* Заглавна страница
  + Университет, факултет, катедра
  + Тема
  + Дипломант, факултетен номер, специалност
  + Ръководител, консултант
  + Дата

Съдържание

* На тази страница всички заглавия на секции и подсекции максимум до трето ниво на вложеност се изброяват със съответния номер на страница. Помислете за нивото на подсекции, които ще включите

[1 Увод 3](#_Toc325907904)

[1.1 Основа, определяне на обхвата, конвенции 3](#_Toc325907905)

[1.2 Цел и задачи на дипломната работа 3](#_Toc325907906)

[1.3 Полза от реализацията 3](#_Toc325907907)

[1.3.1 Състояние на индустрията 3](#_Toc325907908)

[1.3.2 Важност на Софтуерната Архитектура 4](#_Toc325907909)

[1.3.3 Изледване на архитектурното възстановяване 5](#_Toc325907910)

[1.1 Структура на дипломната работа 8](#_Toc325907911)

[1.2 Използвани съкращения и означения (ако има такива) 8](#_Toc325907912)

[1. Подобна работа (Приложима теория) 9](#_Toc325907913)

[2. Анализ на проблема 10](#_Toc325907914)

[3. Модел/метод/технологии 11](#_Toc325907915)

[4. Решение 12](#_Toc325907916)

[5. Резултати, анализ на резултатите (и от експерименти и въпросници, ако има такива), препоръки 13](#_Toc325907917)

[6. Заключение и бъдеща работа 14](#_Toc325907918)

[7. Литература (цитирани писмени и Интернет източници) 15](#_Toc325907919)

[8. Приложения 16](#_Toc325907920)

# Увод

## Основа, определяне на обхвата, конвенции

## Цел и задачи на дипломната работа

## Полза от реализацията

* refs/Atomic Architectural Component Recovery - bible.pdf – Introduction

През 1985-а година Lehman и Belady изложиха така наречените Леманови закони. От тези пет, следните два закона(по-скоро хипотези), имат отношение към текущата работа:

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.

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

### 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. Средната fortune-100 компания оперира с 35 милиона линии код с нарастване 10 процента на година (Buss et al., 1994).

Изучаването на различни случаи показва, че 60-80% разходите по софтуерен продукт произлизат от еволюции на програмите (Nosek and Palvia, 1990). 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 (Fjeldstadt and Hamlen, 1984). 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.

В днешно време много хора смятат, че софтуерните продукти на големите компании трябва да се изработват като сбор от *софтуерни компоненти*. Въпреки че е обещаваща, тази идея не е приложена на практика в голям мащаб до последните няколко години и до появата на индустриално силни компонентни модели като на Майкрософт: COM [R1]; OMG: CORBA и CCM [R2, R3]; Sun: JavaBean [R4, R5] или Enterprise Java Bean [R6]. Компанията „Dassault Systèmes“ (**DS**) – световният лидер в CAD/CAM системи е инициатор в тази област. Компания развива патентован компонентен модел, който се използва успешно от години в развитието на CATIA [R8]. Добре е да се спомене също така отворената инициатива AUTOSAR [R7] набираща сила последните години в автомобилната индустрия, включваща в себе си компонентен модел. Наличността на толкова силни и иновативни компонентно-базирани техники може би ще образува основата на следващата значителна вълна на интерес към индустрията.

### Важност на Софтуерната Архитектура

Големите системи са разделени на подсистеми. Тези подсистеми, още познати като компоненти, и зависимостите м/у тях формират софтуерната архитектура на дадена система. Софтуерната архитектура е ключов актив описващ основните атрибути на дадена системата. Неуместна или влошена архитектура може да се отрази пагубно на поддръжката на една система. От гледна точка на поддръжка основните въздействия на софтуерната архитектура влияят на следните аспекти на дадена система:

**Яснота:** Софтуерната архитектура придава изглед на системата с високо ниво на абстракция. Този изглед представя на високо ниво, ограничения на системата, които трябва да са обозрими за поддържащия персонал и позволяват възможност за търсене само в архитектурната информация. Доста често неследването на оригиналния дизайн и последствията от това става ясно само на това ниво.

**Преизползване:** В архитектурата, разработчика не само може да открие компоненти, които могат да бъдат преизползвани, но той също може да открие зависимости към други части на системата, които трябва да бъдат удовлетворени при използването на даден компонент. Архитектурния дизайн доставя възможността за преизползване на компоненти и рамки, в които се интегрират компонентите. Извличането на архитектура също дава възможност за прилагането на продуктови линии основани на една и съща генерализирана архитектурна рамка.

**Еволюция:** Софтуерната архитектура може да бъде разглеждана като скелет на система, чието описание дава възможност за лесно идентифициране на потенциални проблеми при промяна на системата. Още повече описанието на зависимостите позволява промяна в компонент без да се нарушава останалата част на системата освен това дава възможност за анализ на продуктивността, съвместимост и преизползване. При наличието на тази информация не е необходимо да се третират грешките при компонентите, в които възникват, вместо това може да се проследи истинския им произход и да се коригира.

**Анализ:** Архитектурния анализ дава възможност за изследване на високо ниво на консистентността на системата, съответствието й към даден архитектурен стил, съответствие по дадени качествени критерии, също така специфични за дадения домейн анализи.

**Управление:** Задачи за поддръжка могат да бъдат давани на базата на дадена подсистема. Още повече софтуерната архитектура дава основа за реалистични естимации на разходите и рисковете на дадена промяна. Качеството на дадена система може да бъде оценено на базата на капацитета на архитектурата. Слабите части на системата могат да бъдат идентифицирани и измерени и подобрени. За дадени компоненти може да се вземе решение да бъдат редактирани или пренаписани отново.

Тъй като всички тези фактори са важни за развиването на системата, описание на архитектурата трябва да бъде възстановявано в случай, че такова липсва или е „изчезнало”. Най-добре такъв документ да се поддържа постоянно, включващ бъдещи промени. Въпреки това е възможно да се наложи инспекция на архитектурата на системата за да се видят разлики с документираната архитектура.

### Изледване на архитектурното възстановяване

Архитектурното възстановяване се състои в откриване на **компоненти** (изчислителни части) и **конектори** (точки на комуникация) в система.

**Възстановяване на компоненти:** Основна тема във възстановяването на компоненти е разпознаването на подсистеми (Schwanke, 1991), друга е възстановяването на обекти и абстрактни типове на данни. Въпреки, че разпознаването на абстрактни типове и обекти често се използва като техника за преизползване информацията, която ни дава е на ниско ниво.

**Възстановяване на конектори:** Конектори за последователни и паралелни системи са основните цели на възстановяването на конектори. Въпреки, че наследените системи са често последователни и монолитни. Извикване на функции е най-първичния и популярен тип конектор в такъв тип системи. Друг често срещан тип комуникация м/у подсистеми е чрез споделени глобални променливи. След конекторите на по-високо ниво откриваме атомарните компоненти.

## Структура на дипломната работа

## Използвани съкращения и означения (ако има такива)

* Примерни въпроси, на които отговаря тази секция са:
  + Извършвана ли и някаква работа в тази област преди и ако да, каква?
  + Какви са целите на тази работа?
  + Защо тази работа би била интересна за някоя организация?
  + Има ли някаква специална нотация, която се използва в дипломната работа?

# Подобна работа (Приложима теория)

* + Тази секция е за да даде теоретична основа, която да позволи да се разберат аргументите в следващите части от работата.
  + Тази секция също обобщава друга подобна работа, която съществува в областта, как тази работа е била свършена, какви резултати съществуват. Също референции към съответната литература.
  + Тази секция е за да покажете, че сте компетентен в областта на дипломната работа и че познавате съществуващата работа.
  + [RIGI](http://www.rigi.csc.uvic.ca/)
  + [Software Architecture Mining](https://researcher.ibm.com/researcher/view_project.php?id=469)
  + refs/ An Orchestrated Multi-view Software Architecture Reconstruction Environment.pdfw
  + refs/ Workshop on Dynamic Analysis.pdf - Program Analysis: A Hierarchy (1. Introduction)
  + refs/Reverse Engineering a Large Component-based Software Product.pdf

## Компонентният модел на DS

В средата на 90-те, когато DS започват развитието на CATIA V5, се разбира, че ОО технологията има сериозни ограничения и че C++ не отговаря на всички изисквания. Двата най-важни аспекта са следните:

* **едновременно проектиране.** Единиците С++ са прекалено сродни: дори една минимална промяна може да предизвика голям брой прекомпилирания. За продуктите на големите компании и ограниченията при едновременното 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 „Обектов моделер“ (ОМ). Въпреки името си, ОМ се разглежда най-успешно като компонентен модел. Този раздел представя главните ОМ концепции и дава информация относто реализирането им.

### Концептуално ниво

ОМ 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 заявления и които ги препращат към компонента, допълващ интерфейса. Концепцията за интерфейса помага в подхода към проблема на едновременното 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* (реализират се 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.

ОМ осигурява и още няколко механизма, които не са описани в този проект. Например подкрепя концепцията за упълномощаване или условно изпълнение.

### Ниво на реализация

Всички концепции на ОМ се имплементират по отношение на С++ единици. Например, и интерфейсите, и реализациите се представляват от С++ класове. Всъщност нивото на реализация е много по-сложно, тъй като картографирането не е едно към едно: реализацията на една ОМ единица може да създаде много С++ единици. Още повече, за дадена концептуална единица има много възможности за реализация: за да подобри изпълнението и да отговори на други нефункционални изисквания, компанията DS проектира и изпробва широк кръг от релизационни техники. Всички те позволяват създаването на подходящи компоненти, но в същото време разработването и поддръжката на тези компоненти е доста сложна задача.

За да се контролира софтуерът, ОМ концепциите се превеждат в С++ код, с помощта на шаблони. Този подход много си прилича с подходите на други компонентни модели (пример [32]). В случая на ОМ, се добавя допълнителна информация в изходния код чрез макрота (macros). Това облекчава постоянното писане на различни кодове. Някои от тях също се създават автоматично.

Допълнителна информация може да се намери и в отделни текстови файлове, наречени „*речници*“, които съдържат кортежи „компонент – интерфейс – dll“. Тези файлове позволяват откриването и зареждането само на нужните DLL, които се изискват по време на изпълнението и следователно – подобряването на това изпълнение.

### Често срещани проблеми

ОМ 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). ОМ концептуалните единици са смесени с огромно количество С++ кодове.
* **Нужда от централизирано описание.** Информацията за една ОМ единица често се разпределя сред много различни файлове, включително и изходен код и речници.
* **Нужда от формализация.** ОМ компонентният модел неформално се определя чрез огромна документация. Независимо от това, че е много ценна, тази документация често се оказва неточна, а много от реализационните ограничения (constraints) 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 ОМ 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 на RE3 домейна. Преди да се опитваме да разработим инструмент за реверсивен инженеринг, първо трябва да намерим точна дефиниция на това какво представлява един компонент. Това е направено в следващия раздел.

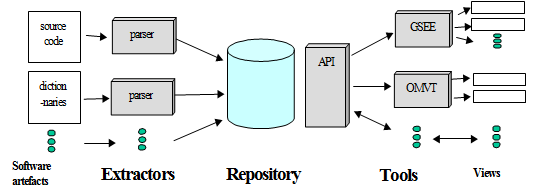
### Създаване на инструменти за реверсивен инженеринг

Създаването на мета-модел не само подобрява разбирането на компонентния модел, но и осигурява много добра основа за изграждане на реверсивно-инженерна платформа, върху която може да се изгради голяма поредица инструменти, вариращи от обикновени инструменти за визуализация до инструменти за сложен анализ или преструктуриране. Това включва, наример, инструменти, засичащи констрейнт нарушение. Разработката на тези инструменти от нулата определено не е рентабилна. За щастие, от мета-модела може да бъде извлечена обща платформа.

#### Реверсивен инженеринг платформа

Фигура показва по-опростен поглед към цялостната архитектура на платформата, която сме изградили. Тази традиционна архитектура за подобна среда на реверсивен инженеринг е съставена от следните части:

* **Екстрактори:** Първата стъпка е да се извлече информация от конкретни софтуерни артефакти. В нашия случай се анализират изходният код и речниците.
* **Репозитори** (repository – склад): Репозитори играе централна роля в обстановката. Една от важните черти на нашия подход е че структурата на това репозитори се извлича директно от мета-модела.
* **Инструменти:** Инструментите генерират различни изгледи на репозитори. 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 - Платформа на реверсивен инженеринг

Следващият раздел показва как мета-моделът е използван за изграждането на изгледи, които показват компоненти, с помощта на различни техники за визуализиране. След това показва как да се откриват несъответствия чрез употребата на специфични изгледи. Експериментът е извършен върху версия на софтуер CATIA, който съдържа 4038 компонента, съставени от 8155 имплементации и имплементиращи 2504 интерфейси. Тези цифри отговарят на едно високо абстрактно ниво. Реализационното ниво е далеч по-сложно. В конкретния случай, са включени 49821 C++ класове в конкретната репрезентация на тези компоненти.

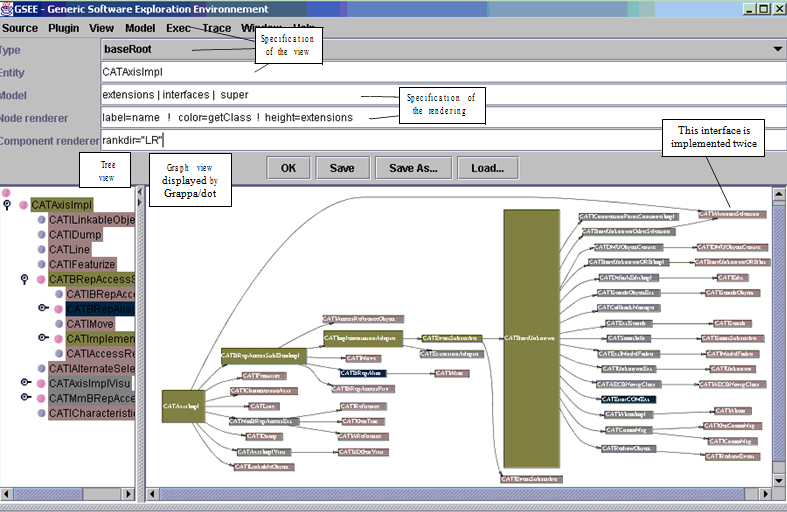
#### Пример за инструменти за визуализация

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 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. Имплементацията й не съдържа нито една LOC (Line of Code), свързанa с ОМ. Петте реда в горната част на прозореца съставят цялата спецификация на изгледа под формата на текст. Всички роли 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: новите изгледи могат да бъдат създадени интерактивно, само чрез промяна на редовете за спецификация и натискане на ОК.



фигура - Компонент, показан от средата за изследване на универсален софтуер GSEE

# Анализ на проблема

* + Тази секция се пише след теоретичната част и след като е формулиран проблема. Тук първоначалния проблем се разделя на няколко под проблема според някакви критерии. Резултата от този анализ служи като основа за избора на модел или метод в следващата секция.
  + refs/Issues in Reengineering the Architecture of Evolving Component-Based Software.pdf – структуриран и пълен текст
  + refs/SW Arch Recovery and Modelling - brief.pdf
  + refs/Towards an Architecture for Refactoring Embedded Software for Ubiquitous Environments - brief.pdf
  + refs/An Environment for Architecture Reconstruction - brief.pdf

# Модел/метод/технологии

* + Тук се описват различните методи и начини за решаване на проблема. Изборът на модел или метод трябва да се мотивира. Защо определени модели, методи и/или технологии са избрани за този проблем? Ако модела не съответства на проблема напълно, то трябва да се представи разширен модел.
  + refs/Atomic Architectural Component Recovery - bible.pdf - Components in the Programming Language C

# Решение

* + Тук се очертава възможно решение и се обясняват в детайли основните принципи. Решението трябва да е свързано с метода, модела и технологиите, избрани в предишната секция.

# Резултати, анализ на резултатите (и от експерименти и въпросници, ако има такива), препоръки

* + В тази секция резултатите се представят, обясняват, обобщават и дискутират. Също така изводи, които могат да се направят на базата на резултатите. Тези изводи служат за основа за препоръките, които представяте в дипломната работа. Критичен анализ на работата ви трябва да присъства, за да може читателя да се ориентира какви са проблемите, с които сте се сблъскали, как сте подходили към тях и евентуално решили.

# Заключение и бъдеща работа

* + Тази секция обикновено е кратка (2-3 стр.) и повтаря части от формулирането на проблема с кратко обобщение на постигнатите резултати
  + Как проблемите са били решени и какво е постигнато в тази работа.
  + Евентуална бъдеща работа - очаква се да дадете препоръки за бъдеща работа, която може да се извърши в тази област.
  + refs/ Workshop on Dynamic Analysis.pdf - General

# Литература (цитирани писмени и Интернет източници)

|  |  |
| --- | --- |
| **Референция** | **Описание** |
| R1 | COM Specification.  (<http://www.microsoft.com/com/default.mspx>) |
| R2 | Corba. Object Management Group.  (<http://www.omg.org>) |
| R3 | “CCM: Corba Component Model”, OMG, August 1999 |
| R4 | R. Englander; “Developing Java Beans”, O'Reilly &  Associates. Jun. 1997. |
| 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, Sun,  (<http://java.sun.com/products/ejb>) |
| R7 | AUTOSAR Technical Overview  (<http://autosar.org/download/R3.2/AUTOSAR_TechnicalOverview.pdf>) |
| R8 | Reverse Engineering a Large Component-based Software Product  (<http://adele.imag.fr/Les.Publications/intConferences/CSMR2001Fav.pdf>) |

# Приложения