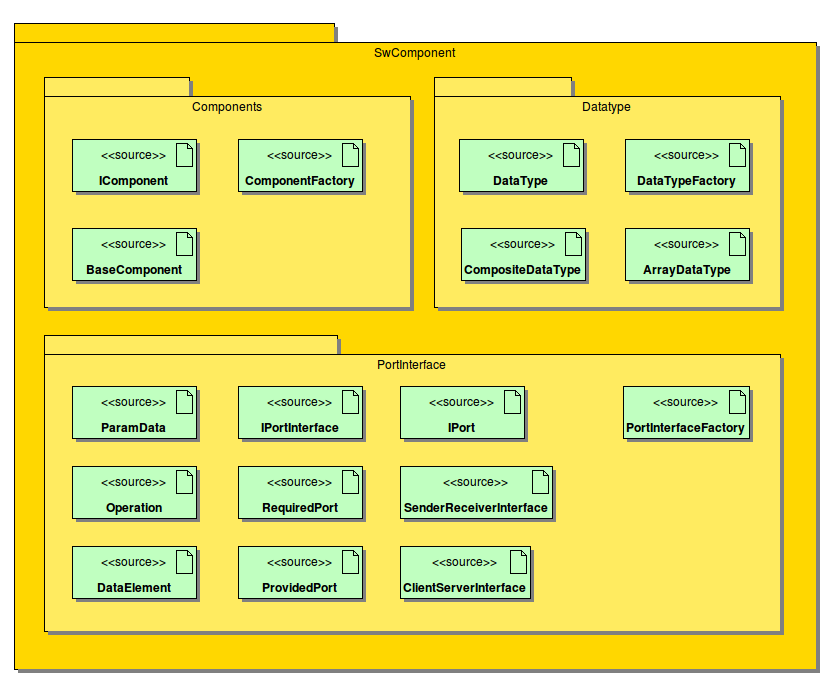
#### Инфраструктура (диаграма на внедряване)



Фигура (Инфраструктура – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.2.1* в *Python* модули.

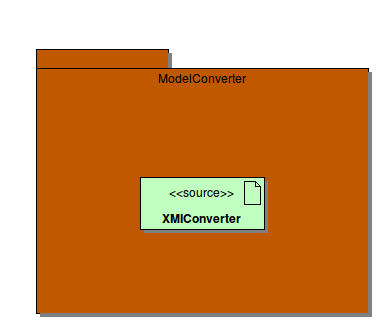
#### Софтуерен компонент (диаграма на внедряване)



Фигура (Софтуерен компонент – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.2.2* в *Python* модули.

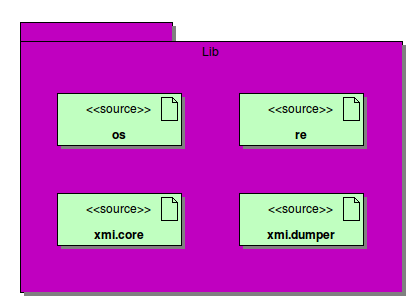
### Сериализатор (диаграма на внедряване)



Фигура (Сериализатор – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.3.4* в *Python* модули.

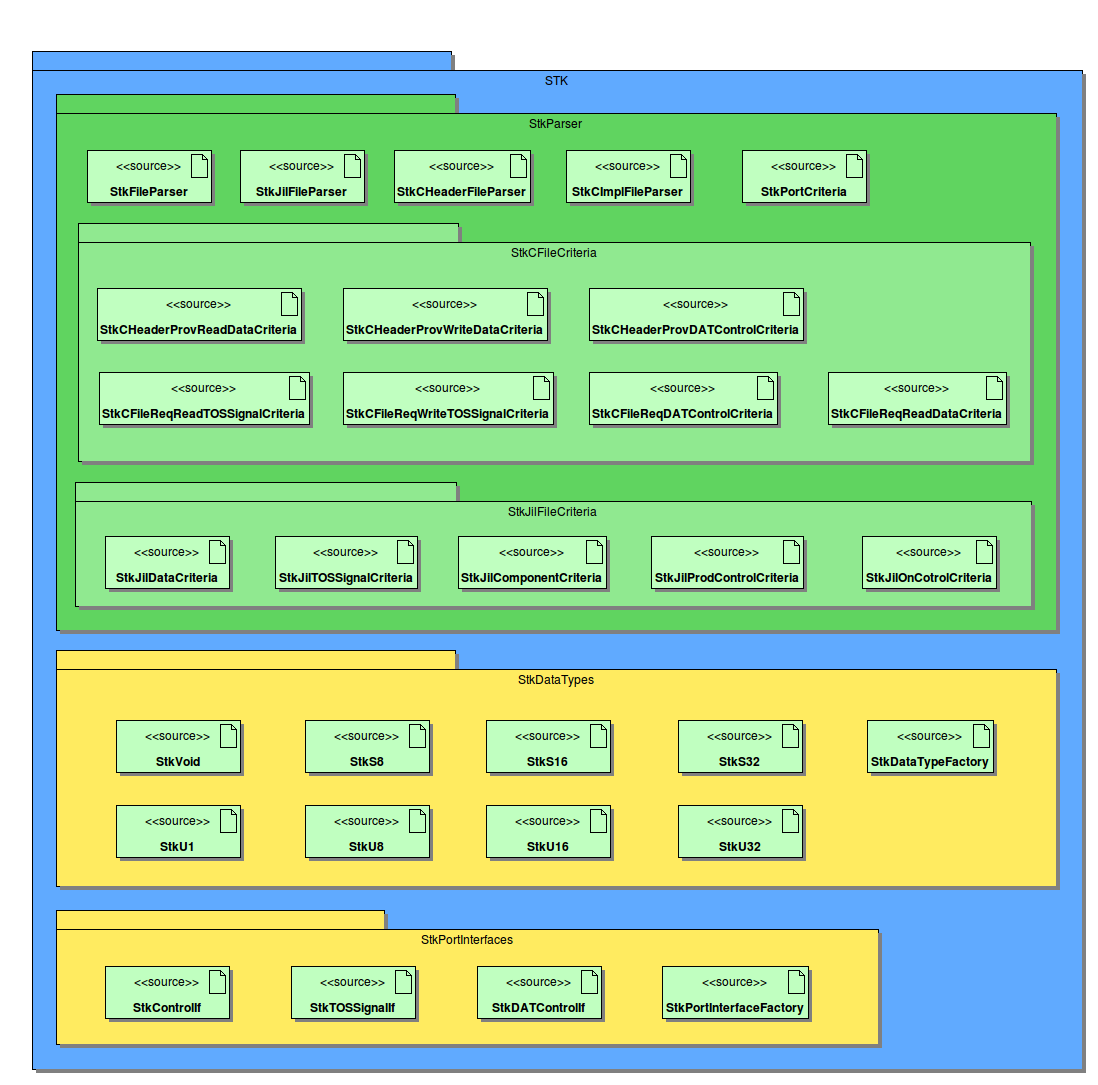
### Спомагателни модули (диаграма на внедряване)



Фигура (Спомагателни модули – диаграма на внедряване)

Представлява интерфейсите и класовете от точка *5.4* във външни *Python* модули.

### Специфични критерии за стандартна архитектура (диаграма на внедряване)



Фигура (Критерии за стандартна архитектура – диаграма на внедряване)

Представлява изискванията от точка *3.3.4* в *Python* модули.

### Генерация на базов код (реализация)

Използвайки *Eclipse* версия 4.3.2 (Kepler) и съответстващия плъгин на *Acceleo* версия (3.4.2), следвайки изискванията от точка *5.3.5*, е съсдаден проект за код генерация на базата на UML модел.

## Системна интеграция (опционално)

## Планиране на тестването - тестови сценарии, процедури, ...

### Цели

Целта на тестовия план е да осигури 100% покритие на кода чрез отделни модулни тестове. Както и да осигури функционални тестове на базата на потребилтеските изиксвания и тестове покриващи нефункционалните изисквания.

### Модулни тестове

За тестване на всички модули от точка *6.1* е създаден Acceleo проект, който генерира тестов шаблон за всеки един модул въз основа модела на системата. Тези тестови шаблони са на базата на стандартния *Python* модул за тестване: *unittest.* Генерират се три типа методи за тестване:

* Инициализация на теста (setUp) – подготовка преди изпълняване на тестовите процедури.
* Tест за всеки метод на класа, който тестване – представлява:

**def** test<име на метода, който се тества>(self):

<ръчно описване на теста>

Pass

* Преключване на тестването (tearDown) – изпълнява процедури необходими при преключването на теста

Покритието на тестовете ще се проверява посредством стандартния модул *coverage* на *Python*.

### Функционални тестове

На базата на точка *3.2* е изготвен ръчен тест проверяващ всички изисквания. След анализ на типичните случаи на употреба съставяме следните стъпки:

* *Изпълнение на анализ* (*3.3.1.5*) на даден проект
* *Сериализиране на UML модела* (*3.3.1.9*) получен от предходната точка
* *Генериране на базов код* (*3.3.1.10*) от сериализирания UML модел

### Нефункционални тестове

Да се провери дали приложението отговаря на всички нефункционални изисквания зададени в точка *3.4.*

## Модулно и системно тестване

## Анализ на резултатите от тестването и начин на отразяването им

## Експериментално внедряване (технологични изисквания, инсталиране, условия, използване, ...)

# Заключение

(1-2стр.)

## Обобщение на изпълнението на началните цели

## Насоки за бъдещо развитие и усъвършенстване

## Използвана литература

(min 10 литературни източника – статии, книги, с, форматирани съгласно MLA Style - <http://www.library.mun.ca/guides/howto/mla.php>)

|  |  |
| --- | --- |
| Референция | Описание |
| R1 | COM Specification.  (<http://www.microsoft.com/com/default.mspx>) |
| R2 | Corba. Object Management Group.  ([http://www.omg.org](http://www.omg.org/)) |
| R3 | CCM: Corba Component Model”, OMG, August 1999 |
| R4 | JavaBeans specification v1.01, Sun Microsystems, August 8, 1997  (<http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/>) |
| R5 | J.M. Favre, “GSEE: a Generic Software Exploration  Environment”, submitted to the International Workshop on  Program Comprehension (IWPC’2001), May 2001.  (<http://www.megaplanet.org/jean-marie-favre/papers/IWPC01F-37-final.pdf>) |
| R6 | Entreprise Java Bean v3.1, Sun Microsystems, November 5, 2009,  (<http://download.oracle.com/otndocs/jcp/ejb-3.1-fr-eval-oth-JSpec/>) |
| R7 | Virtual Functional Bus, AUTOSAR, release 4.2.1,  (http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR\_EXP\_VFB.pdf) |
| R8 | Reverse Engineering a Large Component-based Software Product, Jean-Marie Favre, Frédéric Duclos, Jacky Estublier, Remy Sanlaville, Jean-Jacques Auffret, 2001 |
| R9 | ARES Conceptual Framework for Software Architecture” in M. Jazayeri, A. Ran, F. van der Linden (eds.), Software  Architecture for Product Families Principles and Practice, Addison Wesley, 2000. |
| R10 | Riva C., Reverse Architecting: an Industrial Experience Report, Proceedings. of the 7th Working Conference on Reverse Engineering  (WCRE2000), Brisbane, Australia, 23-25 November, 2000. |
| R11 | Programming languages ranked by expressiveness,  ([redmonk.com](file:///C:\users\crossover\Application%20Data\Microsoft\Word\redmonk.com\dberkholz\2013\03\25\programming-languages-ranked-by-expressiveness\)), Donnie Berkholz, 25 March, 2013 |
| R12 | Regular expression tools / Programming Languages and Libraries (<http://www.regular-expressions.info/tools.html>), Jan Goyvaerts, 22 October 2013 |
| R13 | Domain-Specific Modeling and Model Driven Architecture (<http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Modeling%20Frankel-Cook.pdf>), Steve Cook, January 2004 |
| R14 | An Orchestrated Multi-view Software Architecture Reconstruction Environment, Kamran Sartipi, Nima Dezhkam and Hossein Safyallah, 2006 |
| R15 | Software Metrics - A Rigorous And Practical Approach (2Nd Ed), N. Fenton & S. Pfleeger, 1997 |
| R16 | Structured Design, Yourdon and Constantine, 1979 |
| R17 | An Introduction to Software Architecture, D. Garlan, M. Shaw, 1994 |
| R18 | Foundation for the Study of Software Architecture, D. Perry, A. Wolf, 1992 (<http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf>) |
| R19 | Reverse engineering and design recovery, E. Chykofsky, J. Cross II, 1990 (<http://www.eecs.yorku.ca/course_archive/2006-07/F/6431/Chikofsky.pdf>) |
| R20 | Software maintenance management: Changes in the last decade, Nosek, J.T. and P. Palvia |
| R21 | Application program maintenance study: Report to our respondents, R. K. Fjeldstad and W. T. Hamlen, 1983 |
| R22 | An Embedded Software Primer, David E. Simon, 1999 |
| R23 | Software Architecture in Practice, Second Edition, Len Bass, Paul Clements, Rick Kazman, 2003 |
| R24 | Community Software Architecture Definitions, Carnegie Melon University, 2015 |
| R25 | Component Software: Beyond Object-oriented Programming, Clemens Szyperski, 2002 |
| R26 | UML Infrastructure specification, OMG, formal/2011-08-05 |
| R27 | OMG MOF 2 XMI Mapping Specification, OMG, formal/2013-06-03 |
| R28 | ”Eclipse MDT/UML2 as XMI de facto standard?”, 2009, Jordi Cabot  (http://modeling-languages.com/eclipse-mdtuml2-xmi-de-facto-standard/) |

# Приложения

1. Терминологичен речник

| **Дефиниция** | **Описание** |
| --- | --- |
| AUTOSAR | Софтуерне архитектурен стандарт за автомобилна електроника  (<http://www.autosar.org/about/technical-overview/>) |
| BSW | Basic Software |
| CDD | Complex Device Driver |
| SAD | Software Architecture Design |
| SDP | Software Development Plan |
| SSR | Software Status Report |
| SRS | Software Requirements Specification |
| SDSA | Software Design Safety Analysis report (SDSA) |
| SW-C | Software Component |
| UML | Унифициран език за моделиране (Unified Modeling Language) |
| OMG | Интернационален консорциум с идеална цел, чиито предмет на дейност е стандартизиране на компютърната индустрия ([Object Management Group](http://www.omg.org/)) |
| MOFM2T | Спецификация на OMG (<http://en.wikipedia.org/wiki/MOF_Model_to_Text_Transformation_Language>) |
| MDA | Mоделно разработена архитектура ([Model Driven Architecture](http://www.omg.org/mda/)) |
| XML | разширяем маркиращ език |
| MOF | Стандарт за моделно движена разработа. Meta-Object Facility ([MOF](http://www.omg.org/mof/)) |
| E-MOF | Опростен MOF стандарт (Essential MOF) |
| XMI | XML обмен на метаданни, чиито метамодел може да се изрази чрез MOF |
| GNU | GNU е операционна система, чието име (рекурсивен акроним) означава GNU не е Unix (GNU's Not Unix)  (<http://bg.wikipedia.org/wiki/%D0%93%D0%9D%D0%A3>) |
| GPL | Общ публичен лиценз на ГНУ (GNU)  (<http://bg.wikipedia.org/wiki/GNU_General_Public_License>) |
| Eclipse | многоезична среда за разработване на софтуер, която включва интегрирана среда за разработка (IDE) и плъгин система |
| EPL | Лиценз за отворен код използван от Eclipse |
| Acceleo | Код генератор с отворен код от фондация Eclipse. Имплементация на MOFM2T стандатра (<http://www.eclipse.org/acceleo/>) |
| EMF | Среда за моделиране на Eclipse  (<http://eclipse.org/modeling/emf/>) |
| Perl | универсален, интерпретиран език за програмиране  (<http://www.perl.org/>) |
| PCRE | Библиотека за работа с Perl съвместими съвместими регулярни изрази  (<http://www.pcre.org/>) |
| MDT | Инструменти за разработка на модели на Eclipse  (<http://www.eclipse.org/modeling/mdt/?project=uml2>) |
| OCL | Декларативен език даващ възможност за описание на правила, които да се прилагат над UML. В последствие той може да се прилага и над всеки MOF мета модел  (<http://www.omg.org/spec/OCL/>) |
| PyEMOF | Python имплементация на E-MOF спецификацията  (<http://www.lifl.fr/~marvie/software/pyemof.html>) |
| CAD | Системи за автоматизирано проектиране (Computer-aided design) |
| CAM | Системи за автоматизирано производство (Computer-aided manufacturing) |
| CAE | Системи за автоматизиране на инжинерни задачи (Computer-aided engineering) |
| UUID | Универсален уникален идентификатор (universally unique identifier) |
| EBNF | Разширена Бакус-Наур Форма (Extended Backus–Naur Form) |
| IBM | Американска мултитехнологична и конслултираща корпорация  (http://www.ibm.com) |
| IBM Rational | Подразделение на IBM занимаващо се с модулни архитектури и итеративна разработка  (http://www-01.ibm.com/software/rational/) |

1. Реализирани документи

| **Реф.** | **Име на документ** | **Връзка** |
| --- | --- | --- |
| **[D1]** | UML модел | <https://github.com/dmanev/ArchExtractor/tree/master/ArchExtractor/model> |
| **[D2]** | UML генерирана документация | <https://github.com/dmanev/ArchExtractor/blob/master/ArchExtractor/Documents/umlGenDoc.zip> |
|  |  |  |

1. Степен на изразителност на езиците за програмиране



Фигура (графиката представя средно броя на редактирани (добавени/модифицирани/изтрити) линии код в рамките на една планирана промяна [R11])

*Забележка:* В диаграмата се взима средно предвид броя редактирани линии код в рамките на една планирана промяна. Т.е. колкото по-голям е броя на променените линии толкова езика за програмиране е по-малко изразителен и обратно.

1. “Ръчно” написан код генератор

Този тип код генератор се очаква да обхожда даден съвместим модел и на базата на елементите на този модел да отпечатва и попълва предварително заготвени файлове (шаблони). Като за целта може да се използват класове, които да представляват всеки файлов шаблон, с метод за отпечатване:

void generate(in aModel: Model; out outFile: File)

…

printHeader(aModel, outFile)

printContents(aModel, outFile)

printFooter(aModel, outFile)

…

Характерното за такъв тип генератори е ограничената им адаптивност към промени в модела и не особено лесната промяна при необходимост.

1. Шаблони за генериране на базов код

* RteHeader – rte.h

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Company: Personal use

// ---------------------------------------------------------------------------------

// Copyright: GPL v3

// ---------------------------------------------------------------------------------

// Project: RTE Simualation

// Language: ANSI-C

// ---------------------------------------------------------------------------------

// Component: RTE

// ----------------------------------------------------------------------------------

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#ifndef I\_RTE\_H

#define I\_RTE\_H (1)

//------------------------------------------------------------------------------

// Exported Macros

//

// #define LAYCmpMacroName(...) (MacroDefinition)

//------------------------------------------------------------------------------

#define PVAR(TYPE, VARIABLE) ((TYPE) \*(VALUE))

#define PDATA(VARIABLE) (\*VARIABLE)

//------------------------------------------------------------------------------

// Exported functions

//

// extern tReturnType LAYCmpFunctionName(tTypeArgument1 ArgumentName1, ... );

//------------------------------------------------------------------------------

***<List SenderReceiverInterface's getter/setter methods external definition>***

***<List ClientServerInterface's provided methods external definition>***

#endif // I\_RTE\_H

* RteImplementation – rte.c

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// Company: Personal use

// ---------------------------------------------------------------------------------// Copyright: GPL v3

// ---------------------------------------------------------------------------------// Project: RTE Simualation

// Language: ANSI-C

// ---------------------------------------------------------------------------------// Component: RTE

// ----------------------------------------------------------------------------------//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//------------------------------------------------------------------------------

// Body Identification

//------------------------------------------------------------------------------

#define RTE\_C "RTE\_C"

//------------------------------------------------------------------------------

// Included files to resolve specific definitions in this file

//------------------------------------------------------------------------------

#include "rte.h"

// -----------------------------------------------------------------------------

// Local data

//

// static tType u8VariableName;

// static tType\* pu8VariableName;

//------------------------------------------------------------------------------

***<List SenderReceiverInterface’s DataElement variables>***

//==============================================================================

//=========================== EXPORTED FUNCTIONS ===============================

//==============================================================================

***<List SenderReceiverInterface's getter/setter methods’ implementation>***

// [EXPORTED\_FUNCTIONS\_END]

//------------------------------------------------------------------------------

// End of file

//------------------------------------------------------------------------------

* RteComponentHeader - rte\_<component name>.h:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// Company: Personal use

// ---------------------------------------------------------------------------------// Copyright: GPL v3

// ---------------------------------------------------------------------------------// Project: RTE Simualation

// Language: ANSI-C

// ---------------------------------------------------------------------------------// Component: RTE

// ----------------------------------------------------------------------------------//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#ifndef I\_***<component name>***\_H

#define I\_***<component name>***\_H (1)

//------------------------------------------------------------------------------

// Included files to resolve specific definitions in this file

//------------------------------------------------------------------------------

#include "rte.h"

//------------------------------------------------------------------------------

// Local macros

//

// #define mMacroName (MacroDefinition)

//------------------------------------------------------------------------------

***<List provided SenderReceiverInterface's getter/setter methods>***

***<List required SenderReceiverInterface's getter/setter methods>***

***<List ClientServerInterface's required methods>***

#endif // I\_***<component name>***\_H

* RteComponentImplementation - <component name>.c:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// Company: Personal use

// ---------------------------------------------------------------------------------// Copyright: GPL v3

// ---------------------------------------------------------------------------------// Project: RTE Simualation

// Language: ANSI-C

// ---------------------------------------------------------------------------------// Component: [aComponent.name.toUpper()/]

// ----------------------------------------------------------------------------------//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//------------------------------------------------------------------------------

// Body Identification

//------------------------------------------------------------------------------

#define ***<component name>***

//------------------------------------------------------------------------------

// Included files to resolve specific definitions in this file

//

// #include <system\_file\_name.h>

// #include "project\_file\_name.h"

//------------------------------------------------------------------------------

#include \<rte\_***<component name>***.h\>

//==============================================================================

//=========================== EXPORTED FUNCTIONS ===============================

//==============================================================================

***<List ClientServerInterface's provided methods implementation>***

// [EXPORTED\_FUNCTIONS\_END]

//------------------------------------------------------------------------------

// End of file

//------------------------------------------------------------------------------

1. Легенда на диаграмите за работни процеси



Фигура (Легенда на диаграма за работни процеси)

1. Карта на работните процеси



Фигура (Карта на работните процеси)

**Приложение 2, 3, ....:** напр. код или извадки от кода, наръчник на потребителя, екрани от потребителския интерфейс, спецификации, диаграми, и др.

**Изисквания за оформяне на дипломната работа:**

1. Това е препоръчителен шаблон, в зависимост от конкретното задание.
2. Йерархията на структуриране на съдържанието да не бъде повече от 3 нива, номерирани с арабски цифри – напр. 1.2.3.
3. Чуждестранните термини да бъдат преведени, а където това не е възможно – цитирани в *курсив* и нечленувани.
4. Страниците да бъдат номерирани с арабски цифри, в долния десен ъгъл.
5. Използваният шрифт за основния текст на описанието да бъде Times 12 или Arial 10, и Courier 9 за кода, с междуредие 16pt.
6. Да се избягват пренасянията на нова страница на заглавия на секции, фигури и таблици.
7. Да се избягват празни участъци на страници вследствие пренасянето на фигури на нова страница.
8. Всички фигури и таблици да бъдат номерирани и именовани (непосредствено след фигурата или таблицата).
9. Всички фигури и таблици да бъдат цитирани в текста.
10. Използваните фигури от други източници да бъдат цитирани.
11. Всички цитати да бъдат отразени в списъка на използваната литература.
12. Всички източници от списъка на използваната литература да бъдат цитирани в текста.
13. Използваната литература да се цитира съгласно MLA Style - <http://www.library.mun.ca/guides/howto/mla.php>