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

**ДИПЛОМНА РАБОТА**

на тема

„Име на темата”

Дипломант: **Име Презиме Фамилия**

Специалност: **Софтуерни технологии**

Факултетен номер: **M?????**

Научен ръководител:

**доц. д-р Име Фамилия**

Консултант (ако има такъв):

**Име Фамилия**

София, 20?? г.

**Съдържание** (препоръчителен обем без приложенията: от 60 до 80 стр.)

[1. Увод (3-5стр.) 4](#_Toc397093049)

[1.1 Актуалност на проблема и мотивация (0,5-1стр.) 4](#_Toc397093050)

[1.2 Цел и задачи на дипломната работа (1-2стр.) 4](#_Toc397093051)

[1.3 Очаквани ползи от реализацията (1-2стр.) 4](#_Toc397093052)

[1.3.1 Състояние на индустрията 4](#_Toc397093053)

[1.3.2 Важност на Софтуерната Архитектура 5](#_Toc397093054)

[1.3.3 Изледване на архитектурното възстановяване 5](#_Toc397093055)

[1.4 Структура на дипломната работа (0,5-1стр.) 6](#_Toc397093056)

[2. Преглед на предметната област 7](#_Toc397093057)

[2.1 Основни дефиниции 7](#_Toc397093058)

[2.1.1 Нотация (Atomic Architectural Component Recovery – bible) 7](#_Toc397093059)

[2.1.2 Терминология в софтуерната архитектура 8](#_Toc397093060)

[2.2 Подходи, методи (евентуално модели и стандарти) за решаване на проблемите 11](#_Toc397093061)

[2.3 Съществуващи решения (практически реализации) 11](#_Toc397093062)

[2.4 Избор на критерии за сравнение и сравнителен анализ на решения/методи/стандарти/... 11](#_Toc397093063)

[2.5 Изводи 11](#_Toc397093064)

[3. Използвани технологии, платформи и/или методологии 12](#_Toc397093065)

[3.1 Изисквания към средствата (технологии, платформи и методологии) 12](#_Toc397093066)

[3.2 Видове средства (технологии, платформи и методологии) и начин и място за използването им – сравненителен анализ 12](#_Toc397093067)

[3.3 Избор на средствата (технологии, платформи и методологии) 12](#_Toc397093068)

[3.4 Изводи 12](#_Toc397093069)

[4. Анализ 13](#_Toc397093070)

[4.1 Концептуален модел 13](#_Toc397093071)

[4.2 Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) 13](#_Toc397093072)

[4.3 Качествени (нефункционални) изисквания (като напр. преносимост, използваемост, скалируемост, поддръжка, ...) 13](#_Toc397093073)

[4.4 Работни (бизнес) процеси 13](#_Toc397093074)

[4.5 Изводи 13](#_Toc397093075)

[5. Проектиране 14](#_Toc397093076)

[5.1 Обща архитектура – напр. слоеве, модули, блокове, компоненти... 14](#_Toc397093077)

[5.2 Модел на данните (напр. база данни, файлова структура, ...) 14](#_Toc397093078)

[5.3 Диаграми (на структура и поведение - по слоеве и модули, с извадки от кода) 14](#_Toc397093079)

[5.4 Потребителски интерфейс (опционално) 14](#_Toc397093080)

[5.5 Ресурсни и спомагателни модули (опционално) 14](#_Toc397093081)

[6. Реализация, тестване/експерименти и (евентуално) внедряване 15](#_Toc397093082)

[6.1 Реализация на модулите 15](#_Toc397093083)

[6.2 Системна интеграция (опционално) 15](#_Toc397093084)

[6.3 Планиране на тестването - тестови сценарии, процедури, ... 15](#_Toc397093085)

[6.4 Модулно и системно тестване 15](#_Toc397093086)

[6.5 Анализ на резултатите от тестването и начин на отразяването им 15](#_Toc397093087)

[6.6 Експериментално внедряване (технологични изисквания, инсталиране, условия, използване, ...) 15](#_Toc397093088)

[7. Заключение (1-2стр.) 16](#_Toc397093089)

[7.1 Обобщение на изпълнението на началните цели 16](#_Toc397093090)

[7.2 Насоки за бъдещо развитие и усъвършенстване 16](#_Toc397093091)

# Увод (3-5стр.)

## Актуалност на проблема и мотивация (0,5-1стр.)

## Цел и задачи на дипломната работа (1-2стр.)

## Очаквани ползи от реализацията (1-2стр.)

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 (0,5-1стр.)

# Преглед на предметната област

**(да се замени с конкретно заглавие според заданието)** (10-15стр.)

## Основни дефиниции

### Нотация (Atomic Architectural Component Recovery – bible)

Следващите 3 раздела съдържат стандартна терминология на реинженеринга. Определенията за реверсивен инженеринг, преструктуриране и реинженеринг са предложени от Крос и Чиковски (1990). Фигура 1 изобразява връзките между тези термини.



Фигура 1 (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

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 физическото състояние на системата във връзка с даден предпочитан стандарт.

#### Реинженеринг

*Реинженеринг*, познат още като реновация и регенериране, е изследването и промяната на една субектна 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. Компонентите са изчислителните части и съединители, които описват връзките между тези компоненти (Гарлан и Шоу, 1993; Пери и Улф, 1992). Основни примери за компоненти са абстрактни типове данни, задачи за „производител-потребител“, или фронт енд; примери за конектори са procedure calls, споделени глобални променливи, канали или Unix контакти.

#### Възстановяване (recovery) на архитектурата

Това е дисциплина от реверсивния инженеринг, която има за цел възстановяване на софтуерната архитектура на дадена система. Този термин се разграничава от възстановяване на дизайна. *Възстановяване на дизайна* се въвежда като термин от Тед Бигърстаф (1989) и представлява процес на пресъздаване на проектови абстракции от кодови комбинации, съществуваща документация, личен опит и общи знания относно домейни на приложенията. Бигърстаф твърди, че възстановяването в широкия си смисъл е толкова неструктуриран и непредвидим термин, че не е достатъчна само формална дедукция, следователно трябва да се добавят и други доводи като още един начин за извличане на информация. Тази информация се използва за популиране на домейн модел, който помага за разбирането на софтуера.

Възстановяването на дизайна се разграничава от източниците и информацията, с която трябва да се справи. Както казва Бигърстаф: *„Домейн моделът различава проучването на възстановяване на дизайна от повърхностните подобни постижения като реверсивния инженеринг, което автоматично абстрахира код към специфично ниво, така че да може спецификациите да се променят и автоматично да се възстанови преразгледаният код.“ (Бигърстаф, 1989 г.)*

Не става ясно кое определение за реверсивен инженеринг е имал предвид Бигърстаф. Определението за реверсивен инженеринг от Чиковски и Крос (Раздел 2.1.2) също включва възстановяване на дизайн, но то се появява след определението на Бигърстаф. Още повече, че терминът възстановяване на дизайн само предполага, че дизайнът ще бъде възстановен в буквалния смисъл на думата, а дефиницията на Бигърстаф изисква домейн модел, което ненужно стеснява значението на термина. Затова предпочитам по-неутралния термин „архитектурно възстановяване“.

#### 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 груприане 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 (queue). 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** (***cohesion***)на модул е степента, която трябва да достигнат индивидуалните му компоненти, за да извършат същата задача (Фентън и Флигер, 1997 г.). **Функционалното обвързване** (***coupling***)е степента на взаимна зависимост между модулите (Йордън и Константин, 1979).

Не съществува стандартно определение за модул. Йордън и Константин (1979), например, предлагат следната дефиниция:

***Модулът*** *е прилежаща поредица от програмни изрази, свързани чрез елементи и имащи сумарен идентификатор.*

Това определение от края на 70-те, когато структурният дизайн беше предложеният метод, днес много прилича на определението за *функция*. Днес имаме програмни езици, които потвърждават концепцията за модул. Пример е Модула-2, който не само съдържа думата „модул“ в синтаксиса, но и в името си (Уърт, 1985). Модулът в съвременните програмни езици е синтактична единица, която поддържа капсулирането. Съставена е от интерфейс на изнесените части и незадължителна скрита имплементация. Изнесените 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 **атомен 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. **Атомен компонент** наричаме не-йерархически компонент, съставен от свързани общи константи, променливи, подпрограми и/или потребителски типове. За разлика от атомния компонент, една **подсистема** е йерархичен компонент, съставен от свързани атомни компоненти и/или подсистеми от по-ниско ниво.

Целта на преструктуриране на една система е да се реализира атомен компонент от един модул. Един модул имплементира само един атомен 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 на модули

Йордън и Константин (1979) изброяват следните степени на свързаност на отговорностите:

* **Функционални:** модулът извършва само една определена функция.
* **Следващи:** 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.
* **Временни:** 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: атомният компонент.

Модулите, базирани на абстрактни данни образуват отделна категория, а Макро и Бъкстън (1987) разширяват класификацията на свързаността, за да я включат. Те твърдят, че един модул притежава **абстрактна свързаност (кохезия)** точно когато е от типа абстрактни данни. Можем да обобщим по следния начин: „Един модул има абстрактна свързаност точно когато е атомен компонент.“ Добрият дизайн и преструктуриране трябва да се стремят към абстрактни кохерентни модули.

## Подходи, методи (евентуално модели и стандарти) за решаване на проблемите

* 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

### Реинженеринг и компонентно-базиран софтуер

Гореспоменатото определение не бива да обезкуражава читателят, който се интересува от CBSE. Компонентната технология все още прохожда, но пък има успехи. Тя 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.

Проблемите, за които споменахме, трябва да се възприемат като предизвикателство за RE (реинженеринг) обществото. Както казахме в раздел 2, има два основни начина, по които RE техниките могат да помогнат в контекста на една току-що появила се 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. Подкрепата на 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 същности. 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 изясниархитектурно 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 моедл 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 решения (практически реализации)

* 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

### An Orchestrated Multi-view Software Architecture Reconstruction Environment

#### Свързани дейности

Предложеното изследване в тази книга е свързано с подходите за възстановяване на софтуерна архитектура, които извеждат повече от един изгледи на софтуерната система.

Васконселос [17] представя динамичен анализ, който извежда процесен изглед и scenario изглед на Java приложения под формата на диаграма на UML поредици и диаграма на случаите на употреба. Тези изгледи допълват изведения статичен изглед чрез интеграция в средата Odyssey [2].

Рива предлага техника за възстановяване на архитектура, използвайки комбинирана статична и динамична информация. Тази техника се базира на избиране на архитектурни концепции и прилагане на абстрактни техники върху изходния код, с цел манипулиране на тези концепции на архитектурно ниво. Тази техника позволява създаването на свързани с домейна архитектурни изгледи за архитектурното описание на системата. По подобен начин ние използваме в подхода си сценарии с черти, извлечени от дизайна, за да направляваме multi-view възстановителен процес.

В подобен контекст Deursen и др. предлагат базирана на изгледа софтуерна преструктурираща рамка, която осигурява обща рамка за докладване на действията при реконструкция и сравняване на подходи за реконструкция.

Рихнер предлага доклад за извличане на статични и динамични изгледи от Java програми. Статичният изглед се добива от класови файлове и се визуализира с помощта на Rigi среда на реверсивен инженеринг. Динамичният изглед, който се представя като диаграми на сценарии, се прикача към статичната Rigi диаграма. Припокриващата се информация образува връзка за информационен обмен между изгледите.

Като цяло, значението на нашата техника се проявява в това, че генерираме три изгледа за една софтуерна система (дизайн, поведение и структура), където сценариите играят ключова роля в извършването на multi-view възстановителен процес.

#### Предложения за multi-view рамка

Предложената рамка се състои от multi-view модел и multi-view процес (фиг. 2 и съответно *фиг. 3*). Multi-view моделът предтавлява връзките между трите изгледа *дизайн, поведение и структура* в една диаграма на класовете, където сценариите са основните елементи за извличане и съвместна работа на трите изгледа. Multi-view процесът във *фиг. 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 обикновения синтаксис на израза. Структурираните сценарии се анализират, за да се генерира изглед за проектиране на софтуерната система, представлявана от 2 типа диаграми: *диаграма на същост-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 оборудваната софтуерна система образува черти, които ще извлекат *изпълнителни шаблони*. Всеки такъв шаблон е поредица от сорс код функционални искания, които са често срещани във всички сценарии.



Фигура 2 (Примерен multi-view модел [диаграма на класовете],представящ връзките между три изгледа)



*Фигура 3 (Примерен multi-view процес за извличане на 3 изгледа от една софтуерна система)*

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

Цялостният процес в multi-view рамката ни позволява да свързваме диаграми с абстрактен дизайн към конкретната имплементация на функционалните елементи на изгледа за проектиране. Трябва да се отбележи, че всеки изглед (на проектиране, поведение, структура) има собствено приложение, което ще бъде разгледано в съответния раздел. И все пак тази книга се концентрира върху съвместната работа на тези три изгледа, която се организира от сценариите.

#### Генериране на изглед за проектиране

В този раздел ще обсъдим стъпките за трансформиране на знанията в текста на сценариите в информация, свързана с проектирането и представлявана от 2 типа диаграми: диаграма „същност-връзка“ и диаграма на дейностите, използвайки процеса, изобразен на Фиг. 5. Този подход генерира и структурира редица сценарии, след което използва сценариен домейн модел, за да первърне изградените сценарии в съставки на усвоените проектни диаграми. Този процес се състои от 3 стъпки:

**Стъпка 1 (генериране на сценарий)**

Сценариите са основните елементи, които огранизират предложената multi-view възстановителна рамка на архитектурата. В тази книга приемаме структурирана текстова репрезентация на сценарии, която съответства с обикновения синтаксис на Фиг. 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 Фиг. 6. Този модел трябва да осигури класова информация в сценарии от различни области на приложение. Приложихме този модел на 3 системи, включително система от ресторанти за бързо хранене, инструмент за рисуване Xfig и банкомат. Текстът на структурираните сценарии се анализира с помощта на този домейн модел и равнодействащите примери за класове в модела се записват в базата обекти. Схемата на тази база има вход за всеки клас на домейн модела, както и индексен запис като основен ключ.

Както показва Фиг. 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 констрейнт класове.



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



Фигура 5 (Генерация на изглед на дизайна базиран на задачния сценарий и сценарий базиран на домейна на модела)



Фигура 6 (Сценарий за домейн базиран модел, който се използва за сканиране на сценарий и попълни обектната база)

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 и завимисост на действието.*

Предложеният домейн модел на Фиг. 6 включва класов *констрейнт (класово ограничение).* Той се състои от информация за ограниченията, които могат да се свържат със случаи от всеки подклас на *Информация, Действие, Зависимост.* Примери за тези констрейнти включват: *капацитет, обхват на стойност, редно 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 базата.

* *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 7 (Възтановка на поведенчески изглед базиран на шаблони на изпълнение използван да идентифицира функционалности на дадена част от кода)



Фигура 8 (Колекция от 7 опитини изпълненя. Различните типове оцветени полета отговарят на три различни шаблона на изпълнение)

Поради ограниченото място тук не са представени подробности за генериране на диаграми „Същност-връзка“ и диаграми на дейностите.

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

#### Възстановяване (recovery) на поведението

Фиг. 7 показва стъпките за рекавъри на поведението на една софтуерна система като начин за идентифициране на реализацията на софтуерните черти/свойства от функциите на изходния код. Стъпките са:

**Стъпка 1** Както споменахме в раздел 3, важните черти на една софтуерна система се идентифицират като резултат от процеса на генериране на дизайна. Диаграмата на дейностите ни помага да определим редица сценарии, които изследват едно софтуерно свойство. Наричаме ги поредица от сценарии със специфични свойства. Например, в случая на софтуерна система за Drawing Tool, група сценарии, които споделят операцията „премести“, за да преместят начертана фигура на комп. екран ще устрои подобна сценарийна система със специфични свойства. Софтуерната система трябва да изработи execution traces, когато се извърши сценарий в системата. Основна пречка в анализа на една система е големият размер на execution traces, които затрудняват тази задача (анализа). Ефективното проследяване на функции за замисления сценарий е затрупано от голям брой функции от оперативната система, помощни инструменти и повторения, предизвикани от програмни луупове (loop). За да може да се работи с големия размер на чертите, премахваме всички излишни функции, причинени от луупа на програмата. Тази операция помага за разширяемостта на подхода.

В останалата част на раздела, ще опишем прилагането на техниката на извличане на последователни шаблони, за да намерим групи функции в execution traces, отговарящи на определени системни черти. В литературата 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 execution traces на програмата. Извлеченият последователен шаблон се състои единствено от прилежащи части на execution traces, а разнообразяването с други функции в рамките на шаблоните не е позволено. Тази характеристика създава смислени изпълнителни шаблони, които отговарят на основните функции, имплементиращи специфични черти на системата. Чрез предложения подход различаваме 2 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 целевата черта 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 2 типа шаблони се обсъжда по-нататък в раздела.

Фиг. 8 показва колекция от 7 опитини изпълненя и отговарящите извлечени изпълнителни шаблони, включващи и отличителни, и общи шаблони.



Фигура 9 (Извличане на структурен изглед базиран на максимална асоциативност и клъстерна оптимизация)

**Стъпка 2** (анализ на изпълнителен шаблон). Ще използваме стратегия, за да се фокусираме върху изпълнителните шаблони, които отговарят на определени черти на сценария. В този контекст, разработваме структурните характеристики на концептуалната мрежа, за да отделим функциите, характерни за една специфична черта от групата функции, които имплементират общите черти. 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 софтуерните черти сред структурните модули; 2) оценяване на структурната свързаност на софтуерните модули и 3) визуализиране на функционалното разпределение на специфични черти в мрежата. 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

Фиг. 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 определя същностите от архитектурното ниво за възстановяване на софтуерни модули и техните връзки, създаваме графична репрезентация на софтуерната система и я запазваме във фактбазата. Имайки предвид броя същности в една средно голяма софтуерна система (обикновено повече от 1000 същности), търсенето на източник на графиката е сериозен проблем. Оттук трябва да ограничиим търсещия домейн за всеки модул към група приемливи същности. Прилагаме алгоритми за разработване на данни, базирани на асоциации върху графиката в по-малки райони (търсещи домейни), където всеки такъв домейн се състои от няколко същности, които са свързани със същност от този домейн, а именно основата. След това ограничаваме търсенето на всеки модул към един или повече от тези райони.

**Стъпка 2** *(реконструкция на модула)* Извършваме оптимизация на групирането, която постепенно генерира софтуерни групирания, като модули на функции, свързани помежду си чрез внасяне и изнасяне на функции. Всеки модул се състои от едно или повече основни звена (main seeds), докато основните функции на модула и неоптимална версия на A\* търсещия алгоритъм се използват за събиране на групата от високо асоциирани функции в един модул. Мястото за търсене на модула е ограничено до функциите в търсещите домейни на съответното основно звено. Определяме метрическа система на сходства, базираща се на група същности с максимална асоциация на свойствата. Тя се определя във формата на максимална поредица същности, които имат еднакви връзки с всеки член на друга максимална поредица от същности. Оттук тези групи притежават по-голяма свързаност и са подходящи за създаване на групирания, базирани върху ENTITY ASSOCIATION SIMILARITY METRIC. Тя кодира структурното свойство на групите на максимално асоциираните същности.

Структурното възстановяване без съдействието на предложените multi-views, разчита на улеснения чрез инструменти, които генерират списък с най-високо квалифицираните основни звена (main seeds) за следващото възстановяване. Докато този метод създава свързани модули на функции, не може да създаде смислени такива, тъй като основните функции се избират на 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 Фиг. 9 основните функции от изгледа на поведение се използват, за да се създадат семантично смислени групи като системни компоненти.

Тези 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 е съставена от висококачествени модули по отношение на високата свързаност и ниското функционално обвързване. Тази 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. Тази техника 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 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

Фиг. 10 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 10 (Платформа за 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

##### 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.

Фиг. 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. 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 11 (основен изглед на GSEE)

## Избор на критерии за сравнение и сравнителен анализ на решения/методи/стандарти/...

## Изводи

# Използвани технологии, платформи и/или методологии

**(за практическото решаване на проблема)** (10-15стр.)

* 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

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

## Видове средства (технологии, платформи и методологии) и начин и място за използването им – сравненителен анализ

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

## Изводи

# Анализ

(10-15стр.)

## Концептуален модел

## Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...)

## Качествени (нефункционални) изисквания (като напр. преносимост, използваемост, скалируемост, поддръжка, ...)

## Работни (бизнес) процеси

## Изводи

# Проектиране

(10-15стр.)

## Обща архитектура – напр. слоеве, модули, блокове, компоненти...

## Модел на данните (напр. база данни, файлова структура, ...)

## Диаграми (на структура и поведение - по слоеве и модули, с извадки от кода)

## Потребителски интерфейс (опционално)

## Ресурсни и спомагателни модули (опционално)

# Реализация, тестване/експерименти и (евентуално) внедряване

(10-15стр.)

## Реализация на модулите

## Системна интеграция (опционално)

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

## 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 | 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>) |
| 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. |

**Приложения (опционално)**

**Приложение 1: Терминологичен речник (опционално)** (1-2стр.)

**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>