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

ДИПЛОМНА РАБОТА

на тема

„Изследване на взаимовръзки между софтуерни шаблони за проектиране и практическото им използване”

Дипломант: Симеон Димитров Ангелов
Специалност: Софтуерни технологии
Факултетен номер: M22891

Научен ръководител:
проф. д-р Боян Бончев

София, 2014 г.

Съдържание

Глава 1. Увод 4
1.1. Актуалност на проблема и мотивация 4
1.2. Цел и задачи на дипломната работа 6
1.3. Очаквани ползи от реализацията 7

Глава 2. Преглед на шаблоните за дизайн и взаимовръзка по между им в практиката. Предимства и недостатъци при използването им. 9
2.1. Основни дефиниции 9
2.2. Преглед на шаблоните и основни видове взаимовръзки по между им 10
2.2.1. Pattern complements (допълващи се шаблони) 11
2.2.2. Pattern compounds (модели на съединение) : 13
2.2.3. Pattern sequences (модела на последователността) : 13
2.2.4. Pattern languages (Шаблонен език) : 14
2.3. Избор на критерии за сравнения при селекция на шаблони за конкретно приложение.. 15
2.3.1. Business vs. Software 16
2.3.2. Quality attributes (качествени атрибути) 16
2.3.3. The context-sentitive 17
2.3.4. Link 17
2.3.5. Collaboration 17
2.3.6. Каталози 17
2.3.7. RADM — Reusable Architectural Decision Model 17
2.4. Изводи 18
2.4.1. Плюсове и минуси от използването на софтуерните шаблони 18
2.4.2. Защо да използваме архитектурните и дизайн шаблоните. 20

Глава 3. Използвани технологии, платформи и/или методологии 23
3.1. Изисквания към средствата (технологии, платформи и методологии) 23
3.2. Избор на средствата (технологии, платформи и методологии) 23

Глава 4. Анализ 26
4.1. Концептуален модел 26
4.2. Качествени параметри постигани с прилагането на шаблони в различни взаимоотношения по между им — анализ и дизайн 28
I. Цел: простота на имплементацията без използването на набор от шаблони 31
II. Цел: силно и стриктно разграничаване на слоевете и компонентите с въвеждането на изграждане на интерфейси; по-ефективна тестова стратегия 33
III.Цел: осигуряване по висока степен на защита, абстракция при работа с плащането и гъвкав избор в контекста на слоева архитектура 35

IV. Цел: Пълно удовлетворяване на функционалните изисквания в шаблонно ориентирано програмирано. Достигаме на набор от качествени атрибути, включително и архитектурните драйвери заложени в системата ... 38
4.3. Пример за създаване на изцяло нов шаблон : Integration Payment Provider шаблон, като комбинация от G&F шаблони ... 41
4.4. Бизнес процеси ... 44
4.5. Извод ... 46

Глава 5. Проектиране ... 48
5.1. Обща архитектура — слоеве, модулна декомпозиция и deployment стуктура ... 48
5.1.1. Слоеве ... 48
5.1.2. Модулна декомпозиция ... 50
5.1.3. Deployment структура - изглед ... 54
5.2. Модел на данните (база данни) ... 54
5.3. Диаграми (на поведение - по модули, с извадки от кода) ... 55
5.4. Изключения и обработване на грешки ... 61

Глава 6. Реализация, тестване/експерименти ... 64
6.1. Реализация на модулите ... 64
6.3. Планиране на тестването - тестови сценарии, процедури ... 65
6.4. Модулно и системно тестване ... 65

Глава 7. Заключение ... 67
7.1. Обобщение на изпълнението на началните цели ... 67
7.2. Насоки за бъдещо развитие и усъвършенстване ... 67

Източници ... 68

Глава 1. Увод

1.1. Актуалност на проблема и мотивация

Създаването на софтуер в съвременността безспорно се свърза с бързина, качество, ефективност, преизползваемост и други не-функционални качествени характеристики, които се явяват като предусловия за започването на какъвто и да е софтуерен проект.

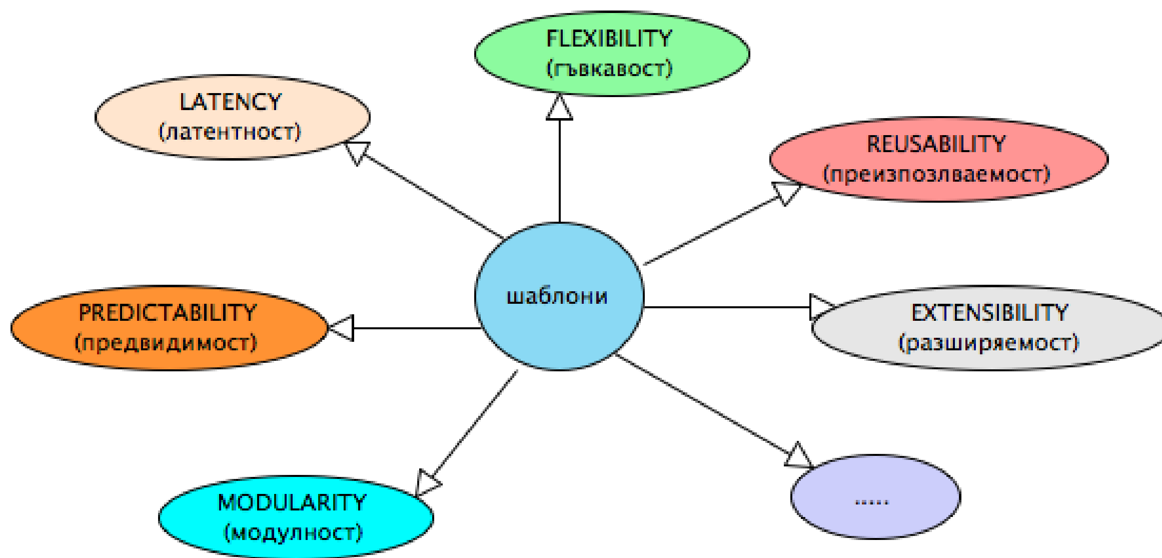
Това предсказва основата на имплементирането на вече готови/или полуготови решения, представящи се най-често с дефиницията на шаблон. Шаблонът е образец на софтуерно решение на проблем в определен контекст и домейн. Той е от решаващо значение при имплементирането на комплексни казуси при разработката на индустриални програмни приложения. Обектно-ориентираните шаблони за проектиране касаят дизайна, комуникацията и синхронизацията между обекти. Те предлагат елегантни решения на типови проблеми в проектирането с възможност за многократно използване. С тях се постига единен речник в софтуерния инженерен бранш. Непрекъснато развиващата се терминология улеснява навлизането и разбирането на домейна на проекта от всички ИТ разработчици. Този речник е и основа на т. нар. шаблонен език: структуриран метод за описание на добри дизайн практики в конкретна област [цитат]. С това този подход става предпоставка за решаването на много големи и сложни проблеми при проектирането на софтуер. В дизайните за шаблони, езикът е описание, съвкупност от шаблони. С тях може да се структурира изцяло темплейт-ориентирана софтуерна разработка.

При използването на шаблони трябва винаги първо много добре да се изследва домейна и проблема, който се решава. В резултат на задълбочения анализ, ще произлезе и структурата, връзките от шаблони и тяхната последователност на използване.

Често дори самите приложни рамки използват шаблони. В резултат на това, разработчиците използват дефинираната структура и правила, за да могат ефективно да изграждат проекта.

Преимуществата на използването на шаблони са:

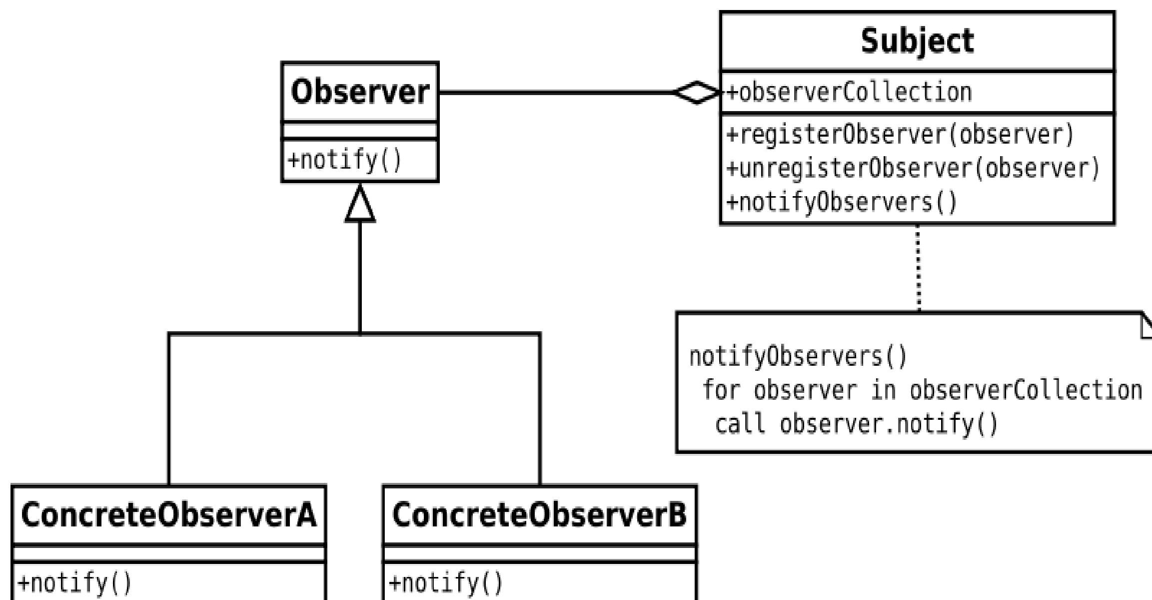
- предоставят елегантно структурирани парчета софтуер (като класове, методи и комбинация от тях), които при правилна употреба решават ефективно проблем в даден контекст
- те са мощно средство за постигане на баланс между ключовите дизайн не-функционални изисквания:



фиг. 1.1 Качествени атрибути в софтуера

Шаблонът, както и правилната съвкупност от тях, е пряк път към всяко от посочените в горната диаграма фиг.1.1 софтуерните качествени характеристики

- Улавят повтарящи се структури и взаимоотношения между софтуерните елементи, с което спомагат преизползването на успешния дизайн. Решават проблем като реализират взаимно свързани структури/взаимоотношения и модели в специфичен домейн/област. Observer шаблонът на фиг. 1.2 е пример за такава структура:



фиг. 1.2 Observer Pattern

Източник: GaF, Design Patterns: Elements of Reusable Object-Oriented Software

Същност на Observer шаблона: «Поведенчески шаблон за дизайн, който се използва в обектно ориентираното програмиране. Използва се в случаите, при които група от обекти (наблюдатели) след регистрацията им към регистриращ обект биват оповестявани за промени в неговото състояние».

- Кодират експертни знания в дизайн стратегиите и най-добри практики
- Налична е изчерпателна информация, описваща най-доброто от шаблон ориентирания дизайн, което подпомага широкото им приложение:
    - «Elements of Reusable Object-Oriented Software», G&F
    - «Pattern Oriented Software architecture», Wiley, Frank Buschmann, Regine Meuner, Hans Rohtnert
    - «Enterprise integration pattern», Robert Doigneau, Addison Wesley, 2011

Шаблоните притежават и много, много други предимства.

1.2. Цел и задачи на дипломната работа

Дипломната работа е в областта на проучване и изследване на шаблоните за проектиране на обектно-ориентиран софтуер. Целта на работата е *изследване на взаимовръзки между софтуерни шаблони за проектиране и практическото им използване*. Ще бъдат разгледани различните типове шаблони - градивни, структурни и поведенчески, и многократното им

използване при изграждане на софтуерни приложения. Ще бъде разгледана и взаимовръзката между дизайн и архитектурните шаблони, или по-точно как от комбинация от дизайн шаблони може да се построи архитектурен такъв.

Така представената обща цел на работата определя следните задачи за изпълнение:

1. Ще се направи подробен анализ на различните взаимовръзки по между им, как тези връзки са взаимозаменяеми в зависимост от търсените качествени характеристики. Ще разгледаме различни типове връзки, като специално внимание ще отделим на т. н. *Pattern languages* (шаблонни езици). Те разкриват набор от комбинации (алтернативни по между си композиции от шаблони), с които се постигат различни цели — най-вече качествени параметри. С този анализ ще можем да определим какво е подходящо да се използва, имайки в предвид не-функционалните, а и функционалните изисквания на системата.

Представянето им ще става основно с използване на унифицираният език за моделиране (UML). Специално внимание ще се обърне на разработката, документирането, тестването и многократното използване на шаблони и библиотеки от шаблони.

2. Ще се имплементира софтуерно приложение с обектно ориентиран език: Java. Трябва да се знае, че шаблоните работят на абстрактно ниво и са приложими при всеки обектно-ориентиран език.
3. Ще бъде създаден един изцяло нов шаблон като комбинация от други такива. С това ще се представи силата им в създаването на нови темплейт решения с помощта на други такива като съвкупност от тях.
4. Ще се оценят и анализират и качествените параметри, постигнати чрез прилагането им, като се направи и оценят софтуерните приложения с и без шаблони.
5. Ще обобщим практическата им полза при решение на проблематиките, с които се сблъскват софтуерните разработчици. Ще покажем как комбинацията от различни видове шаблони генерира софтуер с постигане на най-често изискваните качествени характеристики.

1.3. Очаквани ползи от реализацията

Ползата от дипломната може да доведе до по-задълбоченото и разширено разглеждане на шаблони на архитектурно ниво, където често комбинация от дизайн шаблони, с допълване на една, две тактики, имплементира архитектурен шаблон.

Примерно при използването на архитектурния шаблон Посредник (Broker), се прилагат още и шаблоните:

- Защита (Proxy) — за скриване, отдалечената връзка при клиента;

- Адаптер (Adapter) — за адаптиране на клиентската заявка (request) при сървъра;
- Фасадата (Facade) повишава портативността към различни APIs, използвани от сървъра;
- Стратегия (Strategy), имплантиращи различни механизми и логика;
- Абстрактна фабрика (Abstract Factory), избираща подходящи стратегии в комбинация от тях;
- Слоева система (Layering) при вътрешна имплементация на сървъра, с която се делегират отговорности на различни нива в софтуера;
- Конфигуратор (Configurator manager), прилагайки различни конфигурации в зависимост от средата, в която се прилага.

Тази и друг вид комбинации ще се изследва по-детайлно понататък при самия анализ и проектиране на бизнес приложението.

С това решение и при прилагане на добре дефинирани параметри и изисквания, би могло да се имплементира и софтуер (чисто уеб приложение, на пример), който да генерира софтуерни шаблони за дизайн, отразявайки го примерно на чист UML.

Част от материалите използвани тук, ще бъдат приложени в разработката на книга към курса «Обектно-ориентиран анализ и дизайн с използването на шаблони», воден в магистърска програма «Софтуерни технологии» към катедра «Софтуерни технологии», Софийски университет «Св. Климент Охридски».

Глава 2. Преглед на шаблоните за дизайн и взаимовръзка по между им в практиката. Предимства и недостатъци при използването им.

2.1. Основни дефиниции

Pattern (шаблон)

Софтуерен шаблон за дизайн представлява концепция за разрешаването на често срещан проблем в софтуерната разработка в конкретен контекст и домейн.

Източник: GaF, Design Patterns: Elements of Reusable Object-Oriented Software

Patterns relationships (модели на взаимоотношения — връзка между шаблоните)

Patterns complements (допълващи се шаблони)

Шаблони при които единият изисква съставна част от друг, за да може системата да работи напълно коректно или когато шаблон може да се замени с друг алтернативен такъв.

Pattern compounds (шаблони на съединения)

Колекция от шаблони, които свързани заедно имплементират нов шаблон

Patterns sequences (шаблони на последователността)

Формират дефинирана последователност от шаблони с цел да разрешат конкретен дизайн проблем.

Pattern languages (Шаблонен език)

Мрежа от взаимно свързани модели (шаблони), които дефинират процес за свързване на решения на проблеми при софтуерното разработване в конкретен домейн.

Видовете шаблони за ОО проектиране според (Гама и съавтори, 1999 год.) биват:

- **Creational design pattern (създаващи шаблони за дизайн)** -Шаблони, свързани с изграждането на механизми при създаването на обекти. Те създават обекти по възможно най-подходящ за конкретната ситуация начин.

- **Structural desgin pattern (структурни шаблони за дизайн) -** Шаблони, свързани с това да подобрят дизайна чрез дефинирането на темплейти за връзки между класовете.

- **Behavioural design patterns (поведенчески шаблони за дизайн) -** Предоставят общи модели на комуникация между обектите и реализация на тези модели. По този начин те повишават гъвкавостта по изпълнението на тази комуникация

Architectural pattern (архитектурен шаблон)

Концепция, софтуерен модел, скициращ основни свързани елементи в софтуерната архитектура. В някои от случаите един архитектурен шаблон е съставен от няколко дизайн такива.

Видове сървиси при плащане онлайн

- **NVP (Name-Value-Pair) плащане**

  Това е вид услуга при плащанията, която се имплементира чрез HTTPS POST метода с подаване на двойки (ключове — стойности) параметри към самата заявка

- **SOAP сървис плащане**

  Това е един от двата основни видове имплементации на сървисите при Service Oriented Arhitecture. Базирано е на SOAP съобщения. Това са XML файлове, предаващи се между клиента и сървъра

2.2. Преглед на шаблоните и основни видове взаимовръзки по между им

Шаблоните се срещат във всички възможни области в софтуерния бранш: мобилни системи, автобизнеса, уеб услуги. И това не е случайно. Те по уникален начин ускоряват процеса на разработване на софтуер с доказани и тествани парадигми и тактики. Ефективният софтуерен дизайн изисква обмислянето на въпроси, казуси, които стават видими за реализация едва в по-късния имплементационен етап на проекта. Шаблоните за дизайн спомагат за откриването и предотвратяването на тънки, неуловими моменти, които биха причинили сериозни проблеми в разработката в по-напреднал стадий.

Често софтуерният разработчик знае как да упражни конкретна проектантска тактика към конкретен казус. Тези техники са по-трудни за прилагане в по-широк диапазон от проблеми. Шаблоните за дизайн предоставят генерално решения, документирани във формат, който не изисква специфичната обвързаност с конкретен контекст.

Отделно, позволяват на разработчиците да комуникират чрез добре познати, и разбираеми имена на софтуерните модули, структури и взаимоотношения между отделните единици.

И не на последно място, шаблоните за дизайн са винаги в процес на развитие, винаги се преоткриват нови и нови, което е доказателство за тяхната ефикасност и ценност. Това ги прави много по-силни от ad hoc програмирането.

Но ако погледнем на шаблоните като самостоятелни единици, бързо бихме установили, че в конкретната проблематика, това не ни дава големи бонуси. Дори би затруднило използването им. Тука идва нуждата от прилагането на тези темплейти “в екип” - нуждата от взаимовръзка по между им. Шаблоните в изолация всъщност предоставят едно ограничено решение на проблем, който се увеличава в момента, когато е обвързан и с конкретен контекст - домейна, в който се имплементира. Те работят заедно. Практически не съществува темплейт, решаващ проблема сам по себе си, т. к. той би дал решение в доста ограничен конкретен аспект. Всеки съществен софтуерен дизайн неизбежно включва много свързани по между си шаблони, организирани по възможно най-ефективния начин.

При стартирането на разработката за конкретен проект винаги е хубаво да се започва с анализирането и определяне на дизайна му: от към модулност, интеракция между шаблоните, управление на процеси, връзки и т.н. С това, разбира се, идват и проблемите, с които трябва да се справим.

Чудесен пример е текстовият редактор, който се разглежда в GangOfFour книгата още във втора глава (цитат). Детайлно се минава през дизайна, в който възниква проблем след проблем по време на разработката му и се говори как различни GangOfFour шаблони биха могли да се приложат, за да се решат тези проблеми на дизайн по ефективен начин.

Много други системи са били разработени по подобен начин. Затова в практиката шаблоните са социални, те работят заедно. Когато се вгледаме в сложни, смислови, съществени дизайни, ще видим как тези шаблони си взаимодействат по между си. Можем да дадем и по-сложен пример от корпоративна (*enterprise)* архитектурата. Пример - Broker, който в себе си прилага Adaptor, Proxy, Strategy.

Това, което е интересното тук, е комбинацията между тези шаблони и начинът, по който се приплитат заедно, за да уловят основни части от дизайна в системите.

Според връзките помежду им шаблоните могат да се категоризират по следния начин:

2.2.1. Pattern complements (допълващи се шаблони)

фиг. 2.1. Допълващи се шаблони

Шаблони при които единият изисква съставна част от друг, за да може системата да работи напълно коректно.

Пример:

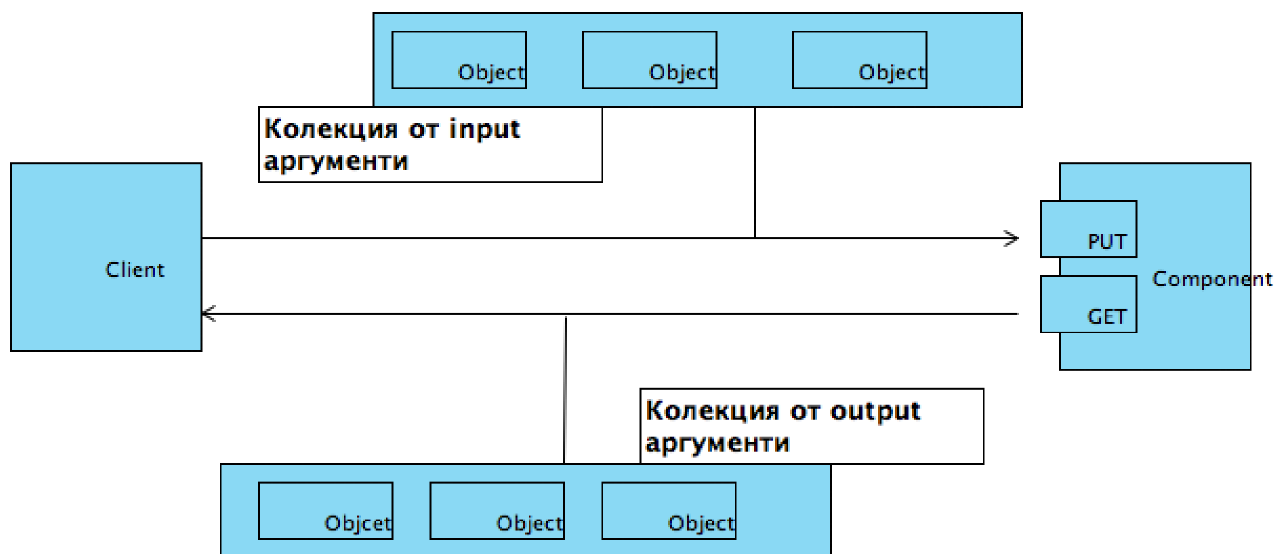
Освобождаващ метод (Disposal method): осигурява освобождаване на ресурси на клас или на самия него. Той е силно свързан с Метод Фабрика, който пък се грижи за неговото създаване (приложение в COBRA – Portable Object Adapter – мениджър за обектните инстанции, който се грижи за това кога се създават и изчистват)

Източник: http://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx

Този вид връзки също така предоставят и алтернатива на взаимно взаимозаменяеми шаблони. Т.с. един шаблон би могъл напълно да се замени с използването на друг и според контекста трябва да се избира кой от вариантите е по-добрият.

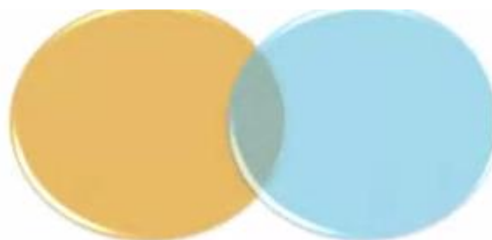
Пример:

Итератор шаблон (Iterator pattern): той достъпва елементите на агрегатор и ги предоставя на клиента. Това решение е добро в еднонишкова среда, но когато говорим за многонишкова среда, където покачването на скалируемостта и бързодействието са от изключителна важност, може би бихме се замислили за Batch method computing шаблона. Той има за задача да групира заедно последователност от операции – collection accesses, с цел избягването на множество от индивидуални достъпи (accessors), с които да се избегне двупосочната латентност, свързвайки се от инстанция в инстанция. Т.е. тука бихме имали свързване към агрегатор, който връща група от елементи, с които може да се итерира локално.



фиг. 2.2 Batch Method

2.2.2. Pattern compounds (модели на съединение) :



фиг. 2. 3 Модели на съединение

Моделите на съединение представляват съвкупност от шаблони, които имплементират съвсем нов шаблон.

Като примерни можем да дадем:

Композитна команда (Composite command): капсулира различни команди в една такава, скривайки ги, чрез предоставяне на общ интерфейс, както за една, така и за последователност от команди.

Партиден итератор (Batch iterator): той комбинира итератора с партиден метод, за да могат да се достъпват отдалечено голям набор от елементи едновременно, т.е. с метода се достъпват и взимат набор от елементи; локално те се итерират и тогава, когато се свърши итерацията, се използва метода, с който да се вземе следващата порция елементи.

Източник: Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages

2.2.3. Pattern sequences (последователност от шаблони) :

фиг. 2.4 Последователност от шаблони

Последователност от шаблони представляват шаблони, които работят с правилно поставена последователност. Тук шаблоните - предшественици се последват от точно определени други такива с цел решаване на по-сложна верига от дизайн проблематики. Дизайнът на системата улавя шаблон по шаблон, като предхождащия шаблон по един или друг начин изисква да бъде следван от определен следващ шаблон. Това е процес на композиране на

съставни шаблони, подредени в ред, който улавя еволюцията, прогресията на системата с постигането на практическо софтуерно дизайн решение.

Тази връзка се откроява когато се имплементира гъвкава приложна рамка или гъвкав *middleware*. Често в подобни случаи е полезно да се използва шаблонна последователност на Стратегията (Strategy), за да се даде възможност да се предоставя *pluggable* имплементация на различни механизми на приложната рамка. Заедно с нея идва и абстрактната фабрика (Abstract Factory), която позволява да се създаде група от семантично обединени компоненти/стратегии. Също така можем да приложим и компонентния конфигуратор (Component Configurator), с който динамично да се конфигурират фабриката и стратегията.

По този начин, например, е разработен COBRA Broker, с който се постига вариация, имплементираща различни транспортни протоколи, конкурентни модели с различни вариации на използване и контекст. Тук Стратегията може да се приложи, за да се имплементират различните видове сериализация/десериализация на данните, различен синхронизационен механизъм, Стратегия за задаване на връзка към сървъра (connection management), Стратегия за транспортния механизъм. Абстрактната фабрика би играла ролята да използва различни метод Фабрики, за да построи семейство от семантично свързани Стратегии с подходящата версия за всяка от тях.

Връзките на шаблоните в последователност разкриват силата на другите във веригата. Примерно Стратегията избира правилната стратегия на marshaling/unmarshaling или конфигуратора осигурява правилната комбинация от стратегии. Тази последователност неимоверно спомага за изграждането на ефективния дизайн и архитектура.

2.2.4. Pattern language (Шаблонен език) :

фиг. 2.5 Шаблонен език

Шаблонният език дефинира мрежа от свързани шаблони; речник; процес на систематично разрешаване на софтуерни проблеми в определен домейн и област. Това са може би едни от най-сложните връзки между шаблоните. В този случай се прилагат в контекста на шаблонния език. Така взаимодействайки по между си, генерират система, документираща успешна прогресия на дизайн решения, трансформации и алтернативи. Те дават

алтернатива на използвана последователност от шаблони при разработването на софтуера. С това се решават проблеми, които имат различни преимущества едно спрямо друг. Примерно в единия случай се постига по-добра скалируемост, докато в другия – по-добра модулност.

Стъпките, с които се прилага шаблонният език включват:

- ключовите проблеми и последователност на решаването им;
- как следва да се решават тези проблеми прилагайки шаблоните и с каква взаимовръзка по между им;
- има се в предвид домейнът, в който ще се прилагат съвкупността от шаблони, който дефинира и качествените параметри, които трябва да се постигнат;
- какви алтернативи биха съществували и какво бихме постигнали за всяка от тях.

Много специалисти, софтуерни архитекти и инженери документират своите знания и опит точно в последователността, дефинирана от езика на шаблоните. С тях се отразява мощта и елегантността на взаимоотношенията между шаблоните, което подпомага софтуерният екип да бъде много по-продуктивен и комуникативен и да произвежда софтуер, който е готов да посрещне нуждите на клиентите и потребителите на системите по по-ефективен начин.

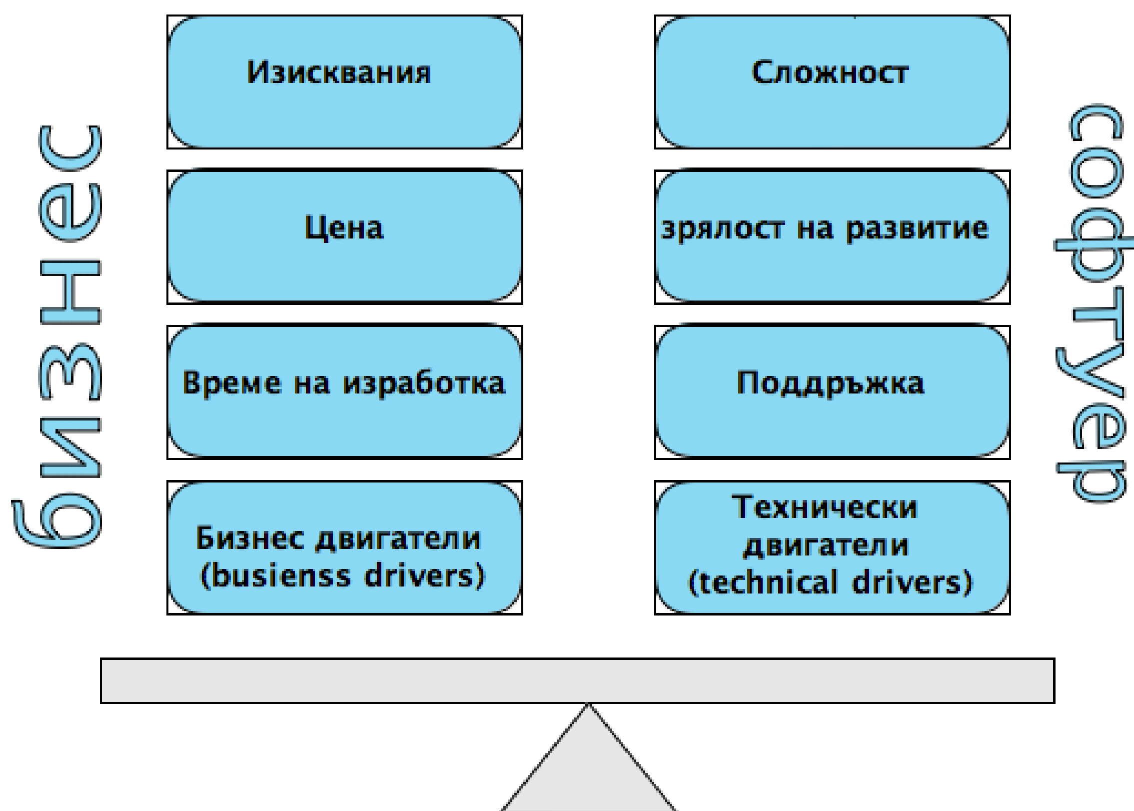
Прилагането и използването на подобни видове връзки са първите стъпки при правилното разбиране на шаблоните, не като самостоятелни единици, а като съвкупност от едно преплетено цяло. Въпрос на добър софтуерен дизайн е всеки програмист или архитект да намери подходящия дизайн и организира по възможно най-ефективния начин тези шаблони, така че да ги направи по-лесни за разбиране и прилагане.

2.3. Избор на критерии за сравнения при селекция на шаблони за конкретно приложение

Шаблоните, както и техният език, са в един непрекъснат процес на развитие и допълнение. Това довежда и до тяхното непрекъснато нарастване като брой и опции. Тънкостта на избора на правилния шаблон/-и за конкретния проблем е важна черта за всеки софтуерен разработчик. Има статии и публикации по въпроса за избор на критерии при селекцията на подходящия шаблон.

Следват извадки от едни от оптималните критерии, които биха се използвали при подобен случай.

2.3.1. Business vs. Software



фиг. 2.6 Business vs. Software

На първо място, винаги търсим оптималното съотношение между бизнес и технически изисквания. Винаги можем да правим добър и още по-добър софтуер. Но до колко е готов бизнеса да плати за това непрекъснато подобряване? Шаблоните спомагат за постигането на този баланс.

2.3.2. Quality attributes (качествени атрибути)

При разработката на даден софтуерен продукт се гонят и т.н. качествени параметри като сигурност, модулност, разбираемост, леснота на поддръжката, скалируемост на кода, висока производителност, бързодействие и други важни от гледна точка на бизнеса качества. Фактически и самите шаблони допринасят за това. Това са на практика едни от най-важните критерии, по които се избира шаблон или съвкупност от шаблони.

Шаблонният език, по който се води избора на правилна комбинация от шаблони, е в зависимост от избраните качествени параметри, както и от функционалните изисквания. В практическите примери ще наблегнем на детайлното разглеждане на различните опции, предлагани от избраните за експерименталната постановка шаблонни езици и какви параметри носи всеки от тях.

2.3.3. The context-sentitive (чувствителност към контекста)

Селекцията е на базата и на контекста на домейна в който работим. Това включва и езика, с който имплементираме софтуера. Прилагането на един шаблон в едно приложение може да е неприложимо, или не дотолкова лесно приложимо в друго такова. *Singelton* е подходящ за еднонишкова среда, примерно десктоп приложение, но не и за клъстърна среда, работеща с много по-голям брой нишки.

2.3.4. Link

Линк е комуникационна връзка, определяща доколко шаблонът е свързан с други шаблони, вече приложени в системата.

2.3.5. Collaboration

Този критерий отговаря на въпроси като: Доколко шаблонът (шаблоните) може да се променя, за да се приложи в системата, т.е. по някакъв начин да се интегрират към конкретния домейн и контекст? Доколко новият шаблон приляга на вече съществуващите такива в приложението?

2.3.6. Каталози

Избор от каталози (вече съществуващ набор от шаблони), събрани под един или друг етикет, таг, домейн на приложимост. Примерно има шаблони за J2EE, G&F шаблоните, POSA 1, 2 , 3, 4, 5 шаблони за многонишкови и клъстърни среди и т. н. Те малко или много дават представа или отразяват колекция от шаблони, свързани по между си за целта на използването си.

2.3.7. RADM — Reusable Architectural Decision Model

RADM споделя множество характеристики с вече описаните каталози, но тук наблюдаваме и елемент на домейн-специфичния избор в различен етап от развитието на приложението. Различните типове шаблони, маркирани като архитектурни, се разглеждат още в самото начало на разработката на приложението, при анализа и концептуалното ниво. Дизайн шаблоните – при чисто технологично ниво, когато вече имаме връзка с технологиите, езика и средата. И шаблони, при тестово ниво. За всеки домейн се създава RADM, наличен като каталог или език. Всеки RADM е създаден от експерти и съдържа шаблони, доказали се, като приложими за дадения домейн. RADM са артефакти, съдържащи домейн специфични моделни решения, водещи разработчика към правилен избор на шаблони и приложения.

Приложените в RADM решения, касаещи имплементацията, трябва регулярно да се обновяват, успоредно с развитието на технологията.

Източник: Pattern languages of Programs - http://goo.gl/sH4mGs

2.4. Изводи

2.4.1. Плюсове и минуси от използването на софтуерните шаблони

Плюсове

- Улавят и абстрахират повтарящи се софтуерни роли и взаимоотношения, за да улеснят систематичното преизползване на успешния дизайн.

- Записват инженерните ни трудове и открития, за да допринесат за развитието и устойчивостта на софтуерните системи. Тук отново се опираме до какви качествени характеристики искаме да постигнем за конкретния проект и кои шаблони, в различна съвкупност, допринасят за тях. Избираме, заедно с функционалните изисквани, най-важните за конкретната система (скалериуммост, ефективност, модуларност, контрол на версиите, различен по типаж оптимизации и т.н.), приоритезираме ги и анализираме. Шаблоните приложими за най-важните за нас качествени характеристики, са и нашето решение.

- Позволяват създаването на единен език, улесняващ разбирането на общите архитектурни и дизайн решения, инженерния труд, и комуникацията в екипа.

- Те предоставят основа за автоматизация. Тук говорим за документация и имплементация на последователност от шаблони, които се използват за разрешаване на конкретен проблем и как това вкарва фактически една автоматизация. Примерно текстовият редактор: за него решение са команда (Command), метод фактори (Method factory), спомен (Memento). Дори приложните рамки се пишат на шаблонно ориентиран софтуер, който автоматично се използва от клиента

- Те ни помагат да преодолеем езикови специфични бариери. Тук няма нужда да навлизаме в конкретните езикови единици или характеристики. При шаблоните, използваме абстракцията на софтуерната архитектура и дизайн, подчертават се наистина важните неща като повтарящите се връзки, още в дизайн фазата, която би била имплементирана в много различни езици и с много различни имплементационни техники с различна цел. Но фокусирайки се в дизайна, ние преодоляваме тази специфика и «късогледство».

- Абстрахират несъществените имплементационни детайли, които не биха били потребни примерно в дизайн фазата.

Лимити (минуси) при шаблоните

- Изискват усилие да бъдат имлементирани. Шаблоните не представят сорс код. Някой трябва да го напише, имплементирайки съответния темплейт. Шаблоните, съответно и тяхната връзка с други, ни очертават как частиците софтуер трябва да се сглобят и как те комуникират помежду си. Това може да изисква повече време за прилагане. Затова, в някои случаи се прибягва до промяна на дефинираната структура, така че да може да се впише в контекста, в който се използва. Разбира се, изисква се и съответното внимание за това до колко се придържаме към качествените параметри, които искаме да постигнем с прилагането на шаблона. Винаги имаме и плюсове и минуси и внимаваме, анализираме, как бихме могли да интегрираме шаблона в нашето решение.

- Друг лимит е, че общият речник, с който идват шаблоните. Той може да бъде примамливо прост. Примерно по-опитните архитекти и софтуерни инженери са наясно какво би представлявало и какво се очаква от имплементацията на конкретен шаблонен език. Докато за по-неопитните програмисти, същият този език може примамливо да значи нещо просто и не времеотнемащо да се реализира. Тук се набляга на важната характеристика на това да може да се дава правилна времева оценка за решение на проблема се губи в погледа на опитния и не толкова опитния, имайки под ръка едни същи инструменти. Тук се включва и момента, в който по опитните биха избрали и по правилния шаблон за конкретния случай, докато за останалите би довело до това да се имплементира наново в по-късен етап. Затова винаги трябва да се внимава и да се има впредвид, че просто названия като Visitor, Proxy, Mediator, CommandManager и прилагането им в дизайна, не винаги означава, че програмистът е разбрал как да се приложат, как тези техники да бъдат ефективно използвани във важни софтуерни приложения.

- Ограничената сфера на познание на шаблоните може да доведе до ограничен избор на решения, поради малкото известни дизайн опции.

- Понякога шаблоните могат да пренебрегнат съществени детайли в имплементацията и по специално в оптимизационните детайли. Т.с. има опасност на тези неща да не им се обърне внимание в ранния етап на дизайна. Ако не внимаваме още в началото и се придържаме прекалено стриктно към конкретен подход, базиран на шаблони които разбираме, можем да се окажем в задънена улица с течение на времето. Един лесен пример бихме могли да дадем с Singlenton, който не е замислен за използване в многонишкова среда.

- Не всички шаблони са приложими в не-обектно ориентирани програмни (ООП) езици. Често мислим, че шаблоните са равнозначещи в какъвто и да е контекст що се отнася до езикови платформи като Java, C++, C#, Objective C и т. н. Това може да се каже, че е истина с някое друго изключение и особеност. Но когато програмираме на

другi езици, като чист C, Fortran, Prolog, много по различни от ООП езиците, можем да видим, че повечето от шаблонните абстракции, по един или друг начин, не се вписват в контекста на не ООП езика. Примерно в език като ADA, още в началото няма понятия като наследяване или прилагане на интерфейс. Много от шаблоните наблягат именно на тези характеристики. В С езика няма концепции като деструктури и класове. Естествено винаги може да се имплементира изкуствено поддържане на тези техники, като примерно използване на структура в С за клас, но това би усложнило проекта значително.

Трябва да имаме предвид, че шаблоните, както и връзката им в шаблонния речник, не винаги се упражняват на сляпо и директно. Идеята им не е да заменят инженерната ни мисъл и креативност с наизустена апликация на сурови дизайн правила и закони в кодирането. Целта им не е и да се опитват да автоматизират софтуерното производство. Нито пък да сменят хората с инструметни. Вместо това, истинската роля на шаблоните е да се опитат да кодират значимите човешки познания и опит, асоциирани със софтуерната разработка. Хората, които в миналото и сега кодират, описват ключовите моменти, ключовите взаимоотношения, структури. Описват ключовите конвенции по начин, по който другите хора могат да учат и разбират.

Добрите шаблони идват от генерализация на практическия опит. Почти е невъзможно да създадем шаблони без опит.

2.4.2. Защо да използваме архитектурните и дизайн шаблоните.

Има доста статии на темата кога е добре и кога не използването на шаблоните за дизайн.

Примерни източници :

- http://stackoverflow.com/questions/85272/how-do-you-know-when-to-use-design-patterns
- Eric Freeman, “Head First Design Pattern”, oct. 2004, O’Really
- И други подобни

Някои разработчици дори се двоумят защо трябва да отделят време за изучаването им, след като може да се мине и без тях. Това мислене е особено разпространено за малки проекти. Шаблоните могат да ни накарат да мислим в една ограничена рамка, поставена от тях и да не ни дадат възможност да измислим нещо ново, може би, по—добро. Тези решения, в същото време, се възприемат като недълготрайни. Езиците и технологиите се менят постоянно, в такъв случай и шаблоните ще са несъвместими с тях.

От бизнес гледна точка се правят изводи, че за тази тематика са нужни много по-опитни програмисти, което неимоверно води и до по-високата цена на проекта. А да не говорим за момента, когато става дума за по-задълбочено навлизане на шаблоните, на връзките между тях, на «издигането» им на по-високо ниво — архитектурното, където не анализираме само

за един конкретен модул, а минаваме към една много по обширна част — проекта от към цялостната му картина, погледнат от възможно най-високата и абстрактна точка.

Горните твърдения могат да навеждат на мисълта, че използването на шаблони е ненужно, скъпо и трудно. Нека оборим това.

Софтуерните шаблони са добри: грешно твърдение!

Шаблоните са неутрални. Това което трябва да се знае е, че не винаги са най-доброто решение. Едни са по-добри от други. Има по добри решения от прилагането на шаблон. Дори някои в определен контекст се превръщат в анти-шаблон (anti-pattern). Примерно Singelton в многонишкова среда. За това винаги трябва да се анализира в дълбочина контекста и изискванията, които трябва да се постигнат в определения проект. Да не забравяме, че шаблонът е решение в определен контекст и домейн. Ако се опитаме да използваме шаблони, просто защото си мислим, че трябва да ги използваме, то неимоверно ще попаднем и оплетем в собствените си мрежи, пишейки наистина сложен и ненужен код за нещо относително малко и лесно. Нека например вземем Командата (Command). Когато заявките са прости и лесно дефиниращи се, много по-ефективно би било използването на един if-else параграф с по три реда код във всеки от случаите. Няма да има нужда да се гради йерархия за Командата и включването на метод-фабрика, създаващ подходящата команда според случая.

Изучаването на езика на шаблоните (Patterns language) е мощно средство за разбиране на правилното им прилагане.

Софтуерните шаблони са сложни : грешно твърдение!

Една от основните идеи на шаблоните е да изчистят кода от излишна сложност. Някои от тях фактически са доста лесни за разбиране, като примерно шаблонния метод (Template method). Някои са сложни като например шаблона Посетител (Visitor). Но в крайна сметка използването на по-лесните шаблони е по-разпространено от по-сложните такива. А доста често за по-трудните имплементации и домейн контекст има имплементирани приложни рамки, които предоставят API за ползване от клиента.

Трябва ли да се използват в малки проекти?

Колкото и да е малък проектът, той може да изисква такива не-функционални изисквания, като модалност, грануларност, възможност за лесно разширяване и т.н., които могат да се постигнат с шаблоните. Може би се използват не толкова сложните и многокомпонентните (като Посетителят и Верига от отговорности), а по—простите от към структура (като Стратегия и Метод фабрика). Може да се погледна и от друга гледна точка: ако се иска проект с написан набързо код, без дизайн, който може да се забрави след това, може би е приемливо да минем и без шаблони. Но ако от този проект се очаква да се поддържа и до-разширява, то тогава се и очакват неизбежните допълнителни главоболия при неприлагане на шаблони.

Имплементирането на шаблони изисква по-опитни разработчици – Невярно

Отново ще се позова на лесни и сложни шаблони. Не за всички шаблони трябва да си минимум старши разработчик, за да се разберат и използват.

Увеличава цената на проекта

Не е вярно, стига шаблоните да се използват правилно. Това което прави проектът по-скъп, е слаб и лош дизайн, особено при не-правилно използване на шаблони.

Не трябва да забравяме, че шаблоните, с или без връзки по-между си, имат една основна сила: комуникацията. Много, много по лесно и бързо е да се каже «използвай стратегия (Strategy) за това», отколкото да се обяснява в детайли какво се има в предвид. Много по-лесно е да се дебатират облагите от *Domain models vs. Transition scripts* ако всички знаем кое какво означава. Ако примерно наречем клас «GamerBuilder», веднага се изяснява, че използвам *Builder* за имплементацията му. Обшият им език е не толкова основополагащ или важен да се научи и знае, но той преоткрива много общи решения на проблеми по лесен и бърз за разбиране начин. Шаблоните ни позволяват да кажем много с малко думи.

Решенията, които шаблоните осигуряват, в много случаи, вече са били до някаква степен използвани или не от разработчика. Той или не е знаел, че това е шаблона X или го е до-имплементирал до ниво, на което му трябва съвсем малко, за да се получи перфектното решение. До есенцията, същността на шаблоните, фактически се стига тогава, когато се спречкаме с проблема и сме мислили за конкретното му решение.

Глава 3. Използвани технологии, платформи и методологии

3.1. Изисквания към средствата (технологии, платформи и методологии)

Основната идея на дипломната работа е не толкова да залага на технологиите и платформите. Набляга се на анализа, дизайна и конструирането на възможно най-подходяща архитектура на проекта според домейна и контекста на работа. Реализира се и конкретен пример, по който ще се работи детайлно в основната точка на анализа от темата. За това ни трябва и обектно ориентиран език. Можем да изберем който и да е от съществуващите такива. Разбира се, можем да боравим и с функционален или какъвто и да е не обектен език. Но в този случай, няма да можем да извлечем максималното от шаблоните и изследванията им върху видовете връзки. А за нас е важно да се забележи и отрази силата на използването им.

От платформена гледна точка е необходимо само и единствено среда, която може да се инсталира на езика върху който ще работим. Понеже няма да включваме многообразие от технологии, не се нуждаем от по специфични среди.

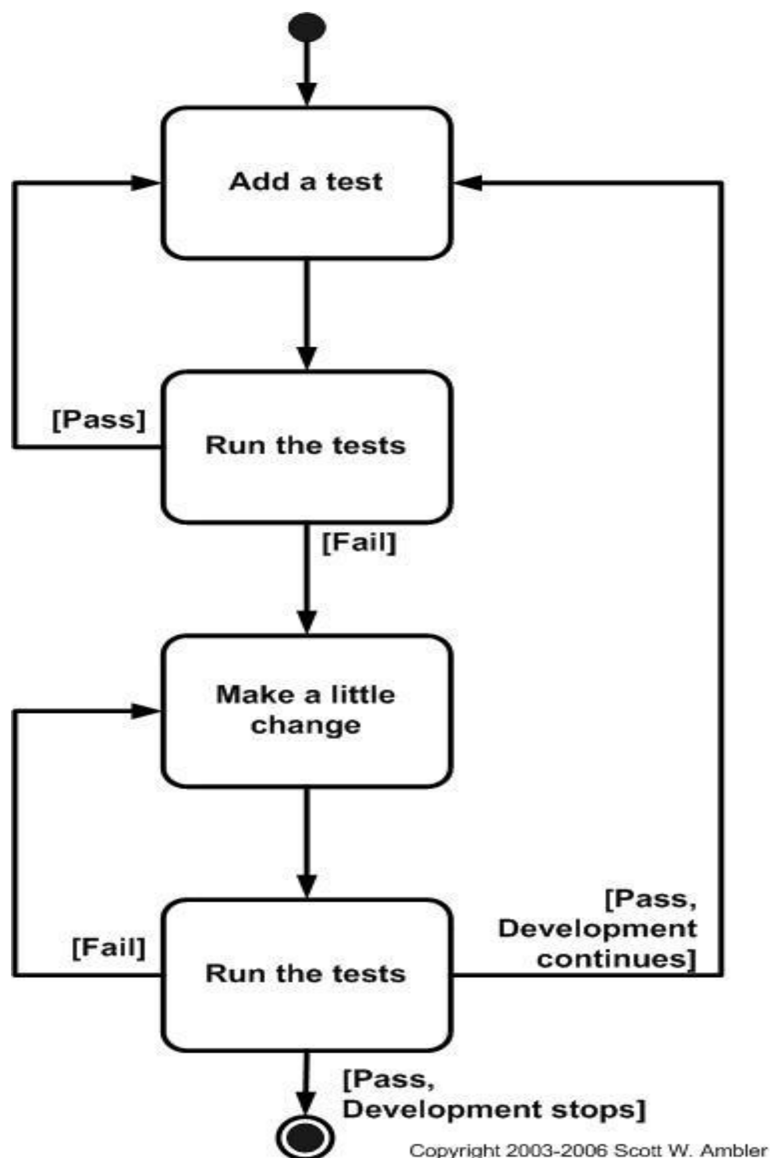
Относно методологията, с която ще боравим, трябва да се избере такава, която е най-подходяща в случаите. когато работим с повече анализ. Тя трябва добре да съвпада с подходите, които избираме да използваме в дипломната работа. А именно: да има частица на анализа, еволюцията и да има подходящата доза сравнителна оценка из между различни варианти.

3.2. Избор на средствата (технологии, платформи и методологии)

За целите на дипломната работа се избира Java - доказал се обектно ориентиран език за програмиране, с който лесно може да се борави в контекста на шаблонния анализ и имплементиране.

Заедно с езика се избира и IDE – *Integrated development environment*, с който ще се пише приложението. В нашия случай то е *Eclipse Kepler* версията – последната излязла 2013 август месец.

Следващия много важен момент е изборът на методология. За подобен вид приложение, където се държи на качеството на написаното, е подходящ *Test Driven development* подхода за програмиране. При него първо се пише теста, след това се пише имплементацията, която се тества. Рефакторира се кодът ако теста се провали и така докато тръгне успешно. Много добро описание се дава със следната диаграма фиг. 3.1.:



фиг. 3.1 Test Driven Development

Източник: http://www.agiledata.org/essays/tdd.html

Следваща, основополагаща причина да се избере TDD подхода е и приликата му с шаблоните. В основата на двете стратегии стои промяната, нещо което е винаги с нас в разработването на софтуер. Ние, като софтуерни инженери, държим в себе си устрема да избягваме каквито и да е бъдещи промени. А TDD и шаблоните ни помагат да попадаме в примките на непрекъснатата промяна.

Докато тестваме се фокусираме върху един единствен проблем, една операция или множество от операции, завършваща конкретна бизнес логика. Това правят и шаблоните – те работят за разрешаването на един конкретен проблем. Дори работейки с TDD, ние се

фокусираме върху неща като: силна кохезия, ниска взаимовръзка между модулите и фокусирането върху един единствен проблем. Това, което и шаблоните правят.

След като имаме на лице *Test Driven Development* подхода и дизайн шаблоните, най-подходящата методология за работа за нас е спираловидната. В нея имаме 4 основни момента, които се въртят, докато не получим най-доброто решение: анализ, еволюция – оценяване, разработване и планиране. В нашия случай, основно ще се набляга на оценяването и анализа. Което е и основа на архитектурата и дизайна на софтуера.

Глава 4. Анализ на начините за използване на шаблони

Тази глава от дипломната работа представя концептуален модел за описание на качеството на софтуерните системи и прави анализ на различните начини за използване на съществуващи шаблони и комбинации от тях, както и създаване и използване на нов шаблон за софтуерна разработка. Също така и разглежда какви качествени атрибути се постигат при съответната комбинация от шаблони. На базата на анализа се дават и конкретни примерни от имплементираната примерната система

4.1. Концептуален модел за описание на качеството

В основата на функционалните изисквания при кой да е проект стоят прилагането на т. нар. софтуерни качествени параметри. Те имат всеобщ характер и обхващат цялостната картина на софтуера — както в режим на дизайн, така и в етап на изпълнение. Те включват в себе си решението на проблемни, съществени области във всички слоеве и части на приложението. Дори в повечето случаи в самите бизнес спецификации се опиват и изискват изрично ред от качествени атрибути, с които да се гарантира качеството на софтуерния продукт.

В постигането на това всеки софтуерен инженер предприема различни по идея и концепция подходи. В много от случаите, най-вече в дизайн контекст, реализацията се изразява именно в прилагането на софтуерни шаблони за дизайн. И както вече знаем — в различна по същност комбинация по между им.

В следващата част ще разгледаме в детайли шаблоните, които се използват за имплементацията на счетоводната ни система за управление на служители. Ще се използват различни тактики, взаимно-подменяеми съвкупност от шаблони, с които се постига и различна съвкупност от качествени атрибути. Кои от тях ще бъдат избрани, е именно задача на определянето на архитектурните двигатели — на основните и най-важни атрибути, които трябва да се търсят и реализират в системата. В анализа ще използвам различните типове взаимоотношения, описани по горе, а именно: модели на съединение, допълващи се шаблони, шаблони на последователността и най-вече ще се наблегне на езика на шаблоните, т. к. именно тук може да се говори за алтернативен избор за реализация на набор от качествени атрибути.

Първо ще направя кратко резюме на различните софтуерни качествени атрибути с цел по ясна представа какво представляват те и какво допринасят за системата:

Качествен атрибут (някъде наричан и не-функционално изискване)	**Крато описание и приложимост**
Maintainability	Възможност това системата да се променя, поддържа по лесен и бърз начин. Да има ниска свързаност между модулите. Този атрибут включва компоненти, сървиси,

	интерфейси, при добавянето или променято на функционалност или поправянето на грешки от страна на софтуерния разработчик
Reusability	Отговаря на въпроса – до колко софтуерните компоненти могат да се преизползват в други системи, след като се адаптират по подходящ начин. С този атрибут се минимизира дублирането на код и най-вече времето за имплементация
Predictability	Системата да се имплементира по начин, който да предвижда нови промени или съвсем нови модули, които лесно би било да се добавят
Security	Способността на системата да предотврати неоторизирано използване, т.с. извън описаните случаи и начини. Сигурна система има за цел да защити данните и неоторизираното им модифициране
Testability	Способността системата да се тества функционалността й, както интегрециаонно, така и компонентно, функция по функция, с добре описани пре-дефинирани условия и пост фактор изпълнение на функциите – резултат.
Adaptibility	Способността софтуерните компоненти да са взаимноозаменяеми
Interoperability	Възможността да се комуникира, обменя информация с външни системи – т.н. наречените трети лица (3-th party systems)
Understandability	Хората разбират системата с малко, или почти никакво обучение
Fault tolerance	Способността на системата да продължава работа дори и в момент на грешка в един от компонентите
Maturity	Доказването на компоненти в системата да бъдат стабилни от други такива. Т.с. стабилна, здрава, връзка между отделните компоненти – модули.

таблица 4.1 Качествени атрибути (параметри)

Могат да се разгледат и много други атрибути, като Performance, Availability, Scallability и т.н., но тук няма да можем да наблегнем и върху тях, т. к. за тях е трудно да се приложат дизайн шаблони. Може би за изброените горе атрибути са приложими шаблони от по-горно ниво (архитектурни) и някои други тактики и конфигурации на самата система при имплементирането й.

Анализът, през който ще минем ще доведе до различни резултати, набор от качествени атрибути. Ще се спрем на този резултат, който идеално отговаря на архитектурните ни драйвери: тестваемост, сигурност, гъвкавост, устойчивост на откази, зрялост, съвместимост, променяемост.

4.2. Качествени параметри (атрибути), постигани с прилагането на шаблони в различни взаимоотношения по между им — анализ и дизайн

Както споменахме по горе, ключовите инструменти с които ще си служим в анализа са четирите вида връзки между шаблоните: модели на съединение, допълващи се шаблони, шаблони на последователността и най-вече ще наблегна на езика на шаблоните. Защо на тях именно? Защото те представляват една мрежа от шаблони, свързани по между си. Под мрежа ние имаме предвид, че шаблоните са свързани един с друг, те взаимно си влияят или се взаимоизключват по определен начин. Те създават речник - речник на шаблонния език. И не само това. Всъщност те ни предоставят много повече от просто речник, единен език - те създават процес на подредено, систематично разрешаване на дизайн проблеми в конкретния домейн и контекст, в който работим.

Хубаво е да подчертаем, че езика на шаблоните не се прилага само в софтуера. Той е приложим и в много други домейни, като архитектура на сгради и строителство.

При шаблоните на последователността имаме една линейнообразна последователност от шаблони, подредени в конкретна поредност, за да получим еволюцията на системата и постигнем определен резултат. Докато при шаблонния език ние имаме една разнородна последователност - мрежа, която може да бъде обходена по различен начин, да се трансформира по различен начин. Да предостави алтернативи - с цел да се адресират специфични проблеми при разрешаване на проекта.

Нека, на първо време, да направим един обобщаващ преглед на шаблоните (таблица 4.2), които ще бъдат включени в анализа при изграждането на системата:

Дизайн / Архитектурен шаблон	Описания и приложимост в системата
Layering	Може да си го представим като група от класове, които имат еднакви зависимости към други модули в системата. Те групират концепция, която може лесно да бъде взаимозаменяема по отношение на

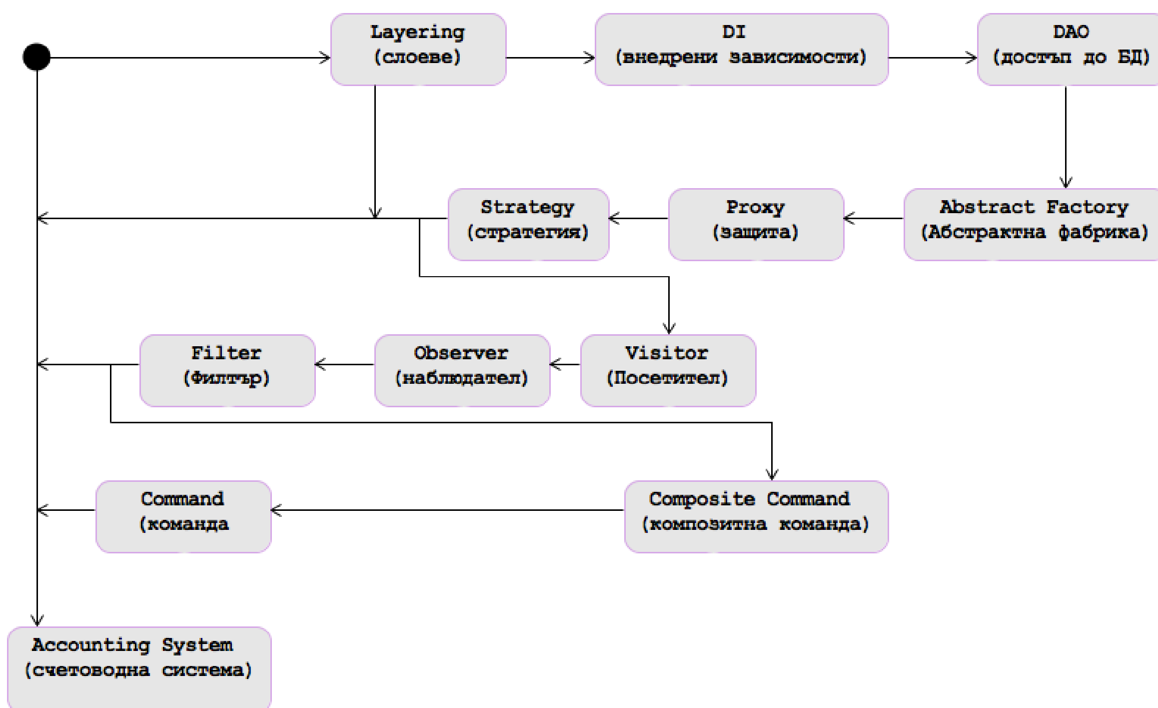
	имплементация. В нашата система слоеве могат да се приложат примерно за достъп до данните, бизнес имплементация, и слой за имплементация на потребителската взаимодействие със системата – конзолата при нас.
Dependency Injection	Нашата система се занимава с плащания. Това от своя страна носи със себе и желанието да бъде добре грануларно тествана, а за това се иска и модулите да са възможно най-добре независими един от друг. Шаблон, допринасящ за премахването на хардкоднати зависими модули, позволявайки им да променят имплементациите им както по време на компилация, така и в режим на изпълнение. Есенцията тук е използването на интерфейси, договори, за конкретен модул.
Data Access Object	Конвенционален шаблон за работа с базата данни (в нашия случай, ще бъде in-memmory). Това е отделен слой, който предоставя имплементация на основните операции с данни за определена таблица: добавяне, изтриване, редактиране и четене на записи от таблицата
Abstract Factory	И тук идва момента да комбинираме имплементации на стратегии/класове и в зависимост от семантиката в момента на изпълнение. Абстрактната фабрика е шаблон, с който ние строим съвкупност от имплементции на класове, които са взаимноосвързани в конкретния момент и изпълняват в момента конкретно свързана задача
Proxy	Предоставя една обвивка, контролираща достъпа до конкретен обект/ресурс. С това искаме да увеличим сигурността и намалим връзките между модулите в системата ни. Ще го приложим в имплементацията за извикване на командите за плащания, където ще ограничим достъпа до конкретното плащане, ако не сме сигурни в идентичността и оторизацията на текущия

	потребител
Strategy	Ние ще искаме гъвкавост за нашата система. Ще искаме различни имплементации на механизмите за връзка с плащанията (SOAP/HTTPS VNP). Съответно стратегии за сериализация и десериализация. Стратегията ще ни помогне с това да избираме различни имплементации на логики по отношение на различните механизми, (сериализация/десериализация) с които ще боравим. Всичко това ще става по време на изпълнение.
Observer	Ще използваме в случая когато искаме да позволим конкретен служител да се представи/изпише в конзолата по различен начин, т.к. този шаблон предоставя именно това: идеално отделяне от субекта (Subject) с данни, служителя от представянето му (Observer)
Filter Iterator	Филтрация на служители по конкретни критерии. Това се постига с филтър итератора, който ще имплемнтира логиката по филтрация на списък от служители и връща такива отговарящ на критерия. Примерно както е случаят с избирането на под-служителите в една йерархия по вид на банката, към която са свързани
Composite Command	Композитната команда е идеален инструмент за изпълнение на последователност от команди/заявки
Command	Командата е заявката – обект, в която ще обвием конкретно искане от написаното в конзолата от потребителя

таблица 4.2 Архитектурни шаблони

Всички тези шаблони показваме чрез диаграма, изобразена във фиг. 4.1. С нея ще можем да представяме и връзката по между им, както и последователността им. Те са подредени като шаблонен език с различен път и алтернативи, през които може да се минава, в зависимост

от това какво се опитваме да направим и да постигнем. Заедно с разглеждането на различните алтернативности, ще наблегнем и също така на другите видове видове взаимоотношения шаблони, които се появяват. Накрая, след като изберем нашата алтернатива, постигайки нашата цел, а именно архитектурните драйвери, ще направим обобщение каква би била системата с и без шаблони и каква добавена стойност ни носят те. Ще подчертаем и случаите, в които освен архитектурни драйвери, с шаблоните се имплантират по елегантен начин и функционални изисквания. Т.с. тука откриваме и друг плюс при прилагането на шаблони.



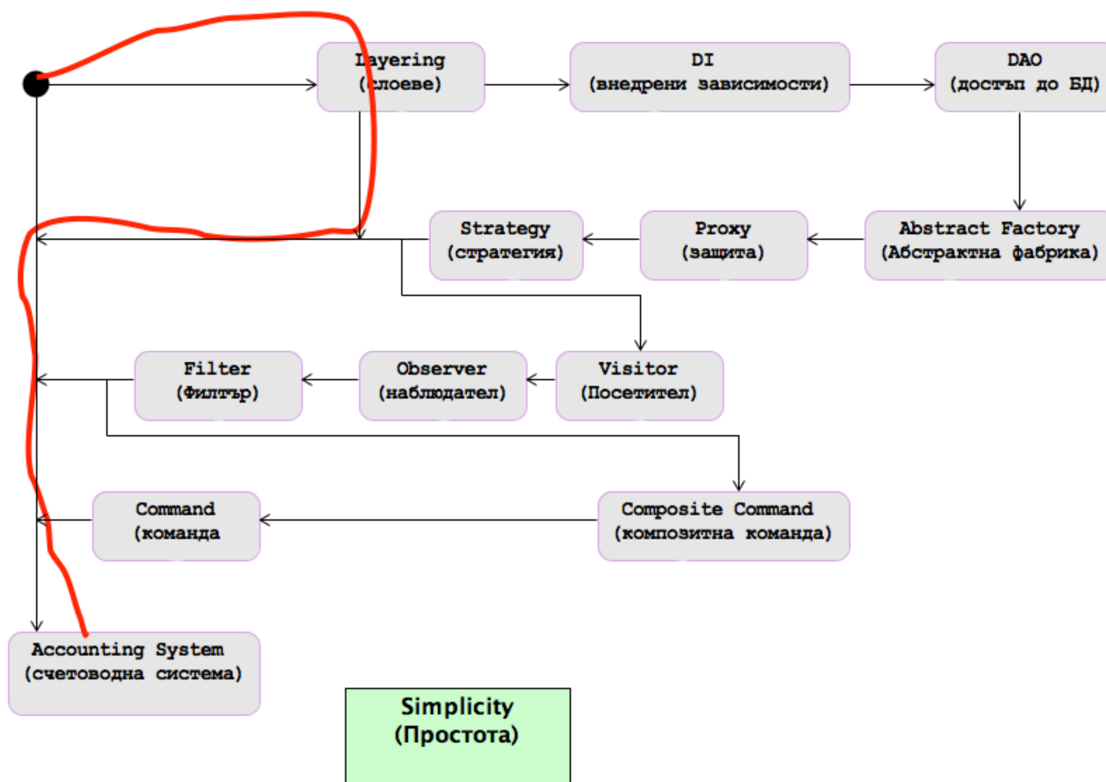
фиг. 4.1 Скелет на шаблонен език

С тази диаграма ще покриваме нашите изисквания от към анализ. Ще съумеем да видим какво имаме под ръка, какво постигаме, покриваме и каква алтернатива бихме могли да изберем.

I. Цел: простота на имплементацията без използването на набор от шаблони

Нека започнем с първата алтернатива, първия път на шаблонния език, с който бихме могли да имплементираме нашата система:

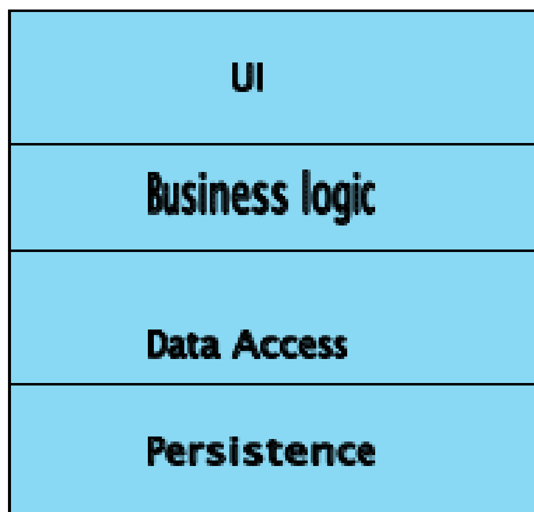
I.1. Обща диаграмова структура



фиг. 4.2 Accounting system - простота

I.2. Използвани шаблони

Този път минава само и единствено през *layering* (слоеве), фиг. 4.3:



фиг. 4.3 Слоева архитектура

Като за нашия случай те са *UI* — презентационния слой; *Business Logic*: слой за бизнес логиката; *Data Access*: слой за *Create Read Update Delete* (създаване, четене, обновяване и

изтриване) операциите към базата данни и *Persistence*: слой грижещ се за създаването на POJO *Plain Old Java Object*, които на Java клас се асоциира таблица от базата данни. В нашия случай това е Java RAM паметта. Но би могла да бъде разбира се, файлова, или SQL — релационна или NoSQL база данни управлявана от *database server*.

I.3. Постигнати качествени атрибути / изводи

Този подход предоставя едно много лесно и семпло решение на системата. Не предоставя, обаче, възможност да се разширява или модифицира софтуера по ефективен и ефикасен начин. Това може да допринесе до силна зависимост между отделните модули, което трудно би се и разбирало, поддържало или дори тествало. Споменахме, че искаме да имаме възможността да тестваме системата както на интеграционно ниво, така и компонентно (*unit base*) базирано, абстрахирайки се от модулите, с които се връзва този, който тества. В този случай не можем да говорим за зрелост или предсказуемост на системата. Трудно бихме могли да кажем или докажем, че подобен софтуер би могъл да се справи със задачи като да се добави нов модул и да се обвърже с останалите, или въобще да се смени съществуващ такъв с друга имплементация. Примерно добавяне на нов вид плащане и връзката с модула за плащания.

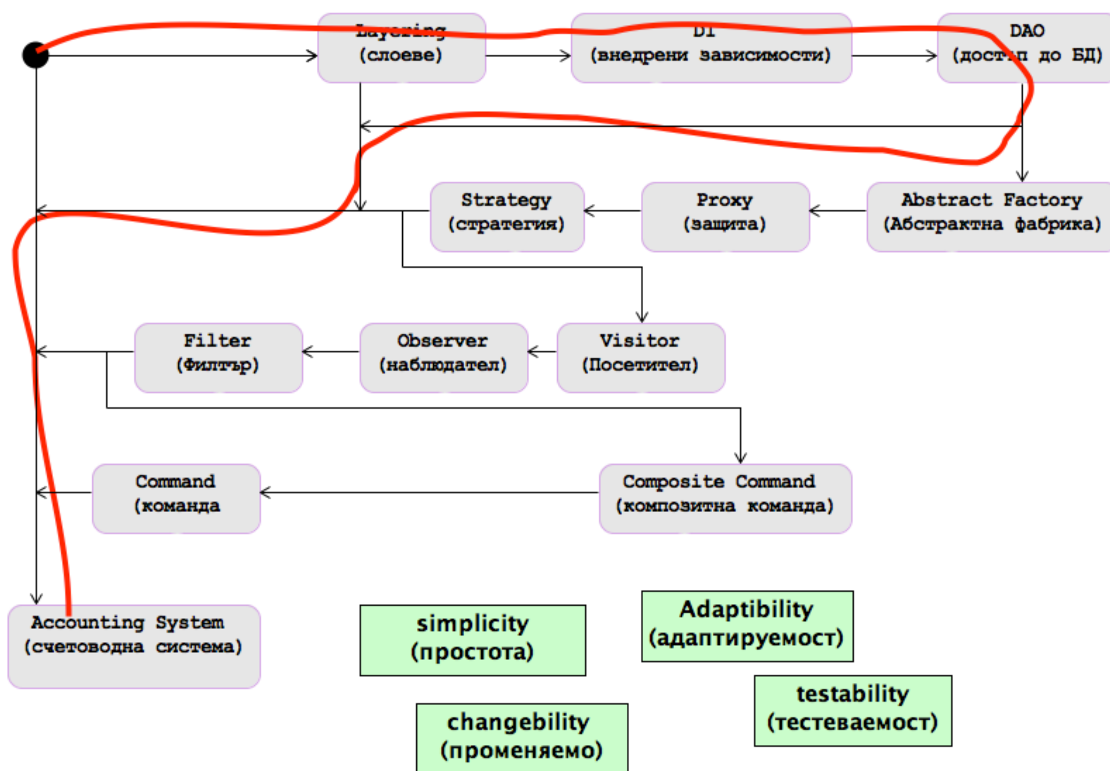
Бихме се спрели на тази алтернатива, този път извадка от нашия шаблонен език, ако искаме да работим с опростена версия на системата от към архитектурни решения. Но минусите, които ни носи това решение, довеждат и до ниско качество на кода и на софтуера като цяло.

II. Цел: силно и стриктно разграничаване на слоевете и компонентите с въвеждането на изграждане на интерфейси; по-ефективна тестова стратегия

Гледайки все още от «птичи поглед» архитектурата и не навлизайки в имплементационните детайли на отделните модули, бихме могли да допринесем за осъществяването на драйверите, мислейки от страната на интерфейса. Т.е. правейки системата още по-силно необвързана в различните слоеве, не само чрез въвеждане на layring шаблон. Както споменахме и по горе - искаме да можем да тестваме различните модули самостоятелно, отделени от техните зависимости — други класове с които работят, които бихме макетирали.

II.1. Обща диаграмова структура

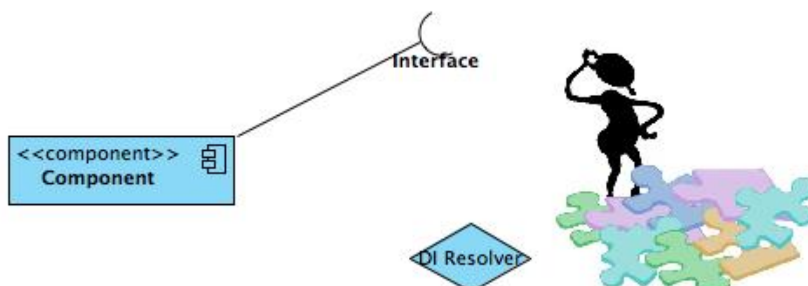
Нека разгледаме следващата алтернатива в нашия шаблонен речник, с която можем да постигнем желания по-горе резултат:



фиг. 4.4 Accounting System – слоеве и внедрена зависимост

II.2. Използвани шаблони

Освен слоевете, като тяхно допълнение, въвеждаме още един софтуерен шаблон силно залегнал в последните години в практиката — *Dependency Injection* или внедрени зависимости. Този шаблон практически ни предоставя обектите (имплементации на интерфейси), които са нужни на друг свързан обект (неговите зависимости). Това се прави вместо да се конструират в него самия и да се търси там съответната имплементация. Като реализация това може да стане примерно чрез Фактори метод шаблона. Ще имплементираме този шаблон чрез просто подаване на имплементации на интерфейси в конструкторите — ръчно внедряване. За нашата система най-същественото разделение е най-вече това между отделните слоеве. Т.е. един бизнес модул работи с конкретен *Data Access* модул, който се внедрява при конструкцията му. Разбира се, има и много приложни рамки предоставящи това, с които би могло да се работи и много по-ефективно.



фиг. 4.5 Внедрена зависимост

На фиг. 4.5. се представя компонента картинка на главните участници във внедрените зависимости шаблона, където т.нар. *DI Resolver* се грижи за това коя имплемнтация на интерфейса нужен на компонента да предаде.

С това и самата ни тестова стратегия се обогатява. Ще имаме възможност да макетираме интерфейсите като не викаме конкретната му функция, а предефинираме резултата, който искаме от нея. Което от своя страна може да имплементираме както *unit*, така и интеграционни тестове и докажем и поддържаме качественото на нашия софтуер. Дори да се правят промени в бъдеще, а това е почти сигурно, по бърз и ефективен начин доказваме, че цялата система работи погледната от всичките си страни дори и след направените промени.

II.3. Постигани качествени атрибути/изводи

С въвеждането на DI допринасяме и за добавянето на още два важни архитектурни качествени атрибути — *Adabtibility* и *Changability*. След като работим с интерфейси, то промяната на зависимите импементации става само на едно място — при *DI Resolver*. Примерно дали плащаме с SOAP или NVP базирано плащане, това няма да доведе до никакви промени в модула, който работи с нашата имплементация на плащането, т.к. внедряваме интерфейс.

II.4. Други видове връзки между шаблони

С въвеждането на внедрената зависимост като надграждане на слоевете забелязваме и друг вид взаимовръзка между двата шаблона - *Pattern complements* или допълващи се шаблони. С тези два шаблона (Слоеве и Внедрените зависимости), спокойно можем да кажем, че имаме една достатъчно голяма абстракция на архитектурата на нашето приложение, а и не само това. Също както в примера, който дадох при обяснението на Допълващ се шаблон, където *Dispolsal Method*, се грижи да освобождава инстанциите, създадени от *Creational method* при тяхното завършване на работа, така и тук виждаме как *Dispolsal Method* фактически допринася за липсващото парче от пъзела в слоевете — пълна и цялостна независимост между Слоевете и Внедрените зависимости.

III.Цел: осигуряване по висока степен на защита, абстракция при работа с плащането и гъвкав избор в контекста на слоева архитектура

Нека сега минем през път в нашата картина на шаблонния език, в който не участва Вградената зависимост. Това ще го направим с цел да се види ако оставим Слоевете като решение и помислим от друг аспект. До какво би ни довело това в системата, до постигането на какви качествени атрибути?

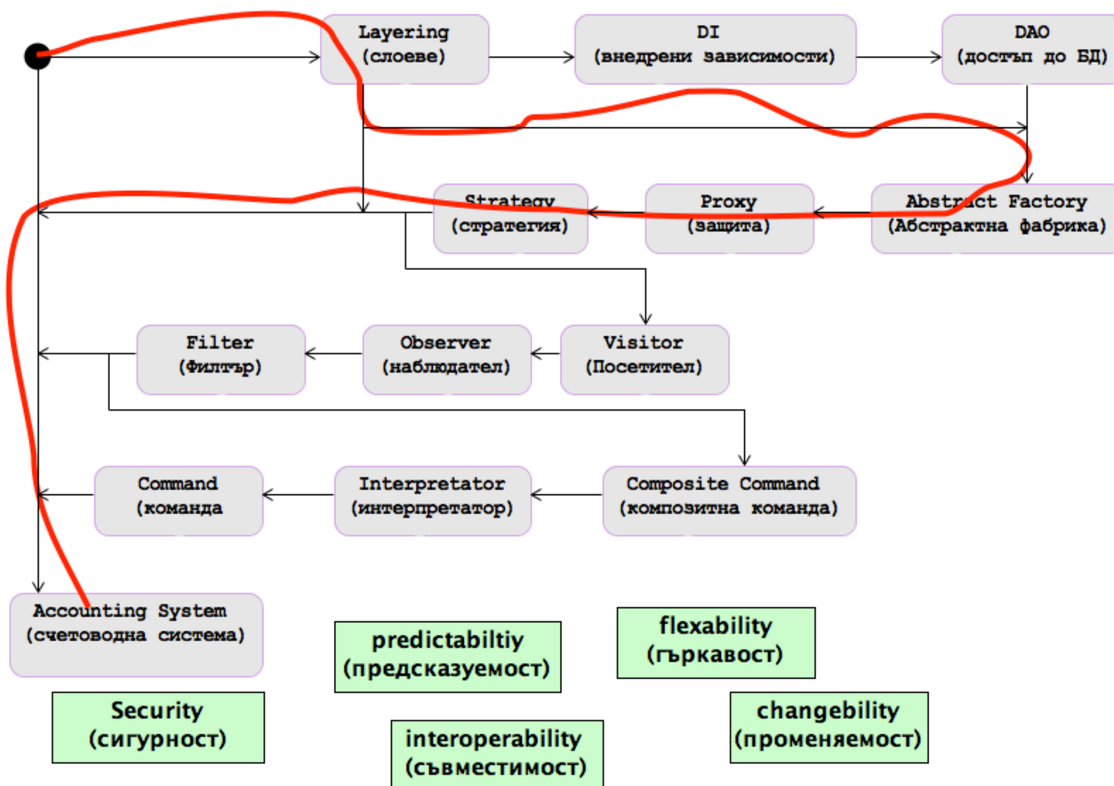
Наясно сме, че ще имаме различни видове свързване към доставчици на плащания (*Payment Providers*) — SOAP и NVP базирани. За двата видове връзки ще искаме подходяща сериализация на заявката за плащане и десереализация на отговора от доставчика. Ще

изискваме и подходяща връзка към канала, който се предоставя за връзка. Това предоставя набор от класове (сериализатор, десереализатор и връзките), който трябва да се създава при съответния вид платежена услуга. Ще трябва по време на изпълнение да се предоставят и подходящи стратегии за построяване/четене на заявките/отговорите.

От друга страна, системата трябва да осигури и достатъчно високо ниво на сигурност чрез подходящата за това оторизация на потребителя, желаещ да осъществи заявката. В крайна сметка става дума за една от най-чувствителните части на нашата система.

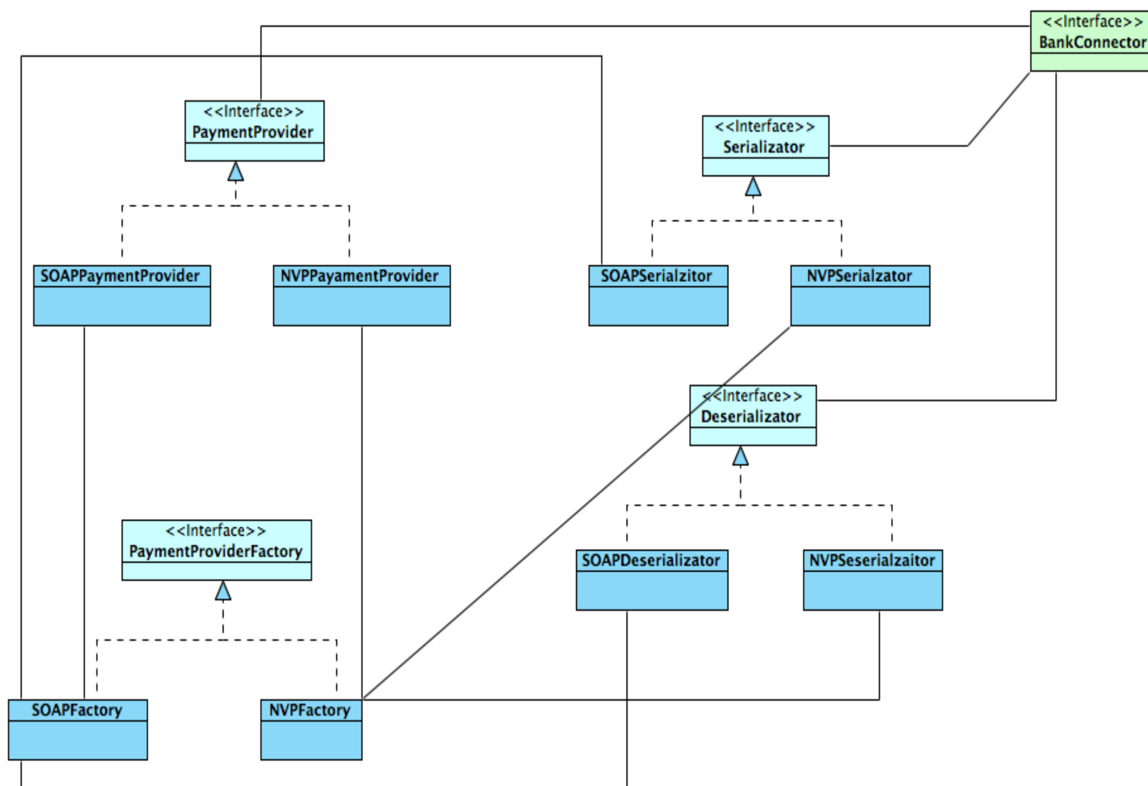
III.1. Обща диаграмова структура

В следната диаграма на фиг. 4.6 представяме шаблонния език, през който минаваме, за да имплементираме изискваните по горе архитектурни атрибути (качествени характеристики)



фиг. 4.6 Accounting System – сигурност, съвместимост, предсказуемост

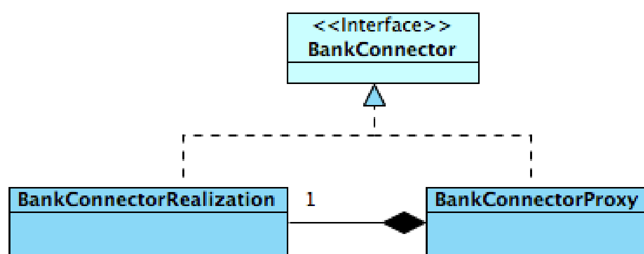
На фиг. 4.7. се отразяват основните интерфейси и класове на казаното до тук по отношение на *payment provider* имплементацията ни:



фиг. 4.7 Payment Provider, класова диаграма

III.2. Използвани шаблони

Тук използваме Абстрактната фабрика, за да получим семейството от класове, с които клиента, *BankConnector* модула при нас, ще си послужи, за да извърши плащане с определен вид сървис връзка (SOAP/NVP). Като добавим и самото Proxy при този клиент:



фиг. 4.8 Proxy в Payment Provider

ни се добавя и сигурността, с която искаме да осигурим достъп до същинската реализация на връзката (BankConnectionRealization), само тогава когато BankConnectionProxy допусне потребителя, оторизирайки го да извърши определената услуга.

III.3. Постигнати качествени атрибути/изводи

Построявайки по този начин системата ни, добавяме сигурни архитектурни качествени атрибути като сигурност (*security*): така няма да позволим на потребителя да извърши операцията, ако няма права за нея; гъвкавост (*flexability*): с лекота можем да сменяме типовете услуги за плащане — SOAP или NVP; предсказуем (*predictability*): няма да представлява затруднение да добавим нова типова връзка за плащания, примерно чрез JSON модела.

III.4. Други видове връзки между шаблоните

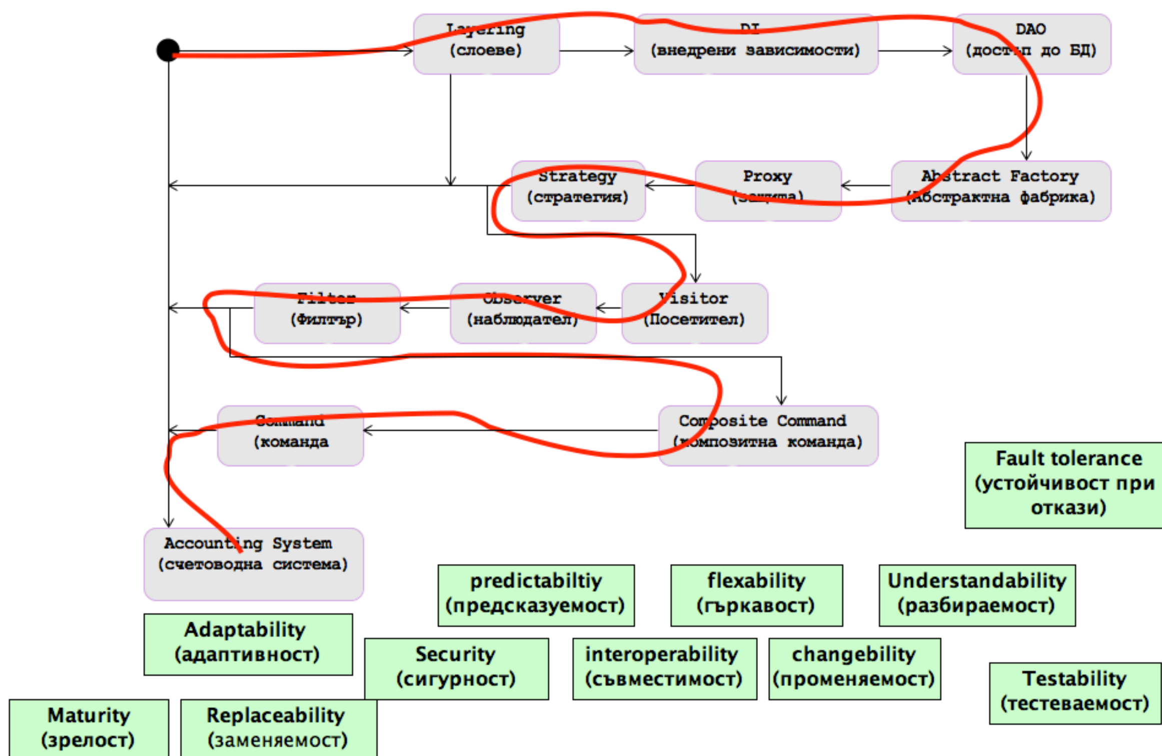
В тази картинка можем да видим и друг вид връзка между двата шаблона: *pattern sequences* или Модел на последователността. Няма нужда Абстрактната фабрика да строи семейството от класове за клиента си, ако този клиент не допусне до него логиката. Т.е. абстрактната фабрика е пост фактор на Проксито в *Payment Provider* модула.

IV. Цел: Пълно удовлетворяване на функционалните изисквания в шаблонно ориентирано програмирано. Достигаме на набор от качествени атрибути, включително и архитектурните драйвери заложени в системата

Нека в този пример включим в шаблонния ни речник както пътя от втора точка, така и пътя от 3-та. В допълнение ще добавим и още шаблони, нужни за пълна функционална имплементация, изисквани от приложението ни.

IV.1. Обща диаграмова структура

Нека разгледаме този път във фигура 4.9. отразяваща пълна имплементационна картина на шаблони в нашето приложение.



фиг. 4.9 Accounting System - Зрялост

IV.2. Използвани шаблони

Запазваме основните шаблони, с които започнахме още в началото: слоева архитектура (*Layers*), вградени зависимости (*Depedency Injection*) и *Data Object Access* (DAO). Следват Стратегията (*Strategy*), Защитата (*Proxy*) и Абстрактната фабрика (*Abstract Factory*), с които увеличаваме флексибилността. С това се конфигурира подходящата съвкупност от имплементации, задоволяващи изискванията на конкретен клиент с конкретни нужди.

IV.3. Постигнати качествени атрибути/изводи

Имаме изискване гласящо: нека виждаме конкретен служител с данните, принадлежащи към него като заплата, имена, отдел, банкова регистрация и т. н. в различен вид - табличен и списъчен. Не е трудно да се забележи нуждата от Наблюдателя (*Observer*), с който ще можем гъвкаво (*flexability*), без да се налагат големи промени, да сменяме наблюдателите на Employee класа, нашият *Observable,* който ще се грижи за изобразяването на информацията за служителя. Има и друг плюс при прилагането на този шаблон: няма да се налага затормозяване на самият Employee, където ще се пише логиката по отношение на отразяване на съдържанието му. Друг качествен атрибут, който можем да забележим е предсказуемостта (*predactability*) в такъв вид дизайн решение. Няма да представлява затруднение добавянето на нов вид изглед на служителя или премахването на съществуващ такъв.

Следващия шаблон Филтър (*Filter*) също играя ролята на допълнителна функционалност към нашия Employee. Неговата задача е да обхожда служителите на инстанция на Служител

класа, които са под неговата йерархия и да ги филтрира по тип банка, към която са вързани. Това е идеален инструмент, когато искаме да платим заплатите на всички в компанията, които са регистрирани в конкретна банка. С този шаблон отново допринасяме за гъвкавост и предсказуемост на нашата система.

Така се постига променяемост (*changability*) или *Maintainability*. След като имаме функционално ориентиран модул декомпозиция (различни видове изгледи и филтри, имплементирани в различни класове като реализация на общ интерфейс) ще може лесно да се намери частта, където ще е необходима промяна или поправки на сорс кода.

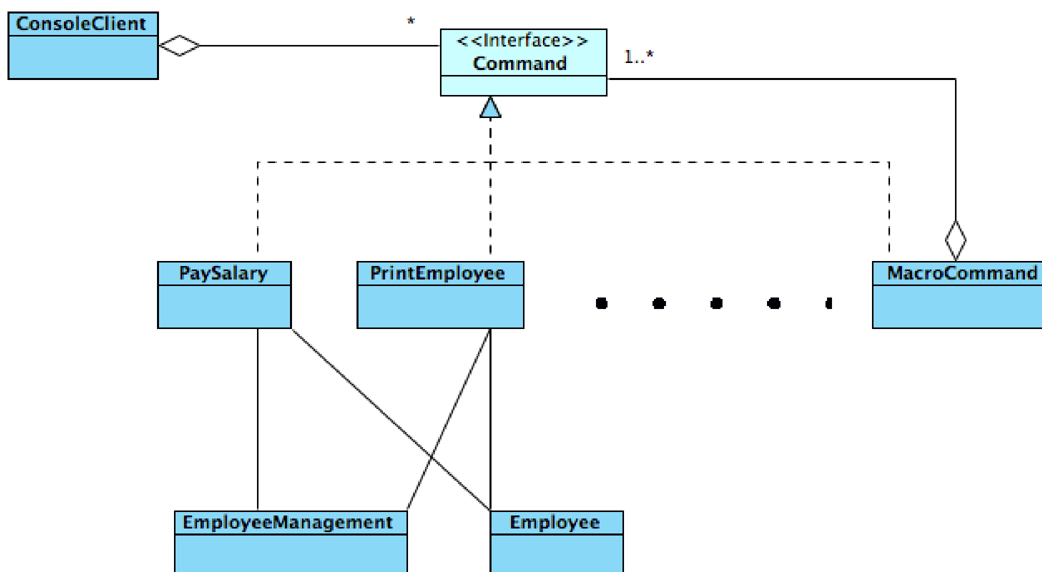
Ако вземем всичките до тук шаблони, които включихме по пътя си, можем да подчертаем и други качествени атрибути, които удостоверяват качеството на продукта. Разбираемост (*Understandability*), съвместимост (*interoperability*) и Устойчивост на откази (*Fault-tolerance*) са един от тях.

Системата се декомпозира до модули, по идеален начин гранулиращи функционалните изисквания. Наименованието на модулите и класовете, както и конкретностите, с които се свързват е сорс код коментирано — самият код документира и написаното.

Приложението прилага и доказан подход при сериализация и десереализация на заявката и отговора от *Payment Provider*, с който се свързваме. С това увеличаваме *interoperability*: възможността да комуникираме с различни системи и да разменяме информация помежду им. С дефинирането на стратегия и модулно-интерфейсно базирано моделиране на сериализационни и десериализационните логики се увеличава и лекотата, по която превръщаме сървисните съобщения (XML, JSON или HTTS POST *variables Pairs*) в локални бизнес модели и обратно.

Продължаваме по нашия път в дефинирания от нас шаблонен език и стигаме до имплементирането на потребителското взаимодействие. Понеже сме избрали конзолна такава чрез въвеждане на команди в командния интерпретатор лесно можем да достигнем до извода да използваме Команда (Command) и Интерпретатор (Interpretator) шаблоните.

За да можем да изпълняваме поредица от команди, ще включим и Композитната команда: *Compsite Command* при която имплементираме заявка чрез обект, с който можем да боравим в бизнес логиката и предаваме инстанцията й между различните функции.



фиг. 4.10 Client Console – Композитна команда

С лекота можем да направим връзката между низ (*string*), с който интерпретираме командата и имплементацията на интерфейса *Command*, изпълняваща тази команда. Повиквателя (*Invoker*) е ConsoleClient класът, който предава въведеното в конзолата на клиента заявка за изпълнение. Получателите (*Receiver*) са според вида команда, ако е за изписване на информацията на служителя, то тогава Employee ще е получателя. Ако е свързана задача с управлението на служителите, то това ще е EmployeeManagement класът.

Когато комбинираме всичките шаблони, изредени до сега, ще забележим как те се поддържат един в друг и правят цялостната имплементация много по здрава и качествена. А това от своя страна е и дефиницията за Зрялост (*maturity*) шаблона. Зрялост по отношение на приложението, по отношение на това, че погледнато от към почти всяка имплементационна страна, която би довела до подводен камък в софтуера, е детайлизирано проучена и анализирана с намирането на подходящо решение за нея. Зрялост по отношение на това, че са създадени шаблонни връзки, които имплементират решение идеално вписващо се в нашия домейн и проблемна област.

4.3. Пример за създаване на изцяло нов шаблон : Integration Payment Provider шаблон, като комбинация от G&F шаблони

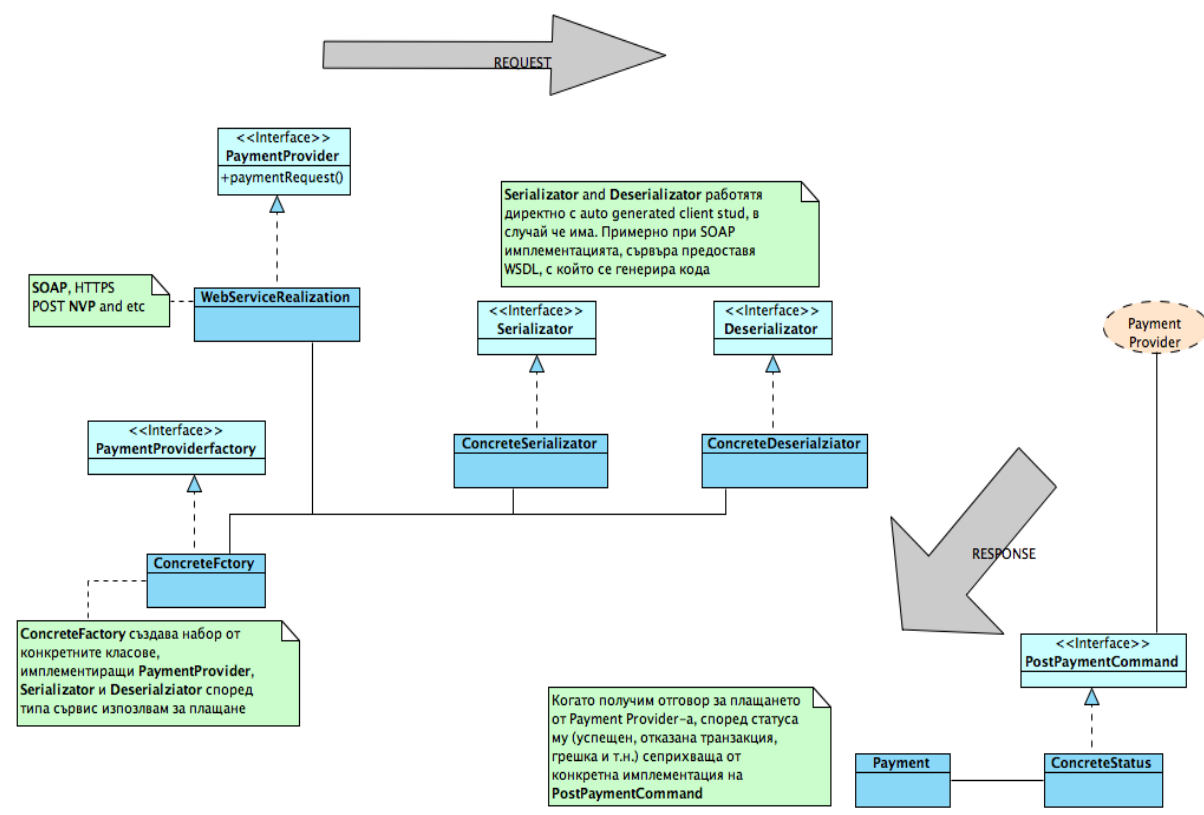
В горната подточка на главата (4.2) разгледахме всички видове взаимовръзки на шаблони като най-вече наблегнахме на шаблонния език, т. к. с него се прави детайлен анализ на това какво имаме като ресурси и знания - архитектурно и шаблонно базирани. Избира се тази алтернатива, която най-подходящо се вписва във функционалните изисквания и се достигат качествените атрибути, определени като архитектурни драйвери за системата.

В тази подточка на главата ще наблегна на една също така привлекателна и развиваща се с бързи темпове шаблонна връзка, а именно *Pattern compounds* (модели на съединение).

Както подчертахме още в началото, Модели на съединение представляват съвкупност от независими един от друг шаблони, вече съществуващи такива, които заедно генерират един нов шаблон. По-простите примери, които дадохме в уводната част са композитната команда, която е композиция от команда и композит шаблоните. От бизнес гледна точка, можем да дадем примерно като *Enterprise Service Bus* или *Service Broker*, които силно навлизат в сървис ориентирана архитектура в последните години.

Тук ще наблегнем на проектирането на нов шаблон като комбинация от съществуващи такива за темплейтно решение на интеграция на Payment Provider в какво да е приложение (web, mobile, enterprise и т. н.), идеално съвпадащо с домейна и проблемната област, с която боравим в момента. С това даваме скелет от добре известни части на разработчика, имащ за задача да имплантира плащания в системата.

Нека погледнем фиг 4.11, отразяваща нейната композиция от шаблони:



фиг. 4.11 Payment Provider Pattern

Шаблоните, с които ще работим са Абстрактната фабрика (*Abstract Factory*) и Командата(*Command*). По избор в зависимост от контекста, могат да се добавят и Проксито (*Remote Proxy*) и Стратегията (*Strategy*). По нататък ще дадем предположения с каква цел.

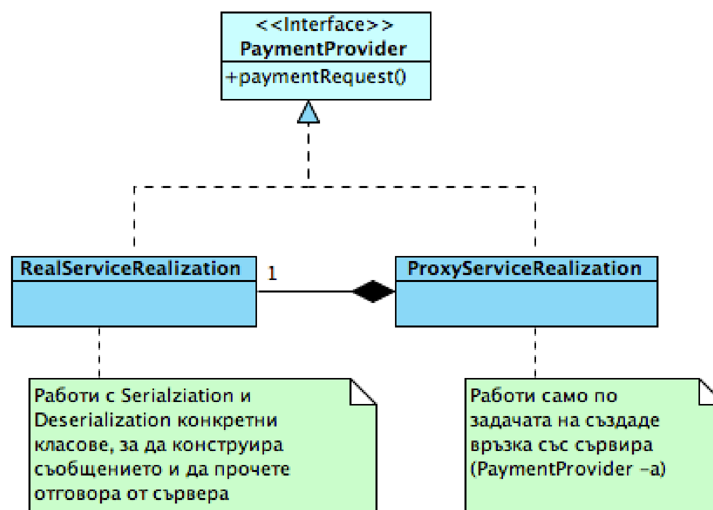
Използваните шаблони в диаграмата по горе са първите два – Абстрактна Фабрика и Командата. На клиента, който използва сървис имплементациите, се връщат конкретна

инстанция на WebServiceRealization и на ConcreteStatus, които имплементират зададените интерфейси.

С PaymentProvider се дефинира договорът, който трябва да се изпълнява от всяка негова имплементация, примерно най-известната: SOAP, или по-олекотената и опростена версия: *Name Value Pair* с HTTPS POST заявка. Всеизвестният PaymentProvider — PayPal предлага и двата вида интеграция и клиентът избира коя от тях да приложи. Клиентът от своя страна прави този избор отново в контекста, в който работи. Примерно ако се изисква високо ниво на сигурност, то SOAP имплементацията предлага по сигурна такава с добавени *WS - Security* техники. При прилагането на подобни решение се има предвид, че и времето за имплементация ще е повече от NVP в случая.

В нашето конкретно приложение тези конкретни реализации се казват: SOAPPaymentProvider и NVPPaymentProvider, т. к. банките, с които работим предлагат един от двата вида сървис плащания.

В случай, че самата връзка с *gateway* на сървъра на PaymentProvider е по сложна за имплементиране се изисква по-голямо внимание за оторизация, четене от конфигурационни файлове и т. н. Можем да добавим в нашия композитен шаблон и Отдалеченото Прокси (*Remote Proxy*), което ще се грижи за тази част от бизнес логиката — фиг. 4.12



фиг. 4.12 Payment Provider Proxy

Това е нашия PaymentPrivder договор, който е обогатен с поне още един клас имплементиращ го: ProxyServiceRealization. Той ще се грижи за това да осъществи връзка със сървъра с конструираното от реалната реализация за PaymentProvider: RealServiceRealization, която от своя страна ще работи с Serialziation и Deserialziation инстанциите, за да оформят самото съобщение (SOAP файла ако е SOAP имплементация).

Абстрактната фабрика в този случай ще има за задача и да знае и инстанциира подходящата двойка Прокси — Реална имплементация.

Абстрактната фабрика в нашия шаблон ще има за цел да създаде правилната комбинация от всички конкретни класове в зависимост от типа уеб сървис имплементация. Ако това е SOAP, то ще се създаде семейство от SOAPPaymentProvider, SOAPSeralization, SOAPDeserialziation и ProxySOAPPaymentProvider, в случай че и проксито е включено, като SOAPSeralization и SOAPDeserialziation са необходими фактически на SOAPPaymentProvider. На клиента се подава SOAPPaymentProvider (или ProxySOAPPaymentProvider). Разбира се същите разсъждения се отнасят и за NVP имплемнтацията.

При повечето доставчици за плащания едно плащане или превод на пари се осъществява в следната последователност:

1) Изпраща се заявка за плащане

2) От сървъра се връща един *redirect* URI, с което потребителя има примерно 15 минути време да осъществи същинското плащане

3) Ако го направи, сървърът връща отново отговор към клиентското приложение, с което се връща и статуса, представляващ отговор на самото плащане (успешно, отхвърлено от банката, грешка в сървира и т. н.)

За тази цел допълваме нашия шаблон с още един от G&F шаблоните: Командата. Това ще го направим с цел да имаме гранулярност по отношение на това къде се обработва отговорът от *gateway* сървъра. Т.е. ще имаме отделен клас, който ще се грижи за логиката при успешно плащане, при грешка в сървира, при отхвърлена заявка и т. н. Идеята е, че в различните случаи има различна бизнес логика — в единият случай искаме да направим транзакция за успешното плащане, докато в другия да изтрием от базата данни за отказани плащания да речем. В случай че имаме общи стъпки за всички видове отговори — писане в логове — можем да направим PostPaymentCommand абстрактен клас и там да реализираме тези стъпки. Командата като шаблон не ни забранява да направим това.

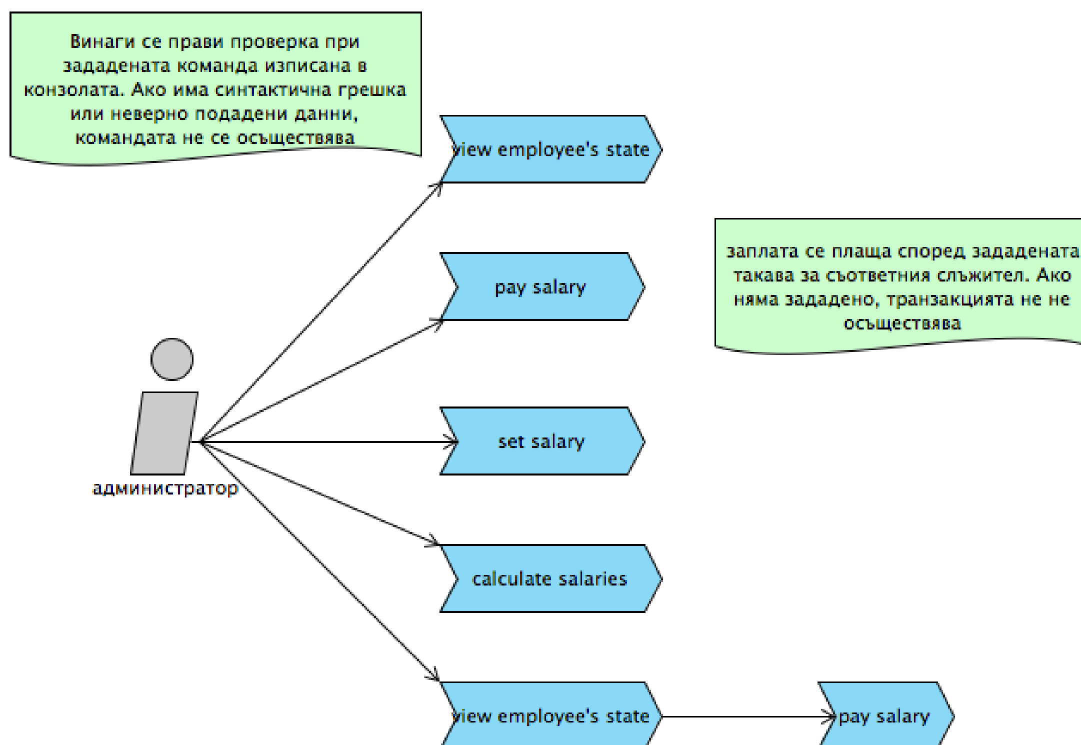
4.4. Бизнес процеси

По отношение на бизнес процесите точно нашето приложение няма конкретно дефинирани такива, тъй като целта ни с него е да покажем как шаблоните работят заедно. В тази подточка на главата ще се опишат на кратко основните функционални изисквания, като се наблегне главно на процесите и как те биха могли да бъдат зададени.

Приложението е конзолно ориентирано. Нямаме графичен потребителски интерфейс. Взаимодействието с потребителя е на конзолно ниво чрез низове — команди написани в командния ред, които съответно се интерпретират от програмата и се изпълняват. Едновременно може да се изпълнява повече от една команда. Към момента командите и техните значения са:

- **viewemployeeaslist <arg1>**: изписва на конзолата статуса на служител с идентифициран номер arg1 във вид на списък от двойки: име на аргумента — стойност
- **viewemployeeastale <arg1>**: изписва на конзолата данните на служител с ИД arg1 в табличен вид
- **setsalary <arg1> <arg2>**: задава заплата, зададена чрез arg2 на служител с ИД arg1
- **paysubordinatessalariesbybanktype <arg1> <arg2>**: изплаща заплатите на служителите, които са под йерархията на служителя с ИД <arg1> , които са с банкова принадлежност обозначена с arg2. Това е команда, ако искаме да платим на всички служители заплата към дадена банка, задаваме ИД-то на директора на компанията
- **addemployee <arg1> <arg2> <arg3> <arg4>**: добавя нов служител с първо и второ име съответно arg1 и arg2, заплата arg3 и тип банка arg4
- както и комбинация от команди.

Процесите бихме изобразили под формата на диаграма чрез фиг. 4.13



фиг. 4.13 Use Case на процесите

Тъй като няма ниво на идентификация и оторизация, няма да има големи if-else нива. Примерни команди следвайки диаграмата са :

```
> viewemployeeastale 1
> setsalary 1 50000
> addemployee Pesho Peshev 1000 DSK
> viewemployeeastale 1 setsalary 1 60000
> |
```

4.5. Извод

В тази глава, най-дългата и обстойно анализирана, като имаме предвид спецификата на темата на дипломната, разгледахме едни от най-екзистенциалните моменти при всяка софтуерна архитектура: дизайн на модули, избиране на подходяща стратегия, достигане на качествени атрибути, и ефективно имплементиране на функционалните изисквания в контекстната област и домейн.

Минахме през обяснение на едни от широко прилаганите и търсени качествени атрибути, дефинирахме шаблони от различен тип и най-важните две подточки: анализиране с шаблонен език и други типове връзки като шаблони на съединение и даването на пример за създаването и дефинирането на нов шаблон — Integration Payment Provider.

По отношение на шаблонния език детайлно минахме през различни пътища и алтернативи, даващи ни различни резултати.

Стъпките, които можем да приложим и над които трябва да се замислим при генерирането на шаблонен език биха могли да са:

- Какви са ключовите проблеми, които искаме да разрешим и в какъв ред;
  - В нашия случай, имахме проблеми с това как да интегрираме едновременно два вида *payment providers*;
  - Как да постигнем високо ниво на сигурност, особено по отношение на плащанията;
  - Как компонентите на различните видове сървиси, с които превеждаме заплатите, да бъдат конструирани по ефективен начин;
  - Как да постигнем високи нива на качествени атрибути като модифициране, адаптивност, гъвкавост, тестваемост и т. н.

- Как да решаваме всеки индивидуален проблем в зависимост от контекста:
  - Тук боравим с домейна, в който прилагаме шаблонно ориентирания подход. Примерно нека си представим, че нашия *Integration Payment Provider* шаблон новодефиниран от нас, се прилага във високо конкурентна среда на нишки. Тогава трябва да се добавят и механизми за синхронизация към така конструираната картина. Може да се добави т. н. *half-synch, half-asynch* шаблон, който има грижата за синхронно обработване за заявките.

- Какви алтернативи съществуват за разрешаването/имплементирането за всеки проблем:
    - тук вече идва и моментът на богатите знания и опит, придобили се с времето и практиката. Построяването на различни пътища води до различни решения с различно качество, въпрос на функционални и не-функционални изисквания.

Ако трябва да обобщим какво е шаблонния език, то това е интеграционен подход, групиращ в себе си група от взаимосвързани шаблони, за да позволи високо качествено преизползване на софтуерен дизайн и архитектура. Не просто използваме индивидуални шаблони в изолация един от друг. Не просто използваме шепа шаблони събрани в едно, както при другите видове връзки (Шаблони на съединения или Допълващи се шаблони). Вместо това имаме шаблонен език, позволяващ ни да навигираме из между дизайн решения и избираме алтернативи, които дават най-голям смисъл, имайки предвид ключовите ни архитектурни и дизайн изисквания.

Друг момент на който наблегнахме е дефинирането на нов шаблон от вече съществуващи такива. В случая опростихме нещата с използването само на G&F шаблони. С това даваме пример как шаблоните могат, стига да се използват правилно, да дадат съвсем нови решения само и единствено като използваме правилната комбинация от вече добре познати такива. А ако искаме да усложним нещата в нашия нов шаблон, можем да добавим и други решения, като се анализира съвкупността им отново в шаблонен език. Примерно може да се анализират пътища/алтернативи във високоскаларна среда и в съвсем последователна еднонишкова среда. С това да избира различни пътища, където трябва да се грижим и за синхронизационни модели при скаларни потенциали. В този случай бихме добави и разглеждането на път минаващ през друг шаблон като Реактор или *Half-synch / half-asynch*, които спомагат за синхронизация на данни.

Като крайно заключение мога да добавя, че е въпрос на избор и отговорност на всеки софтуерен разработчик как и до колко задълбочено да погледне на анализ етапа на продукта/сървиса и на стратегиите и архитектурните решения, през които може да избира.

Глава 5. Проектиране на примерно приложение с използване на шаблони

Настоящата глава от дипломната работа описва проектирането на примерно софтуерно приложение с изполването на шаблони за проектиране съгласно направения в Глава 4 анализ. Разработеното примерно приложение представлява базов прототип с ограничена функционалност, понеже целта е единствено да се демонстрират практически анализираните дотук начини за използване на шаблони. Поради същата причина не е представен анализ на бизнес изискванията към примерното приложение. Описват се и качествените атрибути, постигащи се с изпозлваната комбинация от шаблони. Понеже разботим с чист Dependency Injection, може да се изпозлват конкретни имплементации (приложни рамки и технологии), с което приложението може да се адаптира и в практиката.

5.1. Обща архитектура — слоеве, модулна декомпозиция и deployment стуктура

От архитектурна гледна точка приложението лежи на шаблонно ориентирания софтуер, какъвто е и контекста, в който се реализира дипломната работа.

Основните принципни са:

- архитектурата е на базата на интерфейси с цел използване на вградени зависимости, имплементирана ръчно. Не се използва готова приложна рамка (*framework*) за това;
- конструирана е от напълно независими един от друг слоеве;
- комуникацията между слоевете става чрез интерфейсите;
- вградената зависимост решава коя инстанция да върне на извикващия компонент;
- приложени са два вида тестови стратегии: *unit* тестване и интеграционен, тестващ всичките компоненти, включени при заявката;
- пакетирането става в архивен файл с .jar разширение.

По долу са изброени основните моменти при изграждането на архитектурата и описанието й.

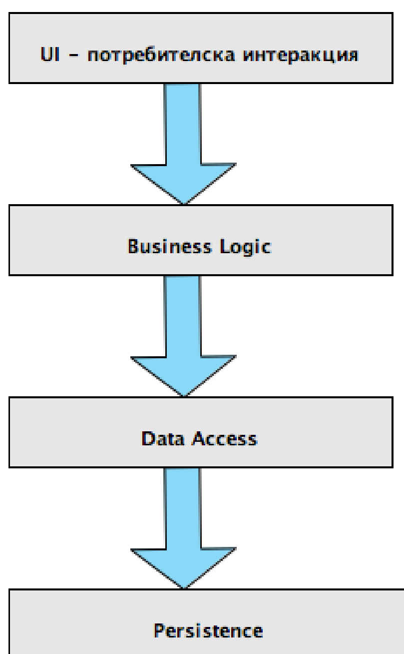
5.1.1. Слоеве

Спазването на един от основополагащите принципи — напълно независими един от друг слоеве — се имплементира като:

1-во: извикването става само и единствено в едната посока.

2-ро: кой слой коя компонента от по-долния слой ще използва се определя от вградената зависимост или нашето Фактори, имплементиращ връщането на подходящата съвкупност от инстанции.

Фигура 5.1 дава ясна представа за дефинираните слоеве и начина на комуникация по между им:



фиг. 5.1 Архитектурни слоеве

Представените на фиг.5.1 архитектурни слоеве са както следва по долу:

- **UI** — презентационен слой, който ще се грижи главно за взаимодействието с потребителя. Ще имплементираме чист конзолен такъв, който ще бъде достатъчно независим от по-долните слоеве, за да може да се смени примерно с графичен интерфейс като *swing* или дори *web* такъв. Той работи с договорите (интерфейсите) на компоненти от бизнес слоя, предоставени от имплементирана вградената зависимост;

- **Business Logic** — тука ще включим бизнес логиката, стъпките от действия, които ще реализират същинската част от приложението: бизнес изискванията. Ще работим в парадигмата на сървисите, възможно най-гъвкаво и преизползваемо, за да можем в бъдеще да имаме възможност да оркестрираме и бизнес потоци с тях. Ще наблегнем отново на разделянето на интерфейсни и импементиращи ги компоненти. На клиента се предоставят договорите. Този слой работи със слоя за достъп до базата данни (*data access layer*), който може да е *in memory, sql relation* с *JPA, NoSQL* и т. н. като достъпа до него става отново чрез вградената зависиост. Тя преценява какъв вид

имплементация на контракта да предостави, в зависимост от това дали приложението е в режим на *in memory* или различен по типаж диск базирана база данни;

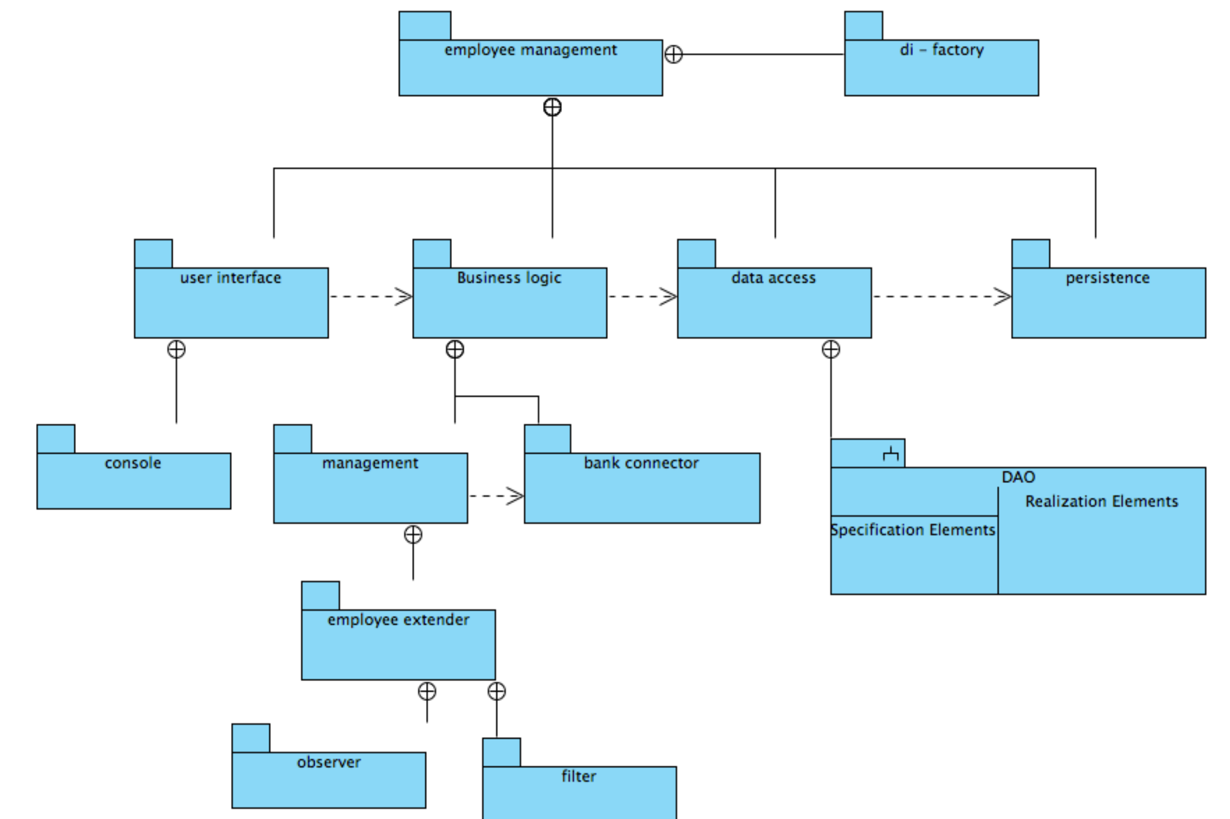
- **Data Access** — предоставя достъп до *Persistance layer* — базата данни по унифициран и опростен вид. Предоставя имплементация на основните операции с база данни: CRUD — *Create Read Update Delete* или създаване, добавяне на нов ред в таблица; чете на данни; обновяване на данни и изтриване. Дефинират се различни по вид заявки с необходимите критерии, като филтри страниране, които *data access* слоя се грижи да построи в зависимост от типа база данни използвана в контекста на режим на работа. Отново на клиента се предоставят интерфейси. Коя имплементация зависи от стратегията дефинирана във вградената зависимост;

- **Persistence** — слоят, в който ще се описват като прости *POJO — Plain Old Java Object — database* класовете представящи в *Java* вид таблиците в базата данни, в случай на диск базирана стратегия. Ако е в *in memory*, ще работи с текущия *Java Heap* и отново ще предоставя същите класове.

5.1.2. Модулна декомпозиция

В тази подточка ще разгледаме модулната декомпозиция на проекта. Или как е конструиран в пакети, под-пакети, класове и съответно договори за тях — интерфейси. Кой къде се намира и в каква йерархия са. Ще започнем от най-горното ниво, наречено *employee management* и ще слизаме надолу в йерархията поетапно.

На фигура 5.2 е показана базовата йерархия на пакети (модули) и под модули, на които системата лежи:



фиг. 5.2 Модули в Accounting System

В основата лежи employee manegement или главата директория от която се влиза в приложението.

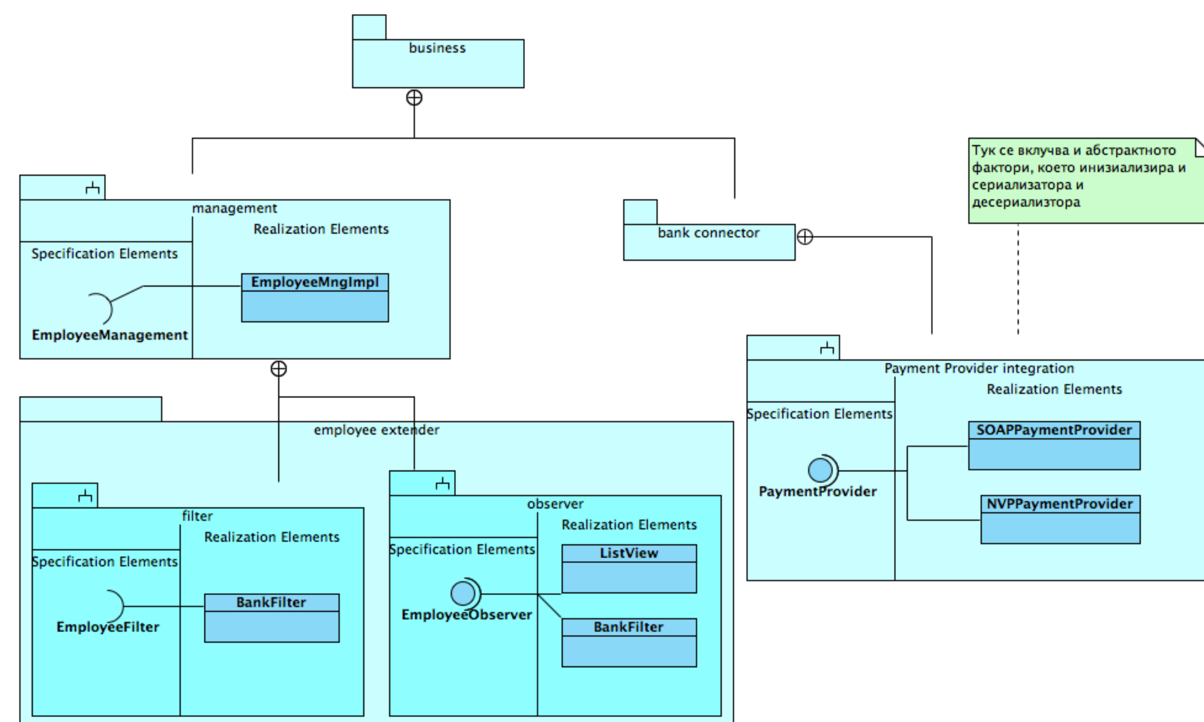
Веднага след нея са подредени именно четирите слоя в различни подпапки: *user interface, business logic, data access* и *persistence*. С това се дава ясна представа на разделението на сорс кода по значение и принадлежност, с което се постига и ниска зависимост и смесване на контекстна логика. По такъв начин конструирана архитектура (слоева) на разработчика ще му е лесно да се ориентира, в зависимост от функционалната спецификация, къде в кода ще са нужни промени или модификации. Ролята на модула и неговата основна функционалност съвпадат с наименованието му, зададено в гореописаната структура.

Заедно до тези четири модула стои и още един, наречен *di — factory*. Тук се намира ръчната имплементация на вградената зависимост (*dependency injection*), с която ще си боравим в приложението. Конструктор на клас, който ще се създаде, ще съдържа в себе си инстанции на класовете от които зависи. Т.е. задава се вътре в себе си. Ще се задържаме на бизнес ниво, т.е. вградената зависимост ще управлява зависимостите между класове с бизнес логика в тях. В нашето Фактори ще се съдържат почти 90% от new операциите. То ще се грижи за графовите инстанции на класове с корен, по изисквания от клиента. Няма да

съдържа нищо повече - като писане в изхода/записване на информация, хвърляне на изключения и т. н.

Това се прави с идеята, ако клас А зависи от друга инстанция на Б, то само и единствено А и *factory* трябва да знаят за това и да се справят с промяната. Това е удобен подход за реализация на *dependecy injection*. Съществуват разбира се и други стратегии за имплементирането му. Ако приложението става по сложно, могат да се направят повече от едно фактори, с цел да се задоволят различните графови конструкции, които биха се изисквали от приложението. С това бизнес класовете имат само за цел да имплементират бизнес логика. Няма да се намери *new*() в тях, грижещо се за това как се създава зависимия от него клас. Дори няма да знае за конкретните имплементации. Това е грижа за факторито.

С помощта на фиг. 5.3. ще раздробим още повече *business logic* модула, за да видим какво по точно се съдържа в него:



фиг. 5.3 Модулна декомпозиция

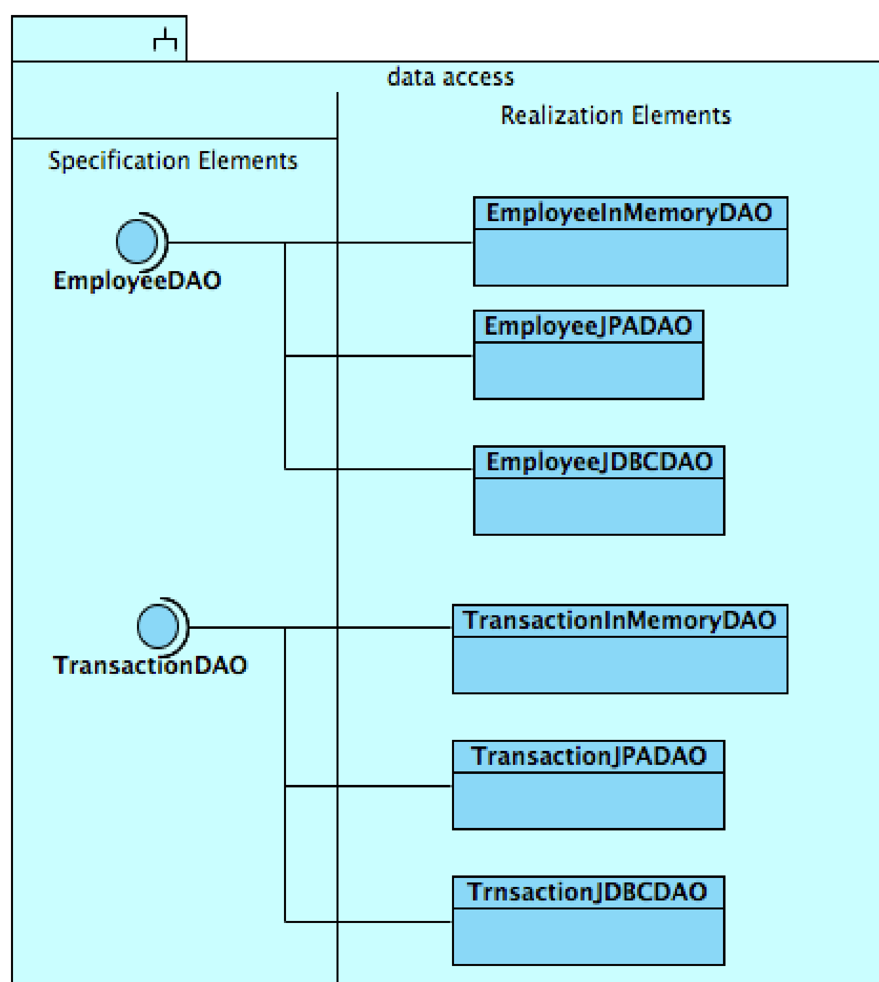
Това е един от съществените модули в системата. Сърцевината, мястото където се имплементира същинската бизнес логика.

На клиента се подава интефрейса EmployeeManagement, като неговата имплементация ще се грижи за операциите, които се изискват от него. Единият подмодул: management, предоставя тази интерфейс. Той е зависим от BankConnector модула, който се грижи за връзка със съответната банка. Която от своя страна предоставя начин на плащане чрез експериментиран доставчик на плащанията - Payment Provider integration модула, който се грижи да интеграцията на конкретен Payment Provider уеб сървис имплементация: SOAP

или Name Value Pair стратегията. Той се грижи да се свърже с клиентския stub, който в повечето случаи е автоматично генериран от java инструмент.

От там нататък, management модула има също така и подмодул, наречен employee extender. В него се съдържа едно разширение на Employee POJO клас, което се намира в persistence слоя. Това е направено с цел да добавим бизнес логика към класа с набор от полета и методи за достъп до тях. Тук се имплементират филтрацията на елементите на агрегатора Employee и с помощта на шаблона Observer, наблюдателите, които ще имат за цел да изписват в конзолата по различен начин съдържанието на конкретен служител.

Във фигура 5.4 се отразява какво съдържа *data access* модула:



фиг. 5.4 Data Access класове и интерфейси

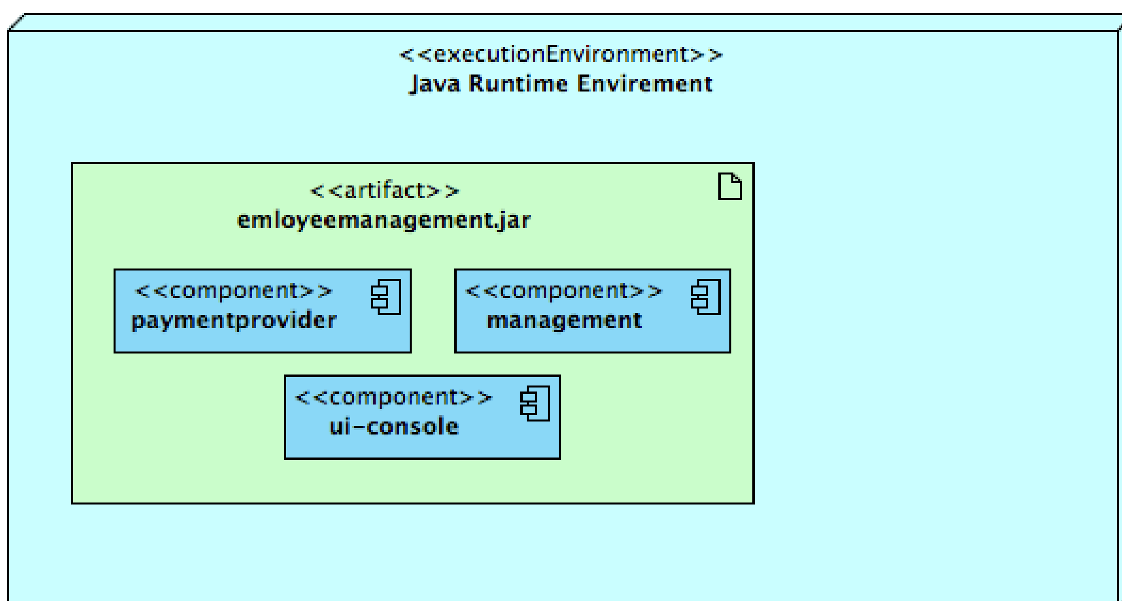
Виждаме разделение на интерфейси (договори) и техните видове имплементации. Клиентът работи с интерфейс. В случая с EmployeeDAO и TransationDAO. А в зависимост от това с какъв вид база данни работим, вградената зависимост връща съответната имплементация: InMemory, JPA, JDBC, NoSQL и така нататък. Това е идеален пример за пълна

независимост от бизнес слоя от реализацията на базата данни. Той работи с договор, не с имплементация.

Останалите два под-модула: взаимодействието с потребителя (*ui*) и *persistence* такъв са по-просто устроени. Единият съдържа различни видове имплемнтации на взаимодействиието с потребителя. Като в случая ние ще използваме само конзолната такава. Но може да се добави и графична като за пример. Докато при другия под модул – *persistence*- съдържа всички *Object Relational mapping* – класове, отговарящи по подходящ начин на данните от базата данни.

5.1.3. Deployment структура - изглед

Следващата структура, която ще изобразим е структурата на разположение или *deployment view*, показваща на каква машина може да се изпълнява програмата:



фиг. 5.5 Accounting System – изглед на разгръщане

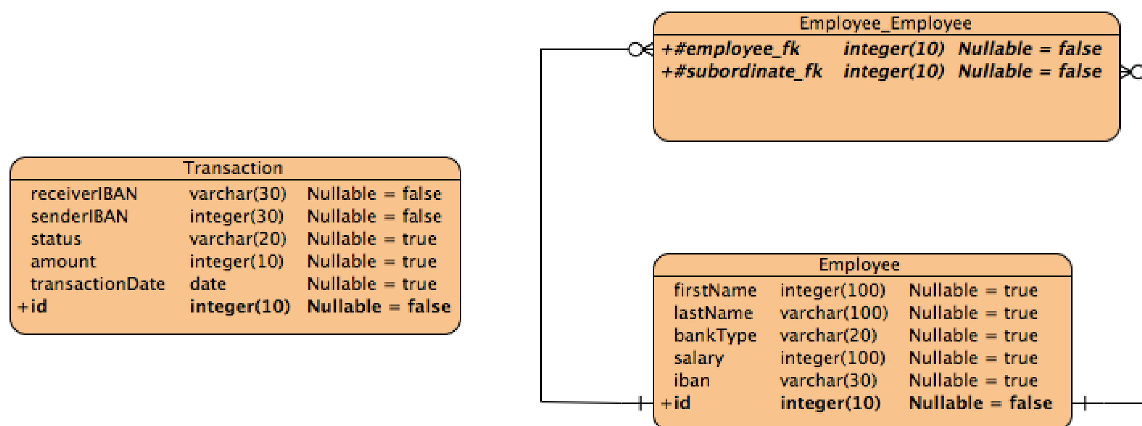
От фигура 5.5 се вижда че единственото, което е нужно за да се стартира програмата е машина да има инсталирам JRE – Java Runtime Envirement от 5-та версия нагоре. Пакетирането е с .jar разширение, като в него сме показали основните компонента, които съдържа.

5.2. Модел на данните (база данни)

Независимо с какъв тип база данни ще работим — диск базирана или *in memory*, ще работим с всеизвестните POJO класове интерпретиращи данните в *java* класове с *get/set* функции за всяко поле. В приложението ще има два основни такива класа, които примерно

в релационна база данни бихме били две таблици с полета асоциирани със съответните полета в класовете.

Фигура 5.6 изобразява диаграма на базата ни данни в релационен стил:



фиг. 5.6 Accounting System – Database view

Както можем да видим, имаме два таблици, едната съдържаща транзакциите, които се извършват в системата и другата данните за служителите й.

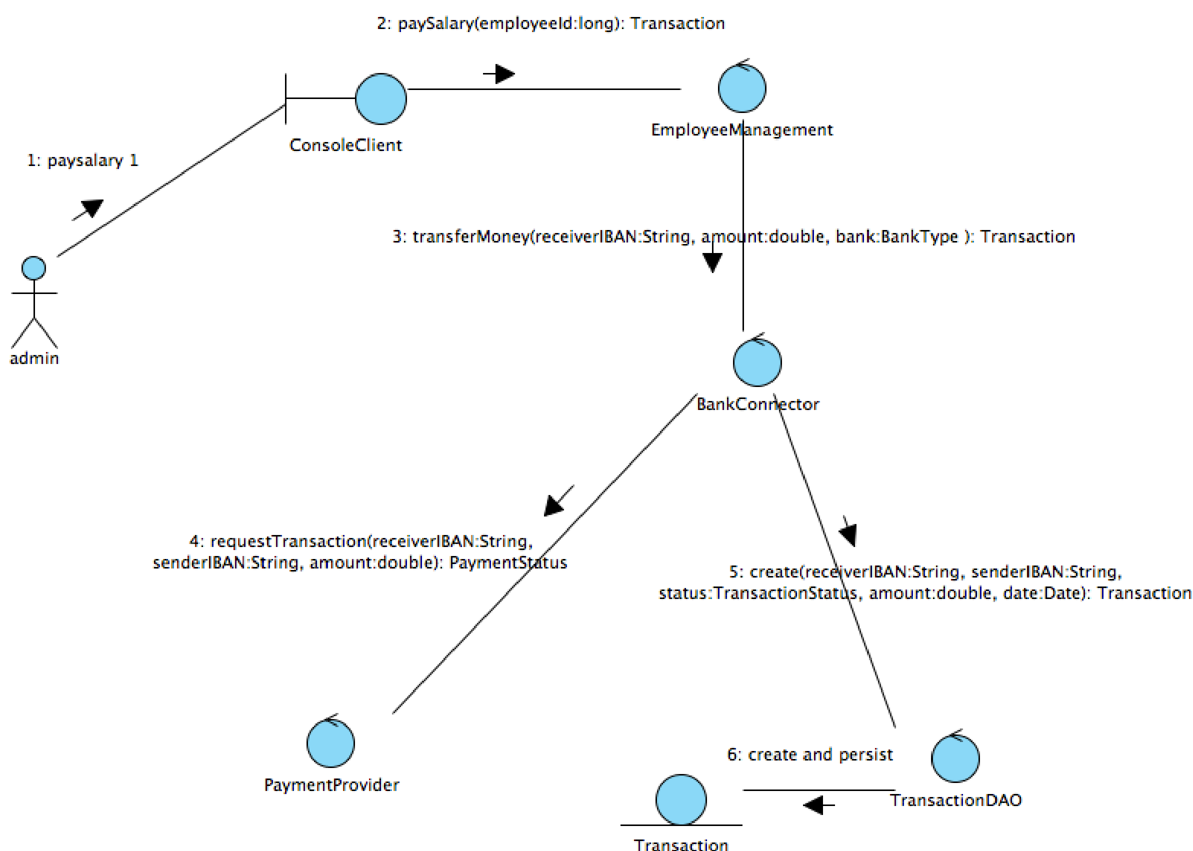
Като цяло в имплементацията ще се работи с POJO интерпретациите им с *in memory* база данни, запазвайки данните *в java heap*.

5.3. Диаграми на поведението

Биха могли да се направят доста диаграми и анализи по отношение на структурата и поведението на системата. От структурна гледна точка в по-горната такава, предоставихме модулната декомпозиця чрез пакетите и главните участващи класове и интерфейси в имплементационния етап.

Като допълнение и разяснение може да се добави една комуникационна диаграма с извадка от кода на интерфейсите, които си взаимодействат по между, с цел да изпълнят подадената от конзолата заявка. Няма да се набляга на конкретната имплементация, т.к. самото приложение е договор ориентирано.

Като примерна операция ще използваме една от най-често задавани операции в системата: плащане на заплата на конкретен служител. Фигура 5.7 илюстрира компонентите, които си взаимодействат при изпълнението й:



фиг. 5.7 Accounting System – Communication View

Първата стъпка, за да се осъществи това плащане, е разбира се писането в командния интерпретатор желаната операция. Понеже сме конзолно базирано приложение, всякаква взаимодействие потребителя ще става през конзолата. Командата за изплащане на заплата е:

```
> paysalary 1
> |
```

където 1 е идентификатор на служител. Всички команди си имат строго дефинирани синтаксиси и в случай, че не се знаят, то тогава *help* командата ще ги изпише и опише, заедно с аргументацията.

Командите се прихващат от ConsoleClient класа и се интерпретират от интерпретатора. *Factory* модула създава конкретните инстанции на модулите и под модулите, участващи в изпълнението на заявката. Ще се спрем на главните стъпките, отнасящи се към осъществяването на плащането.

ConsoleClient извиква метода paySalary на EmployeeManagement реализацията, чрез имплемнтираната команда (в случая чрез инстанция на PaySalary командата – конкретна имплементация на Command контракта).

EmployeeManagement е модулът, главната точка на изпълнение на поведението на системата. Представлява интерфейс, който дефинира всичките сървиси, с които може да работи системата. Ето и извадка от кода на този модул, като показваме само сървиса, отнасящ се за нашата примерна заявка:

```
/**
 * A contract that should describe all available services for employee
related operations.
 *
 * It's a main behavioral point delivered to the clients in order to
execute commands
 * in the context of the application.
 *
 * The business layer of the application is realized vis usage of a simple
dependency  injection pattern, which in some way intercepts the requests
and delegates them to the appropriate implementations of the related
dependencies
 *
 * @see Factory
 *
 * @author simo
 *
 */
public interface EmployeeManagement {

      // payment transactions related operations

      /**
       *  Pay monthly salary for employee with passed ID
<code>employeeId</code>
       *
       * PRE-CONDITIONS:
       *   the Employee with identified id should have set up salary and
IBAN   *    number
       *
       * @param employeeId
       *          the id of the Employee monthly salary is going to be
passed
       *
=      * @return
       *          an instance of Transaction with information about the
paying
       */
      Transaction paySalary(long employeeId);
```

Следващата стъпка е да се подадат необходимите аргумент, нужни на BankConnector модула, за да може да осъществи транзакцията. Това са банковия номер на служителя, типа уеб сървис която принадлежащата му банка използва за осъществяване на преводи и самата

сума. Тези данни се теглят от базата данни за конкретния служител, посочен с идентификационния му номер.

Следващото парче сорс код показва именно дефинирания в контракта метод, както и самия модул:

```
/**
 * A contract identifies all available services for bank transaction
related operations.
 *
 * Takes care about the business logic to connect to a concrete bank, to
execute a payment
 * / transaction service with specific web service implementation : SOAP,
NVP, REST and etc.
 * depends on the service delivered by the concrete bank
 *
 * @author simo
 *
 */
public interface BankConnector {

      /**
       * Transfer <code>money</code> from the company's bank account to
the
       * IBAN identified with <code>receiverIBAN</code> using bank
connection indicated with
       * <code>bank</code> argument
       *
       * @param receiverIBAN
       *          IBAN, where the amount of money will be transfered
       * @param amount
       *          the amount of going to be transfered
       * @param bankType
       *          the BankType indicates the bank going to be used
       *
       * @return
       *          an instance of Transaction indicating the result of the
money transaction
       */
      Transaction transferMoney(String receiverIBAN, double amount,
BankType bank);
}
```

След като сме в BankConnector модула и знаейки конкретната банка по типа, той знае коя уеб сървис реализация да извика. Към момента това са: HTTPS POST method с двойка параметри – ключ: стойност и SOAP базирания такъв. Единствените ни данни, за да се извика логиката за превод на пари е разбира се банковия акаунт номер на този, който дава парите. Това в случая е банковия номер на компанията. Това може да се вземе от конфигуратора в системата.

Ето го и парчето сорс код към следващия модул в редицата извиквания:

```
/**
 * A contract for all implementations of a payment provider going to be
integrated in the system Implementations means in a web service type they
are connected and collaborate with the external world
 *
 * So far, they are :
 * <ul>
 *          <li>SOAP base</li>
 *          <li>HTTPS Name Value Pair (NVP) POST </li>
 * </ul>
 *
 * The are some other strategies for integration, but at least they are
approved as one of the most secure ways.
 *
 * For some of the payment provider integration an auto generated stub
could be required.
 * Like in the case for SOAP base, where the WSDL delivered by the payment
provider is used in order the java source code to be generated
 *
 *
 * @author simo
 *
 */
public interface PaymentProvider {

      /**
       * Perform a transaction for amount of money : <code>amount</amount>
from bank account
       * <code>senderIBAN</code> to bank account <code>receiverIBAN</code>
       *
       * @param receiverIBAN
       *          the IBAN number of the bank account where the money will
be transferred
       *
       * @param senderIBAN
       *          the IBAN of the bank account from where the money will
be taken
       *
       * @param amount
       *          the amount of money which are going to be used in the
transaction operation
       *
       * @return
       *          a PaymentStatus indicates the end transaction status of
the payment
       *
       */ PaymentStatus requestTransaction(String receiverIBAN, String
senderIBAN, double amount);
```

Ако е нужно, този модул работи с автоматично генериран сорс код. Пример е при SOAP имплементацията, при която обикновено всеки сървър предоставя WSDL – *Web Service Definition Language* – файл, заедно с принадлежащите към него .XSD файлове, от които се генерират асоциираните на контракта java класове и интерфейси. В случая на NVP имплементацията, ще се построи HTTP POST заявка с добавените в нея параметри и съответни стойности.

Следва сорс кода към интерфейса, който след като се приключи заявката и се върне резултата на BankConnector модула, който записва резултата на транзакцията в Transaction entity.

```
/**
 * A contract for a Data Access Object for CRUD operations related to a
Transaction entity
 *
 * Implementors of this contract should be separated in meaning of the
data base context the application
 * works: desk base, like NoSQL server base or relational like MySQL
implementation, or in-memory one
 *
 *
 * @author simo
 *
 */
public interface TransactionDAO {

      /**
       * Create and persist a new Transaction into the data base used with
used data
       * state passed via method arguments
       *
       * @param receiverIBAN
       *         IBAN number of the bank account of the receiver - payee
       *
       * @param senderIBAN
       *         IBAN number of the bank account of the sender - payer
       *
       * @param status
       *         a TransacionStatus indicates how the transaction passed
       *
       * @param amount
       *         amount of this transaction
       *
       * @param date
       *         the Date when the transaction has been executed
       *
       * @return
       *         new created and persisted instance of Transaction
       */
```

```
      Transaction create(
                String        receiverIBAN,        String        senderIBAN,
TransactionStatus status, double amount, Date date);
}
```

Това е задача на TransactionDAO, чиято задача е да конструира правилно с подадените данни новия запис за транзакция в базата данни.

5.4. Изключения и обработване на грешки

В следващата графа ще се обобщи стратегията за обработване на грешки или на изключенията, изхвърлящи се в програмата. Изключение (*exception*) е ситуация, която изисква специално внимание. Може да се случи в следствие поява на грешка, която предвещава бизнес процеса да спре преждевременно, или някои не специфични случаи, които могат да бъдат прихванати и обработени, за да може да се продължи процеса на работа.

Изискванията наложени в приложенията към изключенията са:

- консистентен подход трябва да бъде приложен за всички компоненти;
- ситуацията, в която изключението се обработва трябва да е проста и гъвкава;
- всички видове типове изключение трябва да могат да се идентифицират лесно;
- всяко изключение трябва да предоставя информация като : къде, какво и защо – за да може да се дебъгва и лесно.

Специално в *java* програмния език има много добра наложена йерархия от класове за обработване на всякакъв вид грешки. Всички класове отговарящи на даден типаж изключения наследяват *Trowllable* класа. От там имаме още два наследника *Error* и *Exception*. *Error* не се причиняват от програмата. Те са по скоро за работа от JVM. По отношение на *Exception*: показва условие или ситуация, което приложението иска да „хване“. Представляват анормално състояние, с което трябва да бъде правилно прихванато и обработено, с цел избягване прекратяване от програмата. Подкласовете на Exception са *ClassNotFoundException*, *RuntimeException* както и други такива. Специално за *RuntimeExceptions*, ако те не се проверяват и ако не се прихванат ще излезне от действие самата програма.

В нашата програма ще работим с два главни подкласа на *RuntimeException* класа, изключение което както казахме, се разбира по подразбиране, че се изхвърля от коя да е функция и не се налага винаги да се дефинира в нея. За това и е удобна за използване и прихващане й на под класовете в йерархията.

Категоризацията е както следва:

BusinessException – или бизнес изключенията. Те се свързват с това как системата се използва и как състоянието на бизнес обектите се променя и менажера. Те се появяват в изключителни ситуации, за които разработчиците или поддържащите софтуера хора, трябва да внимават и отделят.

Примерни ситуации, които са в тази категория са:

- Обект или таблица от базата данни със специфично ИД не е намерено
- Невалидна заявка от потребителя по отношения на обект със моментно състояние
- Потребител няма права или достъп до данните, които иска

SystemException – или системни изключения. Отнасят се до това как системата и техническата селекция работят.

Примерни ситуации за този вид са:

- Определена заявка не е достъпна;
- Грешка при достъпа на базата данни – за четене или запис;
- Грешка при връзка с базата данни или изгубването й.

Заедно с видовете изключения е дефинират и видове грешки, идентифицирани с кодове. С тях ясно се дефинира със низово описание каква по точно е грешката. Примерно INCORRECT_BANK_NUMBER с ИД 1 и описание „IBANK NOT VALID“.

5.5. Реализация на инжектирането на зависимост (dependency injection)

Приложението в основата си не изпозлва технологии или готови приложни рамки в имплементацията си. Това се отнася и за инжектирането на зависимостите между модулите. Реализацията й също става ръчно, без изпозлване на готов софтуер за това.

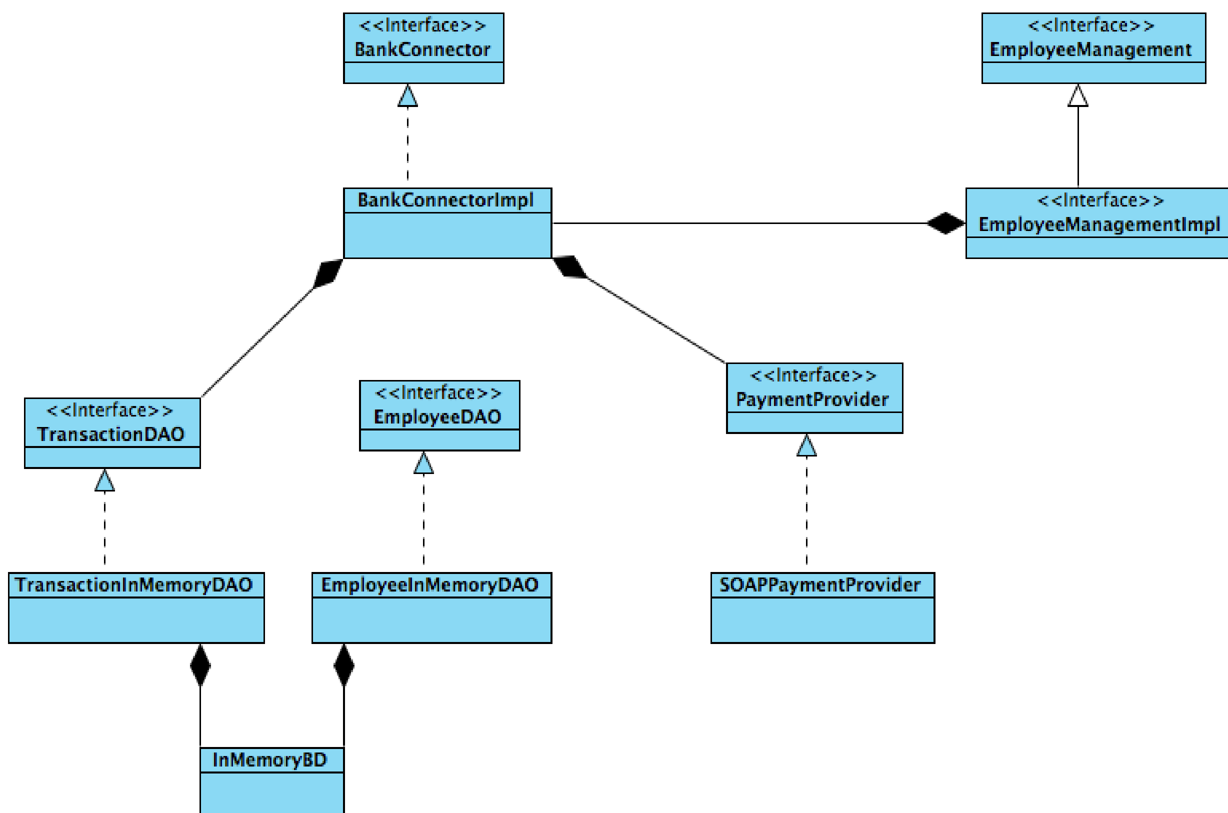
В имплементацията си стоят следните овновни стъпки, които се следват при писането на софтуера в програмта:

- За всяка отделна, самозначеща функционална единица, от която би могла да се отдели и компонент, се пише договор чрез създаването на интерфейс. Това като примери са:
  - адаптъра към базата данни (тя може да е in-memory, дискова или файлова дори)
  - модулът с който се свързваме към доставчика за плащането (Payment Provider)

- всички DAO (Data Access Object) обекти
- основният клас имплементиращ сървисите за менажиране на системата (EmployeeMenagement)

- Системата комуникира с интерфейсите (договорите). Тя не се интересува от инициализирането на конкретните имплементации. Всъснoст, в това е и основния принцип на инжектирането на зависимости
- Ръчното вграждане на имплементации става чрез инжектирането на опрелените зависимости за всяка инстанция още при инициализирането й
- Има един основен клас, който се грижи за това вграждане: Factory, който представлява абстрактна фабрика като шаблон за дизайн. За всяка колекция от изискващи класове се създава метод, който се грижи за инициализирането им. Например, ако искаме да работим с In Memoery база данни и с SOAP сървиси за работа с Payment Provider, ние го указваме на факторито и то се грижи за създаването на цялата фамилия от зависимостие. Само този клас знае за тези зависимости.

Примерната колекция от класове и зависимости представям с фиг. 5.8 по долу



фиг. 5.8 Пример на зависимости от класове

На нея ясно се отличава кой клас без кой не може да съществува. Примерно който и да е Data Access Object клас не може да не знае с коя имплементация на база данни работи или BankcConnector с коя имплементация на сървиси. За тези заивисмости знае само и единствено Factory класа, който се грижи за създаването на тази комозиция от класови инстанции.

Глава 6. Реализация, тестване, експерименти

6.1. Реализация на модулите

Модулите в приложението са реализирани по такъв начин, че предоставят по уникален начин тестването им както на системно, покриващо цялата функционалност из между всички модули, така и модулно тестване, абстрахиращо се от модулите, включени в него. Дават се както интерфейси, договори, така и имплементации към тях. Всеки модул реализира в себе си функционалност, принадлежаща единствено към контекста, за който е дефиниран.

Това си има и своята цел, а именно гоненето на два високо ценени и качествени показателя:

- **High Cohesion (Висока кохезия):** „представлява измерител за степента, в която отделните редове програмен код в даден модул се сработват за да предоставят специфична функционалност. Кохезията е измерител и обикновено се характеризира като „висока“ и „ниска“. Тенденцията е да се предпочитат модули с висока кохезия, защото тя е свързана с желани черти на софтуера като устойчивост, сигурност, надежност, лесна разбираемост“. Източник: http://goo.gl/CUZXFo
- **Low coupling (Ниска свързаност):** „под свързаност или зависимост (на английски: coupling или dependency) се разбира степента, в която един програмен модул разчита на друг програмен модул. Свързаността може да бъде „висока“ или „ниска“. Ниска свързаност имаме когато един модул не е нужно да бъде заинтересован от вътрешната имплементация на друг модул, и взаимодейства с него чрез стабилен интерфейс. При този вид свързаност промяна в имплементацията на един модул не изисква модификация в имплементацията на друг модул. Ниската свързаност е знак на добре структурирана компютърна система“.

  Източник: http://goo.gl/pK13Bd

Освен запазването на минималното взаимна зависимост между модулите (*low coupling*), към всеки такъв се запазва и високата кохезия.

За пример може да се даде EmployeeDAO, като част от семейство *data access layers* за връзка базата данни и операциите за създаване, четене, изтриване и обновяване на данни. В този модул, задачите са му единствено да се свърже към базата данни, която му е предоставена и да върши четирите, посочени по горе, операции с нея. И нещо повече - тези

операции са предназначени единствено и само за служителите (данните обработващи се с Employee класа).

6.3. Планиране на тестването - тестови сценарии, процедури

Системата се тества в два главни аспекта – модулно и интеграционно, включващо всичките нива на сорс кода. Ще следваме следната рамка и за двата вида тествания, с цел ясно представяне и на тази част от приложението:

- Име – името на теста ще подсказва каква част от функционалността се подлага на проверка и какъв случай ще се разиграе. От името на createEmployeeSuccessfulTest метода разбираме, че на проверка се подлага създаването на нов служител; че самият метод е тестови (с постфикса Test);
- Предусловия: това са действия, които винаги трябва да бъдат изпълнени преди да се извика тестовия метод. Тук говорим и за създаването на необходимите данни за всеки тест;
- Действия: т.е. самият набор от действия, с които се и минава през самата тестова модулност;
- Проверка на очакван резултат: проверяват се резултатът, който би трябва да бъде върнат от самия метод. Това най-често става със сверяването му с създадените преди това обекти;
- *Fail* описания. При момент, в който теста не минава, задължително се дава и детайно описание. Целта е поддържащия разработчик, лесно и бързо да може да модифицира и промени имплментацията на модула.

6.4. Модулно и системно тестване

В приложението ще се приложат два основни метода за тестване на функционалността и доказване за коректността й : модулно и системно/интеграционно.

Модулното тестване (*unit test*) е метод, с който се тества парче код, индивидуално. Тестват се както функционалността, която се изпълнява от него, така и данните с които работи. За по голяма централизация върху тестваният компонент, ще използваме и т.н. макетиран метод. Това е инструмент, с който могат да се предефинират извикването към избрани функции на класове, с цел избягването на самото й извикване. С тези два подхода, ще можем да проверим и докажем коректност на работата на съществените компоненти в програмата в два аспекта – в интеграционно и чисто модулно ниво.

За нас тези компоненти са:

- GenericDAO – като основен на всички Data Access класове в системата

- PaymentProvider – като основен за превод за паричните операции с реалните плащания
- EmployeeManagement – в качеството си на първичния модул, с който работят клиентите на приложението

Отделно от компонентното тестване, ще се имплементира и системното/интеграционно такова. Това е подход, в който модулите се групират и тестват как те работят заедно. С тях ще верифицираме коректността на имплементация на функционалните изисквания с цялостната им представа.

По този начин, включвайки доказуемостта за работа на системата с два метода ще можем лесно, при каквато и да е промяна на сорса, да проверим дали приложението остава да работи коректно, така както е било преди неговото модифициране. Ще се режисират и имплементират същински (с голямо значение за проверка на работата на приложението) тестови сценарии.

Глава 7. Заключение

7.1. Обобщение на изпълнението на началните цели

Дипломната работа разгледа и задълбочи в изследването на взаимоовръките между шаблоните за дизайн. Как те могат да се прилагат в софтуерната практика и как при правилното има комбиниране се постигат архитектурните двигатели на системата, с което се печели качество и зрялост на продукта.

Обърна се специално внимание на начини на създаване и прилагане на архитектурни шаблони като композиция от софтуерни такива.

Разгледаха се основните видове взаимоовръзки – модели на съединение, модели на последователността, допълващи се шаблони и шаблонни езици. На последните отделихме специално внимание, т.к. с тях по чисто образен и открит начин се виждат в кои случаи какви качествени атрибути се постигат.

Също така, се отдели време и се даде пример за създаването на един изцяло нов шаблон като комбинация от други такива. С това се представи силата им в създаване на нови темплейт решения.

7.2. Насоки за бъдещо развитие и усъвършенстване

Работата би могла да се разшири и задълбочи в изследването на архитектурните шаблони. Да се изследва до какви качествени атрибути се стига при композирането на дизайн и архитектурните шаблони.

Част от материалите използвани тук ще бъдат приложени в разработката на книга към курса «Обектно ориентиран анализ и дизайн с използването на шаблони», воден в магистърска програма «Софтуерни технологии» към катедра «Софтуерни технологии», Софийски университет «Св. св. Климент Охридски».

Източници

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissibes. “*Design Patterns: Elements of Reusable Object-Oriented Software*”, USA: Addison Wesley, 1994
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. “*Pattern-Oriented Software Architecture Valume 1: A System of Patterns*”, Wiley, 1996
3. Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. “*Pattern-Oriented Software Architecture Valume 2: Patterns for Concurrent and Networked Objects*”, Wiley, 2000
4. Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, “*Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, Volume 5*”, Wiley, 2007
5. Birukou, Aliaksandr, “*A Survey of Existing Approches for Pattern Search and Selection*”. DISI - University of Trento, Italy, 2010
6. Ambler Scott, *Introduction to Test Drivern development* , 2003, Saturday 11 January 2014
7. Coupling, *Coupling (Computer programming),* Web, 4 Dec. 2014
8. Cohesion, *Cohesion (Computer programming),* Web, 4 Dec. 2014