Софийски университет
“Св. Климент Охридски”

Факултет по математика и информатика
катедра “Софтуерни технологии”

Влияние на практиката „Разработване, базирано на тестове” (Test-driven development), върху качеството на дизайна на софтуерна система

Дипломна работа на Теодор Клисурски,
спец. “Информатика”,
маг. програма “Софтуерни технологии”,
фак. № М-21547

Научен ръководител: доц. Силвия Илиева, СТ – ФМИ
Консултант: докт. Ива Кръстева, СТ – ФМИ

гр. София,
октомври 2007 г.

СЪДЪРЖАНИЕ

1 Въведение

1.1 Увод

Разработката на качествен софтуер е немислима без добре дефиниран и измерваем процес. Една от най-важните фази, от процеса за разработка на софтуер, е дизайнът. Качеството на дизайна е определящо за качеството на крайния продукт. Поради това се полагат много усилия за подобряване на качеството на дизайна, чрез изследване и модификация на процеса за разработка.

Една от съвременните гъвкави методологии за разработка на софтуер е „Екстремното програмиране" (Extreme Programming). То включва практиките „Програмиране по двойки" (Pair Programming), „Честа интеграция" (Continuous Integration), „Подобряване на дизайна" (Design improvement), „Малки доставки" (Small releases), „Правила за кодиране" (Coding standard), „Планиране" (Planning game) и „Разработване, базирано на тестове (Test-driven development)". Характерно за практиката „Разработване, базирано на тестове" е, че модулните тестове се написват преди да са готови дизайна от ниско ниво и кода. Това дава на софтуерния инженер още една възможност за преглед и анализ на проблемната област, от една друга гледна точка – как софтуерната система би работила при различни обстоятелства. Това позволява, още на фаза дизайн, да бъдат забелязани детайли, свързани с работата на готовата система, което от своя страна води до по-добър дизайн.

1.2 Цел и задачи

Тази дипломна работа има за цел да внедри практиката „Разработване, базирано на тестове", в съществуващ фирмен софтуерен процес и да изследва влиянието й върху фазата на дизайна на софтуерна система. Основните задачи, произтичащи от целта са:

- Сравнителен анализ на съвременните процеси за разработка на софтуерни системи и ролята на фазата дизайн в тях;

- Представяне на практиката „Разработване, базирано на тестове” – същност, предимства и недостатъци;

- Идентифициране и представяне на подходящи метрики и критерии за оценка на резултатите от прилагането на практиката;

- Прилагане на практиката „Разработване, базирано на тестове” в съществуващ фирмен софтуерен процес, при изграждане на реално приложение;

- Анализ на събраните данни, съгласно избраните метрики и препоръки, на базата на получените резултати.

1.3 Полза

Дипломната работа изследва влиянието на практиката „Разработване, базирано на тестове” върху дизайна на софтуерна система. За целта практиката е внедрена в софтуерния процес и резултатния работен продукт е сравнен с работния продукт, получен чрез изходния софтуерен процес. На база на събраните данни, се правят изводи за приложимостта на предложения подход и се дават препоръки, които да улеснят прилагането му и да доведат до подобряване на дизайна на софтуерния продукт.

Изследването може да бъде от полза на софтуерните специалисти, които се интересуват от теоретичните и практически аспекти на разглежданата практика. Теоретичната част е подходяща за специалисти с интереси в областта на съвременните процеси и методологии за разработка на софтуер. Описанието на експеримента би подпомогнало разработчиците на софтуер, като им покаже реален пример и им даде препоръки за ефективно прилагане на практиката. Резултатите от

изследването биха били от полза на специалистите, занимаващи се с дефиниране и подобряване на софтуерния процес в организациите, разработващи софтуер.

1.4 Структура

Първата глава въвежда читателя в проблемната област, резюмира предимствата на разглежданата практика и набелязва целите на настоящата дипломна работа.

Във втората глава е направен обзор на проблемната област, чрез сравнителен анализ на съвременните процеси за разработка на софтуерни системи и ролята на фазата дизайн в тях. В разглежданията са включени както традиционни, така и съвременни процеси. Представено е Екстремното програмиране и са обобщени неговите предимства и недостатъци.

Третата глава представя в детайли практиката „Разработване, базирано на тестове". Посочени са нейните предимства и недостатъци. Дефинирани са основни понятия и концепции, свързани с тестването.

Четвъртата глава въвежда читателя в проведения експеримент. В началото й е представено подобно изследване и резултатите от него. Следва описание на проведения експеримент, с внедряването на практиката в съществуващ фирмен софтуерен процес, при изграждане на реално приложение.

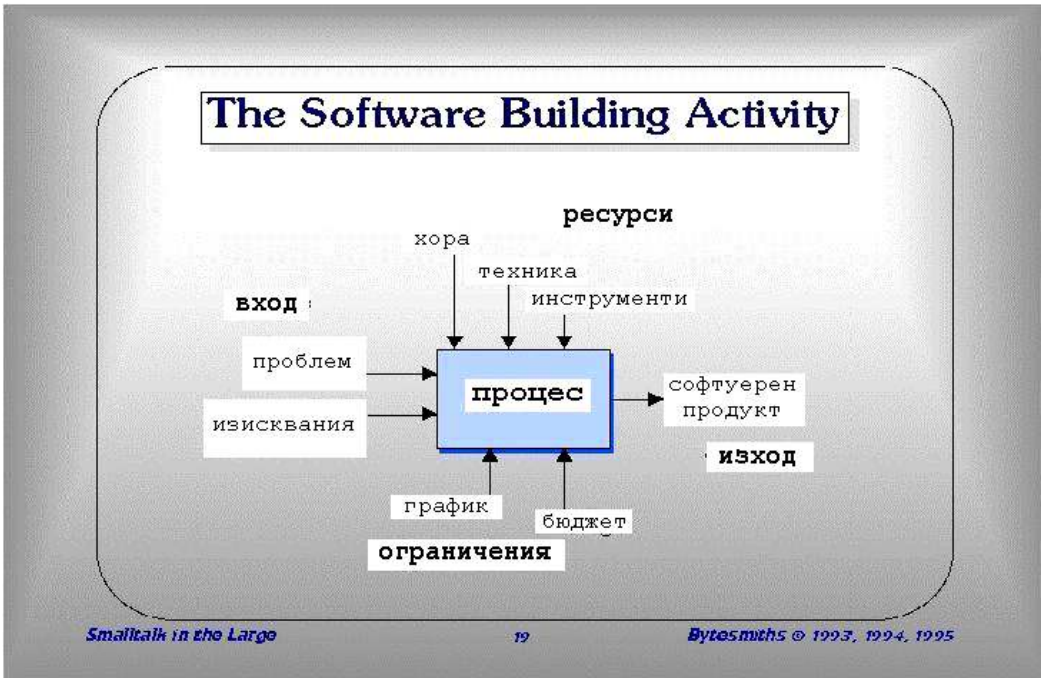
В петата глава са представени и изследвани резултатите от експеримента. Дефинирани са подходящи метрики и критерии за оценка, с помощта на които резултатите са анализирани и са дадени препоръки за подобряване на процеса за разработка.

Заключението прави обобщение и дава препоръки за бъдеща работа, с цел подобряване на процеса за разработка на софтуер.

В края е представен списък с използваната литература и термини.

2 Съвременни процеси за разработка на софтуерни системи – сравнителен анализ

Както споменахме в уводната част, процесът е неразделна част от разработката на съвременния софтуер. Той определя стъпките и насоките за организация на работния процес и персонал, с цел създаване на качествен краен продукт и удовлетворяване на изискванията на клиента. Една от най-общите дефиниции за софтуерен процес дава Съмървил [3], според когото, софтуерен процес е множеството от дейности, които водят до произвеждането на софтуерен продукт. По-критичен е Пресман [2], според когото, софтуерният процес е набор от задачи и дейности, които трябва да се извършат, за да се създаде висококачествен софтуер. Фигура 1 показва основната цел на софтуерния процес – да организира производството на софтуерен продукт по зададените изисквания, с наличните ресурси и при съответните ограничения.



Фигура 1: Процес на изграждане на софтуера

Поради голямото разнообразие на проблемните области, за които се разработва софтуер, съществува и голямо разнообразие на процеси за разработка. Въпреки това съществуват дейности, които са характерни за всички процеси. Това са т. нар. “общи дейности” (generic activities), които дават основни насоки за развитието на продукта. За конкретен процес, всяка от тях се допълва с дейности, които са специфични за процеса и проблемната област. Те се наричат “допълнителни дейности” (umbrella activities). Съвкупността от общи и допълнителни дейности се нарича рамка на софтуерния процес. Освен дейностите, които трябва да се извършат, процесът определя как и на кой етап от развитието на проекта да се извършат, както и от кого да се извършат. Последното се дефинира чрез “роли” по проекта (анализатор, програмист, тестер и др.).

Според Пресман [2], общите дейности са:

- Комуникация – включва комуникация и взаимодействие с всички участници в проекта;

- Планиране – включва създаване на план за работа, на базата на техническата работа, която трябва да се извърши, потенциалните рискове, необходимите ресурси, работните продукти, които трябва да се произведат и времевия график;

- Моделиране – включва създаването на модели, които да осигурят на клиента и разработчика разбиране на софтуерните изисквания. На базата на тези изисквания се изгражда дизайн на софтуера, който дефинира структурата на софтуерната система, интерфейсите между отделните софтуерни модули, структурата на отделните модули, специфичните алгоритми и данните, които системата ще ползва;

- Конструиране – включва създаване на програмния код и тестването му;

- Внедряване – включва доставяне на софтуера на клиента и въвеждането му в експлоатация.

Тези общи дейности съответстват на етапите на развитие на проекта (инициализация, планиране, планиране на фазата, изпълнение и контрол, завършване на фазата и проекта).

Допълнителните дейности служат за контрол и следене на проекта. Типични допълнителни дейности са:

- Следене и управление на софтуерния проект – включва оценка на прогреса съгласно плана на проекта и предприемане на подходящи действия, за да се спази графика;

- Управление на риска – включва оценка на рисковете, които могат да засегнат планираните резултати от проекта;

- Осигуряване на качеството – включва дефиниране и провеждане на дейности за осигуряване на качеството на продукта;

- Технически прегледи – включва оценка на междинните и крайните работни продукти, с цел навременно откриване и отстраняване на дефекти;

- Измерване – включва дефиниране и събиране на метрики за процеса, проекта и продукта, с цел оценка и предприемане на адекватни мерки;

- Управление на софтуерната конфигурация – управлява последиците от промени по време на жизнения цикъл на проекта;

- Управление на повторното използване – дефинира критерии за повторното използване на работни продукти и установява механизми за създаването им;

- Подготовка и генериране на работни продукти – включва дейности, необходими за създаване на работни продукти.

Многообразието от процеси се получава, понеже допълнителните дейности са застъпени в различна степен в различните процеси. Съществуват два типа процеси:

- Традиционни – базови процеси, които дават предписания за протичане на работния процес и създаване на продукта. Основни цели са осигуряване на качеството, управляемост и предсказуемост на проекта;

- Съвременни – процеси, които дават предписания за протичане на работния процес и създаване на продукта, но отчитат спецификата на конкретния проект и дават възможност за по-голяма адаптация на процеса към проекта. Използват се при изграждането на съвременни софтуерни системи. Основни цели са гъвкавост, простота, удовлетворяване на клиента и приспособяемост на процеса към проекта.

В изложението, ще представим процесите, като се придържаме към следната структура:

- Процесен модел – показва основните дейности в процеса, последователността им на изпълнение и връзките между тях; Може да съдържа други съществени елементи, свързани с дейностите и важни за процеса – работен продукт, ресурс, период от време и др. ;

- Характеристики – показват отличителните черти на процеса;

- Предимства и недостатъци.

Представянето на всяка група процеси, ще бъде последвано от сравнителна таблица, съдържаща относителни оценки по следните неколичествени показатели:

- Сложност на концепцията – показва степента на сложност на концепцията на процеса. Възможни оценки – ниска, средна и висока;

- Степен на итеративност – показва степента на итеративност на процеса. Възможни оценки – ниска, средна и висока;

- Трудност на прилагане – показва степента на приложимост на процеса, като взема предвид необходимите ресурси (време, бюджет, квалифициран персонал). Възможни оценки – ниска, средна и висока;

- Приложимост в рискови проекти – показва доколко съответния процес е подходящ при рискови проекти. Възможни оценки – неподходящ, приемлив и подходящ;

- Приложимост в динамични проекти – показва доколко съответния процес е подходящ в динамични проекти. Възможни оценки – неподходящ, приемлив, подходящ и много подходящ;

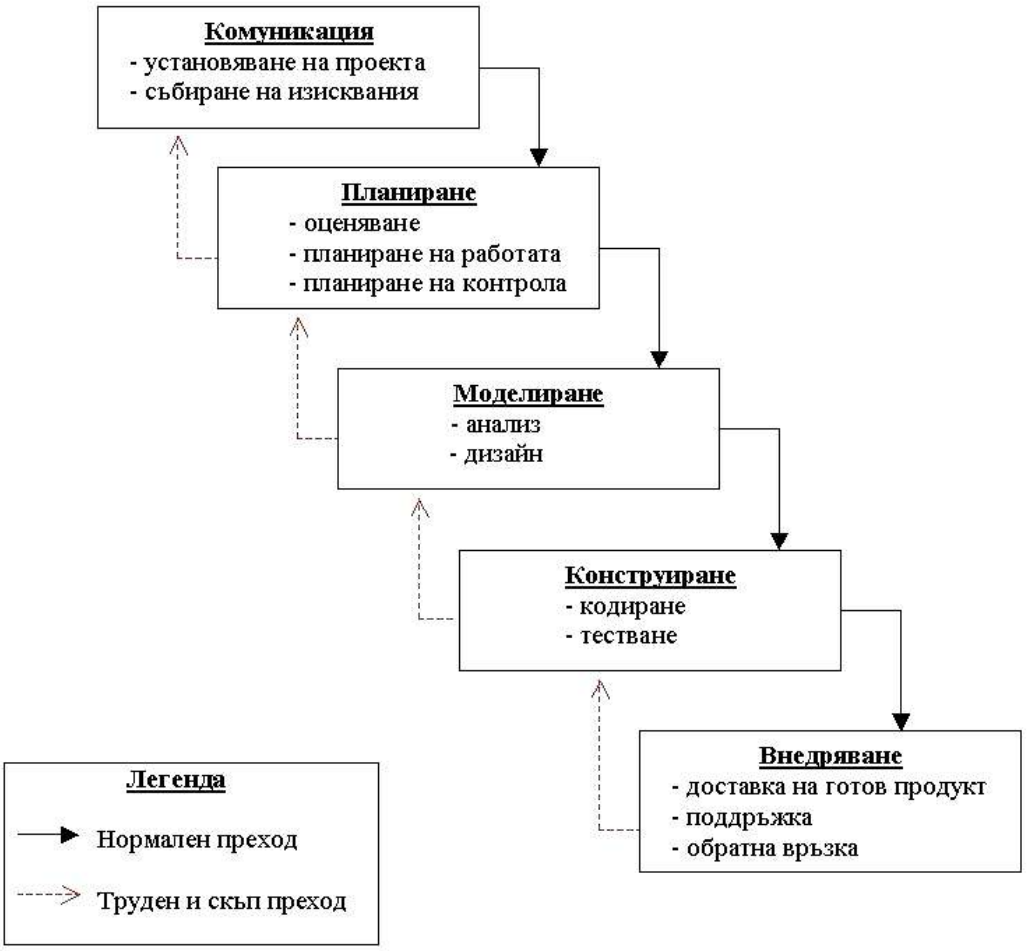
2.1 Традиционни процеси

Традиционните процеси дефинират множество от дейности, задачи, контролни точки (milestones) и работни продукти, които са необходими, за да се изгради висококачествен софтуер. Тези процеси се адаптират към нуждите на организацията, проблемната област и проекта, след което се следват.

2.1.1 Каскаден модел (Waterfall)

Каскадният модел предлага систематичен и последователен подход за разработка на софтуера [2]. Петте общи дейности (Комуникация, Планиране, Моделиране, Конструиране и Внедряване) протичат последователно и еднопосочно, като всяка започва, след като предишната е приключила. Всяка фаза завършва с работен продукт, които е база и вход за следващата фаза. Фазите и дейностите са ясно дефинирани и документирани. Ролите на хората по проекта са

определени и не се променят. Връщане към предишна фаза не е предвидено в модела, но ако при определена ситуация, обстоятелствата наложат корекция на работен продукт от предишна фаза, връщането е трудно, скъпо и застрашава успешния завършек на целия проект. Фигура 2 показва схемата на каскадния модел, както и дейностите, характерни за отделните етапи.



Фигура 2: Каскаден модел

Каскадният модел е прост по същество, но сравнително бавен, тромав и изисква повече ресурси, в сравнение с други методи. Освен това е по-бавен, поради изискването да има работен продукт в края на всяка фаза. Подходящ е за високорискови проекти с достатъчно налични ресурси, при които изискванията са добре дифинирани, ясни и не се променят във времето. Основните проблеми при

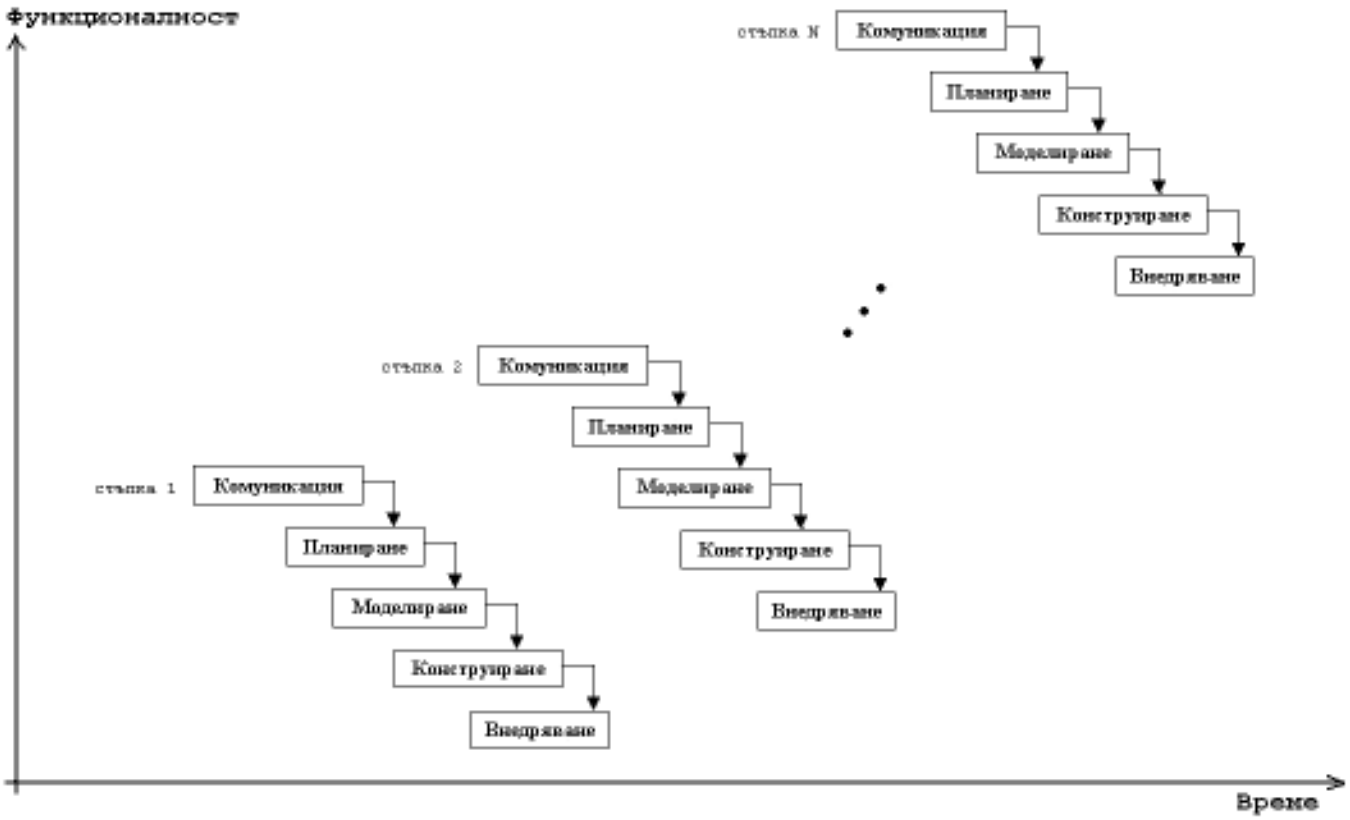
използване на този модел са следствие от факта, че много малко от реалните проекти могат да отговорят на изискванията на модела – добре дефинирани и статични изисквания, достатъчно налични ресурси за изпълнение на проекта и увереност в успешния край от страна на клиента. От последователността, която е заложена в модела, следва че всички грешки, допуснати на която и да е от фазите, се пренасят в работните продукти на следващите фази и следователно в крайния продукт. Моделът е неподходящ в случаите, когато проектът не може добре да се раздели на фази, съответни на фазите на модела. Друг недостатък е фактът, че работеща версия на продукта се получава в края, като резултат от последната фаза. Така липсва възможност за получаване на обратна връзка от клиента по време на разработката, което може да доведе до отклоняване на продукта от реалните нужди на клиента. Сред предимствата на каскадния модел е, че дейностите са добре дефинирани и обособени, което улеснява контрола на графика, бюджета и документацията.

Дизайнът е включен във фазата на моделиране и заема централно място в каскадния модел. Качеството му е от особено значение, тъй-като връщане към предишна фаза не е предвидено в модела и всички дефекти се отразяват на крайния продукт. Качеството на дизайна пряко влияе на разработката на продукта и удовлетвореността на клиента. То е от решаващо значение за крайния изход от проекта. При установяване на дефекти в дизайна по време на следващите фази (конструиране и внедряване), е много вероятно да няма достатъчно време, за да се вземат адекватни мерки и проектът да бъде провален.

2.1.2 Постъпков модел (Incremental)

Както споменахме, сред основните недостатъци на каскадния модел, са липсата на възможност за връщане към предишна дейност и разработката на целия продукт наведнъж, които често се оказват голям проблем в реалните проекти. Именно тези недостатъци преодолява постъпковият модел. При него изискванията се изясняват и групират, като за всяка група изисквания се извършва разработка по каскадния модел. Това позволява разработката да стартира с важните и ясни изисквания и да се даде време за доизясняване на останалите изисквания [3]. Така

разработката протича на стъпки, като на всяка стъпка последователно се извършват петте общи дейности. В резултат клиентът получава доставка в края на всяка стъпка, като готовата функционалност нараства. Възможно е, при наличие на достатъчно ресурси, различните стъпки да протичат паралелно и независимо една от друга. Фигура 3 показва схемата на постъпковия модел:



Фигура 3: Постъпков модел

Постъпковият модел се доближава до естеството на реалните проекти, като същевременно е прост и близък до каскадния модел. Дава възможност за максимално бърз старт на работния процес с наличните ясни изисквания. Тези изисквания са групирани така, че да определят част от системата и да позволят старт на съответната стъпка от процеса. В резултат, първа работеща версия на системата е готова бързо и това позволява на екипа по проекта да получи обратна връзка от клиента още в началния етап от развитие на системата. Последното е изключително важно за успешното развитие на проекта и крайната удовлетвореност на клиента. Обикновено, при първата стъпка се разработва

сърцевината на системата, а останалите функционалности се оставят за следващите стъпки. Постъпковият подход дава възможност за гъвкавост при промяна на условията и невъзможност за спазване на сроковете за изпълнение – едно решение е да се предоговори доставяната функционалност за съответната стъпка. Разработката на стъпки позволява да се реагира адекватно на промените, които се случват по време на жизнения цикъл на проекта.

Дизайнът е включен във фазата на моделиране на всяка стъпка. Качеството му влияе на резултата от съответната стъпка. Корекции на дефекти са възможни на следващата стъпка, заедно с нововъведенията. При постъпковия модел е много важно дизайнът да бъде разширяем и лесно да се преизползва (reusable), тъй – като на всяка итерация се добавя нова функционалност. Разширяемостта на дизайна е критичен фактор за проекта, защото от нея пряко следва разширяемостта на продукта. В модела е заложена възможност, след старта на проекта, да се доизясняват изисквания и да се добавят нови. Липсата на разширяемост може да доведе до невъзможност за провеждане на следваща стъпка, което да блокира проекта или да доведе до голямо забавяне поради нужда от преработка.

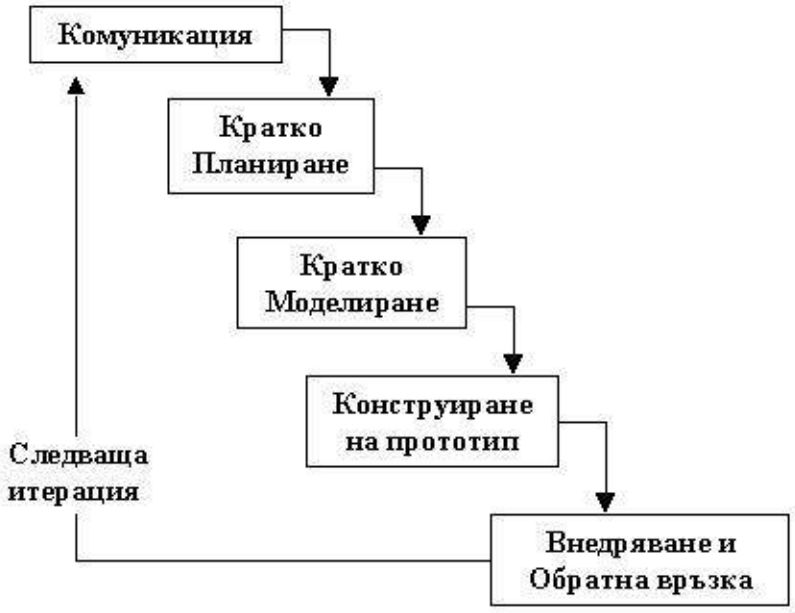
2.1.3 Прототипен модел (Prototyping)

Динамичността в реалните проекти налага да се търсят подходи, които се доближават до естеството на проектите и отговарят на техните нужди. Оказва се, че по време на разработката изискванията се конкретизират и променят според индивидуалните нужди на клиента. За успешната реализация на проекта е важно да има обратна връзка от клиента през цялото време на разработка. Прототипният модел предлага решение в тази посока. Той е итеративен и на всяка стъпка се разработва и доставя на клиента частично работеща версия на системата (прототип), след което екипа по проекта получава обратна връзка. Основна цел на подхода е да подпомогне събирането, разбирането и изясняването на софтуерните изисквания. Всички основни дейности са представени в модела, като на планирането и моделирането е отделено по-малко време, с цел бърза направа на прототипа. Съществуват два варианта на прототипния модел [2]:

- Резултатът от всяка итерация (прототип) не се ползва при следваща итерация. Той служи, само за да подпомогне събирането, разбирането и изясняването на софтуерните изисквания. На следващата итерация разработката стартира отначало, като се имат предвид опита от предишна итерация и обратната връзка от клиента.

- Резултатът от всяка итерация (прототип) се ползва при следваща итерация. Прототипът е база за следващата итерация, в която той се развива до нов прототип, като се имат предвид опита от предишна итерация и обратната връзка от клиента. В края на последната итерация прототипът еволюира до краен продукт. Такива прототипи се наричат еволюционни прототипи. Поради трудността му, този вариант на прототипния модел се прилага сравнително рядко.

Фигура 4 показва схемата на прототипния модел:



Фигура 4: Прототипен модел

Основни предимства на този модел са краткия цикъл на разработка и наличието на прототип в края на всяка итерация. Те допринасят за по-лесното събиране и разбиране на изискванията и навременно получаване на обратна връзка от клиента. Дават възможност за голяма адаптивност на проекта към реалните условия. Недостатъците са свързани със стремежа за бързо внедряване на прототипа. По–малко време е отделено за планиране и дизайн, което крие рискове, особено ако се разработват еволюционни прототипи. В случая на прототипи, които не се използват като база за следваща итерация, планирането и дизайна имат значение само за текущата итерация и следователно не са критични за проекта. Краткото планиране и дизайн са критични за проекта, в случаите на еволюционни прототипи. Спестеното време на тези фази може да доведе до проблеми при следващите итерации. Най–важно качество на дизайна в този случай е да бъде лесно разширяем, тъй – като е база за развитие на следващата итерация.

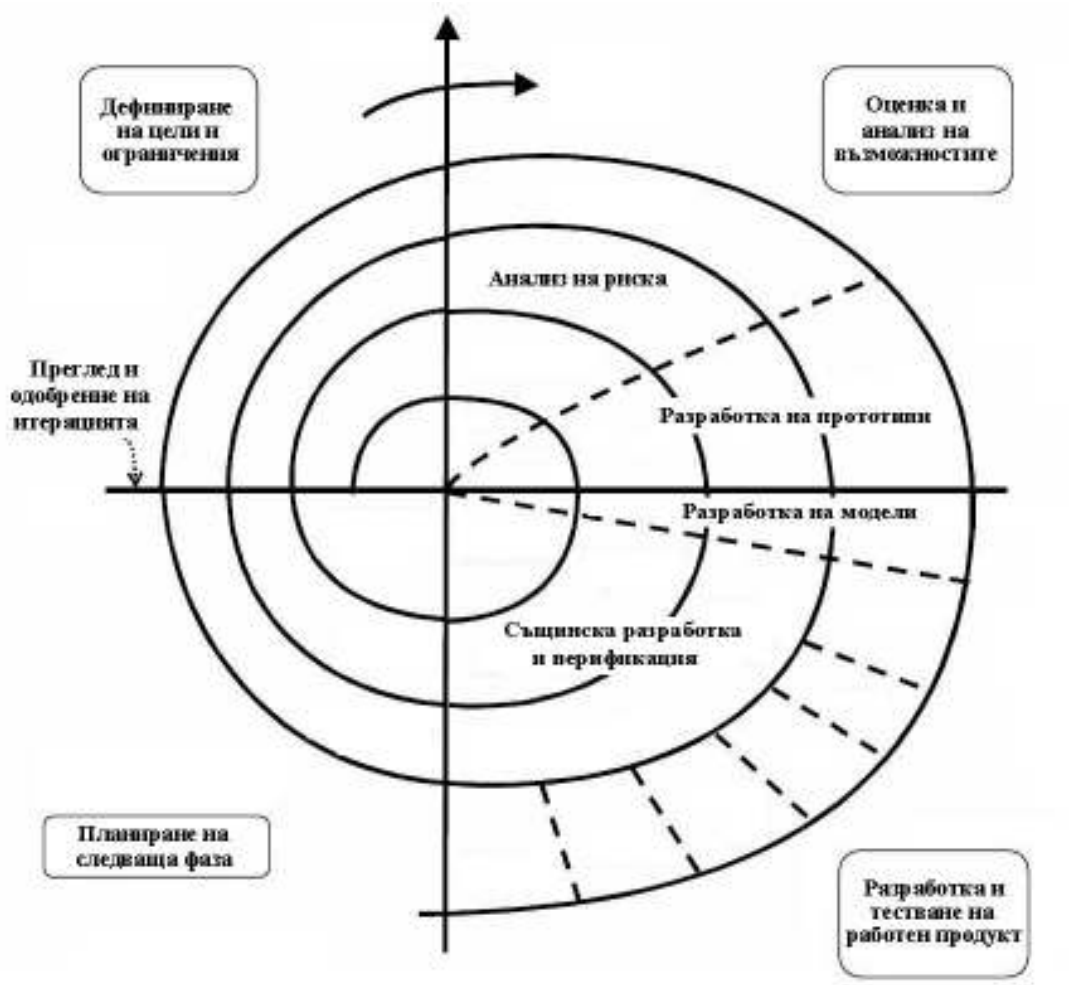
Важна особеност на модела е, че за да работи успешно, клиентът трябва да е информиран за него и да съдейства за правилното му протичане. За клиента трябва да бъде достатъчно ясно, че се доставят прототипи, а не почти готови решения. В противен случай, рискът да загубим подкрепа посредата на проекта е голям, тъй – като за клиента, системата изглежда почти готова още в началото и той няма да е склонен да изчака целия процес на разработка.

2.1.4 Спираловиден модел (Spiral)

Спираловидният модел е еволюционен модел на софтуерен процес, който обединява прототипния и каскадния модел [2]. Основните му характеристики са, че е итеративен и дефинира множество контролни точки. Това дава възможност за по-лесно следене на прогреса. Към всяка итерация е добавен анализ на риска, с цел минимизирането му. Една итерация в модела представя една стъпка от разработката на софтуерния продукт. Моделът дава свобода в тази посока, като при необходимост, една стъпка от разработката може да бъде извършена на няколко итерации. Моделът е либерален и в друго отношение - той позволява някои от итерациите да бъдат извършени като се следва друг модел, който е по-подходящ за

конкретния проект. Тази особеност прави спираловидния модел изключително гъвкав и адаптируем към динамичната природа на съвременните проекти.

Фигура 5 показва схемата на спираловидния модел:



Фигура 5: Спираловиден модел

Според Илиева [1], за всяка итерация моделът дефинира следните дейности:

- Установяване на целите - определят се целите, алтернативите и ограниченията за текущата итерация;

- Анализ на риска - включва идентификация, анализ и оценка на потенциалните рискове за текущата итерация и предприемане на действия за минимизирането им;

- Разработка и валидация - включва избор на модел за разработка на текущата итерация, разработка на работен продукт и валидацията му;

- Планиране - включва преглед и анализ на текущото състояние и изготвяне на план за следваща итерация.

Адаптируемостта към проекта и анализа на риска на всяка стъпка са основните предимства на модела. Те са резултат съответно от итеративната същност на модела и от прототипния подход. Тези предимства позволяват моделът да бъде използван за реализацията на големи и сложни проекти.

Сложността на спираловидния модел налага той да бъде прилаган от квалифицирани специалисти и в проекти с достатъчно налични ресурси. Не е подходящ за малки проекти поради необосновано ангажиране на ресурси. Наличието на обратна връзка от клиента е положителен елемент на модела, но както в случая на прототипния модел е наложително клиентът да е информиран за него и да съдейства за правилното му протичане.

Дизайнът заема важно място в модела и е представен с архитектурен и детайлен дизайн. Итеративността на модела намалява риска, дефекти в дизайна да се окажат критични за резултатите от проекта. Същевременно, както при другите итеративни модели, най–важното качество на дизайна е да бъде разширяем. Липсата на разширяемост може да се окаже критичен фактор за резултатите от проекта, поради невъзможност за извършване на следваща итерация.

Можем да резюмираме характеристиките на традиционните процеси, чрез следната Таблица 1:

Критерий / Модел	Сложност на концепцията	Степен на итеративност	Трудност на прилагане	Приложимост в рискови проекти	Приложимост в динамични проекти
Каскаден	ниска	ниска	средна	приемлив	неподходящ
Постъпков	ниска	средна	ниска	подходящ	приемлив
Прототипен	средна	висока	средна	подходящ	подходящ
Спираловиден	висока	висока	висока	подходящ	подходящ

Таблица 1 : Характеристика на традиционните процеси

2.2 Съвременни процеси

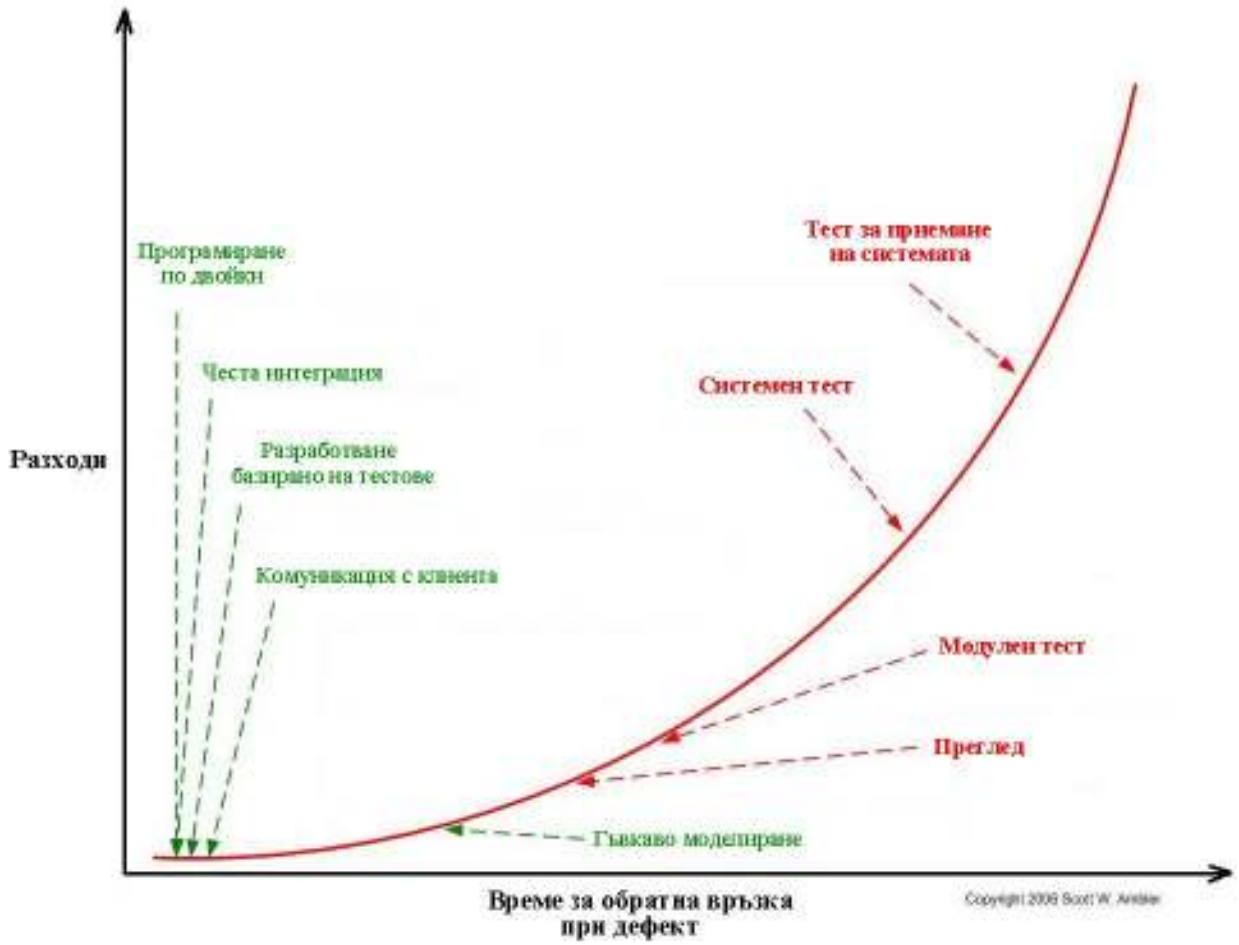
Съвременните процеси дават предписания за протичане на работния процес и създаване на продукта, но отчитат спецификата на конкретния проект, постоянно променящите се условия и дават възможност за по-голяма адаптация на процеса към проекта. Важно място в съвременните процеси заемат т. нар. "гъвкави" процеси, част от които са представени в тази глава. Основни характеристики на гъвкавите процеси са:

- Инкременталност – процесът е инкрементален, с кратък цикъл и в него е заложено често да се доставя работеща версия на клиента;
- Сътрудничество с клиента – екипът по проекта и клиентът работят в условия на добра взаимна комуникация;
- Лекота – процесът е лесен за усвояване и модифициране;
- Адаптивност – процесът позволява адекватно да се реагира на променящите се условия в динамична среда.

При гъвкавите процеси вниманието е насочено към комуникацията с клиента и удовлетворяване на нуждите му. Относно екипа по проекта, се залага на малки, но добре мотивирани екипи, като основна цел е изграждането на работещ софтуер. При хората от екипа, "гъвкавостта" е представена като способност да се реагира на промените по подходящ начин. Има стремеж към опростяване на процеса,

минимизиране на междинните работни продукти и документация. Различията от традиционните процеси са продиктувани от нуждата да се реагира адекватно на постоянно променящите се условия. Основните дейности (Комуникация, Планиране, Моделиране, Конструиране и Внедряване) присъстват, но задачите и дейностите в тях са намалени, така че максимално бързо да се изгради и достави работещ софтуер, които удовлетворява нуждите на клиента. Това понякога измества вниманието от дейностите на анализ и дизайн, но се счита че навременната доставка на работеща система е по–важна. Високо се оценяват индивидуалните качества, умения и комуникативност, защото при гъвкавите процеси фокусът е съсредоточен върху успеха на проекта, а хората са основен фактор за постигането му. Важно е да се отбележи, че изброените модификации на традиционния подход могат да се прилагат към всеки процес за разработка на софтуер [2].

Основна характеристика за всеки проект са разходите за реализирането му. Според Амблър (Ambler) [9], те драстично намаляват при добра комуникация с клиента и вътре в екипа. Причината е навременната обратна връзка при възникнали проблеми и бързата и адекватна реакция за отстраняването им. Сравнение на разходите за отстраняване на един дефект при гъвкавите и традиционните процеси е показано на Фигура 6. Етикетите на стрелките показват начина, по който е открит дефекта.



Фигура 6: Разходи за отстраняване на един дефект

при различни подходи за превенция

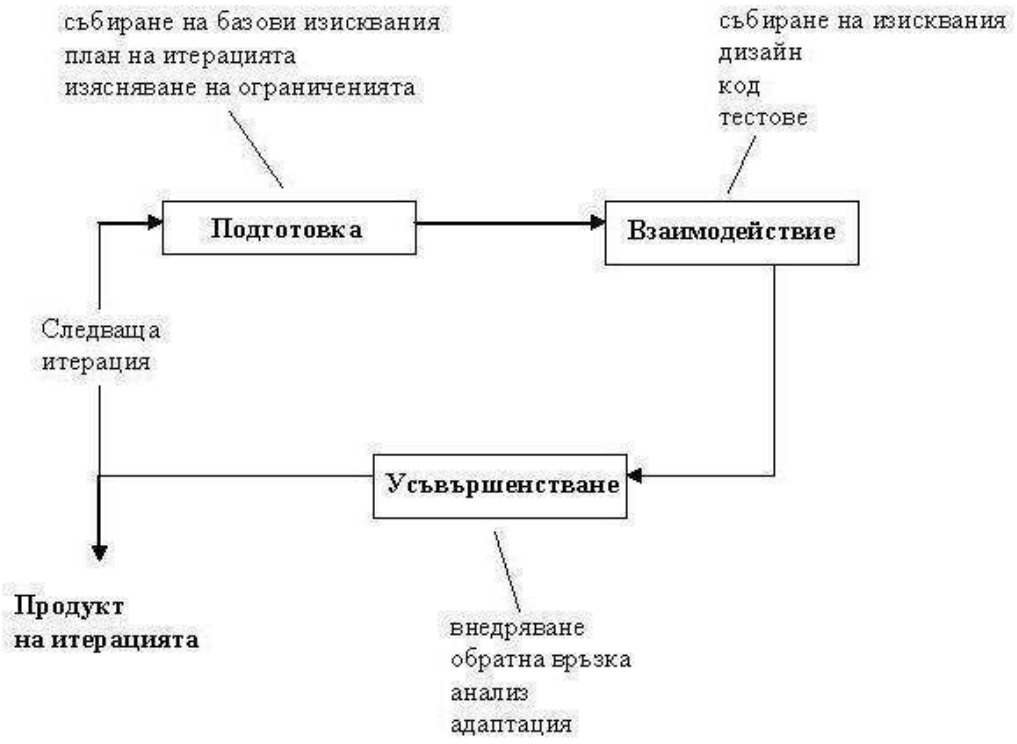
2.2.1 Адаптивно разработване на софтуер

Адаптивното разработване на софтуер (Adaptive software development - ASD) е подход за разработване на големи и сложни системи [1]. Подходът е итеративен и постъпков с използване на прототипи. Стреми се да даде максимална свобода, като същевременно се запазва управляемостта на проекта. Избягва се конкретиката, свързана с точни задачи, които трябва да се извършат, а се набляга на качеството на резултатите. Всяка итерация се състои от три фази:

- Подготовка (Speculation) – заема мястото на комуникацията и планирането, като подчертава динамичността на тези две основни дейности. В тази фаза започва итерацията (проекта). Извършва се планиране, като на базата на дефинираните изисквания и ограничения се определя функционалността за тази итерация (проект);

- Взаимодействие (Collaboration) – подчертава значението на умението за работа в екип, за постигането на добри резултати. Извършва се дизайн, имплементация и тестване на компонентите;

- Усъвършенстване (Learning) – заема мястото на внедряването, като подчертава важността на осъзнаването на грешките и адаптацията към новите условия. На базата на обратната връзка от клиента и техническите прегледи на продукта се изготвят доклади и се правят анализи.

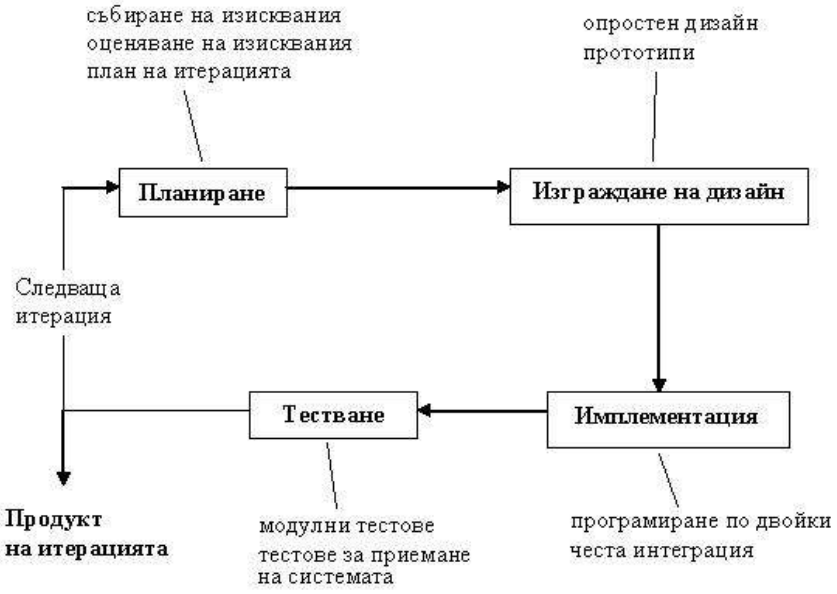


Фигура 7: Адаптивно разработване на софтуер

В основата на подхода са заложени човешките качества – самодисциплина, комуникативност и взаимодействие с околните. Те са база за вземане на адекватни решения на проблемите, които възникват поради динамичността на проектите.

2.2.2 Екстремно програмиране

Екстремното програмиране представлява множество от правила и практики, които са групирани в четири основни дейности – планиране, изграждане на дизайн, имплементация и тестване [6] [8].



Фигура 8: Основни дейности при екстремното програмиране

Планирането стартира със създаване на документи, в които са описани характеристиките и функционалността на бъдещата софтуерна система. Всеки документ описва една главна функционалност. Тези документи се написват от клиента и им се дава приоритет, на базата на бизнес нуждите. Впоследствие, екипът по проекта анализира и оценява всеки документ и определя себестойността

на описаната функционалност. Ако оценката е голяма (над три седмици), функционалността трябва да се прецизира и раздели на по–малки части, след което процесът се повтаря. Екипът по проекта и клиентът работят в тясно сътрудничество и решават кои са функционалностите за следващата доставка, какъв е срока за изпълнение и др. След като функционалностите за следващата доставка са изяснени, екипът по проекта решава в каква последователност да ги имплементира. Вариантите са:

- Всички функционалности се разработват заедно;
- Първо се разработват най – приоритетните функционалности;
- Първо се разработват най – рисковите функционалности.

Ако е имало предишна итерация, се прави анализ на базата на предишния план и постигнатите резултати и ако е необходимо се актуализира плана. Подходът позволява нови функционалности да бъдат добавяни, променяни и отстранявани от клиента по всяко време. Екипът по проекта прави анализ на влиянието на всяка промяна върху проекта и ако е необходимо се актуализира плана.

Дизайнът трябва да бъде възможно най–прост. Основното му предназначение е да дава информация за дадена част от системата. Използва се обектно-ориентиран подход, при който на всеки клас съответства картонче (CRC card), на което са отразени отговорностите на класа. Тези картончета са единствения работен продукт, в резултат на дизайна. При неясноти или проблеми, свързани с дизайна, подходът препоръчва създаване на прототипи, чрез които бързо да се намери решение. Традиционно дизайна се извършва преди да е започнала имплементацията. При екстремното програмиране, той се извършва и след като имплементацията е започнала. Това е съвсем естествено, защото по време на имплементацията се появяват идеи за подобряване на дизайна.

Имплементацията стандартно започва след като дизайна е готов, а след нея се написват тестовете. Екстремното програмиране препоръчва, след като дизайна от високо ниво е готов, на базата на изискванията да се разработят модулни тестове. Така вниманието на разработчиците се фокусира върху проблемната област и те са

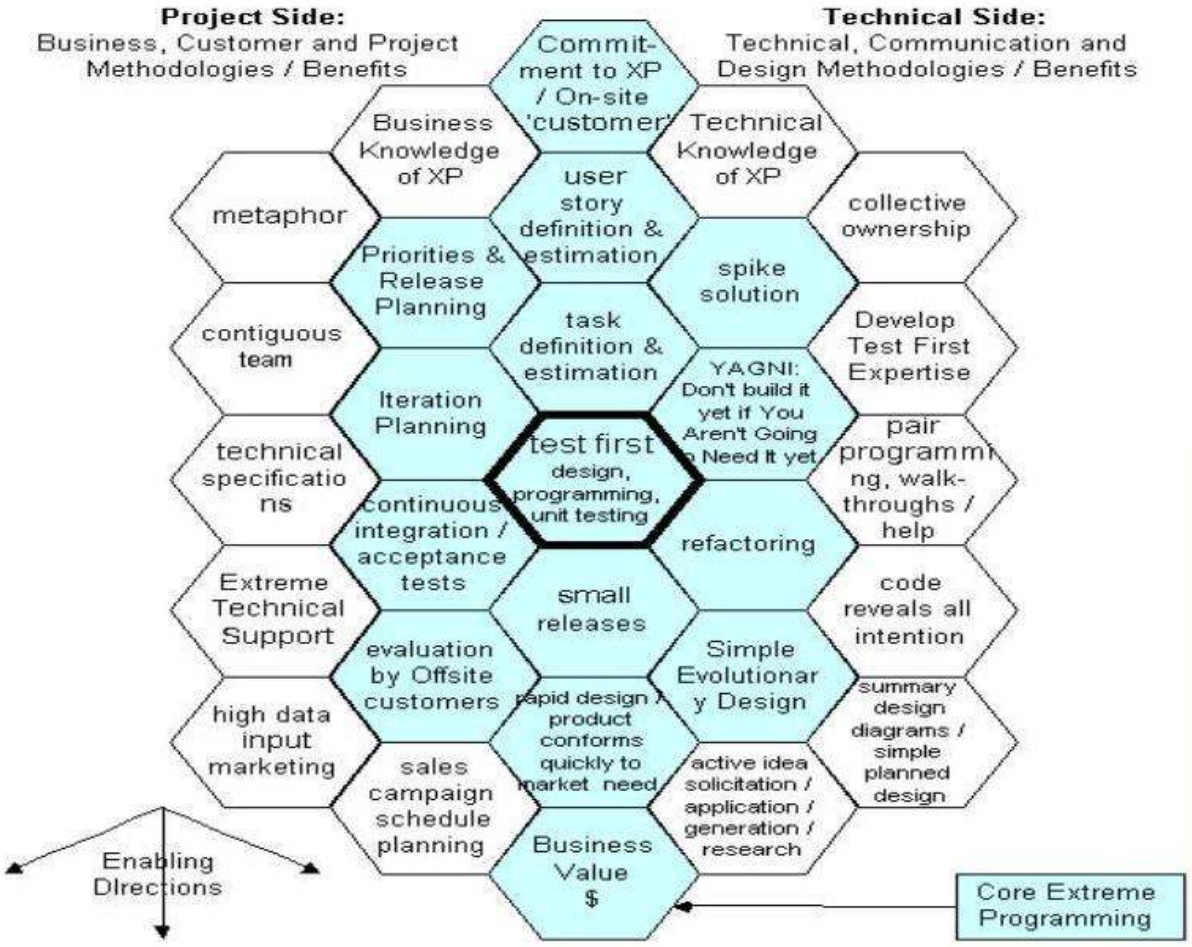
наясно какво да направят, за да удовлетворят тестовете. Така кодът може да бъде тестван, веднага щом е готов. Една от най-специфичните особености на екстремното програмиране е практиката “Програмиране по двойки”. Същността й е, че двама души работят на един компютър, единият пише кода, а другият следи за проблеми, свързани с текущата имплементация (качество на кода, алгоритми, връзки с други модули и т. н.). Това позволява контрол на качеството в реално време и предотвратява последиците от евентуални дефекти. След като някаква част от кода е готова, тя се интегрира с останалата част от системата, което позволява ранно откриване на несъответствия в интерфейсите и други проблеми на ниво архитектура.

Тестването при екстремното програмиране се явява като част от същинската разработка. Подходът препоръчва тестовете да са автоматизирани, за да могат да бъдат изпълнявани многократно. Целта е да се спести време и да се намали риска от регресия при промяна на кода. Така модулните, интеграционните и системните тестове могат да се изпълняват автоматично през определен период от време. Това улеснява поддържането на стабилна версия на системата и дава възможност за бързо откриване на възникнали проблеми. Тестовете за приемане на системата се изготвят на базата на документите от планирането, които описват характеристиките и функционалността на системата.

Причината екстремното програмиране да е толкова близо до реалните проекти и проблемите в тях е, че подходите, които се предлагат, първо успешно са се доказали на практика, а впоследствие са конкретизирани, обобщени и формулирани теоретично. Поради стремежа към минимизиране на всички дейности, различни от имплементацията, екстремното програмиране е подходящо за малки и средни екипи. Процесът силно се придържа към основните характеристики на гъвкавите процеси – инкременталност, сътрудничество с клиента, лекота и адаптивност.

Екстремното програмиране включва практиките „Програмиране по двойки”, „Честа интеграция”, „Подобряване на методите за дизайн”, „Малки итерации, всяка от които завършва с работещ краен продукт”, „Въвеждане и спазване на правила за писане и форматиране на кода”, „Планиране” и „Разработване, базирано на тестове (Test-driven development)”. Гровър (Grover) [5] групира практиките, като показва, че те са взаимно свързани. За успешното прилагане на всяка практика, се изисква да

бъдат налице определени предпоставки, които според него са резултат от прилагането на други практики. Тяхното прилагане от своя страна, изисква прилагането на трети и т. н. Така се стига до базовите предпоставки – желание за прилагане на практиките на Екстремното програмиране, познаване на теоретичните му основи и начина на прилагането му в бизнеса. На Фигура 9 са показани практиките на екстремното програмиране, резултатите и връзката между тях:



Фигура 9: Връзка между практиките на екстремното програмиране

Прилагането на всяка практика е възможно, ако се прилагат практиките, които на фигурата са над нея. Съответно, прилагането на всяка практика дава възможност за прилагането на тази под нея, тази вляво и тази вдясно. Така практиките заемат

горната част, а резултатите от прилагането им – долната. Централно място заема практиката “Разработване, базирано на тестове” (Test – driven development).

Представените гъвкави процеси са обощени в Таблица 2:

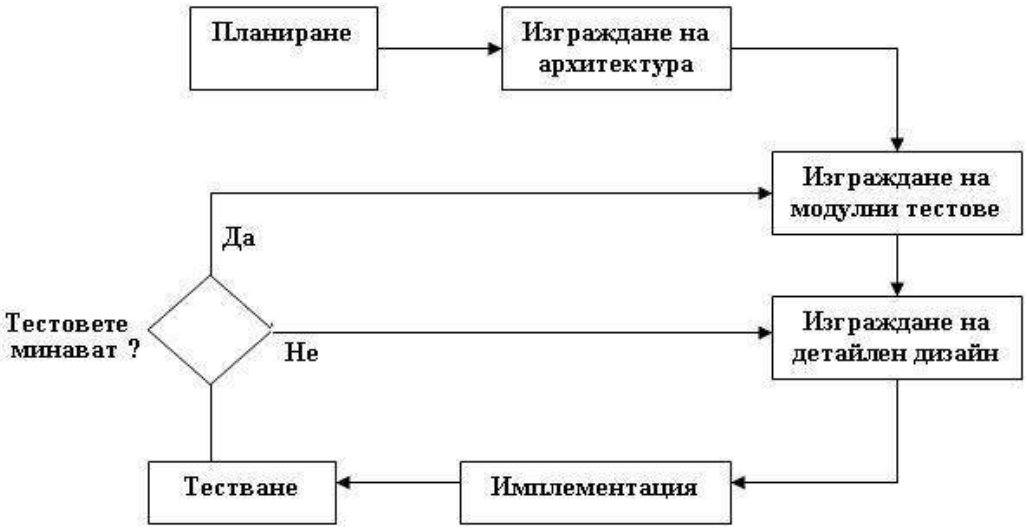
Критерий / Модел	Сложност на концепцията	Степен на итеративност	Трудност на прилагане	Приложимост в рискови проекти	Приложимост в динамични проекти
Адаптивно разработване на софтуер	средна	висока	висока	подходящ	подходящ
Екстремно програмиране	средна	висока	средна	приемлив	много подходящ

Таблица 2 : Характеристика на гъвкавите процеси

3 Разработка на базата на тестове

3.1 Същност

Разработването на софтуер следва процес, който съдържа определени дейности, изпълнявани в определена последователност. Традиционният подход поставя написването на модулни тестове след имплементацията на самите модули. Подходът на екстремното програмиране, като част от гъвкавите методологии, предлага практиката "Разработване на базата на тестове", която в литературата се среща като "test driven development", "test first design", "test first programming" and "test driven design" (всички от [13]). Практиката на разработване на базата на тестове предлага написването на модулни тестове да предшества имплементацията на самите модули. Това е възможно, като отговорностите (responsibilities) на даден модул, описани в архитектурата, се приемат за база за написване на модулни тестове. Концепцията на практиката е показана на Фигура 10:



Фигура 10: Разработване на базата на тестове

Така, при изградена архитектура, с дефинирани отговорности за съответните модули, има възможност да се съставят модулни тестове, преди модулите да са проектирани детайлно. За всяка отговорност, описана в архитектурата, се изграждат един или повече модулни теста, които да я верифицират по подходящ начин. Впоследствие се изгражда детайлния дизайн на всеки модул и се имплементира. Написаните тестове се прилагат още при имплементацията на модула, веднага щом има възможност част от него да бъде тествана. Има два възможни варианта, в зависимост от резултатите от тестването. Ако имплементацията не е удовлетворила съответните тестове, то тя не реализира съответните отговорности и трябва да бъде поправена. Процесът се връща във фазата на детайлен дизайн, който да изпълни отговорностите на съответния модул. Детайлният дизайн се имплементира и отново се правят тестове. Ако имплементацията е удовлетворила съответните тестове, то тя е коректна и се преминава към изграждане на модулни тестове за други отговорности, описани в архитектурата. Итеративно отговорностите на всички модули, описани в архитектурата се "покриват" от модулни тестове, което гарантира тяхната коректна имплементация.

Важен принцип на практиката е, че изготвянето на модулни тестове винаги предхожда имплементацията на тествания компонент. Така, през целия период на разработка имаме тестове, които да верифицират частично изградения продукт. Дадена част от системата се приема за готова, веднага щом преминава успешно написаните за нея тестове. Понеже тестването е итеративно повтаряща се дейност, за ефективно прилагане на практиката, е необходим софтуерен инструмент за автоматизация на тестовия процес.

Практиката препоръчва тестовете да са възможно най–малки и опростени. Те имат за цел да тестват единствено съответната функционалност и то във възможно най–атомарен вид. Направата и поддръжката им са по–лесни, а резултатите от тях носят повече информация за тестваните компоненти.

Модулните тестове тестват модулите на ниво функция. За да се получи глобална визия за състоянието на тестваната система, е необходимо те да бъдат групирани по определен начин и да им бъдат осигурени подходящи условия за работа.

Можем да обобщим, че всеки тест се изпълнява на три стъпки – инициализация, провеждане на тест и завършващи дейности:

- Инициализация – подготвя тествания компонент за теста;

- Провеждане на тест – пробно използване на тестовия компонент и проверка на състоянието и резултатите;

- Завършващи дейности – възстановява първоначалното състояние на тествания компонент и неговата околна среда.

Последната стъпка е от изключителна важност за коректното протичане на тестовия процес. Тя осигурява равнопоставеност на тестовете и елиминира възможността някой компонент да бъде тестван в неочаквана (недефинирана) среда.

Разработката, базирана на тестове дефинира четири основни понятия, свързани с реализацията на тестването и изграждането на тестовата среда:

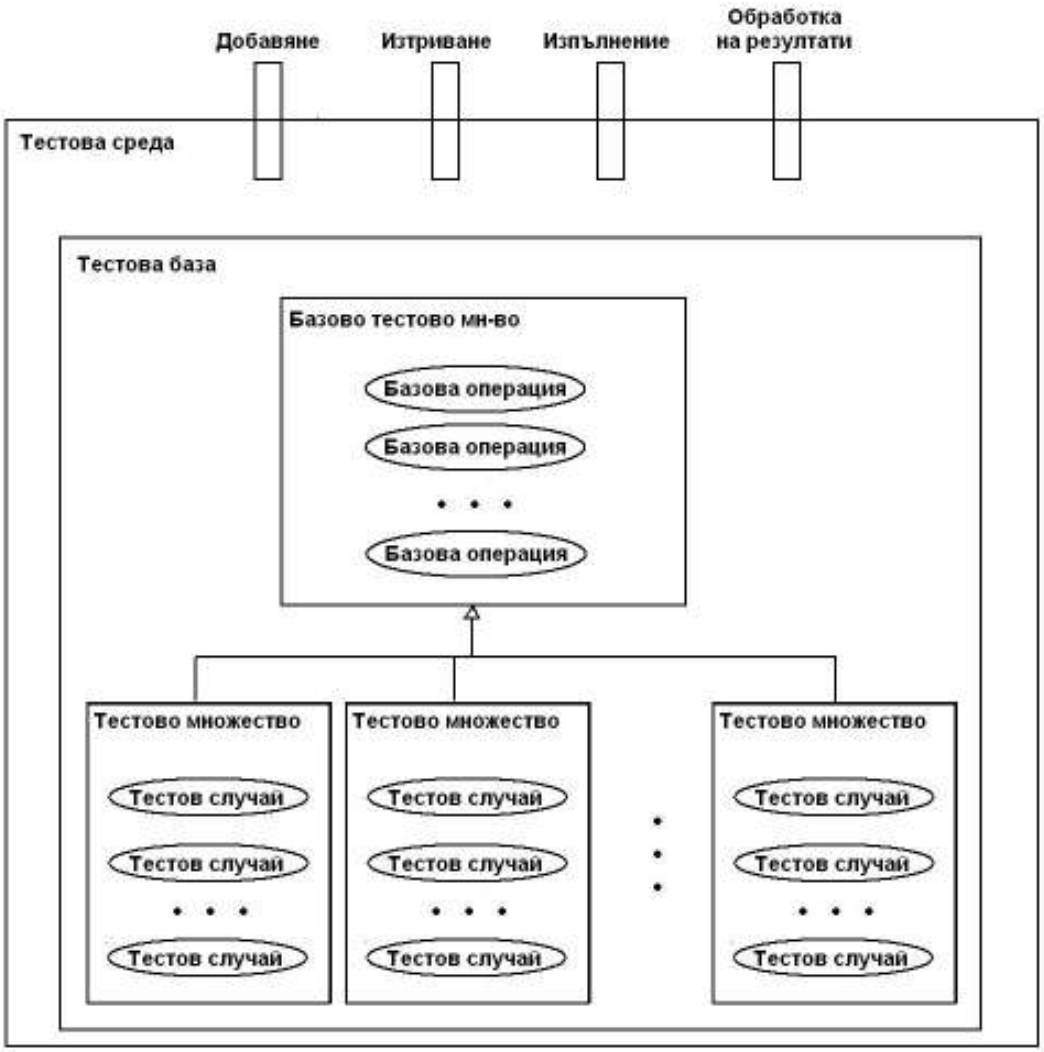
- Тестов случай (Test case) – има за цел да тества поведението на определен модул (клас) в дадена ситуация. Обикновено се реализира чрез тестова функция. Всеки тест поставя тествания обект в подходящо за теста състояние, след което следи поведението и резултатите му в тестваната ситуация. Тестовите случаи извършват същинското тестване и трябва да са независими един от друг и от реда им на изпълнение;

- Тестово множество (Test suite) – състои се от множество от тестови случаи, които са логически свързани. Има за цел да тества поведението на определен модул (клас) в различни ситуации. Показва кои тестове са логически свързани. Реализира се чрез тестов клас, който има достъп до всички данни на тествания. Всеки тест поставя тествания обект в подходящо за теста състояние и следи поведението и резултатите му;

- Тестова база (Test fixture) – представлява обвивка на тестови множества. Има за цел да създаде необходимите условия за провеждане на тестовете. Отговорен е за инициализиращите и завършващите действия, както и за същинското изпълнение на тестовете. Отново, тестовете трябва да са независими един от друг и от реда им на изпълнение. Реализира се чрез йерархия от тестови класове. В базовите класове се дефинират общите операции, свързани с тестването (инициализация и завършване на теста). Те се предефинират в наследниците (тестови множества), ако е необходимо. Всеки тест поставя тествания обект в подходящо за теста състояние, инициализира го по подходящ начин, провежда теста и разрушава тествания обект. Изключително важно е, тестваната система (компонент) да възстанови първоначалното си състояние след завършване на теста;

- Тестова среда (Test harness) – съдържа тестовата база и предоставя условия за създаване, добавяне и изтриване на тестови множества, както и за провеждане на съответните им тестове. Разполага със средства за контрол на тестването, запазване и анализ на резултатите.

Структурата на тестова среда е показана на Фигура 11:



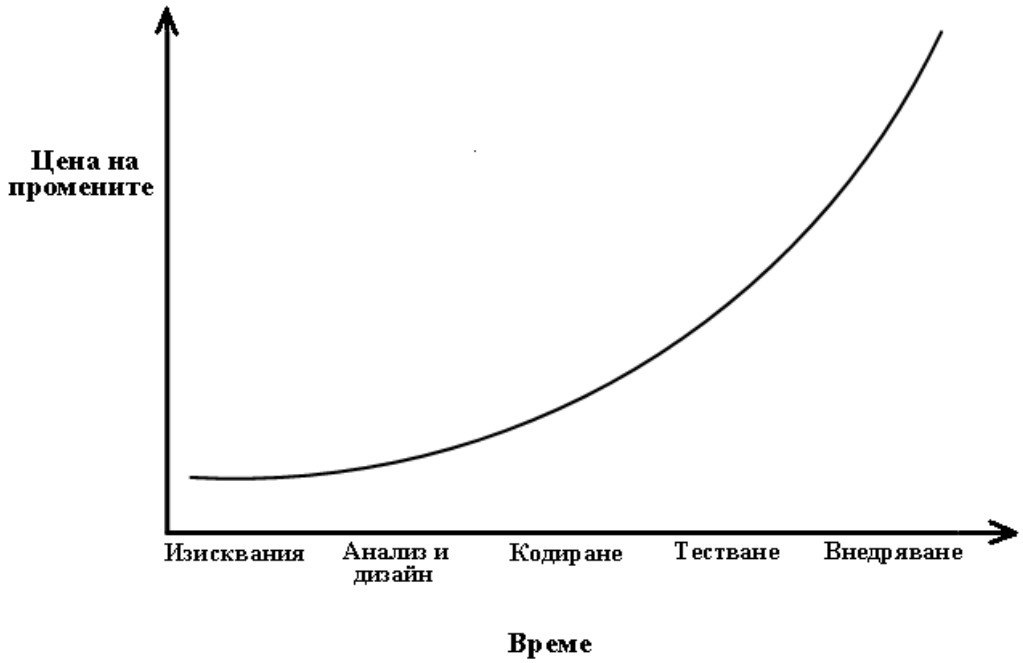
Фигура 11 : Структура на тестова среда

3.2 Предимства и недостатъци

Според Чаплин [4], важен принцип на разработката, базирана на тестове е, че ако човек не може да напише тест за това, което е започнал да имплементира, то все още е рано за имплементация. С други думи, щом изискванията към модулите не могат да се трансформират до тестове, то те не са достатъчно ясни. Следователно, подходът притежава естествен механизъм за проверка на качеството на архитектурата, което подобрява процеса на разработка и повишава качеството на продукта.

След като тестовете за дадена архитектурна единица са готови, водещо при подхода е да се проектира и имплементира код, който ги удовлетворява. Поради това, ще бъдат създадени само класове (в термините на ООП), които са наистина необходими, което от своя страна води до по-добро сцепление (cohesion) и по-слаба свързаност (coupling) на класовете.

Наличието на готови тестове по време на имплементацията спомага за навременен ефективен контрол на качеството на продукта и обратна връзка, което намалява риска от разминаване между дизайна и имплементацията на системата. В резултат, намалява общия брой на дефектите и влиянието им върху продукта и проекта. Освен това, готовите тестове улесняват провеждането на тестове за регресия (regression test), които винаги могат да бъдат пуснати, за да се установи текущото състояние на продукта. Последните предимства са изключително ценни, като се има предвид, че цената на дефектите силно расте във времето:

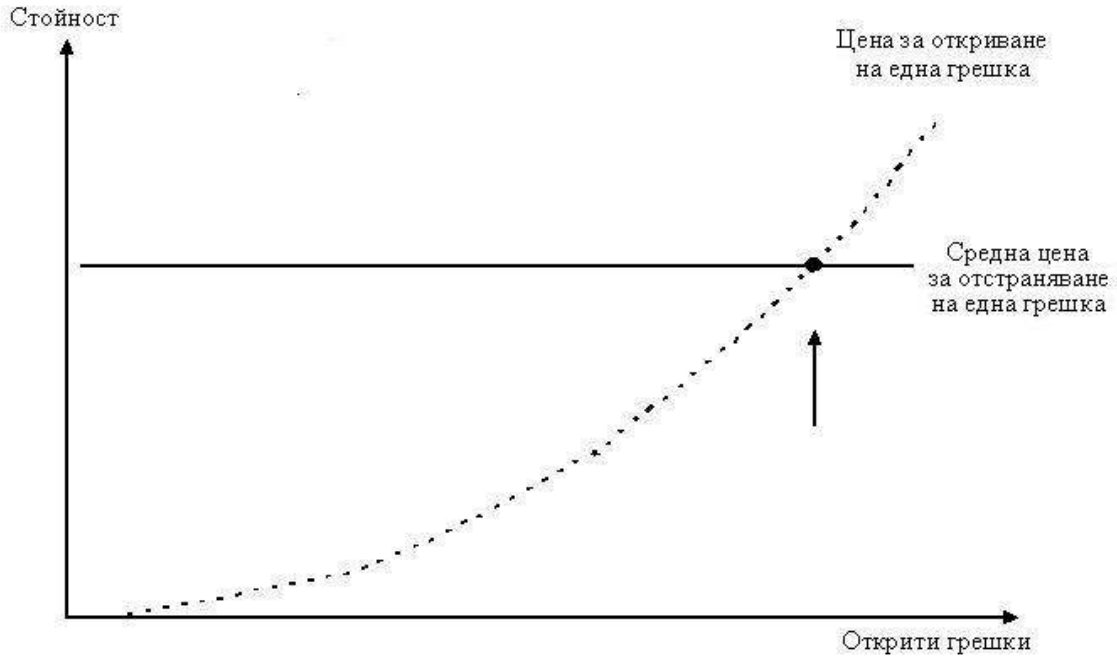


Фигура 12: Изменение на цената на промените във времето [12]

Кривата на последната фигура е представена от Боем (Boehm) [7]. На нея се вижда, че колкото по-късно се установи един дефект, толкова по-скъпо струва

неговото отстраняване. Тази е причината, много усилия да са насочени към подобряване на качеството на работните продукти от началните фази на разработката – документ за софтуерните изисквания, документ за софтуерната архитектура и детайлен дизайн. В резултат на контрола в реално време, практиката на разработване, базирано на тестове повишава качеството и същевременно понижава цената на продукта.

Планирането и реализацията на тестовете също има своята цена, която може да се окаже проблем за проекта, поради липса на достатъчно време или ресурси. Следователно, както при всички видове тестване, трябва да се търси баланс между качеството и цената на продукта. Оказва се, че когато качеството не е критичен фактор, има смисъл от тестване, само ако цената на откритите дефекти е по–голяма от цената на тестовете. Тази зависимост е показана на Фигура 13:



Фигура 13: Момент на спиране на тестването

Имплементацията на базата на тестове, повишава възможностите за автоматизация на тестването, защото продуктът се разработва така, че да премине успешно през определени тестове и следователно е лесен за тестване.

Автоматизацията на тестовия процес повишава качеството на продукта и намалява разходите за тестване.

По отношение на дизайна, подходът предполага инкременталност на малки стъпки, което дава възможност за вземане на адекватни и конкретни решения. Често по време на дизайн се обръща внимание само на интерфейса на отделните съставни единици на компонентите (класове и функции). Стремежът да бъдат удовлетворени определени тестове, налага да се отдели внимание и на поведението на тези съставни единици, което подпомага дейностите на анализ и дизайн.

Акцент на старта на всяка итерация (изграждане на тест), е анализ на функционалността, за която ще бъде изграден тест. По време на изграждането на тестове се извършват дейности, характерни за фазата на моделиране. Подпомага се изучаването на проблемната област, което е важна предпоставка за успешното моделиране на продукта. Най–естествено е да се започне с отговор на въпроса "Как мога да тествам този компонент ?". Отговорът носи информация за функционалността на компонента и начинът му на ползване.

Съществува положителен ефект от наличието на тестове през цялото време на разработка. В началния етап, преди кода да е готов, тестовете описват желанания начин на употреба на компонента, за който са написани. След имплементацията, те са примери за употреба на компонента, за който са написани и така улесняват ползването му.

Забелязваме, че разработчикът разполага с коректив на своята работа още преди да е започнал детайлния дизайн на тествания компонент. Във всеки момент от имплементацията, целия наличен програмен код може да бъде тестван, което носи увереност, че продуктът се развива в правилна посока. Спокойствието и сигурността от възможността за проверка, благоприатстват подобряването на имплементацията (refactoring) и интеграцията. Подобряването на имплементацията е тясно свързано с описваната практика. След като някой от тестовете премине успешно за пръв път, е важно съответният му код да бъде изчистен от излишества, опростен и приведен във вид, съответен на правилата за писане и форматиране на кода. Тестовете гарантират бърза и коректна обратна връзка за резултата от работата.

Готовите тестове определят каква функционалност трябва да се имплементира. Естествено, разработчикът търси най–лесния начин, като критерий

за качество са тестовете, които могат да бъдат изпълнявани по всяко време. Следователно, резултатният код е максимално опростен, откъдето следва и по–добро качество на съответния му модулен дизайн.

Прави впечатление, че итеративността на подхода гарантира, че всички готови елементи от разработваната система притежават тестове. При ползване на наследен (външен) код, инвестицията за изготвяне на тестове се оказва доста голяма, а ползите от нея са по–малки, понеже компонентите са вече готови. В такива случаи, е подходящо компонентите, за които няма тестове да бъдат "обвити" с итерфейс, който е разработен съгласно практиката "Разработване, базирано на тестове".

Основни фактори, които влияят на приложимостта на практиката, са степента на гъвкавост на процеса и количеството съпътстващи административни дейности. "Лекотата" и "прозрачността" на процеса са предпоставки за по–лесното прилагане на изследваната практика. Обратно, тромавия и неадаптивен процес, многото документация и стриктното следване на определени правила, затрудняват прилагането на практиката.

4 Описание на проведения експеримент

Тази глава описва проведения експеримент, който има за цел да внедри практиката “Разработване, базирано на тестове” в съществуващ софтуерен процес и да изследва влиянието й върху дизайна на софтуерна система. В началото е представено подобно изследване и резултатите от него. Разгледан е съществуващия процес за разработка на софтуер, който приемаме за базов и модификациите му, в съответствие с изследваната практика. За провеждане на експеримента е избран модул от разработвана система, който да бъде разработен съгласно двата процеса– базов и модифициран. Включено е описание на тестовата среда, която е използвана за прилагане на практиката. В края на главата са описани детайлите на проведения експеримент, а резултатите са подробно разгледани в следващата глава.

4.1 Подобни изследвания

Проведен е експеримент [14] с няколко групи от по осем програмисти, целящ изследване на влиянието на практиката “Разработване, базирано на тестове” върху процеса за разработка на софтуер. Специалистите са разпределени в две групи на случаен принцип – такива, които да ползват изследваната практика, а останалите да ползват традиционен подход. Целта на втората група е да бъде индикатор за разликите, които са резултат от прилагането на изследваната практика. И двете групи разработват част от софтуерна система, като впоследствие резултатите им ще бъдат сравнени. Няколко тестови сценария са разработени, с цел да се сравнят продуктите, получени по двата метода. След провеждане на експеримента се оказва, че групите, които работят според изследваната практика са направили по–качествени тестове, които могат да се ползват и след промени в системата, докато тестовете на другата група са тясно свързани с текущата имплементация и не могат да бъдат ползвани при бъдещо разширение на продукта. Това е очакван резултат, като се вземе предвид факта, че едните тестове проверяват изисквания към система, която още не съществува, а другите имат за цел да докажат, че вече направеният продукт отговаря на изискванията. Освен това, резултатният код,

получен след прилагане на изследваната практика е по–удобен за тестване и отговаря на изискванията, описани в архитектурата, което е естествено, като се има предвид, че той е написан с цел да удовлетвори фиксирани тестове.

Прилагането на изследваната практика в случая е отнело повече време (~16%), в сравнение с традиционния подход. Трябва да се отчете, че тук не е включено времето за възстановяване на дизайна след имплементацията (Reverse engineering), а при разработката на базата на тестове тази дейност отпада, поради факта, че кода е идентичен с дизайна. Резултатите показват, че групите, които прилагат практиката са получили по–качествен код и са приключили с работата си, едва когато са разполагали с множество автоматизирани тестове и код, който ги удовлетворява. За сравнение, паралелните групи са приключили работа даже без готови автоматизирани тстове.

В изследването е включен анализ на покритието на кода, което е постигнато от двете групи. Оказва се, че групата, прилагаща изследваната практика постига по–голямо покритеие на кода, което се дължи на тестовете, написани в началната фаза на процеса, които обхващат повече случаи на неправилно използване на тествания компонент.

Прилагайки специфични методи за оценка, е получено, че разбирането на изискванията е било с 87% по–добро, отколкото при традиционния подход за разработка. Допълнително, полученият код е бил по–лесен за тестване, а оттам и за дебъгване, което снижава разходите да поддръжка.

Установен недостатък на подхода е, че той е отнел повече време от традиционния, което се обяснва с по–голямото време, отделено за писане на тестове. Трябва да се отбележи, че в този случай, разработчиците, които прилагат практиката са имали по–добра производителност, което дава повече време за анализ и дизайн на кода.

4.2 Базов процес

Базовият процес за разработка на софтуер е итеративен и в края на всяка итерация излиза работещ продукт. Особено внимание е отделено на първата половина от жизнения цикъл на проекта, в която се минимизират предпоставките

за възникване на дефекти. Този период обхваща времето от събирането на изискванията до написването на кода. Качеството на всеки работен продукт от този период се осигурява от прегледи, на които се идентифицират дефекти. Следва корекция на съответния работен продукт, съгласно описаните забележки и преминаване към следващата дейност. Схема на базовия процес за разработка на софтуер е показана на Фигура 14.

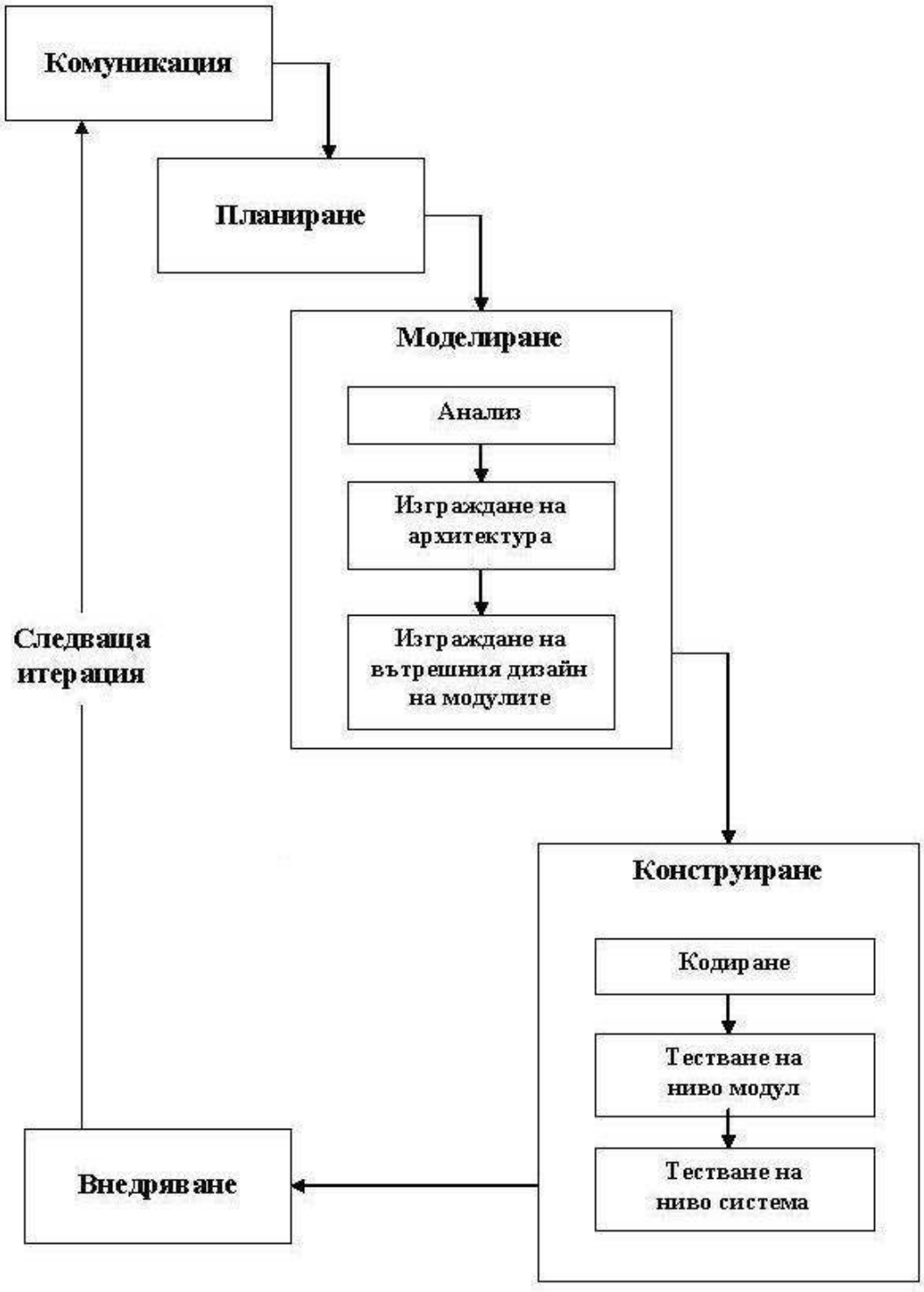
Петте общи дейности (Комуникация, Планиране, Моделиране, Конструиране и Внедряване) протичат в указаната последователност, като всяка завършва с работен продукт, които е база за следващата дейност. Съществени за експеримента са дейностите моделиране и конструиране, които са представени по–детайлно в модела на базовия софтуерен процес. Няма особености при дейностите комуникация и планиране.

Дейността моделиране съдържа три поддейности:

- Анализ – анализиране на изискванията към софтуера. Целта е клиентските изисквания да бъдат трансформирани до функционални изисквания към системата от техническа гледна точка;

- Изграждане на архитектура – на база на техническия анализ, се изгражда архитектура на бъдещата софтуерна система. Целта е да се дефинира изцяло софтуерната архитектура – отговорности, модули, връзки, комуникации, ограничения;

- Изграждане на модулен дизайн – на база на архитектурта, се изгражда модулния дизайн на бъдещата софтуерна система. Целта е да се дефинира структурата на всеки модул – отговорности, подмодули, връзки, комуникации, ограничения, специфични алгоритми, които ще се използват.



Фигура 14: Базов процес

След като модулния дизайн е готов, се преминава към конструирането на проектираното решение. Дейността конструиране съдържа три поддейности:

- Кодиране – имплементация на проектираните компоненти, съгласно модулния дизайн;

- Тестване на ниво модул – на база на модулния дизайн, се изграждат модулни тестове, които да верифицират реализацията на модулния дизайн. Целта е да се провери, че модулът изпълнява отговорностите, заложени в детайлния дизайн;

- Тестване на ниво система – на база на архитектурта, се изграждат системни тестове на цялата софтуерна система. Целта е да се провери, че всички компоненти, събрани заедно, изпълняват задачите, за които са предназначени и между тях няма несъответствия.

Процесът завършва с дейността внедряване. Тя има за цел да внедри системата в реални условия и да докаже, че реализацията на системата като цяло, отговаря на клиентските изисквания. Открити проблеми на това ниво се решават на следваща итерация, като се залагат промени в изискванията, архитектурата и дизайна.

4.3 Внедряване на практиката в базовия процес

Описаните в предишния раздел дейности, определят действащия процес за разработка на софтуер, който приемаме за базов. Модифицираме базовия процес, като внедряваме в него практиката “Разработване, базирано на тестове” и изследваме влиянието й върху дизайна на софтуерната система. Тази практика поставя изготвянето на модулни тестове преди анализа и дизайна на ниво модул. Това става, като изискванията към отделен модул от софтуерната архитектура бъдат трансферирани до модулни тестове, които да ги верифицират. Тестовете трябва да са готови за ползване преди започването на дизайна на ниво модул. След

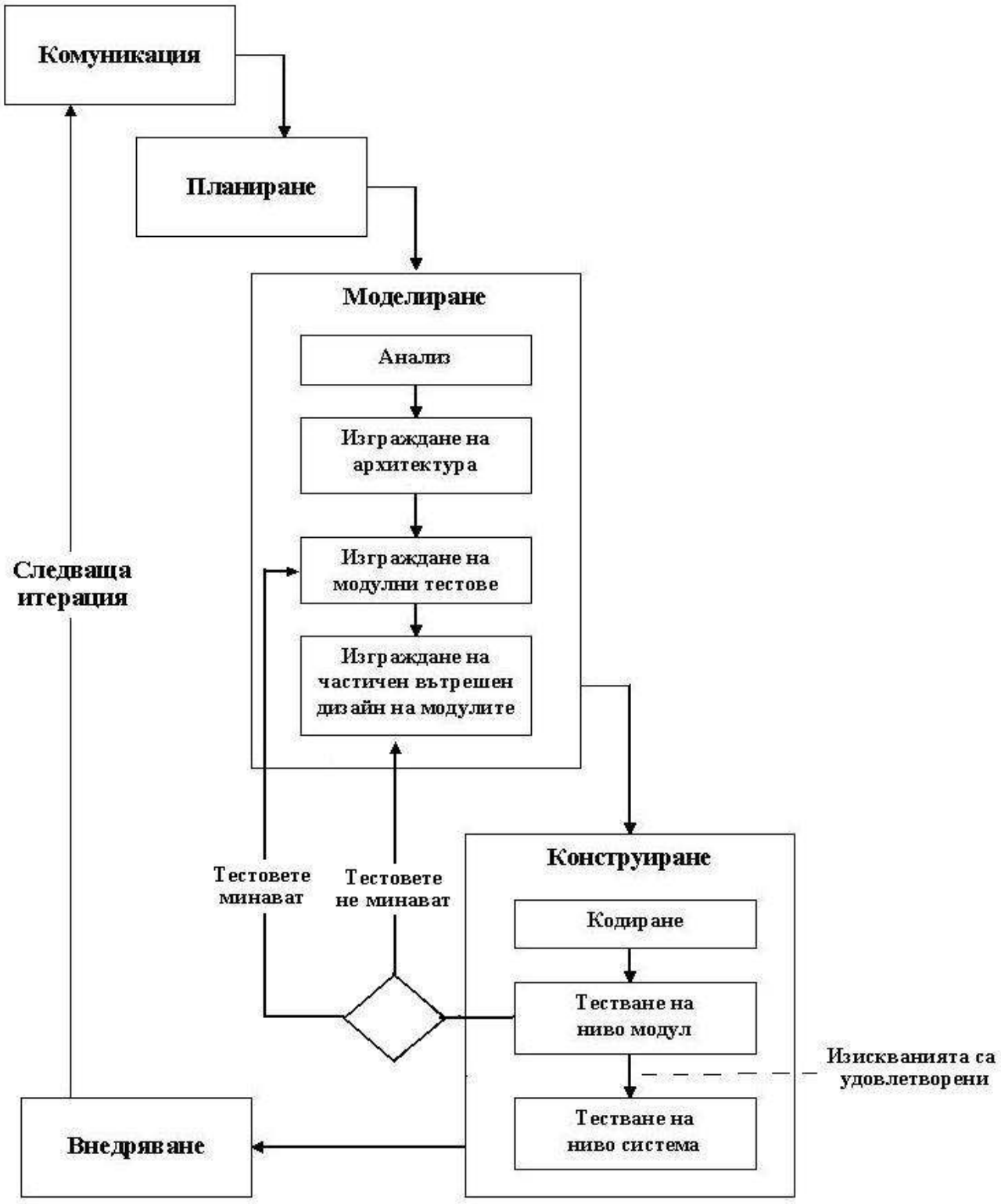
това се преминава към анализ и дизайн на модула с цел да се изготви детайлен дизайн. Критерий, за това доколко дизйнът е готов, е степента, до която съответния му код може успешно да удовлетвори тестовете, които са предназначени за него. Следва имплементация на дизайна и проверка на имплементацията с вече готовите тестове. Ако тестовете не минават успешно, детайлния дизайн и имплементацията се коригират и отново се подлагат на проверка с тестовете. При успешен изход от тестовете, се преминава към изготвяне на нови модулни тестове за същата итерация или се продължава с интеграционни тестове, ако всички изисквания за текущата итерация са имплементирани.

Базовият процес е итеративен, като основните дейности протичат последователно на всяка итерация. Внедряването на практиката “Разработване, базирано на тестове” е в хармония с итеративната природа на процеса и като резултат я усилва. Добавен е цикъл в най–рисковия участък от модела на процеса – дизайна и имплементацията. Положително е, че индикатор за изпълнението на този цикъл е резултата от имплементацията, което осигурява по–добро качество. Високата степен на итеративност, води до по–голама гъвкавост и адаптивност на процеса. Съществен е момента на получаване на първа обратна връзка за коректност на решението – той е максимално рано във времето, което води до по–малки разходи за отстраняване на потенциални дефекти. Модификацията на базовия процес е показана на Фигура 15.

Използвания базов процес и неговата модификация са сравнени в Таблица 3:

Критерий / Модел	Сложност на концепцията	Степен на итеративност	Трудност на прилагане	Приложимост в рискови проекти	Приложимост в динамични проекти
Базов процес	средна	средна	средна	подходящ	подходящ
Модифициран процес	средна	висока	средна	подходящ	много подходящ

Таблица 3 : Характеристика на използваните процеси



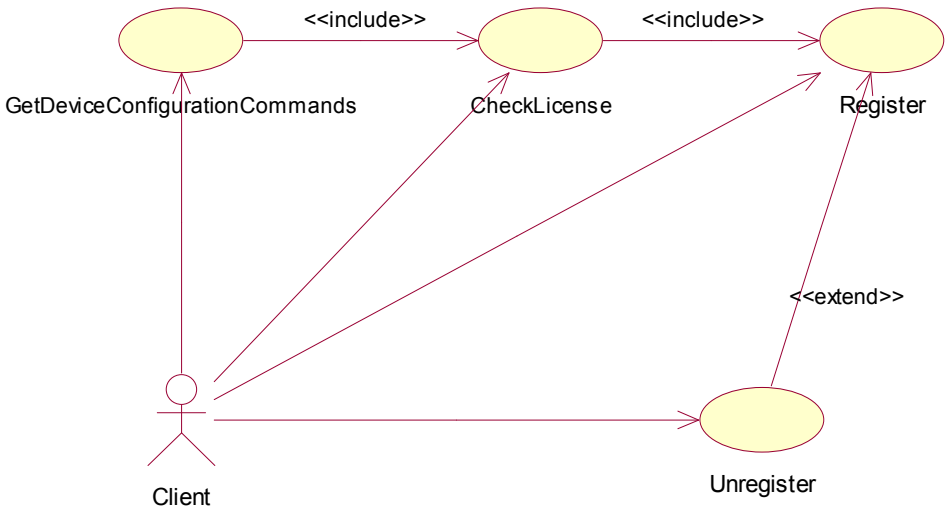
Фигура 15: Модификация на базовия процес с практиката

“Разработване, базирано на тестове”

4.4 Описание на разработвания модул

Изискванията към разработвания модул са описани на ниво архитектура. За база за провеждане на експеримента е избран модулът “Конфигуратор на устройства” (DeviceConfigurator), който е отговорен за конфигурирането на устройства на базата на конфигурационни файлове. Компонентът е предназначен да конфигурира устройства, анализирайки техните характеристики. Като резултат генерира серия конфигурационни команди, които служат за конфигуриране на устройство.

На Фигура 16 е показана диаграма на случаите на употреба (use – case diagram) на модула “Конфигуратор на устройства”:



Фигура 16 : Употреба на модула “Конфигуратор на устройства”

Отговорности на компонента:

- Регистриране / дерегистриране на клиент (устройство за конфигурация);
- Проверка на валидност на лицензa на клиент:

- Мениджмънт на лицензни файлове;
- Интерпретация на лицензен файл;

- Конфигуриране на устройство:

  - Мениджмънт на конфигурационни файлове;
  - Интерпретация на конфигурационен файл.

4.5 Среда за разработка

За провеждане на експеримента бяха използвани средата Visual C++ 6.0 и тестовата среда CPP Unit. CPP Unit е тестова среда, която предоставя възможност за изготвяне и автоматизиране на тестове. За всеки клас с име Class, се дефинира клас с име ClassTestCase, който наследява класа CppUnit::TestCase и притежава методи, които тестват методите на Class. Класът ClassTestCase се регистрира като тестов чрез специални макроси в декларацията. Класът TestCase дефинира методите setUp() и tearDown(), които се извикват съответно преди и след всеки тест. SetUp() извършва подготвителни дейности, за да се осигури подходяща околна среда за провеждането на тестовете. TearDown() има отговорността да възстанови тестовата среда в първоначалния й вид. Важно е да се следи за взаимна независимост на тестовете – всички тестове заедно трябва да се държат по начина, по който се държат поотделно. Примерна декларация на тестов клас е показана на Фигура 17:

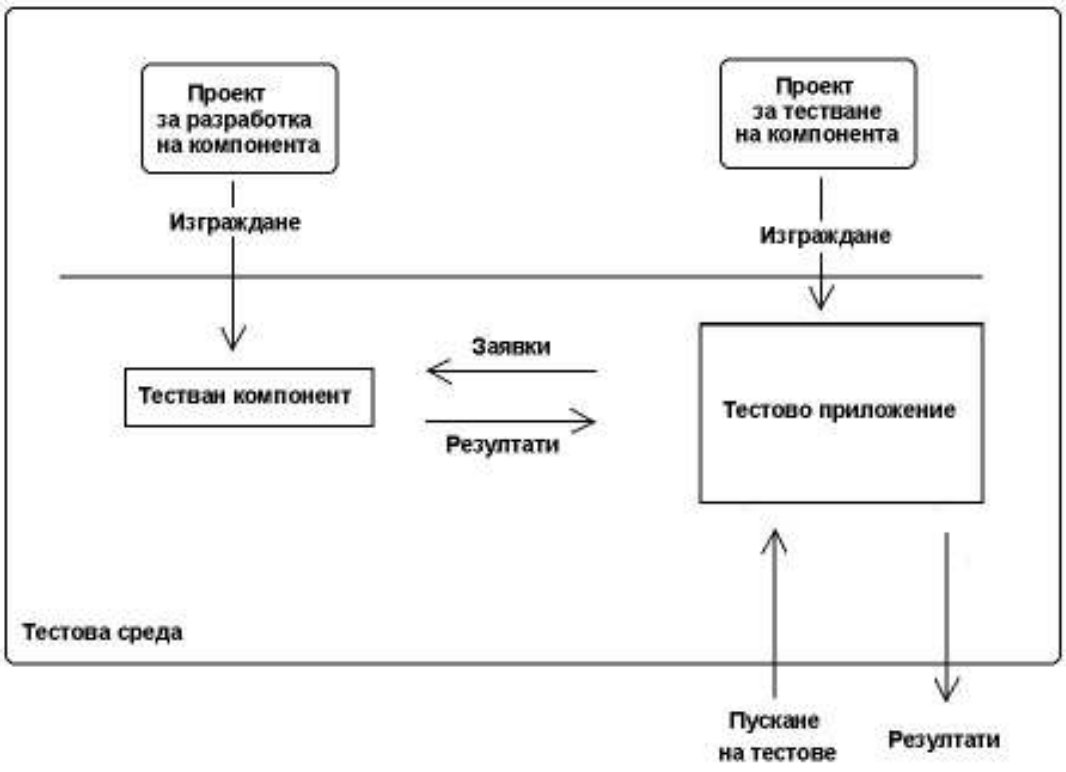
```
class ExampleTestCase : public CppUnit::TestCase
{
  // Регистрация на тестов клас с име ExampleTestCase
  CU_TEST_SUITE( ExampleTestCase );

  // Регистрация на тест с име "example"
  CU_TEST( example );
```

```
  // Макрос за край на тестовете
   CU_TEST_SUITE_END();

public:

  // Функция за инициализиращи дейности
  void setUp();

  // Функция за завършващи дейности
  void tearDown();

protected:

  // тест с име "example"
  void example();
};
```

Фигура 17 : Примерна декларация на тестов клас

Приложението HostApp анализира регистрациите на тестови класове и извиква последователно всички регистрирани тестове. Резултатите от тестовете се появяват в графичен прозорец, като потребителят има възможност да избира кои тестове да пусне. Това е голямо предимство при прилагане на практиката "Разработване, базирано на тестове", тъй-като позволява да бъдат тествани само функционалностите, които се разработват в момента. Друго предимство на CPP Unit е, че за всеки тест, който не е преминал успешно, показва кое условие в теста не е било удовлетворено.

Разработваният компонент (DeviceConfigurator) и тестовото приложение (HostApp) са отделни проекти, разположени в една рамка (workspace) на Visual C++ 6.0. Така итеративното добавяне на тестове и съответната им функционалност, което е характерно за прилаганата практика е силно улеснено. Схема на тестовата среда е показана на Фигура 18:



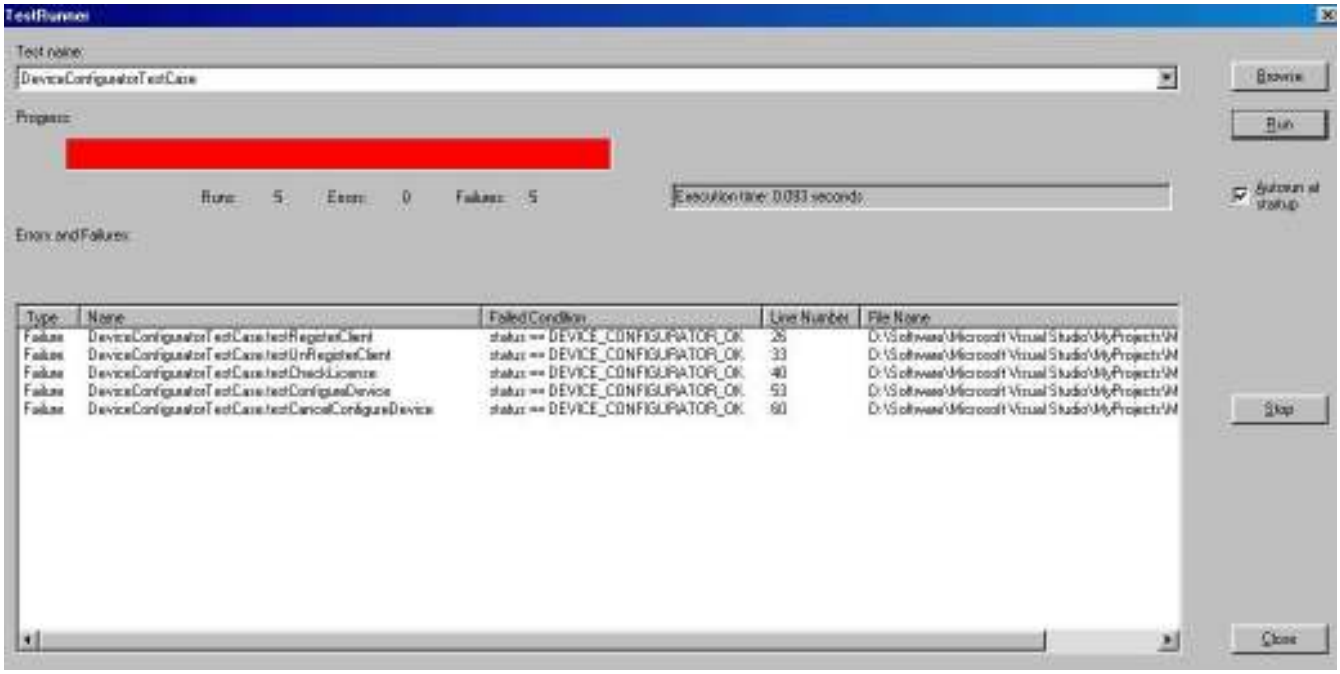
Фигура 18 : Тестова среда

4.6 Провеждане на експеримента

Проведохме експеримента, както е показано на Фигура 10, като взехме предвид архитектурните изисквания от секция 4.4. Създаваме проект (DeviceConfigurator) за конфигуратора на устройства и настройваме CPP Unit да ползва за тестове резултатната библиотека – DeviceConfigurator.dll. Така, след като веднъж е написан тест за някаква функция, можем да я имплементираме и тестваме много пъти, докато имплементацията удовлетвори теста.

Стартираме на базата на първата отговорност от архитектурата, написвайки тест за функцията DC_RegisterClient(). Той съдържа обръщение към несъществуващата функция, която тества и тестът не се компилира. Продължаваме, като написваме празна имплементация на функцията DC_RegisterClient(). В резултат, тестът се компилира, но не минава успешно. Повтаряме тези стъпки за

всички отговорности от архитектурата. На Фигура 19 е показано състоянието на CPP Unit в този момент:



Фигура 19 : CPP Unit в състояние на неуспешен тест

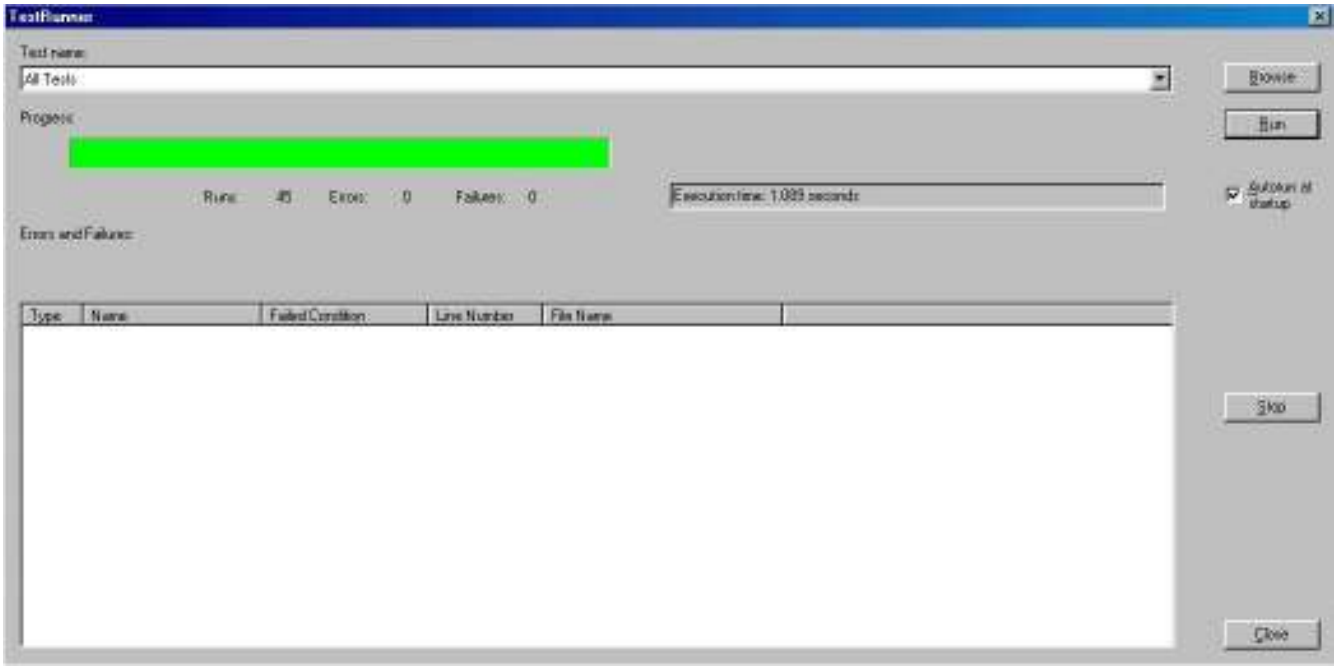
Тук е моментът да започне имплементацията на всяка от функциите. Придържаме се към правилото, тестовете да предхождат кода, т. е. винаги когато ни трябва функция, първо написваме тест за нея и после я имплементираме. Всеки тест трябва да тества точно един тестов сценарий и да е независим от останалите тестове.

Опитвайки се да имплементираме функциите от интерфейсния слой, имаме нужда от класове и функции, на които те да делегират отговорности. Следвайки практиката, която сме избрали, първо определяме техните отговорности, после написваме тестове за тях и накрая се опитваме да ги имплементираме. Ако има нужда от нови класове, процесът се повтаря и спира, когато нямаме нужда от нови функции и успеем да преминем последния написан тест успешно. Прави впечатление, че получаваме множество тестове, които не минават успешно и множество функции, които са написани частично. Когато спрем с добавянето на тестове и функции, имплементираме функциите в ред, обратен на тяхното създаване. Тогава и тестовете започват да се удовлетворяват в същия ред.

Редът на появяване на класовете и функциите показва, че този подход за дизайн е от вида “отгоре-надолу” (top-down), а удовлетворяването на тестовете става в обратна посока (bottom-up).

Удобно е да изберем от менюто на CPP Unit опцията “All tests”. Тогава на всяко тестване ще се изпълняват всички тестове и максимално бързо ще получим сигнал ако се получи регресия – имплементацията на текущ компонент е повредила имплементацията на друг.

На Фигура 20 е показано състоянието на CPP Unit след имплементацията на всички функции от модула “Конфигуратор на устройства”:



Фигура 20 : CPP Unit след успешно тестване

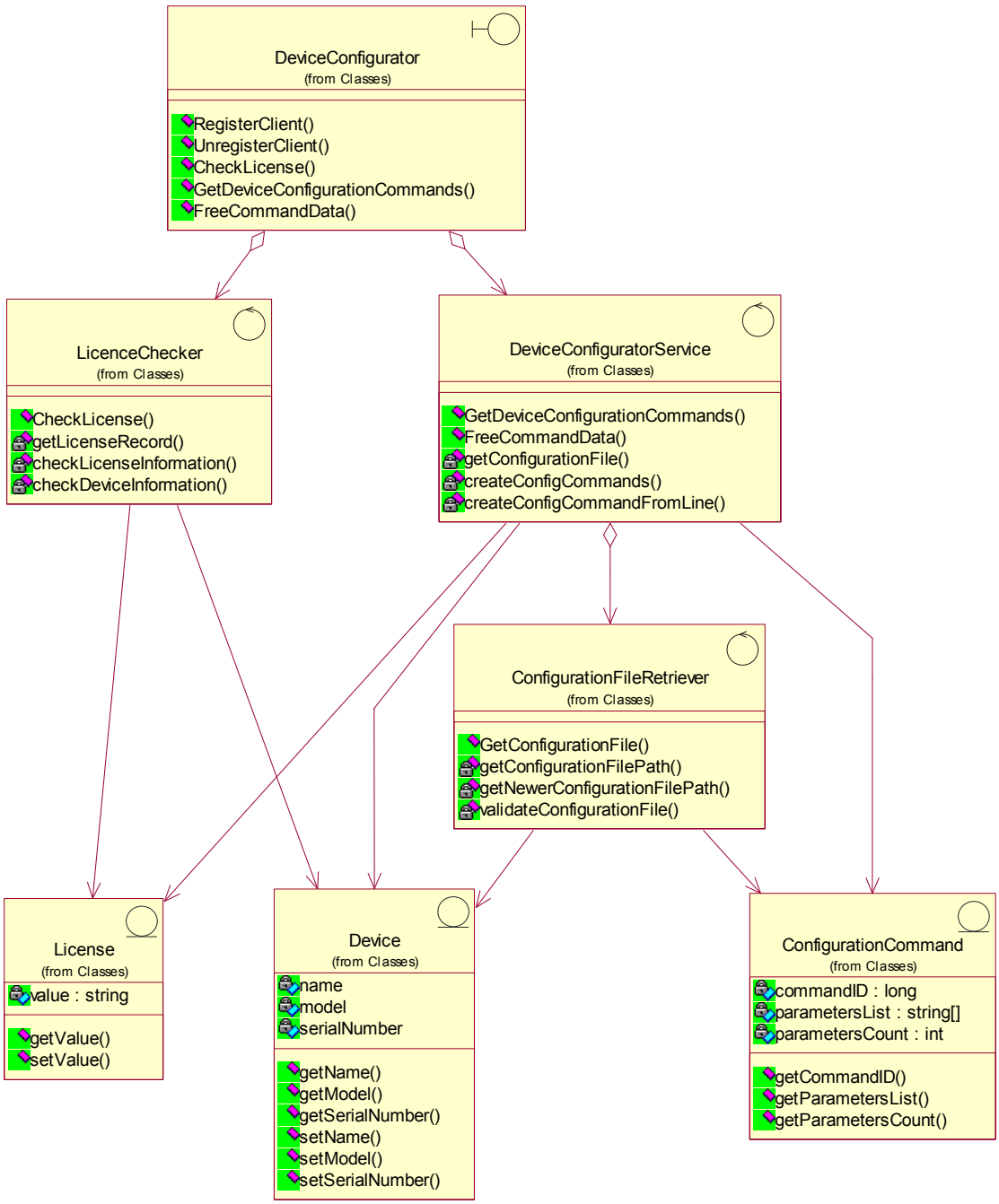
5 Изследване на резултатите от внедряването на решението

В тази глава са представени резултатите от внедряването на практиката “Разработване, базирано на тестове” в софтуерния процес. Представени са метрики и критерии за оценка на получените резултати. Направен е анализ на събраните данни и са дадени препоръки за прилагане на изследваната практика.

5.1 Резултати

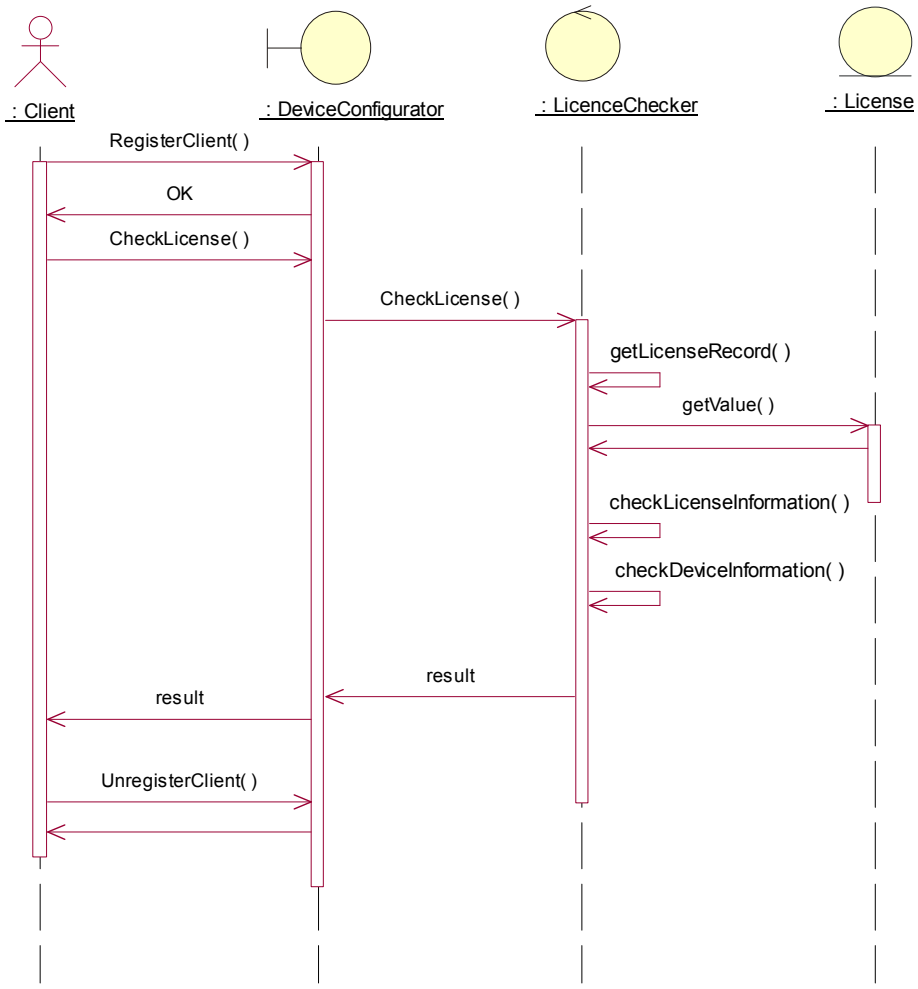
5.1.1 Дизайн при прилагането на базовия процес

Следвайки трислоен модел на архитектура и съгласно базовия процес за разработка на софтуер, беше изготвен дизайн със следната диаграма на класовете:

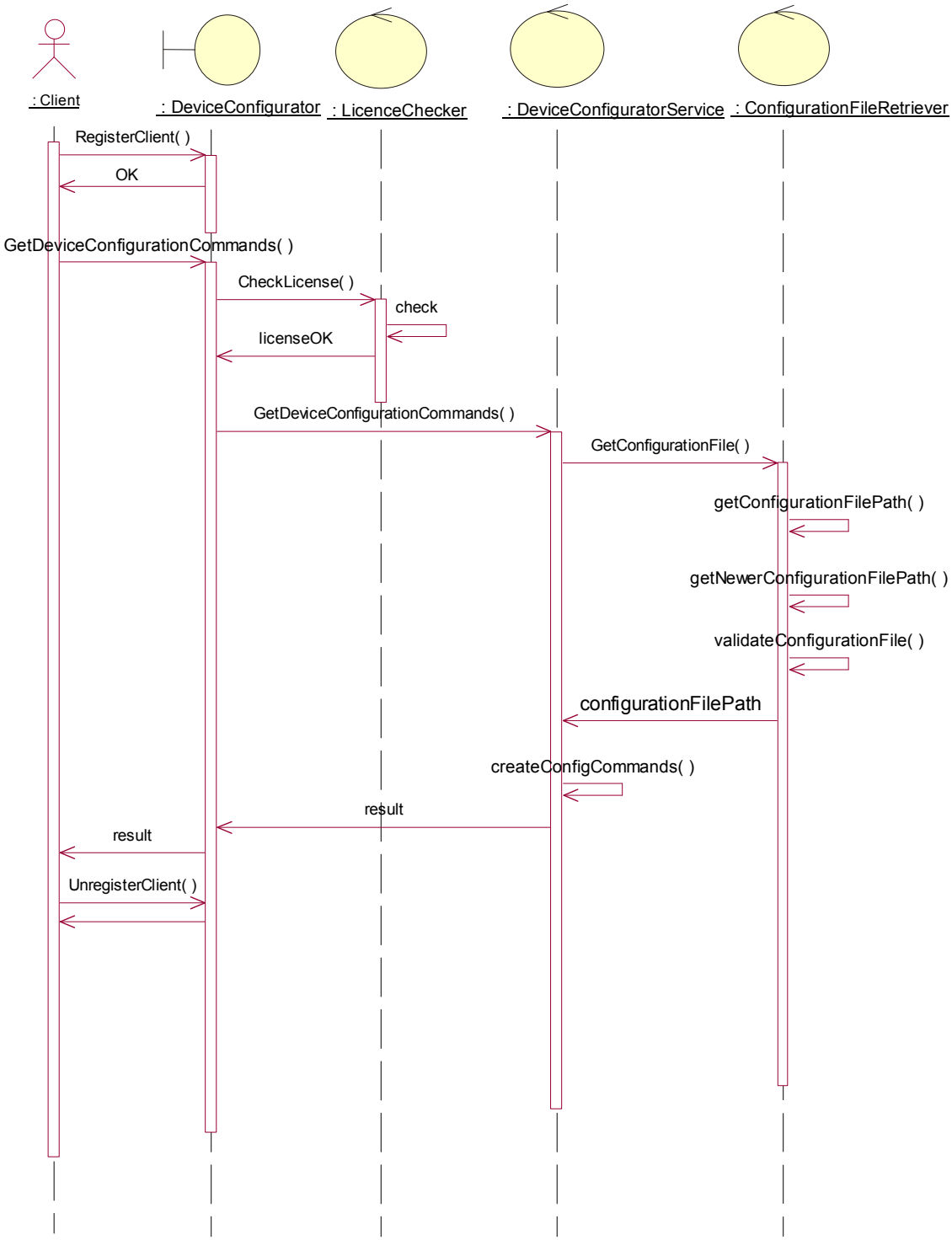


Фигура 21 : Диаграма на класовете в дизайна, получен според базовия процес

На Фигура 22 и Фигура 23 е показан дизайна, получен по базовия процес, на основните операции – проверка на лицензa на клиент и генериране на конфигурационни команди:



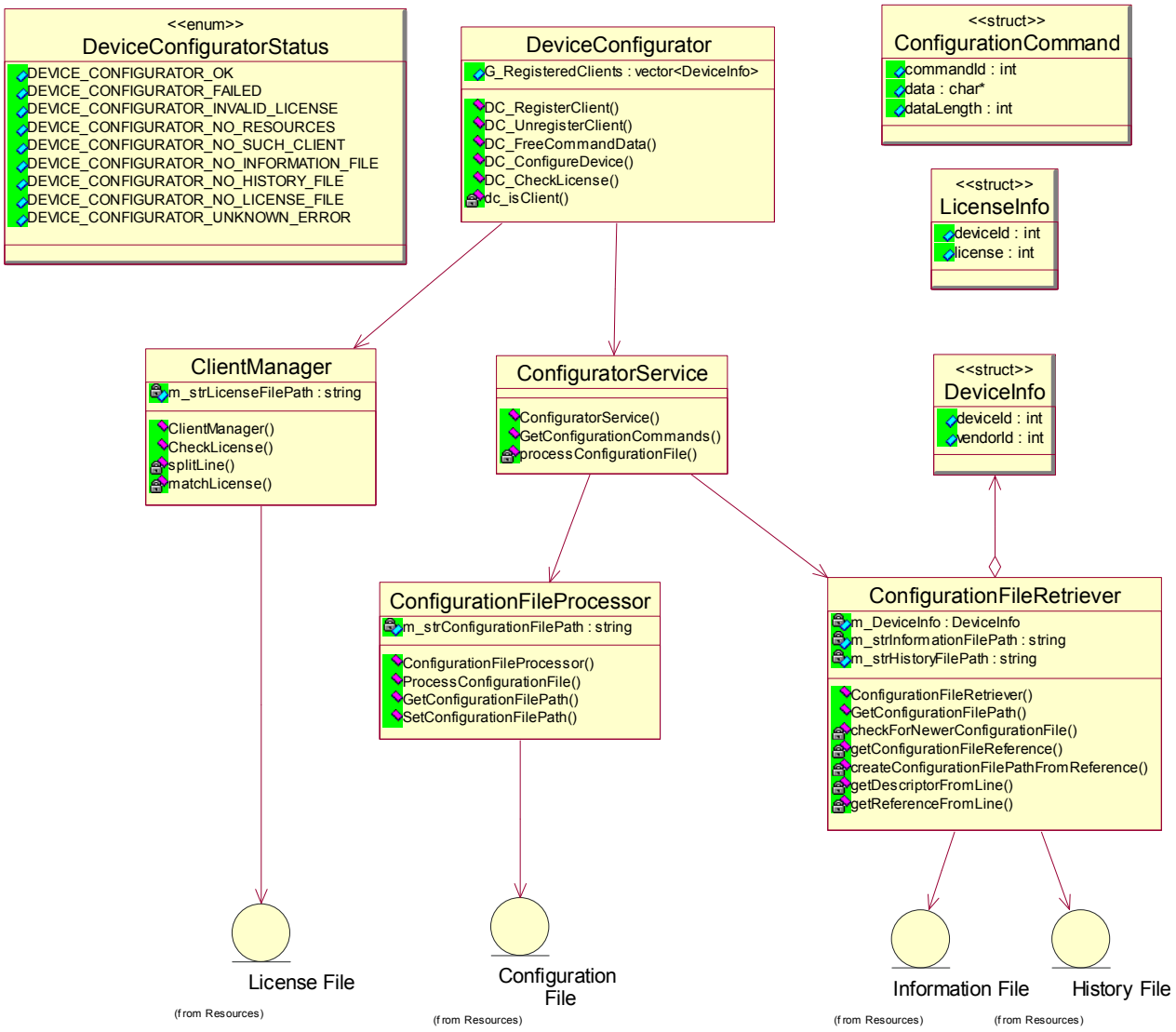
Фигура 22 : Проверка на лицензa на клиент – базов процес



Фигура 23 : Генериране на конфигурационни команди – базов процес

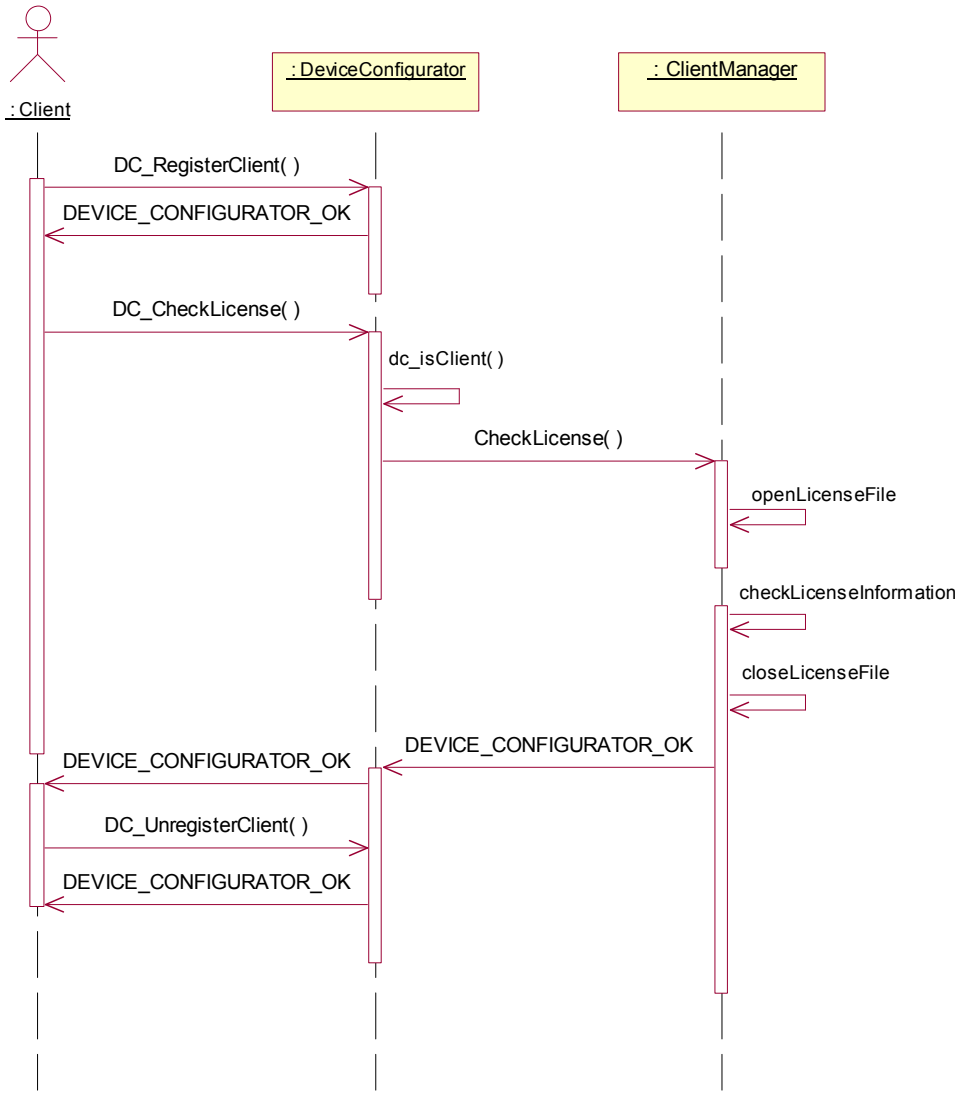
5.1.2 Дизайн при прилагането на модифициран процес

Прилагайки практиката “Разработване на базата на тестове”, беше изготвен дизайн със следната диаграма на класовете:

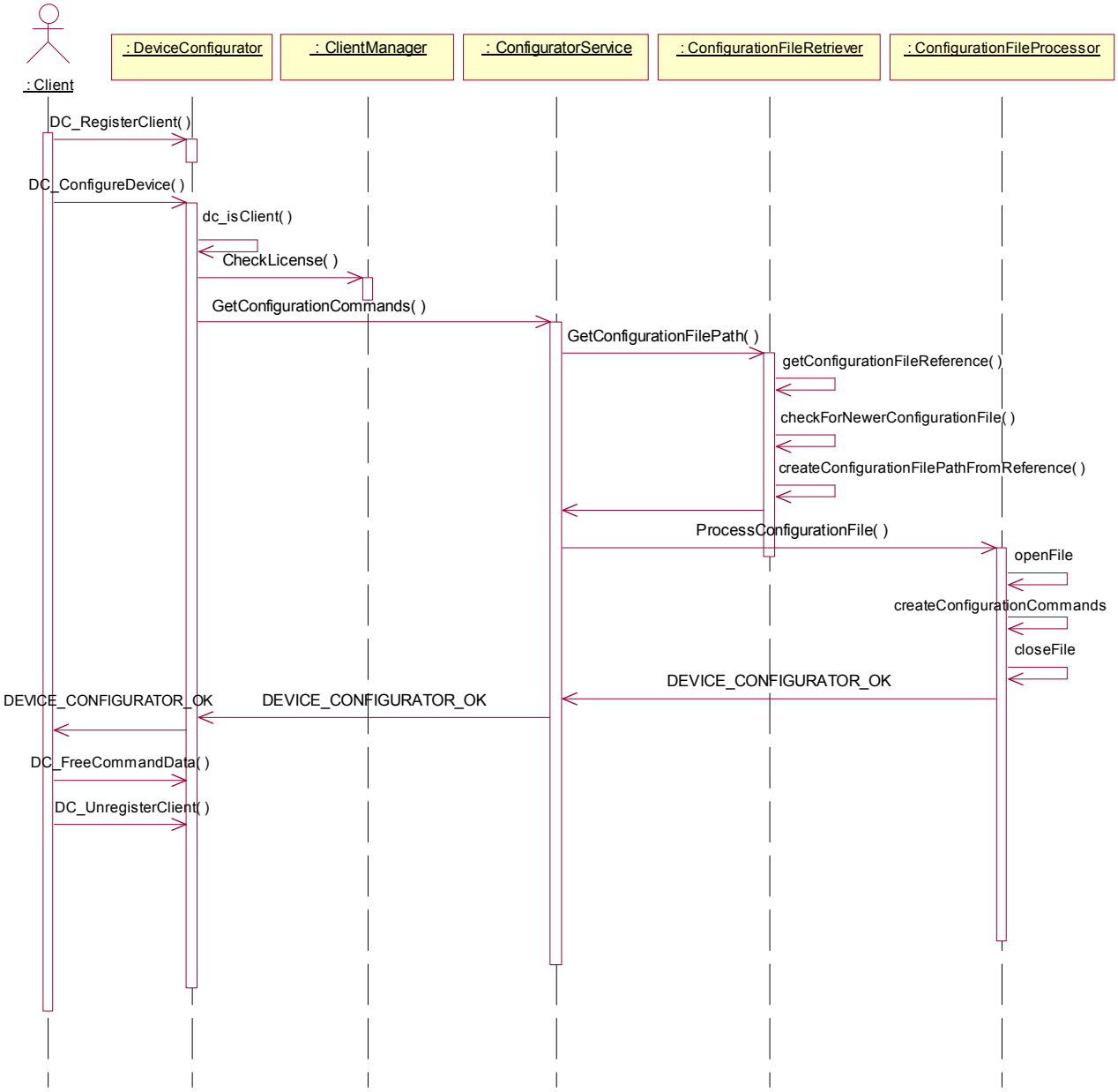


Фигура 24 : Диаграма на класовете в дизайна, получен след модификация на базовия процес с практиката “Разработване на базата на тестове”

На Фигура 25 и Фигура 26 е показана реализацията на основните операции на компонента, прилагайки модифицирания процес:



Фигура 25 : Проверка на лицензa на клиент – разработка на базата на тестове



Фигура 26 : Генериране на конфигурационни команди – разработка на базата на тестове

В дизайна, получен според модифицирания процес, класовете описващи конфигурационна команда, лиценз и устройство са заменени от структури и съответно функциите за достъп са отпаднали. Появил се е изброен тип (enum), който описва възможните грешки. Отразени са файловете, необходими на модула и

връзките им с класовете от дизайна. Слоя на управляващите класове (service layer) е ясно разделен на две части – операции с данните и логика на приложението (бизнес логика). В дизайна, получен според базовия процес, има връзки от всички управляващи класове към класовете-същности (entity class), докато използвайки модифицирания процес, имаме ясно отделен клас за логиката на приложението (ConfiguratorService), който управлява комуникацията и делегира отговорностите на останалите управляващи класове.

5.2 Софтуерни метрики

За да изследваме резултатите от проведения експеримент, които са представени в секция [5.1], трябва да ги оценим. Нуждата от обективност и коректност на оценката, налага дефинирането на методите за оценяване и тълкуване на съответните резултати. Процесът на събиране, анализ и оценка на данни, свързани с процеса или продукта се нарича измерване. Процесът на измерване на софтуера дефинира:

- Предмета на измерване – това е продукта или процеса, за който се събират данни;
- Участници в проекта, които извършват измерването, анализа и оценката на събраните данни;
- Етап от проекта, на който ще се извършва измерването – време на измерването, според жезнения цикъл на проекта;
- Начина на измерване – методите, които ще се приложат, за да бъдат събрани, анализирани и оценени данните, на които се базира измерването.

На базата на тези общи характеристики, процесът на измерване на софтуера дефинира критерии за сравнение на предметите на измерването – метрики. След

анализ и оценка на резултатите, се установява текущото състояние на процеса (продукта) и се набелязват действия за неговото усъвършенстване.

5.2.1 Сцепление (cohesion)

Сцеплението (cohesion) е мярка за дизайна на софтуера, която показва доколко отделните компоненти изпълняват целите, за които са създадени и комуникират добре. При добрия дизайн, сцеплението между отделните компоненти е голямо, те добре взаимодействат и изпълняват функциите си. Високото ниво на сцепление означава:

- подходящ интерфейс;
- добра капсулация;
- простота – компонентът отговаря за една логически обособена част от изискванията и функционалността;
- минималност по отношение на отговорностите – отговорностите на компонента са точно тези, които са определени в архитектурата (не са иззети част от отговорностите на други компоненти);
- минималност по отношение на функционалността – предвидената функционалност на компонента е минималната възможна, за да изпълни неговите отговорности. Дизайнът не съдържа елементи, които са излишни.

Съществуват различни начини за измерване на сцеплението. Ние ще използваме предложените от Хендерсън и Грахам [10] метрики за отсъствие на сцеплението – LCOM (Lack of Cohesion of Methods). Те съществуват в няколко разновидности и имат за цел да идентифицират проблемни класове. Високи стойности на LCOM означават ниско ниво на сцепление. В следващите абзаци са представени използваните метрики - LCOM2 и LCOM3.

Нека имаме клас със следните характеристики:

- m – брой методи;
- a – брой атрибути;
- sum(mA) – брой методи, които достъпват някой от атрибутите.

Тогава:

LCOM2 е разновидност на LCOM, която се пресмята чрез израза:

```
LCOM2 = 1 - sum(mA)/(m*a)
```

Формула 1 : Липса на сцепление, вид 2

Ако a = 0 или m = 0, LCOM2 се счита за 0.

LCOM3 е разновидност на LCOM, която се пресмята чрез израза:

```
LCOM3 = (m - sum(mA)/a) / (m-1)
```

Формула 2: Липса на сцепление, вид 3

Ако m = 1, LCOM3 се счита за 0.

Измерванията се извършват след приключване на дейностите, свързани с моделирането, от софтуерни инженери или специализирани софтуерни инструменти, а събраните данни се анализират от софтуерни архитекти.

5.2.2 Свързаност (coupling)

Свързаността (coupling) е мярка за дизайна на софтуера, която показва доколко отделните компоненти са свързани помежду си. Наличието на връзка между два компонента води до зависимост, което е от съществено значение при поддръжката и използването на всеки от компонентите. Под връзка се разбира всякакъв вид зависимост, вследствие на която, промяната на единия от компонентите може да предизвика промяна в другия. При добрия дизайн, свързаността между отделните компоненти е минимална, всеки компонент е максимално независим от останалите. Ниското ниво на свързаност означава:

- добра капсулация – спецификата за компонента е скрита за останалите компоненти и ако се налага е достъпна през дефиниран интерфейс;

- високо ниво на абстракция – представянето на данните и реализацията на операциите са разделени от използването им. Така промени в реализацията на компонента не водят до промени в използването му;

- възможност за повторно използване – слабата свързаност означава по–малка зависимост от останалите компоненти, което позволява компонентите да се използват повторно, като им се осигури подходящата околна среда;

- лесна разширяемост – от ниското ниво на зависимост следва, че ограниченията към компонента са малки и той лесно може да бъде разширен;

- ниски разходи за разработка – увеличените възможности за повторно използване и лесната разширяемост намаляват значително разходите за разработка;

- ниски разходи за поддръжка – капсулацията и високото ниво на абстракция намаляват разходите за поддръжка;

За измерване на свързаността ще използваме използваме “фактор на свързаност” (coupling factor – CF) [11]. Фактор на свързаност е отношението на броя свързани двойки класове към броя на възможните свързвания. При броенето се отчита посоката на връзката между класовете. Така за множество от n класа, броят на възможните свързвания е n*(n – 1). Нека за дадено множество класове имаме:

- AC – брой свързвания;
- MC – максимален брой свързвания.

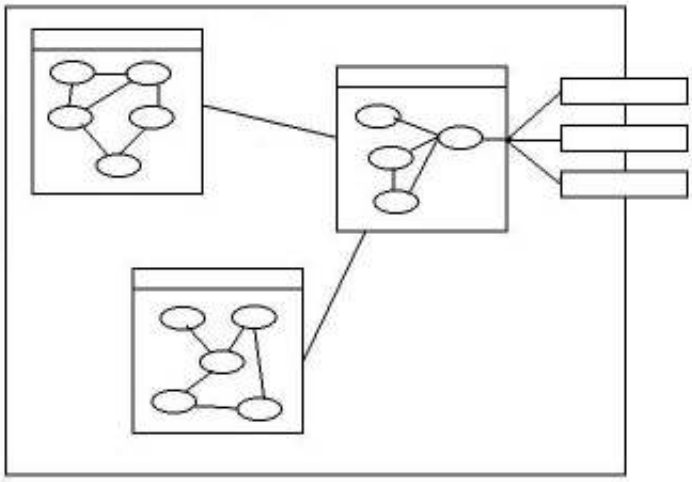
Тогава:

```
CF = AC / MC
```

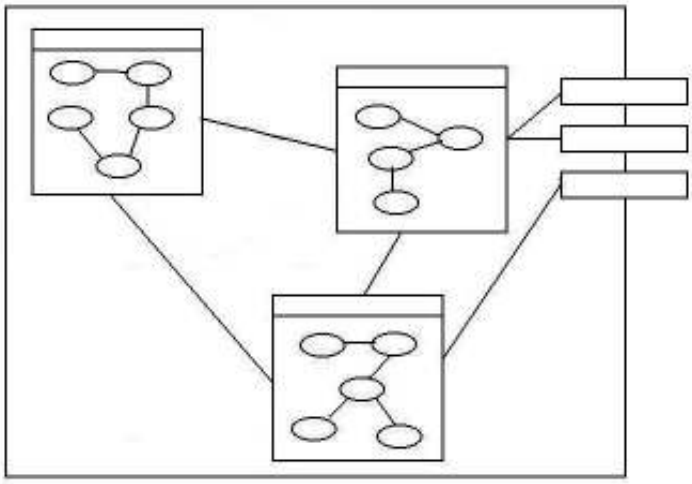
Формула 3 : Фактор на свързаност

Останалите характеристики на метриката – време на измерване и участници в проекта са еднакви с тези при сцеплението.

На Фигура 27 и Фигура 28 са показани архитектури на системи съответно с добри и лоши показатели на сцепление и свързаност.



Фигура 27 : Силно сцепление и слаба свързаност



Фигура 28 : Слабо сцепление и силна свързаност

5.3 Анализ на резултатите

Ще използваме дефинираните в секция [5.2] метрики, за да анализираме и сравним получените в секция [5.1] дизайни.

За дизайна, получен по базовия процес имаме следните характеристики:

- брой класове – 7;
- брой свързвания – 10;
- максимален брой свързвания – 42;
- брой методи – 29;
- брой атрибути – 7;
- брой функции за достъп – 11.

Съгласно Формула 3 за фактора на свързаност имаме:

```
CF = 10 / 42 = 0.24
```

Съгласно Формула 1 за липса на сцепление, вид 2, имаме:

```
LCOM2 = 1 - 11/203 = 0.95
```

Съгласно Формула 2 за липса на сцепление, вид 3, имаме:

```
LCOM3 = (29 - 11/7) / 28 = 0.98
```

За дизайна, получен по процеса, модифициран с практиката “Разработване, базирано на тестове”, имаме следните характеристики:

- брой класове – 5;
- брой свързвания – 4;
- максимален брой свързвания – 20;
- брой методи – 24;

- брой атрибути – 6;
- брой функции за достъп – 11.

Съгласно Формула 3 за фактора на свързаност имаме:

$$CF = 4 / 20 = 0.20$$

Съгласно Формула 1 за липса на сцепление, вид 2, имаме:

$$LCOM2 = 1 - 11/144 = 0.92$$

Съгласно Формула 2 за липса на сцепление, вид 3, имаме:

$$LCOM3 = (24 - 11/6) / 23 = 0.96$$

Обобщени резултати от експеримента са показани в Таблица 4:

процес / метрика	Базов процес	Модифициран процес
CF	0.24	0.20
LCOM2	0.95	0.92
LCOM3	0.98	0.96

Таблица 4 : Резултати от експеримента

Факторът на свързаност при модифицирания процес е по–нисък, което означава по–малка зависимост между класовете. Основна причина за това е, че изследваната практика създава предпоставки за минимизация на връзките между компонентите. Получените връзки са наистина необходими и се използват. Като следствие, резултатния дизайн (респ. продукт) е по–прост, по–разбираем, по–лесно разширяем, с по–малко потенциални дефекти, по–лесен за имплементация и поддръжка.

Липсата на сцепление при модифицирания процес е по–малка, което означава по–добро сцепление, по–добра комуникация между класовете. Причина за това е минималността на подхода, по отношение на елементите на дизайна – добавянето на нов елемент на дизайна (клас, функция), става само когато е необходимо. Това допринася за липсата на излишество в дизайна, което го олекотява и повишава качеството му. Слоя на управляващите класове е ясно разделен на две части – операции с данните и логика на приложението, което минимизира връзките и увеличава сцеплението.

В дизайна, получен след внедряване на практиката "Разработване, базирано на тестове", вместо класове същности (entity class) са използвани структури, тъй – като те са достатъчни, за да изпълнят предназначението си. Същевременно, в дизайна, получен по базовия процес липсва дефиниция на възможните грешки и тяхната интерпретация. В другия дизайн, този елемент не е пропуснат, тъй – като е невъзможно да се напишат тестове, без да се следи за резултатите на тестваните компоненти. Включени са файловете, необходими на модула и връзките им с класовете. Следвайки модифицирания процес, след една или няколко итерации, резултатния дизайн покрива отговорностите на съответния модул и това е критерия за спиране на моделирането. Полученият дизайн е минималният, но достатъчен, които реализира отговорностите на съответния модул. Удовлетворяването на всички тестове е гаранция, че всичко необходимо е включено в дизайна.

От друга страна, итеративното надграждане на малки блокове реализирана функционалност има недостатъци. Възможно е пренебрегване на глобалната визия за системата и съществува опасност да се стигне до момент, в който разширението на текущия дизайн е трудно или невъзможно. В съвременните проекти, много често е необходимо разширяване на съществуващи софтуерни системи. От тази гледна точка, минималността може да се окаже проблем, ако разширяването е наложително, а дизайна на системата е трудно разширяем, поради стремежа към простота и минималност.

Трябва да отбележим, че по време на прилагането на изследваната практика, няколко пъти се случваше, съществуващи тестове, които са преминавали успешно по време на предишни итерации, да преминават неуспешно при реализацията на дизайна от следваща итерация. Това състояние на регресия, веднага беше констатирано от използвания инструмент за автоматизация на тестването и

проблемите бяха отстранени в начален стадий. За бързата им локализация спомогна принципа за простота на тестовете, който е залегнал в основите на прилаганата практика.

Начинът на използване на отделните компоненти беше основна движеща сила във фазата на дизайн. Това благоприятства разглеждането на компонентите от всички страни, а не само от ъгъла на номиналните случаи (nominal cases). По–лесно се стига до случаи на употреба, в които има предпоставки за възникване на грешки (error cases), което води до по–голяма пълнота на дизайна. Разглеждането на подобни случаи във фазата на дизайн е изключително предимство, като имаме предвид изменението на цената на промените във времето (вж. Фигура 12).

Отделянето на повече време за написване на тестовете в началото е инвестиция в качеството, която след това се компенсира от по–малко дефекти в продукта. Освен това, полученият дизайн изцяло съответства на програмния код и автоматично отпада нуждата от корекции в дизайна след имплементацията (reverse engineering).

5.4 Предложения и препоръки

Резултатите от прилагането на практиката “Разработване, базирано на тестове” зависят от процеса, в който е внедрена и от проекта, по който се работи.

Практиката е подходяща за внедряване в процеси, които дават известна свобода на дейностите и възможност за гъвкавост. Липсата на формалности в процеса улеснява прилагането й, като дава възможност за свобода на разработчика и акцентира върху актуалните проблеми и тяхното разрешаване. Колкото “по–незабележим” е процеса, толкова по–лесно и безболезнено би се внедрила практиката.

Съществено значение за резултатите от прилагането на практиката има и проекта, в който се прилага. Екстремното програмиране насърчава индивидуалността и творчеството. Естествено възниква необходимостта от координация и синхронизация на работата, което е важно и при изследваната практика. Тя е подходяща при проекти с малко персонал, при които синхронизацията става лесно и няма нужда от стриктно следене на процеса.

Важни са архитектурните особености на проекта и тенденциите за развитието му. Многото на брой малки итерации, правят практиката подходяща за краткосрочни и средносрочни проекти с повече динамика. Трудноприложима е при големи проекти, които разширяват обхвата си във времето и разчитат на по–сигурни процеси, с повече формалности и изискващи повече ресурси.

Прилагането на изследваната практика е немислимо без софтуерни инструменти за автоматизация на тестването. Наличието на такива е предпоставка за правилното й прилагане и за качеството на резултатния продукт.

6 Заключение

Настоящата дипломна работа описва внедряването на гъвкавата методология „Разработване на базата на тестове” в съществуващ софтуерен процес и изследва влиянието й върху дизайна на софтуерна система. Бяха разгледани по–важните основни процеси за разработка на софтуерни системи и техни модификации с гъвкави методологии. Всеки процес бе представен с модел и характеристика, след което бяха анализирани и представени неговите предимства и недостатъци. Цел на изложението е, читателят да добие представа за проблемната област и по–лесно да се ориентира в палитрата от предлагани процеси и методологии. Обърнато е внимание на често срещани проблеми и предизвикателства при реализацията на софтуерни проекти в динамичните съвременни условия.

Практическата част описва същността на експеримента и провеждането му. Застъпено е представяне на използвания софтуер за провеждане на модулни тестове и е дефинирана средата за разработка, в съответствие с изследваната практика. Дадени са практически стъпки и препоръки за провеждане на експеримента. Тази част цели нагледно да представи прилагането на практиката и да покаже на читателя детайлите, които заедно с резултатите, да послужат за база при оценката на влиянието й върху дизайна на резултатния продукт.

В последната част бяха представени и впоследствие сравнени, резултатите от проведения експеримент. Те послужиха като отправна точка при последвалите анализи и препоръки. Потвърдиха се очакванията ни за по–добро сцепление и по–малка свързаност на дизайна, получен с процес, модифициран с изследваната практика. Този дизайн е минималният, но достатъчен, който реализира отговорностите на разработвания модул.

Поради динамичната природа на проектите, динамичност, гъвкавост и адаптируемост, са качества на процеса за разработка на софтеур, без които той е обречен на неуспех. Практиката „Разработване, базирано на тестове”, като част от гъвкавите методологии, спомага за подобряване на процеса за разработка на софтуер. Многото на брой малки итерации го правят по–адекватен на реалността и по–приспособим към бързо променящите се условия.

Актуалността на екстремното програмиране създава предпоставки за продължаване на изследванията на влиянието на различните практики върху процеса за разработване на софтуер. Подходящо би било да се разшири обхвата на изследваните практики, като бъдат групирани според проблемната област на проекта, в който се прилагат. Резултатите от такива изследвания биха подпомогнали софтуерните специалисти при избора им на процес и група практики за дадена проблемна област. На по–късен етап е възможно те да се обособят в специализирани процеси.

7 Литература

[1] Илиева, С., Вл. Лилов, Ил. Манова. Изграждане на софтуерни приложения. София, Университетско издателство “Св. Климент Охридски”, 2006.

[2] Pressman, R. Software Engineering: A practitioner’s approach, McGrawHill, 2005.

[3] Sommerville, I. Software Engineering, Addison-Wisley, 2006.

[4] Chaplin, D., Test First Programming, TechZone, 2001.

[5] Grover, J., Extreme programming enabling chart, http://www.kiva.net/, 2002.

[6] http://www.agiledata.org/essays/tdd.html, 2007.

[7] Boehm, B., Improving Application Quality Using Test–Driven Development, http://www.methodsandtools.com/archive.php?id=20, 2007

[8] http://c2.com/cgi/wiki?ExtremeProgramming, 2007.

[9] Ambler, S., The Full Life Cycle Object–Oriented Testing Method, http://www.ambysoft.com/essays/floot.html, 2007.

[10] Henderson-Sellers, B., L. Constantine and I. Graham, Cohesion metrics, http://www.aivosto.com/project/help/pm-oo-cohesion.html, 2007

[11] MOOD and MOOD2 metrics, http://www.aivosto.com/project/help/pm-oo-mood.html, 2007

[12] http://www.agilemodeling.com/essays/costOfChange.htm

[13] Janzen, D., Software Architecture Improvement through Test-Driven Development, http://www.acm.org/src/subpages/gf_entries_06/DavidJanzen_src_gf06.pdf

[14] George, B., An initial Investigation of Test-Driven Development in Industry, http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

8 Използвани термини

- контролна точка (milestone) – момент от времето, когато се проверява статуса на проекта. Очаква се той да отговаря на предварителните разчети;

- работен продукт (work product) – резултат от дейност, която е част от процеса – документи, модели, код, тестови отчети и др;

- ООП – обектно–ориентирано програмиране;

- TDD (test-driven development) – разработване, базирано на тестове;

- свързаност (coupling) на класове – в ООП, зависимост между класове, при която промяната в един от тях води до промяна в останалите;

- сцепление(cohesion) на класове – в ООП, отношение между класове, при което всички заедно реализират определена функционалност и са независими от останалите класове;

- среда за разработка – софтуерен продукт, който предоставя възможности за разработка на софтуер на определен език или платформа. Включва редактор, компилатор и др.;

- тестовата среда – софтуерен продукт, който предоставя възможности за разработка и автоматизация на тестове на софтеурни компоненти на определен език или платформа;

- регресия (regerssion) – състояние на софтуерна система, при което един или повече нейни компонента започват да работят некоректно, в резултат на промени в други компоненти от същата система;

- тест за регресия (regression test) – тест, които има за цел да провери, че тестваната софтуерна система не е в състояние на регресия;

- тестов сценарий – съвкупност от предпоставки и действия за провеждане на един тест. Включва дефиниране на очаквания резултат;

- номинален случай (nominal case) – тестов сценарий, при който изпълнението на дадена програмна единица (функция), постига целите, за които е предназначена;

- случай на грешка (error case) – тестов сценарий, при който изпълнението на дадена програмна единица (функция), не постига целите, за които е предназначена, поради грешка заложена умишлено в сценария. Има за цел да провери поведението на тестваната програмна единица в нетипична за нея ситуация.

9 Приложение

В приложение предоставяме част от програмния код, получен при провеждане на експеримента. Представен е кода на тестовия клас на интерфейса на компонента, а след това кода на интерфейса на компонента. Пълният програмен код и дизайн може да бъде намерен на приложения компакт диск. Имената на файловете отговарат на имената на дефинираните в тях класове.

9.1 Декларация на тестовия клас на интерфейса на компонента

```
#ifndef CPP_UNIT_DEVICECONFIGURATORTESTCASE_H
#define CPP_UNIT_DEVICECONFIGURATORTESTCASE_H

#include <cppunit/TestCase.h>
#include <cppunit/extensions/HelperMacros.h>

// tested object
#include "DeviceConfigurator.h"

/*
This is a tester class for DeviceConfigurator.
*/
class DeviceConfiguratorTestCase : public CppUnit::TestCase
{
     // Начало на декларация на тестово множество
     CU_TEST_SUITE( DeviceConfiguratorTestCase );

     // Регистрация на тестове в тестовото множество
     CU_TEST( testRegisterClient );

     CU_TEST( testUnregisterClient );
     CU_TEST( testUnregisterClient_fail_noSuchClient );

     CU_TEST( testCheckLicense );
     CU_TEST( testCheckLicense_fail_noSuchClient );
     CU_TEST( testCheckLicense_fail_invalidLicense );

     CU_TEST( testConfigureDevice );
     CU_TEST( testConfigureDevice_noSuchClient );
     CU_TEST( testConfigureDevice_invalidLicense );

     CU_TEST( testFreeCommandData );
     CU_TEST( testFreeCommandData_fail );

     // Край на декларация на тестово множество
     CU_TEST_SUITE_END();
```

```
public:

     // Декларация на функциите за подготовка и завършване
     // на всеки тест
     void setUp();
     void tearDown();

protected:

     // Декларация на тестовите функции
     // Всяка реализира един тестов случай

     void testRegisterClient();

     void testUnregisterClient();
     void testUnregisterClient_fail_noSuchClient();

     void testCheckLicense();
     void testCheckLicense_fail_noSuchClient();
     void testCheckLicense_fail_invalidLicense();

     void testConfigureDevice();
     void testConfigureDevice_noSuchClient();
     void testConfigureDevice_invalidLicense();

     void testFreeCommandData();
     void testFreeCommandData_fail();

private:

     // Помощни данни и функции за реализиране на тестовете

     bool checkConfigurationCommands(
                    const ConfigurationCommand* commands,
                    const int commandsCount );

     DeviceInfo m_DeviceInfo;

     LicenseInfo m_LicenseInfo;
};


#endif
```

9.2 Имплементация на тестовия клас на интерфейса на компонента

```
#include "stdafx.h"
```

```
#include <cppunit/extensions/TestSuiteBuilder.h>
#include "DeviceConfiguratorTestCase.h"
#include "TestIncludes.h"


CU_TEST_SUITE_REGISTRATION( DeviceConfiguratorTestCase );


void DeviceConfiguratorTestCase::setUp()
{
     m_DeviceInfo.deviceId = DEVICE_ID_DEFAULT;
     m_DeviceInfo.vendorId = VENDOR_ID_DEFAULT;
     m_LicenseInfo.deviceId = DEVICE_ID_DEFAULT;
     m_LicenseInfo.license = LICENSE_DEFAULT;
}

void DeviceConfiguratorTestCase::tearDown()
{
}

void DeviceConfiguratorTestCase::testRegisterClient()
{
     DeviceConfiguratorStatus status =
               DEVICE_CONFIGURATOR_UNKNOWN_ERROR;
     status = DC_RegisterClient( m_DeviceInfo );

     // clean up without getting the status
     DC_UnregisterClient( m_DeviceInfo );

     // Проверка на условие. Ако е лог. истина се приема,
     // че тестът преминава успешно.
     assert( status == DEVICE_CONFIGURATOR_OK );
}

void DeviceConfiguratorTestCase::testUnregisterClient()
{
     DeviceConfiguratorStatus status =
               DEVICE_CONFIGURATOR_UNKNOWN_ERROR;
     status = DC_RegisterClient( m_DeviceInfo );
     assert( status == DEVICE_CONFIGURATOR_OK );

     status = DC_UnregisterClient( m_DeviceInfo );
     assert( status == DEVICE_CONFIGURATOR_OK );
}

void
DeviceConfiguratorTestCase::testUnregisterClient_fail_noSuc
hClient()
{
```

```
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

    status = DC_UnregisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_NO_SUCH_CLIENT
);
}

void DeviceConfiguratorTestCase::testCheckLicense()
{
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;
    status = DC_RegisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );

    status = DC_CheckLicense(
                  m_DeviceInfo, m_LicenseInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );

    status = DC_UnregisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );
}

void
DeviceConfiguratorTestCase::testCheckLicense_fail_noSuchCli
ent()
{
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

    // call without registering
    status = DC_CheckLicense(
                       m_DeviceInfo, m_LicenseInfo );
    assert( status
              == DEVICE_CONFIGURATOR_NO_SUCH_CLIENT );
}

void
DeviceConfiguratorTestCase::testCheckLicense_fail_invalidLi
cense()
{
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

    status = DC_RegisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );

    // set an invalid license
    m_LicenseInfo.license = INVALID_LICENSE_DEFAULT;
    status = DC_CheckLicense(
```

```
                        m_DeviceInfo, m_LicenseInfo );
    assert( status
              == DEVICE_CONFIGURATOR_INVALID_LICENSE );

    status = DC_UnregisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );
}

void DeviceConfiguratorTestCase::testConfigureDevice()
{
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

    ConfigurationCommand* commands = NULL;
    int commandsCount = 0;

    status = DC_RegisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );

    status = DC_ConfigureDevice( m_DeviceInfo,
              m_LicenseInfo, commands, commandsCount );
    assert( status == DEVICE_CONFIGURATOR_OK );
    assert( checkConfigurationCommands(
                        commands, commandsCount ) );

    status = DC_FreeCommandData(
                        commands, commandsCount );
    assert( status == DEVICE_CONFIGURATOR_OK );

    status = DC_UnregisterClient( m_DeviceInfo );
    assert( status == DEVICE_CONFIGURATOR_OK );
}

void
DeviceConfiguratorTestCase::testConfigureDevice_noSuchClien
t()
{
    DeviceConfiguratorStatus status =
              DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

    ConfigurationCommand* commands = NULL;
    int commandsCount = 0;

    status = DC_ConfigureDevice( m_DeviceInfo,
              m_LicenseInfo, commands, commandsCount );
    assert( status
              == DEVICE_CONFIGURATOR_NO_SUCH_CLIENT );
}
```

```
void
DeviceConfiguratorTestCase::testConfigureDevice_invalidLice
nse()
{
     DeviceConfiguratorStatus status =
               DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

     ConfigurationCommand* commands = NULL;
     int commandsCount = 0;

     status = DC_RegisterClient( m_DeviceInfo );
     assert( status == DEVICE_CONFIGURATOR_OK );

     // set an invalid license
     m_LicenseInfo.license = INVALID_LICENSE_DEFAULT;

     status = DC_ConfigureDevice( m_DeviceInfo,
               m_LicenseInfo, commands, commandsCount );

     assert( status
               == DEVICE_CONFIGURATOR_INVALID_LICENSE );

     // configuration commands must not be retrieved
     assert( !checkConfigurationCommands(
                              commands, commandsCount ) );

     status = DC_FreeCommandData(
                              commands, commandsCount );
     assert( status != DEVICE_CONFIGURATOR_OK );

     status = DC_UnregisterClient( m_DeviceInfo );
     assert( status == DEVICE_CONFIGURATOR_OK );
}

void DeviceConfiguratorTestCase::testFreeCommandData()
{
     DeviceConfiguratorStatus status =
               DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

     const int CONFIGURATION_COMMANDS_COUNT = 5;
     const int COMMAND_DATA_LENGTH = 50;
     const char* COMMAND_DATA = "TestData";

     ConfigurationCommand* configurationCommands =
                         new ConfigurationCommand[
                         CONFIGURATION_COMMANDS_COUNT];

     if ( configurationCommands == NULL )
     {
          // fail
```

```
            assert( 0 );
        }

        for (int i = 0; i < CONFIGURATION_COMMANDS_COUNT; i++)
        {
            ConfigurationCommand* configurationCommand =
                                    configurationCommands + i;
            if ( configurationCommand != NULL )
            {
                configurationCommand -> data = new
                                    char[COMMAND_DATA_LENGTH];
                if ( configurationCommand -> data != NULL )
                {
                    strcpy( configurationCommand -> data,
                                    COMMAND_DATA );
                }
            }
        }

        status = DC_FreeCommandData( configurationCommands,

        CONFIGURATION_COMMANDS_COUNT );

        assert( status == DEVICE_CONFIGURATOR_OK );
    }

    void DeviceConfiguratorTestCase::testFreeCommandData_fail()
    {
        DeviceConfiguratorStatus status =
                    DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

        status = DC_FreeCommandData( NULL, 0 );

        assert( status == DEVICE_CONFIGURATOR_FAILED );
    }

    bool
    DeviceConfiguratorTestCase::checkConfigurationCommands(
                    const ConfigurationCommand* commands,
                    const int commandsCount )
    {
        if ( commands == NULL || commandsCount <= 0 )
        {
            return false;
        }
        return true;
    }
```

9.3 Декларация на интерфейса на компонента

```
#ifndef DEVICECONFIGURATOR_H_INCLUDED
#define DEVICECONFIGURATOR_H_INCLUDED

#include "StdAfx.h"

enum DeviceConfiguratorStatus
{
    /**
    OK.
    */
    DEVICE_CONFIGURATOR_OK,

    /**
    Failed.
    */
    DEVICE_CONFIGURATOR_FAILED,

    /**
    Invalid license.
    */
    DEVICE_CONFIGURATOR_INVALID_LICENSE,

    /**
    Device configurator hasn't enough resources.
    */
    DEVICE_CONFIGURATOR_NO_RESOURCES,

    /**
    Device configurator hasn't such client.
    */
    DEVICE_CONFIGURATOR_NO_SUCH_CLIENT,

    /**
    References file is missing.
    */
    DEVICE_CONFIGURATOR_NO_INFORMATION_FILE,

    /**
    History file is missing.
    */
    DEVICE_CONFIGURATOR_NO_HISTORY_FILE,

    /**
    License file is missing.
    */
```

```
    DEVICE_CONFIGURATOR_NO_LICENSE_FILE,

    /**
    Unknown error.
    */
    DEVICE_CONFIGURATOR_UNKNOWN_ERROR
};

struct DeviceInfo
{
     int deviceId;
     int vendorId;
};

struct LicenseInfo
{
     int deviceId;
     int license;
};

struct ConfigurationCommand
{
     int commandId;
     char* data;
     int dataLength;
};


extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_RegisterClient(
                                    DeviceInfo deviceInfo );

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_UnregisterClient(
                                    DeviceInfo deviceInfo );

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_CheckLicense(
                               DeviceInfo deviceInfo,
                               LicenseInfo licenseInfo );

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_ConfigureDevice(
                          DeviceInfo deviceInfo,
                          LicenseInfo licenseInfo,
                          ConfigurationCommand*& commands,
                          int& commandsCount );

extern "C" __declspec(dllexport)
```

```
DeviceConfiguratorStatus DC_FreeCommandData(
                         ConfigurationCommand* commands,
                         const int commandsCount );

#endif
```

9.4 Имплементация на интерфейса на компонента

```
#include "StdAfx.h"

#include "DeviceConfigurator.h"
#include "ClientManager.h"
#include "ConfiguratorService.h"


const int MAX_CLIENTS_COUNT = 10;

vector<DeviceInfo> G_RegisteredClients =
                                vector<DeviceInfo>();

bool dc_isClient( DeviceInfo deviceInfo );


extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_RegisterClient(
                                    DeviceInfo deviceInfo )
{
     if ( G_RegisteredClients.size() >= MAX_CLIENTS_COUNT )
     {
          return DEVICE_CONFIGURATOR_NO_RESOURCES;
     }

     if ( !dc_isClient( deviceInfo ) )
     {
          // add this client
          G_RegisteredClients.push_back( deviceInfo );
     }
     else
     {
          // this client is already registered
     }
     return DEVICE_CONFIGURATOR_OK;
}

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_UnregisterClient(
                                    DeviceInfo deviceInfo )
```

```
{
     bool found = false;
     for ( int i = 0;
               i < G_RegisteredClients.size() && !found;
               i++ )
     {
          DeviceInfo currentDeviceInfo =
                         G_RegisteredClients.at( i );
          if ( currentDeviceInfo.deviceId  ==
                              deviceInfo.deviceId
               && currentDeviceInfo.vendorId ==
                              deviceInfo.vendorId )
          {
               // remove this client
               G_RegisteredClients.erase(
                    G_RegisteredClients.begin() + i );

               found = true;
          }
     }

     if ( found )
     {
          return DEVICE_CONFIGURATOR_OK;
     }
     else
     {
          return DEVICE_CONFIGURATOR_NO_SUCH_CLIENT;
     }
}

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_CheckLicense(
          DeviceInfo deviceInfo, LicenseInfo licenseInfo )
{
     if ( !dc_isClient( deviceInfo ) )
     {
          return DEVICE_CONFIGURATOR_NO_SUCH_CLIENT;
     }
     ClientManager clientManager;
     DeviceConfiguratorStatus status =
               clientManager.CheckLicense( deviceInfo,
               licenseInfo );
     return status;
}

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_ConfigureDevice(
     DeviceInfo deviceInfo, LicenseInfo licenseInfo,
```

```
     ConfigurationCommand*& commands, int& commandsCount )
{
     if ( !dc_isClient( deviceInfo ) )
     {
          return DEVICE_CONFIGURATOR_NO_SUCH_CLIENT;
     }
     DeviceConfiguratorStatus status =
                         DEVICE_CONFIGURATOR_UNKNOWN_ERROR;

     // check license
     ClientManager clientManager;
     status = clientManager.CheckLicense(
                              deviceInfo, licenseInfo );
     if ( status != DEVICE_CONFIGURATOR_OK )
     {
          return status;
     }

     ConfiguratorService configuratorService;
     status = configuratorService.GetConfigurationCommands(
                         deviceInfo, licenseInfo,
                         commands, commandsCount );

     return status;
}

extern "C" __declspec(dllexport)
DeviceConfiguratorStatus DC_FreeCommandData(
                         ConfigurationCommand* commands,
                         const int commandsCount )
{
     if ( commands == NULL || commandsCount <= 0 )
     {
          return DEVICE_CONFIGURATOR_FAILED;
     }

     for ( int i = 0; i < commandsCount; i++ )
     {
          ConfigurationCommand* configurationCommand =
                                                commands + i;
          if ( configurationCommand != NULL )
          {
               if ( configurationCommand -> data != NULL )
               {
                    delete configurationCommand -> data;
                    configurationCommand -> data = NULL;
               }
          }
     }
```

```
     delete commands;

     return DEVICE_CONFIGURATOR_OK;
}

bool dc_isClient( DeviceInfo deviceInfo )
{
     for ( int i = 0; i < G_RegisteredClients.size(); i++ )
     {
          DeviceInfo currentDeviceInfo =
                              G_RegisteredClients.at( i );
          if ( currentDeviceInfo.deviceId ==
                         deviceInfo.deviceId
               && currentDeviceInfo.vendorId ==
                         deviceInfo.vendorId )
          {
               return true;
          }
     }
     return false;
}
```