**Софтуерни технологии**

**Раздел 1:**

Според стандарта IEEE софтуерното инженерство е приложение на систематичен, дисциплиниран, пределен подход за разработка, изпълнение, и поддръжка на софтуер. Тази дефиниция предполага съществуването на модели, поддържащи систематичен развой на софтуера. Разработване на такива модели е една от основните задачи на софтуерното инженерство. Моделите са едно идеализирано представяне на развойния процес. Обикновено те дават само една рамка, която позволява достатъчна адекватност и гъвкавост при разработването на софтуер. Различните проблеми, които трябва да бъдат решавани в софтуерните проекти, предполагат наличието на подходящи модели, които да гарантират удобства и адекватност при разработване на различни типове софтуерни продукти. Прототипирането е един често използван модел.

На Фиг. 1 са показани основните дейности, свързани с моделиране на развойния процес в софтуерното инженерство.

Дейности по моделиране на СП Моделите на развойния процес могат да бъдат дефинирани в следните два аспекта:

* От една по-обща гледна точка - един модел е план за разработка, който специфицира основните дейности за създаване на софтуерни продукти.
* По-прецизно - един модел е дефиниция за развой на софтуер, която представя:
  + Дейностите, които трябва да се изпълнят
  + Хората, които ще ги изпълнят – обикновено в качеството си на някаква роля
  + Редът на изпълнение на действията
  + Продуктите, които ще бъдат разработени
  + Начинът за оценка на разработвания софтуер.

Ролите в развойните модели обикновено се идентифицират с участници в развойния процес, които трябва да изпълнят определени задачи. Примери за роли могат да бъдат инженери за тестване, ръководители на проекти, проектанти, програмисти, софтуерни ергономисти и др.

**Жизнен цикъл на софтуерния развой**

Разработването на софтуерни продукти (както в класическото инженерство) преминава през различни стадии. За характеризиране на това свойство е въведено понятието жизнен цикъл на софтуерния развой. Жизненият цикъл (software life cycle) е процес на разработка на софтуер, който идентифицира всички фази и стадии на продуктите – от развой, въвеждане в експлоатация, поддръжка, до снемане от експлоатация и премахване.

Жизненият цикъл може да бъде характеризиран в следните два аспекта:

* Статичен – фазите на жизнения цикъл (life cycle phase) са ограничени във времето интервали със сравнително самостоятелни ресурси, за които могат да се определят начални ситуации и оценими крайни състояния
* Динамичен – фазите са свързани помежду си във времето. На Фиг. 3.3. е даден един идеализиран цикъл на отделните фази.

**Основни фази на жизнения цикъл**

До възникване на дисциплината софтуерно инженерство (около 1970) процесът на разработка на софтуер обикновено се е разглеждал като черна кутия, където:

* Всеки разработчик е можел да я попълва със собствени методи
* Процесът е бил напълно индивидуален
* Процесът е бил напълно индивидуален
* Нямало е стандартни дейности и стандартни продукти
* Процесът не е бил итеративен, а по-скоро хаотичен.

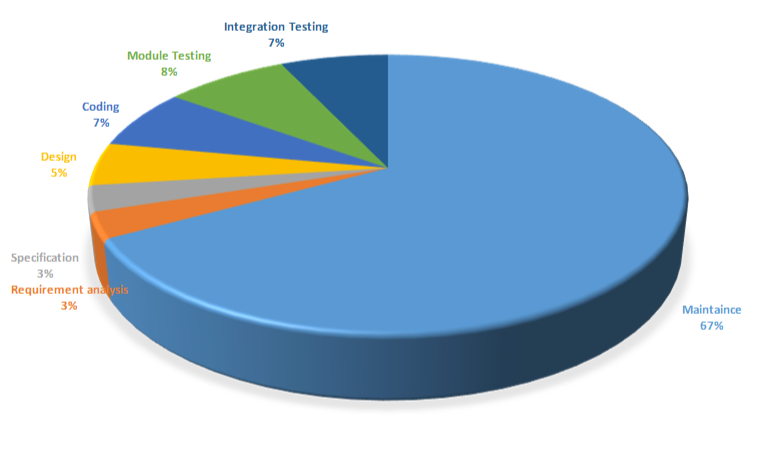
Основният поставян тогава въпрос е какво точно става в тази “черна кутия”. Днес вече съществуват много и различни модели за разработка на софтуер. Въпреки това все още съществуват фирми за развой на софтуер с хаотичен и неопределен процес на разработка.

В началото на седемдесетте години възниква първият модел за систематичен развой на софтуер, който е наречен класическия водопаден модел (waterfall model). Основният принос на този модел е, че поставя началото на еднин систематичен, използващ инженерни принципи развой на софтуер. Водопадният модел дефинира следните основни фази за разработване на програмни продукти, които стават основата на всички последващи модели:

* Анализ и дефиниция – в тази фаза се анализира проблема за решаване и се дефинират изскванията към софтуерния продукт. Фокусът е предимно върху външното поведение на системите. Интензивно се изследва взаимовръзките между клиентите и доставчиците (разработчиците). Тази взаимовръзка е основна за разбиране същността на проблема. За разработката на коректен и ефективен софтуер, софтуерните инженери (анализатори), трябва да разберат информационния домейн (потребителската област) като изисквана функционалност, поведение, производителност и интерфейси. Основните резултати от тази фаза са спецификация на изискванията и модел на продукта
* Проектиране – в тази фаза се специфицира структурата на софтуерната архитектура (спецификация на компонентите и техните връзки). Софтуерното проектиране е процес, който се фокусира върху четири различни атрибута на разработваните програми - информационни структури, софтуерна архитектура, представяне на интерфейсите и процедурни (алгоритмични) детайли. Процесът на проектиране трансформира изискванията към продукта в представа за софтуерна система, качеството на която може да бъде оценено преди да започне кодирането. Основните резултати от тази фаза са софтуерна архитектура и детайлен проект
* Разработка – в тази фаза софтуерната архитектура се „пълни“, т.е. реализират се отделните компоненти на архитектурата. Проектът трябва да бъде преведен в понятна за машината форма – това е задача на кодирането, което се изпълнява в тази фаза. Ако проектът е изпълнен с най-големи подробности, кодирането се превръща в механична задача. Основни резултати от фазата са целеви продукти и програми (код)
* Тестване – в тази фаза се тестват отделните компоненти и тяхната интеграция в единна система. Тестването започва след генерирането на кода. Процесът на тестване се фокусира върху значението (бизнес-логиката) софтуера. Трябва да се осигури тестване на всяка отделна част от софтуера, включително и функционалните разширения (ако има такива). Тестовете са за неоткрити грешки и за сигурност, че с определени входящи данни ще се получат актуални резултати в съгласие с изискванията. Основните резултати от фазата са тестови случаи и тестови протоколи

**Разпределяне на цената в жизнения цикъл на софтуера**

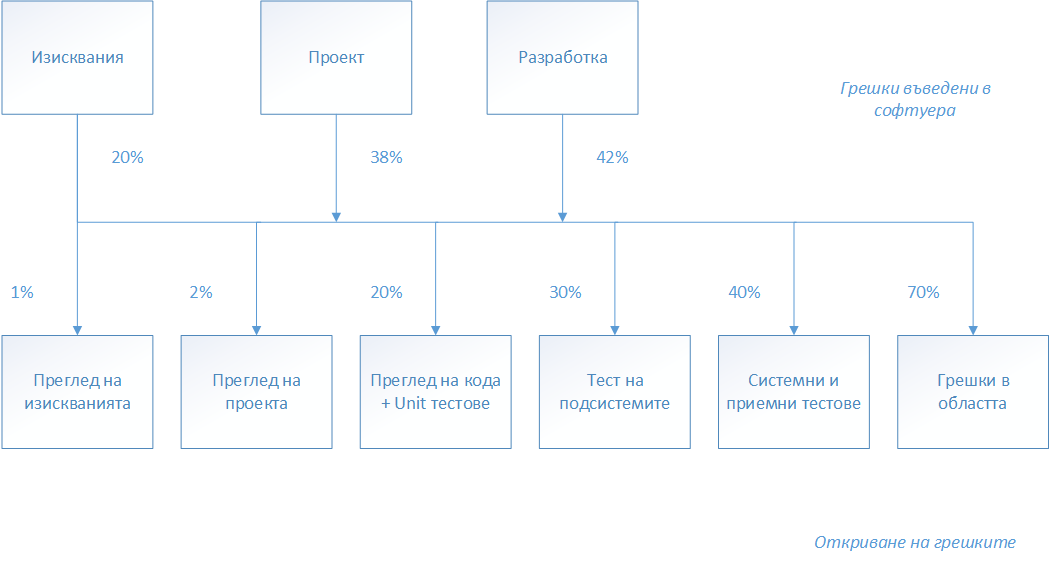
На Фиг. 2 е дадено едно типично разпределение на цената на софтуера според фазите на класическата разработка на софтуер. Повече от половината цена отива в поддръжка на софтуера – 67%. Фазата на анализ и дефиниция е разделена в две отделни дейности: анализ, за която дейност отиват около 3% от средствата и спецификация, за което се изразходват още 3%. За кодиране на приложението обикновено се отделят около 7%. Фазата на тестване е разделена също на две дейности - тестване на модулите и интеграционно тестване. За тестване на всеки отделен модул се изразходват около 8%, а за интеграционното тестване (интеграция на модулите) - около 7%. Може да се отбележи, че средствата, които се изразходват за тестване, са два пъти повече отколкото за кодиране.



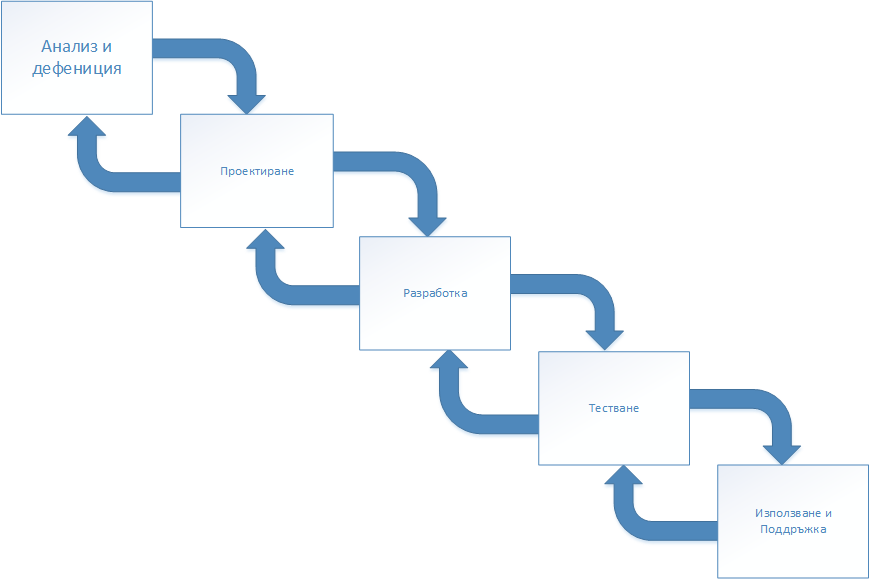
Фигура 2: Разпределение на цената според фазите на СП

На фиг. 3 е дадено разпределение на най-много допуснати грешки при разработката на софтуер и моментите на тяхното откриване. 20% от грешките се допускат по време на събиране на изискванията. При преглед на спецификацията на изискванията се откриват едва 1%. По време на проектиране се допускат около 38% от грешките в проекта, а при преглед на проекта се откриват само 2 %. Най голямо количество грешки се допускат във фазата на разработка (кодиране) – 42%. При преглед на кода и прилагане на unit тестове се откриват около 20%. Повечето от грешките се откриват по време на фазата на тестване на приложението, които са разпределени по следния начин:

* При тестване на подсистемите на приложението се откриват около 30% от грешките
* При провеждане на системни и приемни тестове се откриват около 40% от грешките в приложението – това са тестове при внедряване на приложението в средата в която ще работи
* 7% от грешките се дължат на грешки в проблемната област.



Основната схема на итеративните модели е представена на Фиг. 8. Тези модели избягват някои от основните недостатъци на водопадния модел. При тях е възможно връщане от една фаза в предишна, което беше невъзможно при водопадния модел. В практиката винаги може да се наложи промяна на проекта поради някаква грешка или разширение, т.е. необходимо е връщане в предишна фаза.

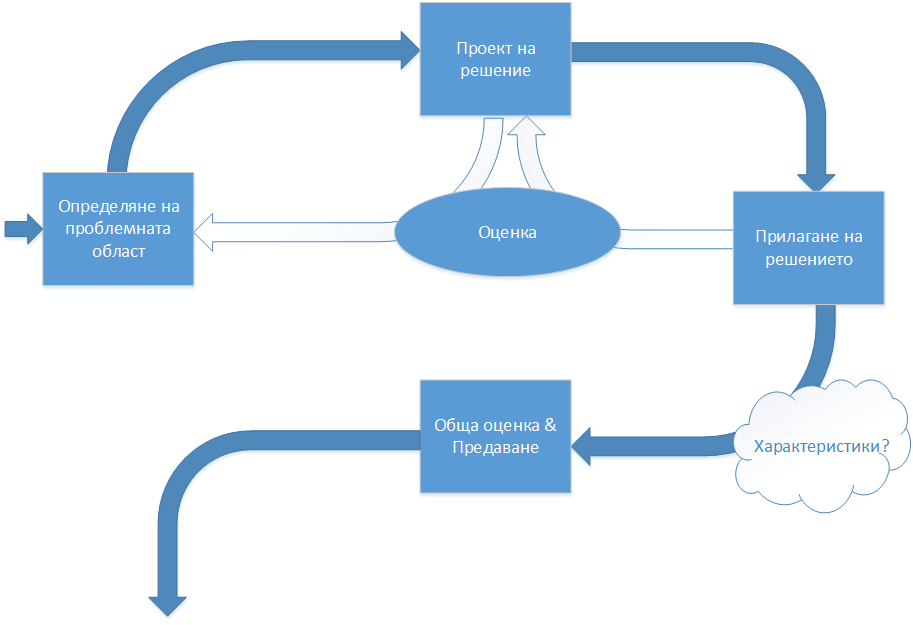


Фиг.8: Итеративен модел

В итеративните модели на разработка на софтуер съществуват също така определени проблеми:

* Връщането назад във фазите на разработка е една доста скъпа дейност
* В итеративния подход както се вижда на фигурата можем да се върнем една фаза назад. Но е възможно например по време на тестването да открием, че спецификацията на изискванията е грешна. Следователно трябва да се върнем във фазата на анализ и дефиниция и да коригираме спецификацията. За целта трябва да прескочим фазата на разработване и фазата на проектиране, а това не е възможно в този модел на процеса на разработка.
* За избягване на някои от тези недостатъци са разработени различни модификации на стандартния итеративен модел. Някои от основните са:
  + Спирален модел
  + Whirpool модел
  + Фонтанен модел.

Прототипирането е тясно свързано с итеративните модели, като то представя една по-различна гледна точка на разработването на софтуерни продукти. Прототип на едно софтуерно приложение може да се направи поради това, че изискванията за приложението в началото са непълни и не достатъчно прецизни. Прототипът обикновено реализира съществени за потребителя компоненти или неясни части от приложението. Съществуват два основни подхода за прототипитане:

* Бързо прототипиране – след като прототипът е бил разработен и е изпълнил своята задача той се изоставя и приложението се разработва на ново
* Еволюционно (итеративно) прототипиране – разработе
* ният прототип се използва като част от разработката на приложението.

Фигура 10: Модел за итеративно прототипиране

На Фиг. 10. е представен модел за итеративно прототипиране, който се състои от четири фази – определяне на проблемната област, проект на решение, прилагане на решението, обща оценка и предаване. Във всяка фаза се прави оценка на решението или областта, което води до избягване на грешки или по-лесното и навременното им откриване. Моделът има следните характеристики:

* Естествена форма за разработка на проекти, базирани на вътрешно отстраняване на грешките
* Разработва се прототип още в ранна фаза, който може да послужи и като демонстрационна версия на продукта
* Оценките ръководят проектните решения.

Когато прототипирането се поддържа от специализирани средства (напр. подходящи генератори) тогава говорим за бързо прототипиране (rapid prototyping). Един метод, базиран на направлявано от потребителя итеративно прототипитране е разработка на динамични системи (DSDM). DSDM e RAD (Rapid Application Development) метод, приет от много компании и университети, за разработка на софтуер. DSDM използва девет принципа, като най важните от тях са следните:

* Изисква итеративно прототипиране
* Изисква сериозно участие на потребителите (те са част от екипа за проектиране)
* Моделът е ориентиран към продукта.

Методът се използва предимно за разработка на комерсиален софтуер. Той е бил разработен като реакция на ограниченията, произтичащи от различните варианти на водопадния модел. Основните характеристики на метода са следните:

* Подпомага приложното проучване
* Съдържа итерация на функционален прототип
* Поставя системата в потребителската среда
* Извършва се оценяване като паралелна дейност по време на целия жизнен цикъл.

По време на приложното и бизнес проучване на приложението се дефинират на общо и абстрактно ниво функционални изисквания към приложението. Резултатът от тази фаза е план на прототипа и установяване на нефункционалните изисквания.

Във фазата на итерация на функционален прототип се определят детайлните изисквания към системата. Обикновено тази фаза завършва с прототип, който демонстрира основната функционалност на софтуерният продукт.

Във фазата на проектен прототип се усъвършенства функционалния прототип.

По време на фазата на разработка се създава ръководство за потребителя и се правят прегледи на документите на проекта. Предоставянето на продукта и обучението на персонала е също част от тази фаза.

Най важната характеристика на този подход е че четирите фази се застъпват и могат да се изпълняват паралелно.

**Раздел 2:**

### Въведение

Във функционалния изглед на системата ще бъдат разгледани три основни концепции. Две от тях – функционални дървета и диаграми на потока на данните (DTF) са подходящи за описание на функционалността на една софтуерна система при използване на структурен анализ. Use case диаграмите са подходящи за описание на функционалност и бизнес процеси при обектно-ориентиран анализ.

### Функционални дървета

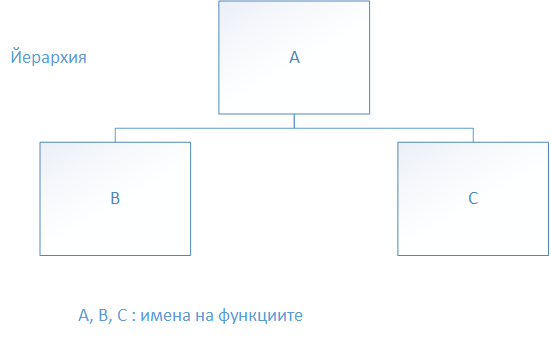
Функциите обикновено описват определени дейности или ясно очертани задачи в рамките на една проблемна област. От гледна точка на софтуерните технологии една функция:

* Доставя определени изходни данни на основата на някакви входни данни или
* Предизвиква промяна на съдържанието или структурата на определени данни.

Във фазата “Анализ и дефиниция" като функции могат да се представят напр. различни задачи, които трябва да изпълнява разработваната софтуерна система (обикновено оформени като функционална спецификация на системата). Във фазата “Проектиране и Разработка” функции могат да бъдат подпрограми (функции, процедури, методи) на някой език за програмиране, който е избран за реализация на системата. Графично функциите ще представяме като правоъгълници.

Декомпозирането на общи комплексни задачи на по-прости функционални идентичности може да се представи като функционално дърво, което представя определена йерархия от функции (Функции à Подфункции à … ). Възможни са различни интерпретации на една такава  йерархия. Така напр., за функционалното дърво на Фиг.1. могат да се направат следните интерпретации:

* Във фаза на дефиниция може да се интерпретира като “Aсе състои отB иC”, т.е. така наречена зададена йерархия
* Във фаза на проектиране интерпретацията може да бъде “AизвикваBиC”, или йерархия на извикване
* Това, че А се състои от B и C не означава, че по време на изпълнението тя задължително извиква B и C.
* Зададената йерархия не е необходимо да съдържа йерархията на извикване  
  (например: задача -> изпълнима функция)



Фигура 1: Йерархия от функци

### DFD

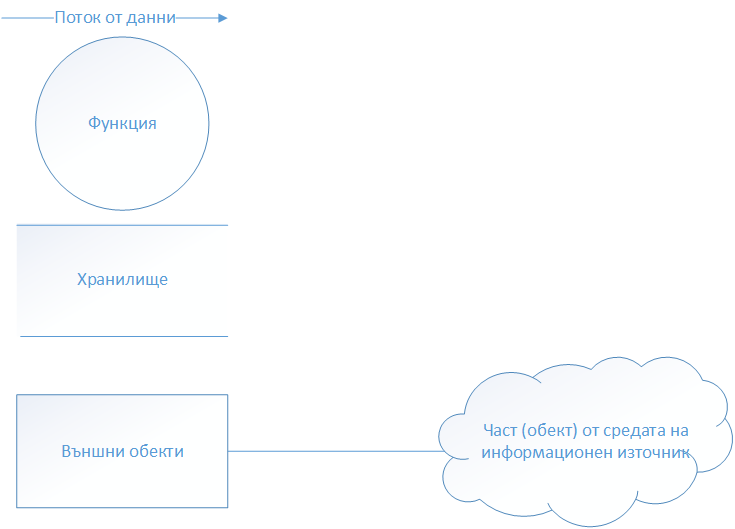
Основната идея на този вид представяне (Data Flow Diagram – DFD) е, че разработваната система е един информационен поток от данни между:

* Функции
* Памет (хранилища за данни)
* Интерфейси към външни обекти (могат да бъдат източници или цели).

При движението на данните те могат да бъдат трансформирани от един вид в друг.

Съществуват различни възможности за представяне на потоците от данни. Едно от най-често използваните е на DeMarco, при което за представяне на базовите елементи на диаграмите се използват следните графични символи (Фиг 2):

* Поток от данни – представя се като именувана стрелка
* Функция (съотв. процес) – представя се като именувана окръжност (или балон)
* Памет (място за съхраняване на данни) – за представянето се използват две паралелни линии, между които се задава името на паметта
* Интерфейс към околната среда (към външни обекти) – представя се като правоъгълник, именуван посредством името на интерфейса.



Фигура 2: Основни елементи в DFD

Базовите елементи могат да се комбинират помежду си за представяне на по-сложни структури на информационни потоци между функции, хранилища и външни обекти .

При използване на тези диаграми трябва да се спазват следните синтактични правила:

* Всяка диаграма (DFD) трябва да съдържа поне един външен обект;
* Всеки външен обект се представя само веднъж, освен ако диаграмата не е достатъчно ясна - тогава външният обект може да бъде повторен ;
* Всеки поток от данни има име (с изключения на потоци, които водят/идват към/от хранилища. Приема се, че тe съдържат всички данни от хранилището);
* Между интерфейсите към външни обекти не се представят потоци от данни;
* Между паметите не трябва да има директни потоци от данни ;
* Между интерфейсите към външни обекти и паметите не трябва да се дават директни потоци от данни;
* Между външните обекти и хранилищата трябва винаги да се намират функции;

Семантичните правила за диаграмите на потоците от данни са следните:

* Общо правило - диаграмите описват потоците от данни, а не контролните потоци. По тази причина те не съдържат възможности за разклонения и цикли
* За интерфейсите
  + Когато един интерфейс представя едно множество от различни инстанции, тогава той се представя като един елемент
  + Ако системата е ограничена от малък брой подобни интерфейси, които обаче са обозначени от различни потоци от данни, тогава е целесъобразно разделено представяне на интерфейсите.
  + Интерфейсите трябва да се избират така, че да дават ясна представа за оригиналния източник или предназначение на информацията;
  + Изборът на интерфейси трябва да се абстрахира от конкретния начин на въвеждане (напр. от клавиатурата) или извеждане (напр. отпечатване) на информацията.
* За имената на потоците от данни
  + Желателно е имената на потоците от данни да бъдат съществителни имена  или като комбинация от  прилагателни и съществителни имена. Пример: ‚N-Сметка‘, ‚валиден N-Сметка‘
  + Трябва да се избягват наименования с общо значение като напр.: ‚Данни‘, ‚Информация‘, ...
* За имената на функциите
  + Целесъобразно е имената на функциите да бъдат глаголи, следвани от имена на  конкретни обектиили съществителни, следвани от глаголи.
  + Добре е да се избягват имена на функции с общо значение, като напр. ‚процес‘, ,управление‘, ...

**Предимства и недостатъци на DFD**

Основните предимства на диаграмите на потока от данни са следните:

* Могат да бъдат лесно създавани и имат добра четаемост
* Лесни са за разбиране и от непрофесионалисти, което улеснява значително контактите между партньорите (клиенти, доставчици)
* Съдържат повече информация от функционалните дървета

Недостатъци на тези диаграми:

* Ако искаме да представим една цяла система диаграмите могат да станат много големи, трудно обозрими и нечетаеми – възможно решение в такъв случай е изграждане на йерархични нива (метод на структурния анализ)
* Трудно се поддържа единно абстракционно ниво за данните и функциите
* В определени случаи означаването на потоците данни само с имена не е достатъчно.

### Use case диаграми

Организацията на работните потоци във фирмите обикновено се представя като бизнес-процеси. Един бизнес-процес се състои от вътрешни дейности, които ще бъдат изпълнени последователно за удовлетворяването на нуждите на потребителите (със или без софтуер). Такива дейности могат да бъдат: сортиране на поща, четене на писмата,  отговаряне на писмата използвайки тексто-обработваща система и т.н.

В софтуерните технологии бизнес-процесите обикновено се описват като сценарии (use cases). Един use case  може да бъде дефиниран като фундаментална подфункция от една система (услуга на системата) със стойност за потребителя (видим резултат), която е реализирана чрез последователност от взаимодействия между потребителя и системата.

Сценариите описват функционални изисквания към разработваното приложение, като функционално описание на системата е множеството от всички use cases.

Use case диаграмите описват бизнес-процеси в една информационна система, връзките между бизнес-процесите и връзките между бизнес-процесите и актьорите на системата.

Когато изискванията към разработваната система са ясни, тогава анализаторите могат да създадат различни сценарии, които да следват начина на функциониране на системата. При  създаване на use cases анализаторите обикновено следват представената по-долу схема.

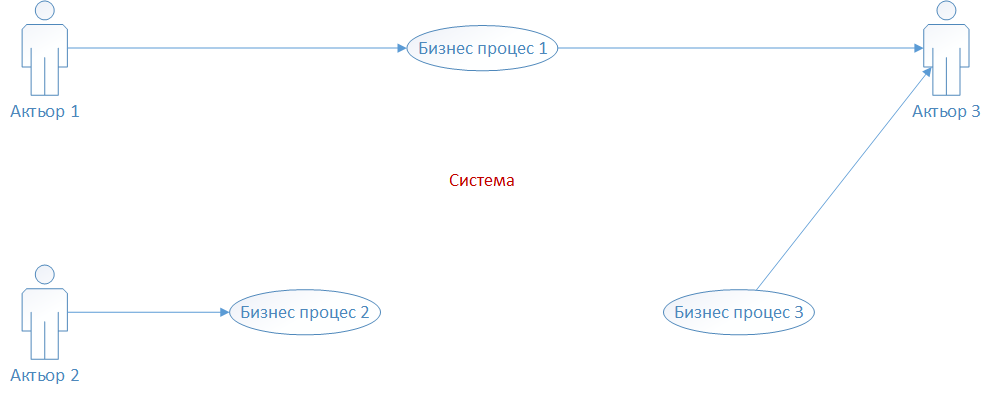
Първо се идентифицират актьорите на системата – това са субектите на предлаганата функционалност, които могат да бъдат определена група потребители или различни видове устройства, външни за разработваното приложение. Актьорите всъщност представят определени роли, които потребителите (или устройства) изпълняват при работа със системата. Формално актьорите се дефинират като субекти, които комуникират със системата и са външни за нея. Важно е да отбележим, че актьор и потребител не са идентични понятия. Обикновено потребителят може да играе няколко различни роли при използване на системата, докато актьорът представя клас от външни за системата субекти, на който се присовява само една роля.

Второ, идентифицират се самите сценарии - use-cases описват начина, по който актьорите взаимодействат със системата. За създаване на сценариите може да се използва следният набор от примерни контролни въпроси:

* Кои са основните задачи или функции, които ще се изпълняват от определен актьор?
* Каква системна информация ще получава или ще променя даден актьор?
* Ще трябва ли актьорът да информира системата за външни промени в средата?
* Каква информация иска да получава от системата разглеждания актьор?
* Трябва ли един актьор да бъде информиран за неочаквани промени?

Всеки use-case предоставя един еднозначен начин на взаимодействие между актьори и система. Той може да съдържа също така спецификация на времеви изисквания или други ограничения. Use-cases могат да бъдат възприемани по различен начин от отделните актьори, като те трябва да се оценяват от гледна точка на всеки актьор дефиниран в системата. На сценариите могат да се присвояват приоритети, които се определят за всеки актьор (обикновено се измерват по скала от 1 до 10). Когато се използва итеративен обектно-ориентиран модел за разработване на софтуер, приоритетите могат да влияят върху последователността, в която се предоставят  функциите на системата, т.е определяйки приоритета на всеки един от сценариите в системата, разработчиците ще са наясно с последователността за разработването им.

На Фиг. 3. е представена система състояща се от три бизнес процеса (use cases). Актьорите, които ги използват, имат различни права на достъп до тях .



За представяне на сценариите (use cases) се използват различни описателни техники, като напр.:

* текстово (текстова схема) – шаблон, по който трябва да се опише всеки use case
* collaboration диаграми – тези диаграми представят сценария като обекти от системата и връзки между тях. Представени са съобщенията, които обектите си разменят, за да се изпълни сценария
* sequence диаграми – представят сценария като множество от обекти и съобщения, които те си разменят, за да бъде той изпълнен. Съобщенията в тези диаграми са подредени последователно във времето
* activity диаграми – представят сценария като алгоритъм , който трябва да бъде следван за неговото изпълнение
* крайни автомати – представят поведението на даден обект в системата, заедно с възможните състояния, които той може да приема по време на изпълнение на сценария
* мрежи на Петри – за представяне на паралелни процеси.

**Принципна схема за изграждане на use cases**

Сценариите обикновено съдържат следните основни компоненти:

* Име (идентификатор) на сценария – обикновено подсказва функционалността на сценария
* Цел - основната цел, която ще бъде постигната при успешно изпълнение на use case
* Категория**–**може да бъдеосновна, второстепенна, или незадължителна
* Предусловия- очаквано състояние, предшестващо изпълнението на use case
* Успешни следусловия- очаквано състояние след успешно изпълнение на use case
* Неуспешни следусловия- очаквано състояние, когато целта не е достигната
* Актьори- роли на потребители или други системи, които вземат участие в use case
* Първоначални събития- когато това събитие се случи use case ще се инициализира
* Описание:
* Първо действие, което трябва да се изпълни в конкретния сценарий
* Второ действие, което трябва да се изпълни в конкретния сценарий
* т.н. (последователност от дейности, които трябва да се изпълнят за осъществяването на сценария)
* Разширение:
  + 1A Разширение (upgrade) на първото действие от споменатите в описанието горе с някоя възможност на дефинираните в системата функции
* Алтернативи:
  + 1A Алтернативно изпълнение на първото действие –  алтернативни дейности, които могат да възникнат при изпълнение на първото действие от описанието
  + 1B Допълнителни алтернативни дейности, които могат да възникната за първото действие от описанието на use case

**Връзки между Use Cases според UML**

В езика за моделиране UML, между отделните сценарии са възможни следните видове връзки:

* **Extend**– това е връзка, която показва разширение на функционалността на един use case с функционалността на друг;
* **Include –**това е връзка, която показва включващо разширение на функционалността на един use case с функционалността на друг. Чрез нея се показва, че една функционалност включва в себе си друга функционалност;
* **Generalize –**това е връзка, която по своя смисъл означава наследяване.

**Предимства и недостатъци на Use case диаграмите**

Основните предимства на използването на сценарии са следните:

* Фокусират се върху основния работен поток, а не върху елементарни функции
* Концентрират се върху стандартните работни потоци
* Лесни са за разбиране и от клиентите (не само от професионалисти)

Недостатък е, че диаграмите лесно могат да се претоварят и утежнят с твърде големи и излишни детайли.

**Раздел 3:**

**Въведение**

Ще бъдат разгледани две основни концепции на ориентирания към данни изглед на системата:

* Речник от данни (Data Dictionary)
* ER – модел (Entity Relationship Model)

**Речник от данни**

Речникът на данни е каталог, който съдържа информация за структурата, свойствата и използването на данните в един развоен процес. Целта е съхраняване на синтактичната структура на оперативните данните от потребителска гледна точка, без изпълними структури като масиви, полета, дървета и др. За описание обикновено се използва модифицирана BNF (Backus-Naur-Form). Тази концепция се използва за по-прецизно представяне на информацията, съдържаща се в диаграмите на потока от данни или диаграмите на класовете

|  |  |  |
| --- | --- | --- |
| **Символ** | **Значение** | **Пример** |
| = | Еквивалентно на | A = B + C |
| + | Последователност | Х = Х1 + Х2 + Х3 |
| [ | ] | Без подредба Избор един ... или | А = [B | C] |
| { } | Повторение | А = {B} |
| M { } N | Повторение от M до N | A = 1 {B} 10 |
| () | Избор от 0 ( )1 | A = B + (C) |
| \*\* | Коментар | A = X + Y \*Коментар\* |

Таблица 1

Таблица 1 показва нотацията за представяне на речник от данни .

Основното предимство на речника на данните е, че с помощта на EBNF структурите от данни могат да се представят формално и сбито (както синтаксиса в модерните езици за програмиране). Тъй като data dictionary не се представя графично, читаемостта за потребителите обаче е ограничена. Едно възможно решение са синтактичните диаграми, които не се прилагат широко в практиката.

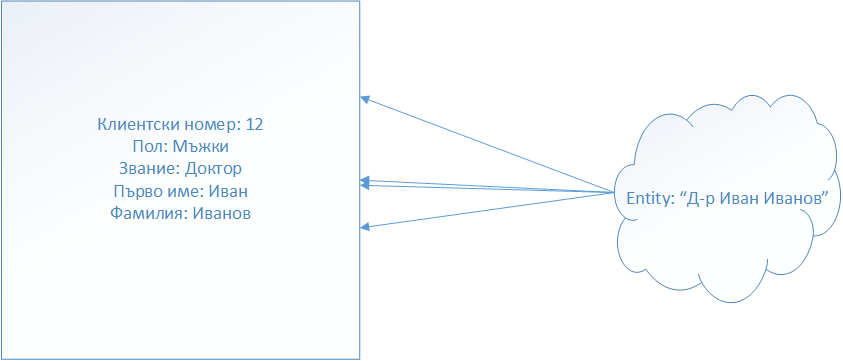
### Entity-Relationship (ER) модел

ER моделите, предложени от P. Chen през 1976 г. за моделиране на данни, са концептуални модели, които са сравнително стабилни към функционални промени. Тези модели се използват предимно за моделиране на бази данни в приложения, опериращи с голямо количество от данни (типично за комерсиалните приложения). Основното предназначение на моделите е описание на постоянните записи от данни (външни данни, файлове) и техните връзки, като могат да се използва в различни методи за разработка на софтуерни продукти (напр. структурен  или обектно-ориентиран анализ).

В следващите раздели ще разгледаме по-подробно основните елементи и използването на ER моделите.

**Идентичности**  
Основните елементи на ER моделите са идентичностите. Една идентичност е едно индивидуално и различимо представяне на обекти от реалния свят или от някакъв абстрактен свят. За визуално представяне на идентичностите се използват правоъгълници.

На примера (Фиг.1) е представен една такава идентичност (обект) от реалния свят, който е потребител на някаква информационна система. Тази идентичност се представя с определени специфични характеристики, като напр. “Клиентски номер”, “Пол”, “Звание”, “Първо име”, “Фамилия”,  адрес (представен като “Улица”, “Пощенски код”, “Град”).



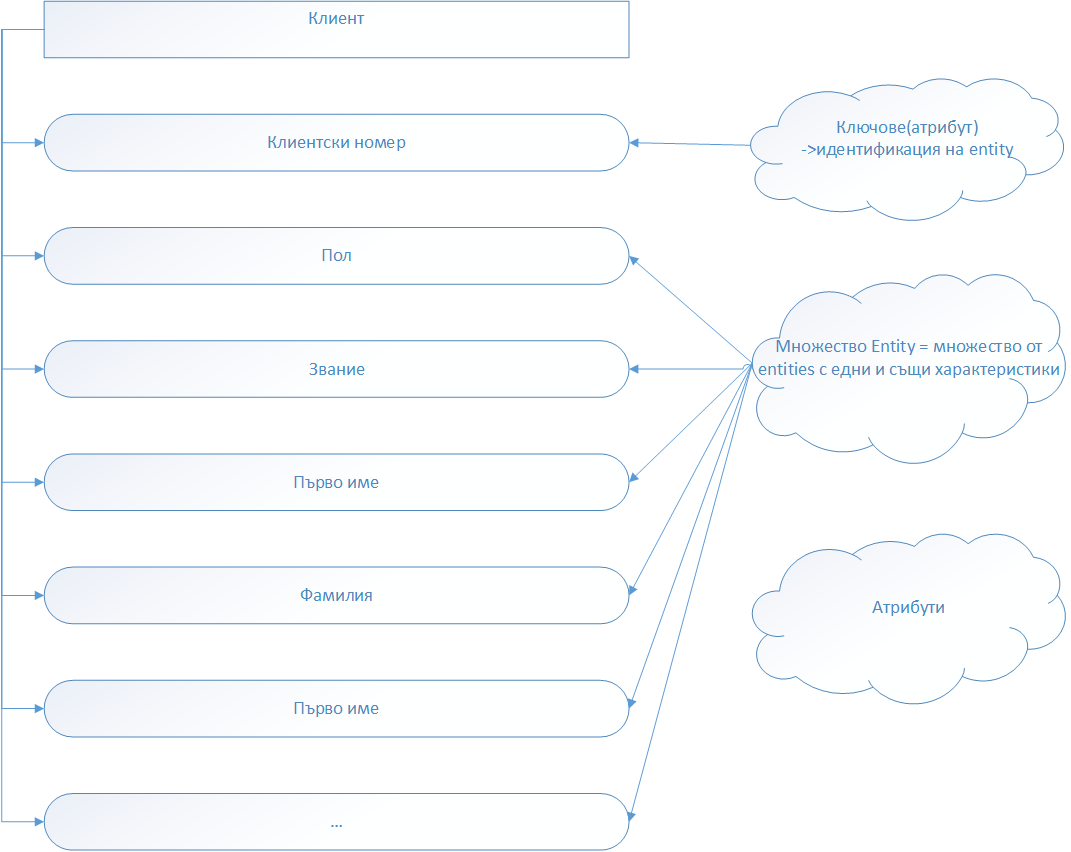
Фигура 1: Обект от реалния свят

**Множества от идентичности**  
Идентичности с едни и същи характеристики могат да се групират в множества. На примера (Фиг.2) е дадено едно множество от потребители. Елементите на множеството са идентичности, характеристиките на които са едни и същи с тези, представени на фиг 1.

Характеристиките на едно множество от идентичности се наричат атрибути. Атрибутите могат да бъдат два вида:

* Описателни – представят съществените за бизнес-логиката на разработваното приложение свойства
* Идентифициращи – използват се за еднозначно идентифициране на идентичностите. В примера такъв атрибут може да бъде “Клиентски номер”.

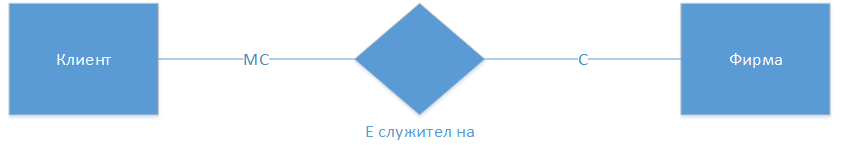
Възможни са повече от един възможни идентифициращи атрибути. Един ключ K е винаги една минимална комбинация от атрибути, с която може да бъде еднозначно идентифицирана една идентичност. Всяко супермножество на тази комбинация се нарича ключ-кандидат. В моделите атрибутите, представляващи ключове, се подчертават.



Фигура 2: Множество от entity (обекти)

**Асоциации (Връзки)**  
Асоциацията е семантична връзка, която съществува между множества от идентичности. Асоциациите се представят като ромбове, в които се задават значенията (интерпретациите) им

**Кардиналности**  
Кардиналността в един ER модел представя сложността на една връзка между множества от идентичности. Основната идея е да се покаже броят на идентичностите от един тип, с които е свързана една единтичност от друг тип.



Фигура 3: асоциация и кардиналност

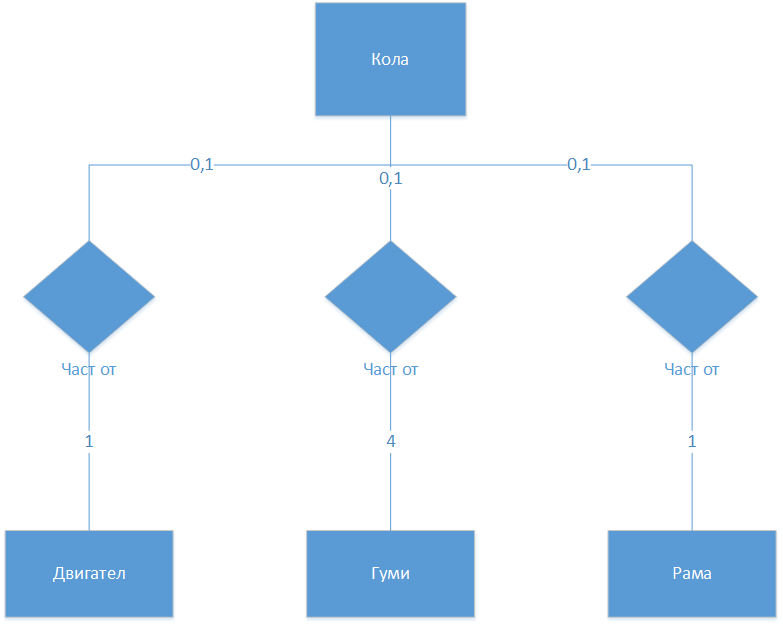
За представяне на кардиналността се използват спецификатори, които могат да бъдат:

* 1 - точно един свързан елемент
* C - 0 или един елемент (C (Choice) = избор – означава избор между 0 или 1 елемент)
* k - точно k елемента
* M - повече елементи (1, 2, …, n) – ( означава избор между 0, 1 или повече (more) елементи)
* MC -      0 или повече елементи (0, 1, 2, …, n)

**Агрегация**  
Агрегацията е специален тип асоциация между множества от идентичности. Обикновено тази връзка се интерпретира като „е част от“ , т.е. едно множество от идентичности е част от друго множество идентичности. На примера (Фиг.4) са дадени следните четири множества от идентичности: “Кола”, “Двигател”, “Гуми” и “Рама”. Агрегацията между тях  се интерпретира по следния начин: множествата от идентичности “Двигател”, “Гуми” и “Рама” са част от множеството идентичности “Кола”. Съответно за всяка от тези агрегации са определени кардиналносттите, които се интерпретират както следва:

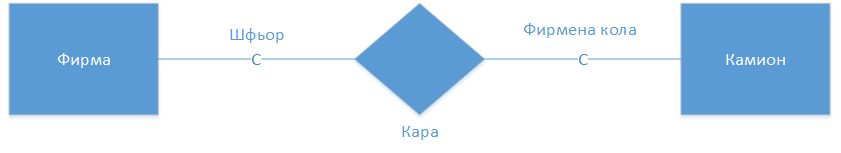
* един двигател може да не бъде част или може да бъде част от една кола
* една кола може да съдържа точно един двигател
* четири гуми могат да не са част от една кола или могат да бъдат част от една кола
* една кола може да има точно четири гуми
* една рама може да не е част или да е част от една кола.
* една кола може да има точно една рама.

В примера различните типовете кардиналности са представени с числа, а не с дадените по-горе означения. Това е начинът на отбелязване на кардиналността в езика UML.



Фигура 4: Агрегация

**Роли**  
Ролята е тип нотация в ER модела, която се използва за означаване така наречената  функция на идентичност в една асоциация. На примера (Фиг.5) са дадени две множества от идентичности “Фирма” и “Кола”, свързани с асоциацията “управлява”. За да бъде уточнена функцията на идентичност за множеството “Фирма” в конкретната асоциация е зададена роля “шофьор”. По аналогичен начин, за да се уточни функцията на идентичността за множеството “Камион” в асоциацията е зададена роля “фирмен камион”. Също така са дадени и кардиналностите на асоциацията – един шофьор може да не управлява или да управлява една фирмена кола и една фирмена кола може да не  бъде управлявана или управлявана от един шофьор.



Фигура 5: Роля на идентичност във връзка асоциация

**Раздел 4:**

**Ориентиран към правила изглед на системата**

Ще бъдат разгледани основните концепции: правила, таблици на решенията и дървета на решенията.

Разгледаните в тази глава основни концепции се използват обикновено във фазата на реализацията. Във фазата на дефиницията в много случаи е целесъобразно представянето на алгоритмични и ориентирани към правила гледни точки на разработваната система или на части от нея да се описва на едно друго ниво на абстракция. Тук ще разгледаме само тези аспекти, които са от значение за фазата на дефиниция. Представените техники са удобни за описание на  зависими от определени условия действия, потоците на управление, функции, задачи или изключения. Така напр., ако се интервюира един възложител или експерт, тогава е целесъобразно техните специфични знания да се обобщят във формата на правила. Във фазата на анализ и дефиниция с помощта на тези изразни форми текстовите описания в спецификацията на изискванията могат да се направят по-прецизни (на синтактично и семантично ниво).

Пример за спецификация на изискванията с помощта на правила – при „плащане с чек“ служител на банката трябва да  следва следните правила при проверка на плащането:

/1/ Ако кредитният лимит на собственика на чека бъде превишен, предишните плащания са били редовни и превишението е по-малко от 500 лева, тогава плащането може да се осъществи   
/2/ Ако кредитният лимит бъде превишен, предишните плащания са били редовни и превишението е повече от 500 лева, тогава плащането по чека може да се извърши, но на клиента се поставят нови условия  
/3/ Ако предишните плащания на чека не са били редовни, плащането не може да бъде извършено   
/4/ Плащането по чека може да бъде извършено, ако кредитният лимит не е превишен.

**Правила**

Правилата обикновено се представят в езика на предикатната логика, като едно правило има следния синтаксис: If условие1 and условие2 … then операция, където:

* Условия – описват определена ситуация, при която могат да се извършат операциите
* Операции – могат да се изпълнят при наличието на описаната в условията ситуация. Операциите могат да бъдат два типа:
  + импликации или дедукции – извеждат се логическите стойности на определени твърдения
  + действия – с тях се променят определени състояния.

Посредством ясния си синтаксис и семантика правилата правят спецификациите на изискванията по-прецизни. Структурата на правилата позволява да определяме елементарни условия и действия като словесни изрази (спецификация на изискванията), като по такъв начин изискванията могат да се декомпозират на повече малки самостоятелни правила. на изискванията става модулна и с това лесно променяема.  Така дефиницията

Пример: нека представим /1/ от горния пример като израз в предикатната логика. Какви елементарни условия и действия можем да специфицираме?

**if**  
Кредитният лимит ще бъде надхвърлен *and*  
плащанията редовни *and*  
надхвърлянето < 500  
**then**  
плащане.

За /3/ кореспондиращ израз в терминологията на предикатната логика може да бъде следният:

**if**  
**not** редовни плащания  
**then**  
не се извършва плащане.

Като резюме можем да дадем следните елементарни действия и условия:

* Възможни условия:
  + C1: Кредитния лимит е надхвърлен? /1/, /2/, /4/
  + C2: Редовни плащания? /1/, /3/
  + C3: Лимит надхвърлен с < 500 евро? /2/
* Следните действия са възможни:
  + A1: плащане /1/, /2/, /4/
  + A2: не се извършва плащане /3/
  + A3: предложени са нови условия /2/

Ако модифицираме правило /3/ както следва: “/3\*/ Ако предишните плащания не са били редовни и*кредитният лимит е надхвърлен*, плащане*няма* да бъде извършено”, тогава кореспондиращо правило може да бъде следното:

При използване на тази форма за представяне на изискванията могат да възникнат определени затруднения, като напр.:

* Трудно се постига яснота върху общия вид на представянето
* Проблеми с пълнотата на спецификацията, т.е. дали ще бъде възможно да се обхванат всички условия?

В такива случаи таблиците на решения могат да бъдат едно целесъобразно подобрение.

**Таблици на решенията**

С помощта на таблици на решенията (DT) могат да бъдат дефинирани кратко, ясно и прегледно действия или операции, които зависят от изпълнението или неизпълнението на определени условия.

Таблиците на решенията се състоят от четири квадранта (Фиг.1):

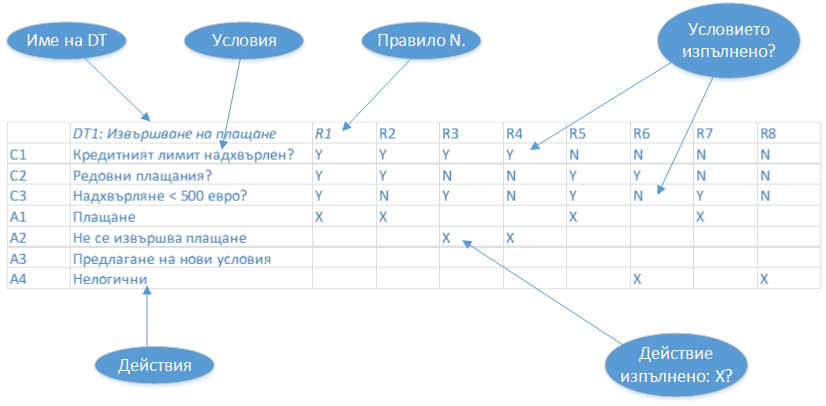
* В първия квадрант (1) се задават условията
* Във втория квадрант (2) се записват действията заедно с техните идентификатори
* Останалите два квадранта - (3) и (4) - се използват за свързване на условията с действията. Свързванията се извършват по колони. Всяка колона се интерпретира като правило и се идентифицира с определен идентификатор.

**if**

|  |  |
| --- | --- |
| **Име на DT** | **Номер на правило** |
| (1) - условия | (3) – знак на условие |
| (2) - действия | (4) – знак на действие |

Фигура 1: Нотация за таблици на решенията

На Фиг.2. е дадена таблица на решенията за примера “плащане с чек”.Фигура 2: Таблица на а решенията за примера „Плащане с чек“



Фигура2: Таблица на решенията за примера “Плащане с чек”

**Оптимиране на таблици на решения**

Таблицата на Фиг. 2. е пълна таблица на решения, понеже в квадрант (3) са дадени всички възможни комбинации от условията. Ако разгледаме по-внимателно правилата R6 и R8 ще забележим, че съдържащите се в тях комбинации са взаимно изключващи се – ако не е надхвърлен определен лимит, тогава не може да съществува надлимитна сума. В този случай двете правила могат да бъдат премахнати. Тогава получаваме една съдържателно пълна таблица (а не формално пълна, както беше в първия вариант), т.е. таблица, съдържаща практически възможните комбинации (Фиг 3).

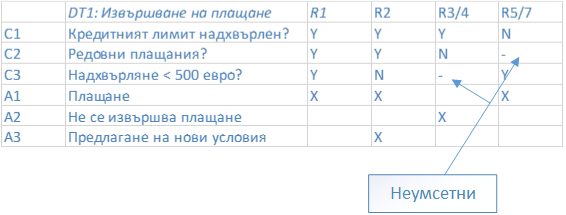


Фигура 3: Оптимирана таблица на решенията

Един съществен проблем при използването на таблиците на решенията е появата на комбинаторен взрив,т.е. при линейно нарастване на условията, кореспондиращите комбинации нарастват експоненциално (при n условия съществуват 2n възможни комбинации). В такива случаи съдържателно пълните таблици могат да бъдат оптимирани по следната схема:

* Търсят се правила с идентични действия – в нашия пример правила (R3, R4) и (R5,R7) имат идентични действия (Фиг.3.)
* Правилата с идентични действия се обединяват, като ако двете правила се различават само  в един ред (условие), тогава правилата се обединяват в едно и в този ред се въвежда незначеща референция  – в примера R3/4 и R5/7 (Фиг.4.)

Така в крайна сметка за примера от 8 правила в оптимираната таблица останаха само 4 (Фиг.4.).



Фигура 4: Оптимирана таблица на решенията

В резюме можем да подчертаем, че при практическо използване таблиците на решения с над 5 условия стават неясни – появяват се повече от 25 комбинации от условия, което значително затруднява създаването на пълни таблици на решения. За решение на този проблем съществуват две възможности.

Първата възможност е декомпозиране на процеса за вземане на решение на подпорцеси, като за всеки отделен процес се изгради собствена самостоятелна таблица. Взаимоотношенията между таблиците се представят посредством съответни връзки. Възможни са следните форми за свързване на таблиците:

* Последователност – две таблици образуват последователност, когато едната таблица е единствен и непосредствен наследник на другата
* Разклонение – една таблица е последвана от повече алтернативни таблици
* Цикъл – съществува поне едно правило, което води до непосредствено повторно използване на актуалната таблица
* Вграждане – за тестване на определено условие или за извършване на определено действие от една таблица се използва друга таблица.

### Дърво на решенията

Както видяхме, освен вертикално (като четири квадранта) правилата могат да се представят също така и хоризонтално. За да могат да бъдат улеснени потребители, които не познават този вид нотация, могат да се използват дървета на решения. При дърветата на решения всички алтернативи се представят явно. Дърветата се обхождат отляво надясно.

**Раздел 5:**

**Въведение**

В тази част ще разгледаме две основни концепции на ориентирания към състояния изглед на системата:

* Крайни автомати
* Activity диаграми

**Крайни автомати**

При много системи изходните резултати не зависят само от входните данни, а така също и от вътрешните състояния на системата или от история на обработката до определен момент. Крайните автомати са адекватно средство за моделиране на такива взаимозависимости. Така напр., вътрешните им състояния могат да се използват за съхраняване историята на обработката на една система. Крайните автомати могат да се използват за моделиране на различни динамични аспекти:

* жизнен цикъл на обектите
* промяна на състоянията на операциите по време на тяхното изпълнение – в спецификациите на операциите на класовете
* промяна на състоянието по време на взаимодействието между потребители и системи – в спецификация на use cases.

Понеже тази проблематика е обект на други лекционни курсове (теория на крайните автомати, дискретна математика) тук ще дадем накратко само основните дефиниции. Математическият модел на един краен автомат се дефинира като A = [ X, Y, Z, f, g ] , където:

* Z  - крайно множество от вътрешни състояния
* X  - входна азбука (символи, сигнали, събития)
* Y – изходна азбука (символи, действия)
* f: X x Z -> Z - функция на прехода
* g: X x Z  -> Y - функция на изхода

Един краен автомат може да се представят също така като граф (*граф на състоянията*).

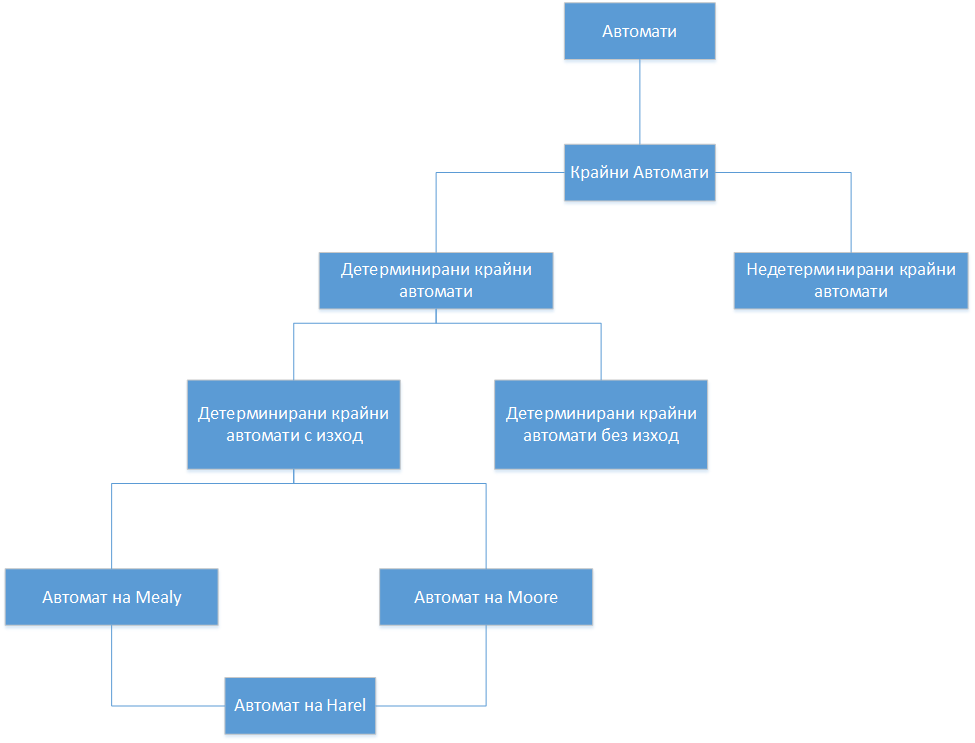
За моделиране на софтуерни системи често се използват различни модификации на класическите крайни автомати. Две основни разновидности на крайните автомати са следните:

* Автомат на Moore – при този вид автомати изходите (действия) са прикрепени към вътрешните състояния на автомата. Една съществена предпоставка за използване на автомати на Moore е, че на отделните състояния може да се съпостави точно един изход.
* Автомат на Mealy - изходите (действия) се прикрепят към преходите. Това отговаря на нашите представи за продължителността на преходите, състоянията и изходите. Състоянията обикновено представят периоди от време или времеви интервали. Преходите от своя страна представляват времеви точки, в които се извършват някакви промени в поведението на моделираната система. Изходите се появяват в дискретни моменти във времето и поради тази причина могат да се асоциират добре с преходите.

Автоматите на Mealy и Moore са еквивалентни детерминирани автомати. Те могат да бъдат трансформирани един в друг:

* Автоматът на Moore може да бъде трансформиран в еквивалентен автомат на Mealy като изходите, генерирани от състоянията в автомата на  Moore, се съпоставят на преходите, които водят към тези състояния
* Автоматът на Mealy може да бъде трансформиран в еквивалентен автомат на Moore като целевите състояния с различни изходи се представят чрез няколко състояния в резултатния автомат на Moore.

На Фиг.1 е дадена една класификация на крайните автомати.

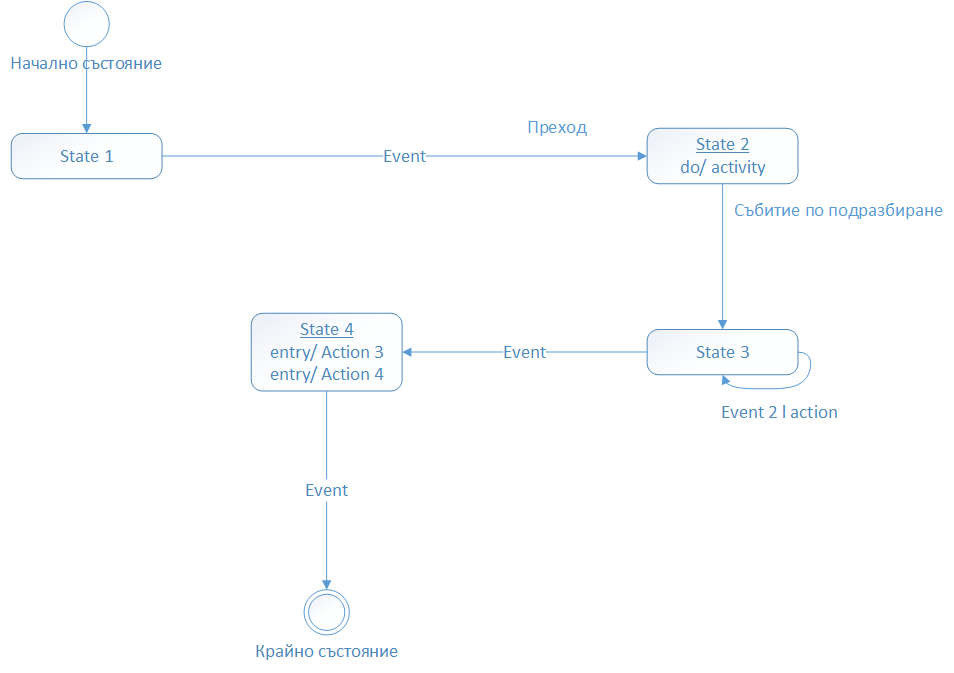


Фигура 1: Класификация на крайните автомати

За моделиране на комплексни и сложни взаимовръзки представените по-горе автомати са недостатъчни. За представяне на такъв вид отношения може да се използва една друга разновидност на крайните автомати, наречена *автомат на Harel*. Този автомат е разширение на един краен автомат със следните концепции:

* Хибриден краен автомат – комбинация между автомат на Moore и автомат на Mealy
* Възможни са условни преходи
* Йерархичен краен автомат – вътрешните състояния могат да се представят като йерархия
* Състоянията могат да имат памет
* Поддържат се конкурентни (паралелни) състояния – едно състояние може да се състои от подсъстояния, в които системата може да се намира в един и същ момент.

На Фиг.2 е дадена принципната схема на функциониране не един такъв автомат (използвана е UML нотация).



Фигура 2: UML нотация на краен автомат

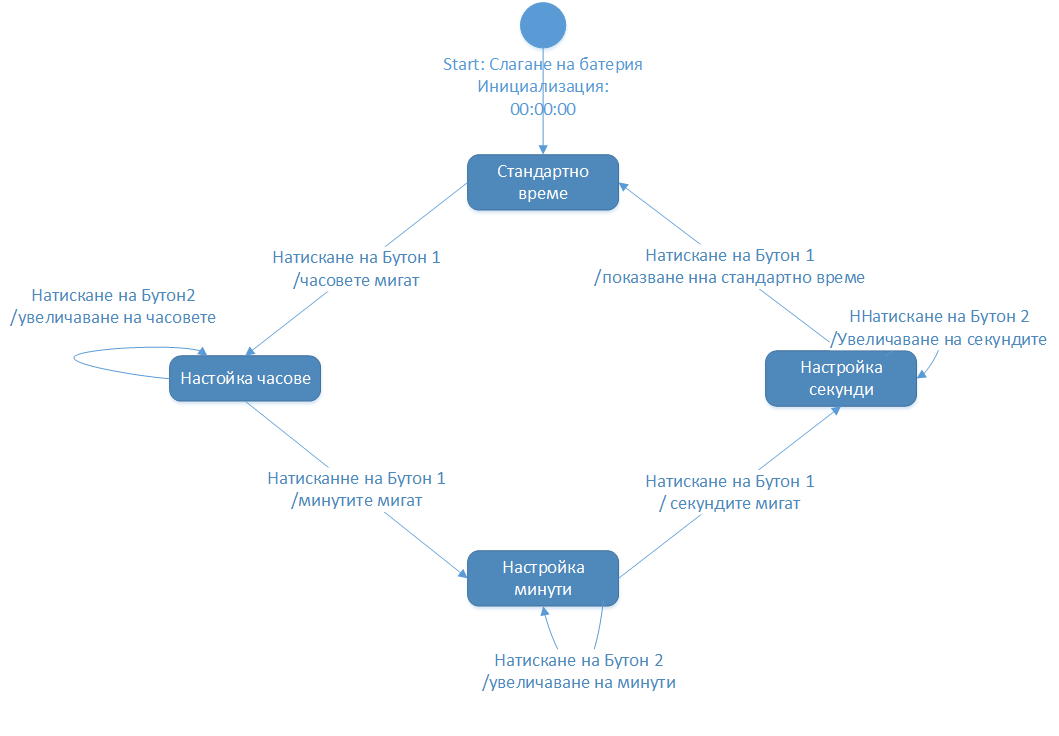
Както вече подчертахме крайните автомати обикновено се използват за моделиране на жизнения цикъл на обектите, т.е. отделните обекти в един клас могат да се представят посредством краен автомат. По принцип не е необходимо създаване на краен автомат за всеки клас. При жизнения цикъл на обектите и в клас-диаграмите само операциите на съответен клас могат да бъдат дейности. Следователно чрез крайни автомати е целесъобразно да се предсатвят само обектите от класове, които имат динамично поведение.

За демонстриране на използването на крайните автомати за моделиране на системи ще разгледаме един пример:

“Настройка на часовник**”,**където занастройването на един дигитален часовник се използват два бутона:

* *Бутон 1* - позволява последователно избиране на настройка (стандартно време, настройка на часа, настройка на минутите, настройка на секундите)
* *Бутон 2* - позволява настройка на времето според избраната настройка (стандартно време, час, минути, секунди)

За този пример ще използваме автомат на Mealy. Графът на състоянията е показан на Фиг.3

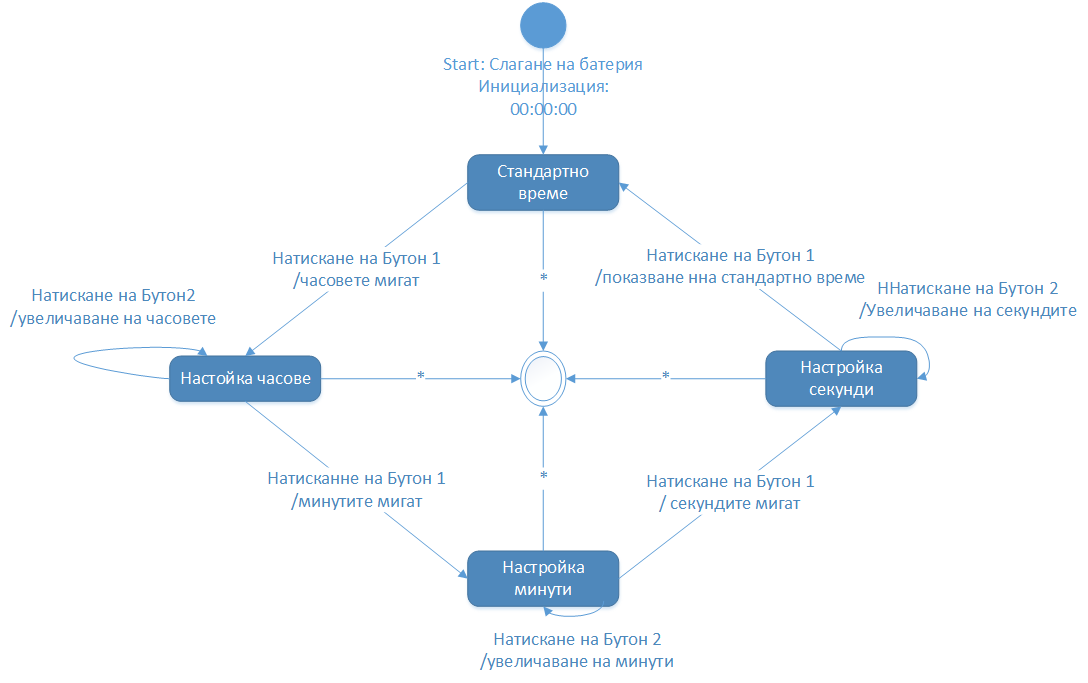


Фигура 3: Граф на състоянията – нотация на Mealy

Изходите при този автомат са прикрепени към преходите, така че ако се активира определен преход, тогава се осъществява кореспондиращият изход. Създаване на краен автоматза моделиране напримера се извършва по следната схема: 

* Идентификация на *състоянията* – ще идентифицираме следните състояния
  + *Стандартно време*: След слагане на батерията
  + *Настройка на часове*:  Часовете могат да бъдат настроени
  + *Настройка на минути*:  Минутите могат да бъдат настроени
  + *Настройка на секунди*:  Секундите могат да бъдат настроени
* Идентификация на *събития* – могат да се идентифицират следните събития
  + *Начален сигнал*: Когато батерията е сложена
  + *Бутон 1 натиснат*:  Когато бутон 1 е натиснат
  + *Бутон 2 натиснат*:  Когато бутон 2 е натиснат
  + Два бутона не могат да бъдат натиснати едновременно
* Идентификация на *резултати*– ще идентифицираме следните резултати
  + *Часовете мигат:*за да индицира, че в момента часовете могат да се настройват
  + *Минутите мигат*
  + *Секундите мигат*
  + *Увеличаване на часовете:*часовете се увеличават с по един на дисплея
  + *Увеличаване на минутите*
  + *Настройка на секундите:*00 се показва на дисплея за секундите
  + *Инициализация:*Показва се на дисплея 00:00:00
* Дефиниция на преходите:
  + Когато *Началният сигнал* се чуе, системата трябва
    - да си смени състоянието на *Стандартно време*и
    - да изпълни действието *Инициализация*
  + Ако *бутон 1*е натиснат и състоянието на системата е *Стандартно време*, системата трябва
    - Да изпълни действието *Часовете мигат*и
    - Да промени състоянието си в *Настройка на часовете*
  + и т.н.

При автоматите на Mealy изходите и входовете са прикрепени към преходите между състоянията. Така могат да се специфицират две специални състояния – начално и крайно състояние, които се обозначават със специални символи   
В някои случаи идентифицирането на крайно състояние е трудно, невъзможно или безсмислено. Така напр. автоматът, моделиращ функцията “Настройка на часовник” няма крайно състояние (Фиг.4.).



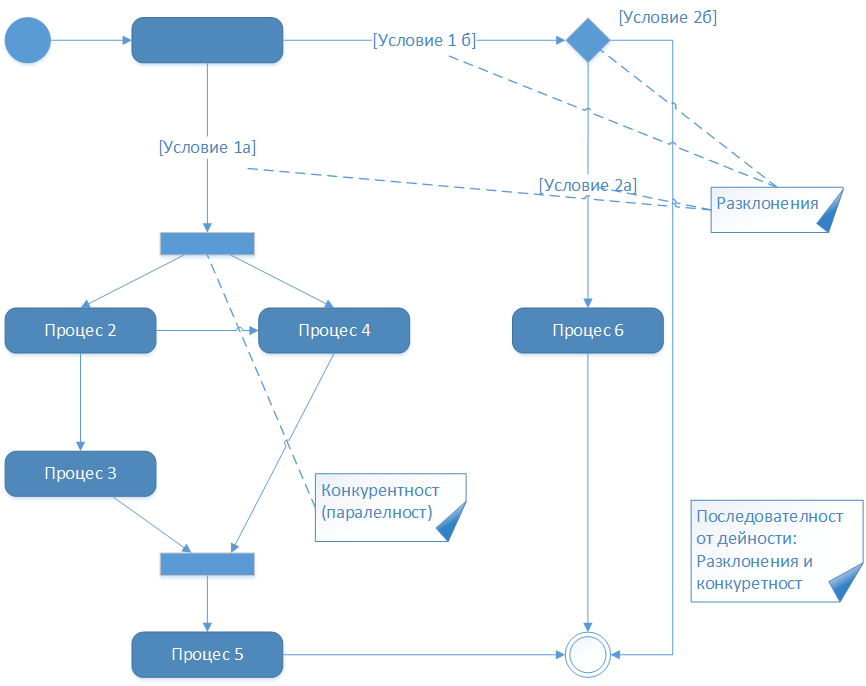
Фигура 4: „Настройка на часовник “ с крайно състояние.

**Activity диаграми**

Activity диаграмите представляват един вариант на крайните автомати, при който:

* Алгоритмите и бизнес-процесите се описват посредством състояния на действията (action states)
* *Action states* са стъпки (дейности) по време на изпълнението на един алгоритъм или бизнес-процес
* *Аction state* е напуснато, когато дейността свързана с него  е приключила (implicit event)
* Подобни са на  другите диаграми, разгледани в предишните глави.

На Фиг.5 е даден един абстрактен пример на Activity диаграма, където се прави по-скоро едно текстово описание на този вид диаграми. Разглеждайки този пример можем да получим една обща представа за същността на тези диаграми. Те описват една последователност от дейности, която може да съдържа разклонения в зависимост от определени условия. Могат да се представят също така и конкурентни (паралелни) дейности. Както се вижда на фигурата след приключване на “Процес 1” работата може да продължи в две различни посоки в зависимост от условията, съдържащи се в прехода. Освен това “Процес 2”, “Процес 3” от една страна и “Процес 4” от друга страна могат да се изпълнят паралелно, като след приключването им  те се обединят в “Процес 5”.



Фигура 5: Абстрактна activity диаграма

**Раздел 6:**

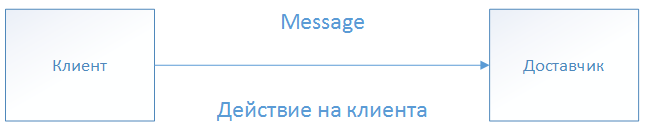
**Ориентиран към сценарии изглед на системата**

**Въведение**

В тази част ще разгледаме две основни концепции – sequence и collaboration диаграми. И двата вида диаграми се използват за моделиране на сценариите на системата при обектно-ориентиран развой. Могат да бъдат разработени по време на фазата на анализ.

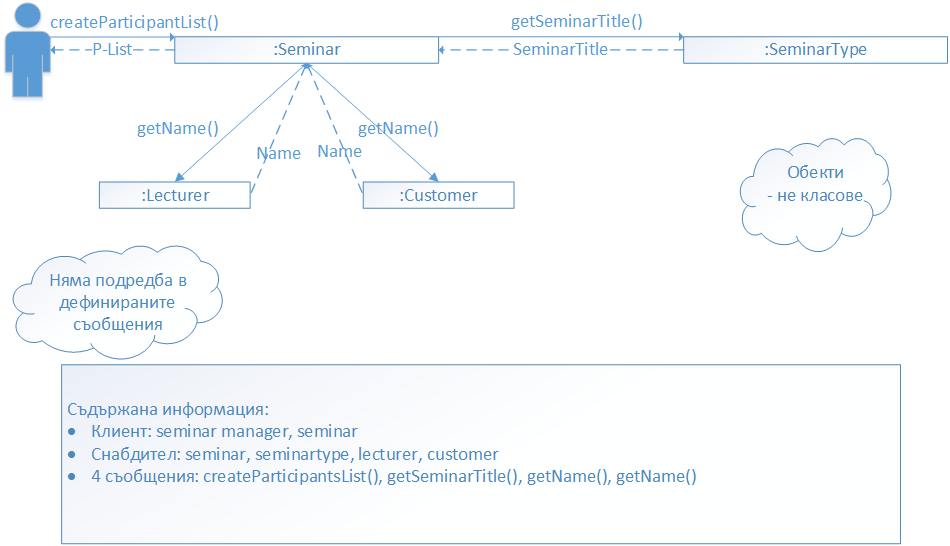
**Съобщения**

С помощта на *съобщения* моделираме заявки на клиенти за  изпълнение на определени услуги от различни доставчици, както е показано на Фиг.1.



Фигура 1: Нотация за съобщение

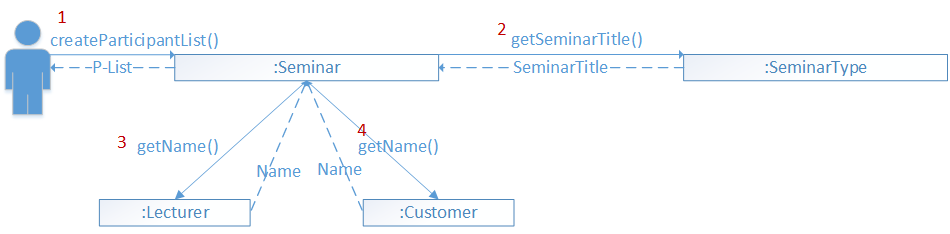
 На Фиг. 2. са представени четири доставчика на услуги за примера “Организация на семинар”. Клиентите могат да бъдат менъджер на семинари и обекта *Семинар*. Също така са представени и съобщенията, които тези обекти си разменят за да се изпълни искания сценарии (съставяне на списък с участниците в даден семинар). С непрекъснати стрелки са показани съобщенията, които се изпращат към обектите, а с пунктирани стрелки – отговорите на съобщенията. Не е ясна обаче последователността, в която обектите си разменят съобщенията.



Фигура 2: Изпращане на съобщения

**Сценарии**

Сценариите описват динамични последователности от операции в една система, в която съществува дефинирана подредбата на съобщенията. Един отделен use case може да бъде документиран чрез множество от сценарии. Ще разгледаме един сценарий, който може да се реализира по време на изпълнението на  createParticipantList() от примера “Организация на семинар” (Фиг.3.)



Фигура 3: Сценарии за създаване на списък с участници на даден семинар

Примерът е аналогичен на този от Фиг.2, като тук е обозначена последователността, в която се изпращат съобщенията между обектите. В началото актьорът изпраща съобщение, с което се изисква списъка на участниците в даден семинар (createParticipantList()), към обекта от тип Seminar. Този обект от своя страна изпраща съобщение към обект от тип SeminarType за получаване името на семинара (getSeminarTitle). Третото съобщение, което следва в този сценарий, е getName(), което се изпраща от обекта от тип Seminar към обекта от тип Lecturer за определяне името на лектора за този семинар. След това се извличат и имената на клиентите (участниците) на семинара, като обектът от тип Seminar изпраща съобщение към обекта от тип Customer (getName()). Имайки цялата информация, получена от трите обекта (Lecturer, Customer и SeminarType), обектът Seminar доставя искания списък с участниците в семинара на актьора “Mенъджер на семинари”.

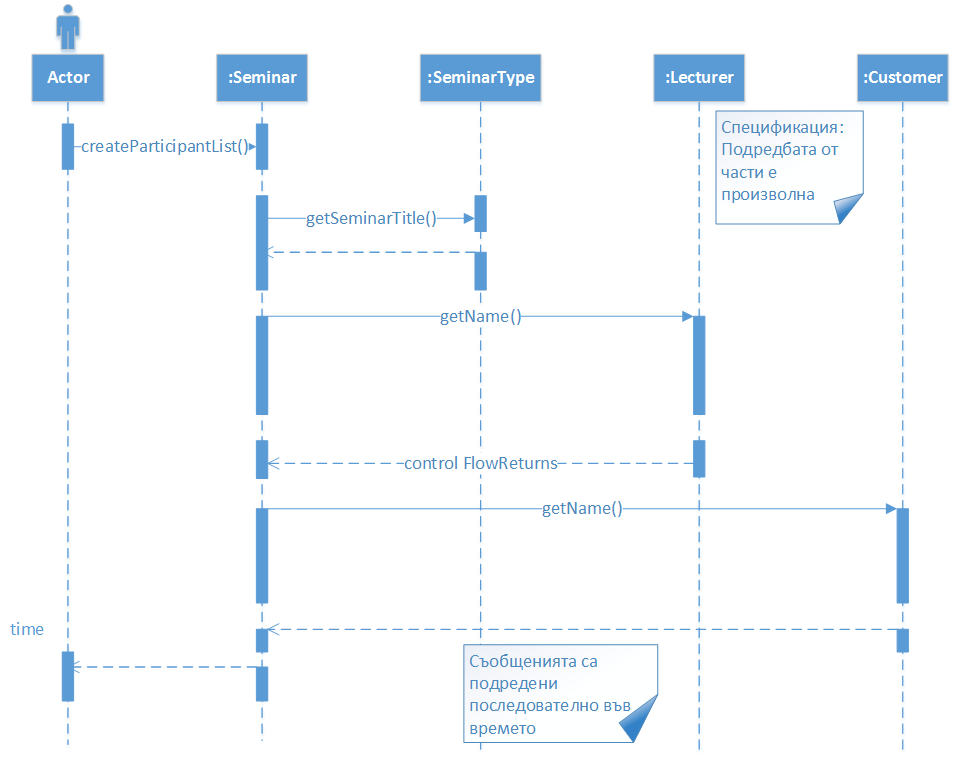
В езика UML сценариите се представят посредством interaction диаграми. Interaction диаграмите могат да бъдат два вида:

* Collaboration диаграми
* Sequence диаграми

**Sequence диаграми**

Sequence диаграмите са специфичен вид Interaction диаграми, които показват:

* Сценарии за някои от процесите в един use case
* Обектите, които участват в тези сценарии
* Съобщенията, които обектите си разменят, подредени последователно във времето.



Фигура 6: Sequence диаграма на сценария за създаване на списък с участници в даден семинар

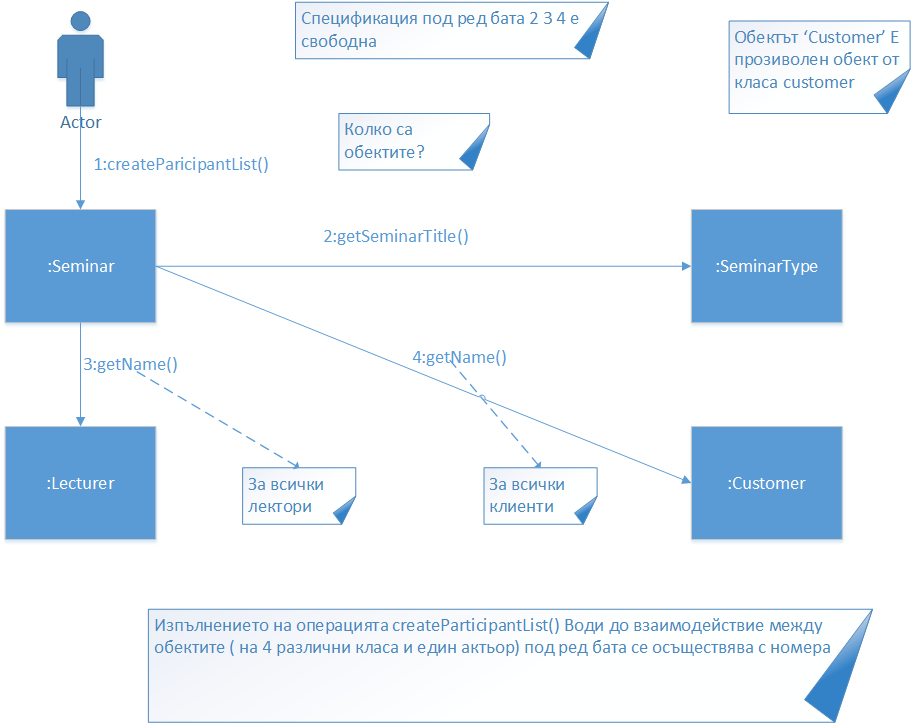
Четенето на тези диаграми се извършва отгоре надолу, като обектите, участващи в диаграмата, са показани най-отгоре. При sequence диаграмите не е необходимо номериране на съобщенията, защото те се изпращат така, както са подредени в диаграмата (първо се изпращта съобщението, намиращо се най-горе в диаграмата).

В примера (Фиг.6) е показана sequence диаграма на сценария за създаване на списък с участници в един семинар. Разликата между тази диаграма и съответната collaboration диаграма е в това, че тук по вертикала  се изобразява протичане на времето. Най-отгоре са разположени обектите, които участват в сценария (:Seminar, :Lecturer, :Customer, :SeminarType). Те си разменят съобщения в последователността, определена от тяхното подреждане.

Sequence диаграма на живота на обектите е дадена на Фиг.7. Диаграмата показва  момента на създаване на изобразения обект. Така например, обектът class3 сe създава след обектите от class1 и class2. За всеки обект в тези диаграми се изобразява също така линията им на живот (представя се като пунктирана линия). Линията на живот показва времевия интервал, в който съществува обектът. Когато един обект изпрати съобщение към друг обект, това се показва със стрелка, сочеща към обекта-получател (напр. operation1() ) . Освен това всеки обект може да изпраща съобщения сам на себе си (например operation3() ). Правоъгълниците върху линията на живот показват продължителността на действие на съответния обект, т.е. те показват интервалите, в които обектът е активен. На тези диаграми може да се представят и моментите на премахване на обектите. Това се показва чрез кръстче върху линията на живота на обекта.

**Collaboration диаграми**

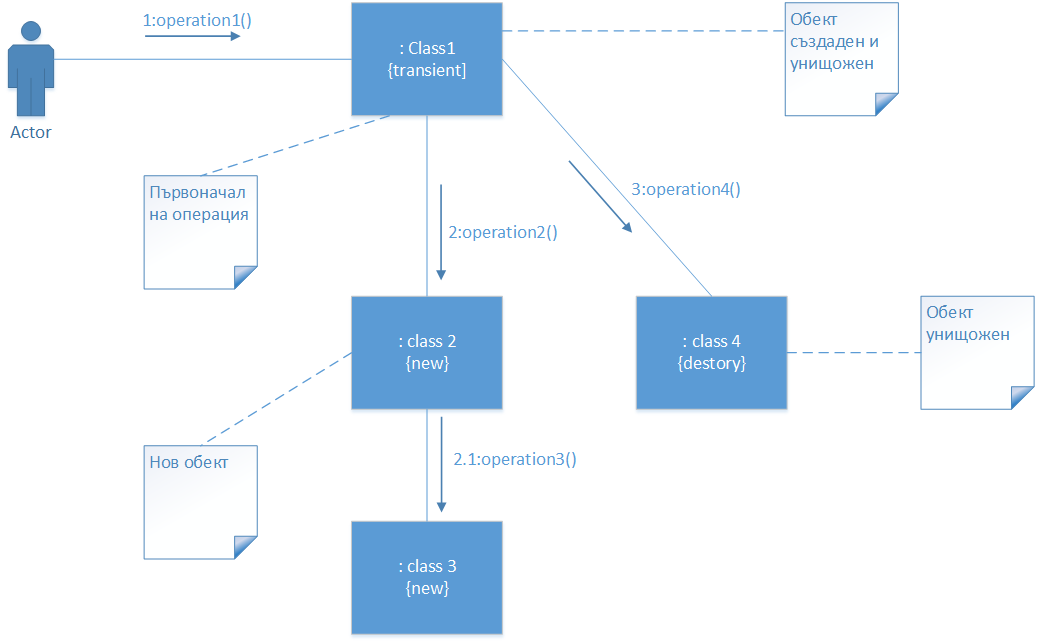
*Collaboration диаграмите* са подобни на нотацията, която използвахме в дадените по-горе примери. Тези диаграми се използват за поетапно моделиране на информационните потоци в един use case. Основните елементи на диаграмите са обектите и разменяните между тях съобщенията в рамките на един информационен поток. За да може да се управлява последователността на изпращане на съобщенията, те се номерират. Collaboration диаграма за сценария на функцията createParticipantList() е показана на Фиг.4



Фигура 4: Collaboration диаграма на сценария за създаване на списък с участници за даден семинар.

Collaboration диаграмите могат да се използват също така за моделиране живота на обектите. На Фиг.5. са моделирани:

* Създаване на обект
* Премахване на обект
* Създаване и последващо премахване на обект.



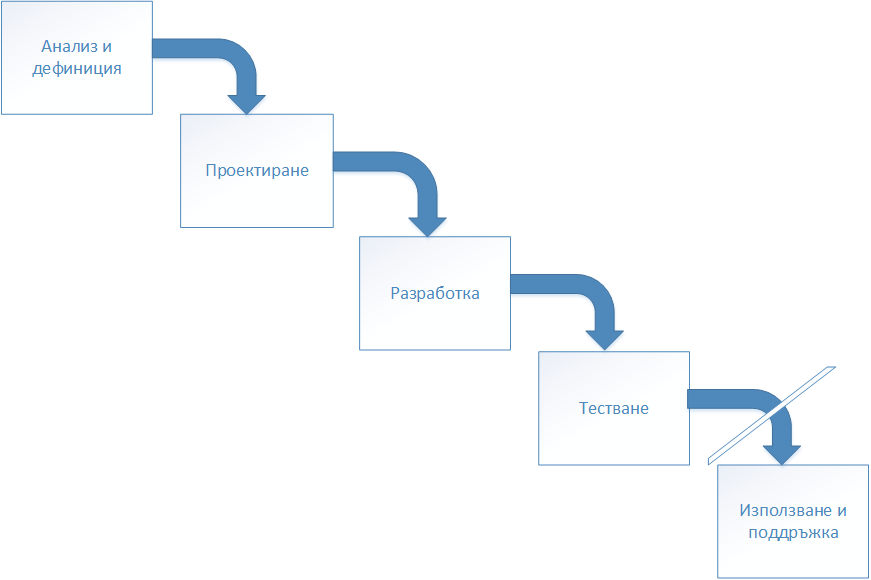
Фигура 5: Абстарактна collaboration диаграма

**Раздел 7:**

**Обща характеристика**

В тази глава се разглеждат резултатите от фазата “Анализ и дефиниция” от класическия “водопаден” модел за разработка на софтуер, представен накратко в глава 3. Трябва да отбележим, че принципите и документите, които разглеждаме за отделните фази на модела, са приложими и за всеки друг модел на софтуерния процес. В главата ще разгледаме подробно как се разделя фазата “Анализ и дефиниция”, какви документи се разработват по време на всяка подфаза и необходимостта за това разделяне. Важно е да се отбележи, че последователността, в която се разглеждат документите, не винаги отговаря на тази на тяхното създаване в реалните софтуерни проекти.

Анализът и дефиницията е първата фаза на класическия водопаден модел, представен на Фиг.1.



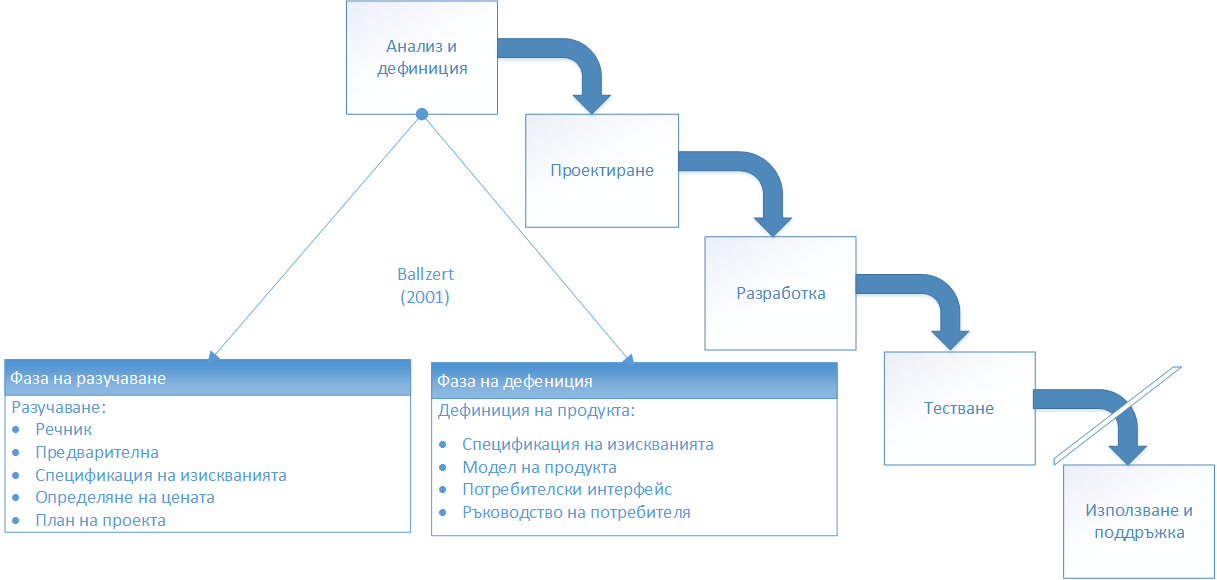
Основната цел на тази първа фаза е да се направи описание на външното поведение на разработваната софтуерна система, като се решат следните две задачи:

* Анализиране на проблема който трябва да бъде решен
* Дефиниране на изискванията към софтуерния продукт

Фигура 1: Водопаден модел

Тази фаза може да бъде разделена на две подфази (Фиг.2.):

* Фаза на планиране
* Фаза на дефиниция



Основният резултат от анализа и дефиницията е едно множество от документи, основните от които са следните:

* Документи които се използват основно за планиране на разработките - такива документиса речник предварителна спецификация на изискванията определяне на цената
* Документи които се използват обикновено за дефиницията на разработваните продукти - напр. спецификация на изискванията модел на продукта потребителски интерфейс

Фигура 2: Фаза на анализ и дефиниция

Резултатите от анализа и проучването на проблемната област се използват като основа за подписване на договор за разработка. Дефиницията на продукта е основен документ за последващия развой. За представяне на документацията не съществува един общ подход – използват се различни нотации за дефиниране и спецификация на софтуерните продукти и изискванията към системите. Използваните представяния и средства до голяма степен зависят от конкретната предметна област и участващите в развоя фирми. Получените резултати по време на фазата анализ и дефиниция могат да бъдат основание и за вземане на решение за прекратяване на проекта.

Фазите се характеризират със следните артифакти:

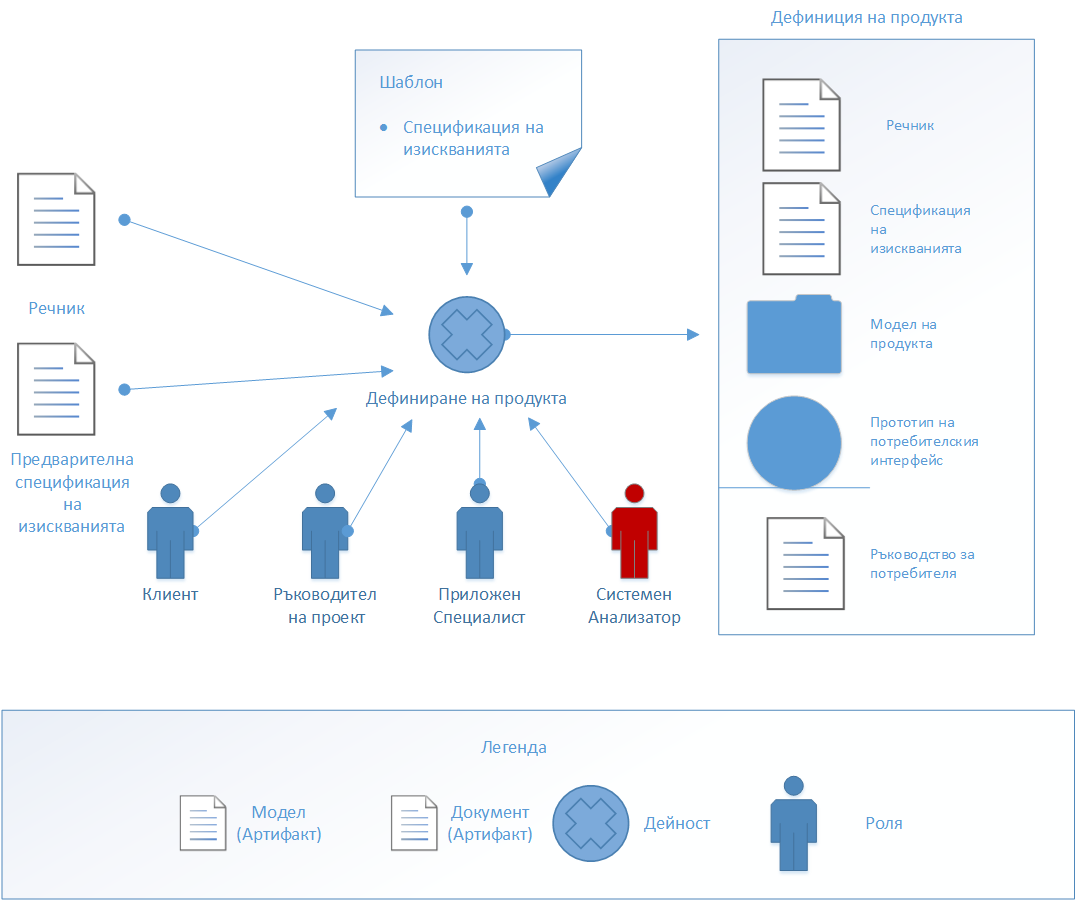
* Дейности
* Роли
* Продукти

**Фаза на дефиниция**

Втората основна дейност във фазата “Анализ и дефиниция” е дефиниране на продукта (Фиг.5). Тази дейност е свързана със създаването на четири основни документа:

* Спецификация на изискванията
* Модел на продукта
* Прототип на потребителския интерфейс
* Ръководство за потребителя.

Определяща роля при дефиницията на продукта имат системните анализатори. Като основа за разработване на посочените по-горе документи се изполват създадените вече (по време на планирането) предварителна спецификация на изискванията и речник на проекта. Както при планирането тук също така могат да се използват шаблони за създаване на документите, които са съобразени с някой от официалните стандарти или конкретен фирмен стандарт.



Основните участници в тази фаза са:

* Потребител (клиент)
* Ръководител на проект
* Приложен специалист
* Системен анализатор.

Фигура 5: Фаза на дефиниция

Таблица на Фиг.6 показва взаимозависимостите между ролите и артифактите за фазата на дефиниция, породени от отговорностите и сътрудничеството.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Дейности** | **Клиент** | **Ръководител проект** | **Приложен специалист** | **Системен анализатор** |
| Речник | с | c | o | c |
| Спецификация на изискванията | с | c | o | c |
| Модел на продукта | с | c | c | o |
| Прототип на потребителския интерфейс | с | c | o | c |
| Ръководство за потребителя | с | c | o | c |

Легенда:

* с – сътрудничество
* о - отговорност

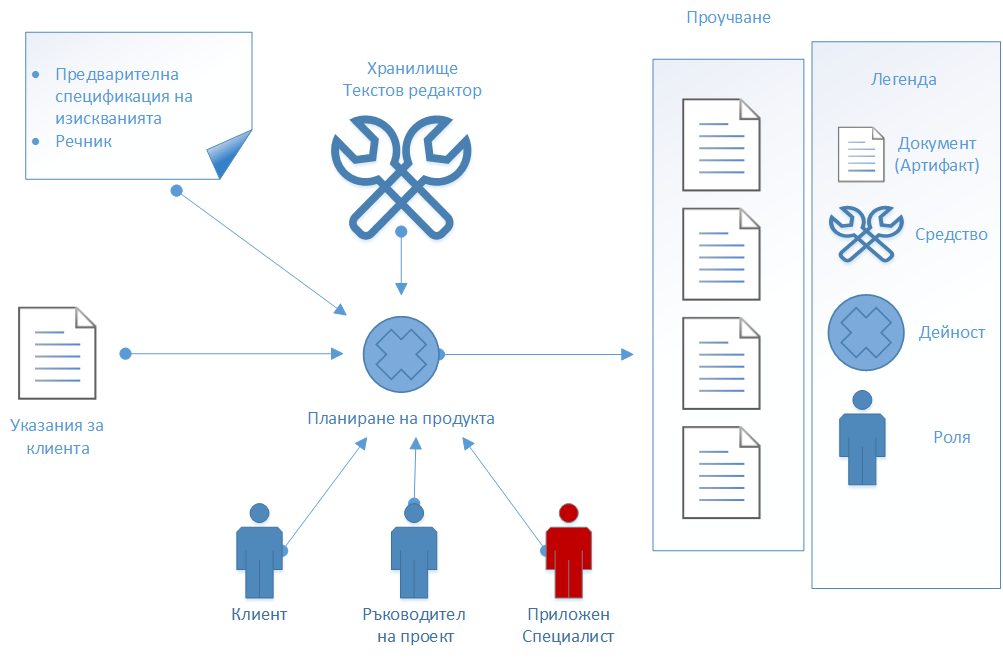
Фигура 6:Отговорности и сътрудничество между участниците във фазата на дефиниция

Речникът дефинира общата терминология, използвана по време на развойния процес. Той се използва за разработване на потребителския интерфейс, модулите за помощ (help) и ръководството за потребителя.

Определянето на цената на проектите се разглежда в следващата глава. Планирането на проектите е една по-специфична тематика, която би излязла извън рамките на учебника.

**Фаза на планиране**

Взаимозависимостите между трите основни елемента, характеризиращи фазата на планиране са дадени на Фиг.3.



Основната дейност в тази фаза е планиране на продукта. Планирането е свързано със създаването на четири основни документа:

* Предварителна спецификация на изискванията
* Речник на проекта
* Определяне на цената за проекта
* Разработване план на проекта.

Фигура 3: Фаза на планиране

Основен източник за разработване на тези документи в планиращата фаза са изискванията на потребителите. Основните субекти на фазата са потребителите, ръководителят на проекта и приложните специалисти. За целите на планирането участниците в развойния процес могат да използват различни шаблони и помощни средства. Шаблоните представляват формуляри на документи, които трябва да бъдат попълнени от участниците в развоя. Една добра практика е шаблоните да са разработени според някой от официалните стандарти (например: IEEE) или според някакав вътрешен фирмен стандарт на разработчика. За планирането на продукта са целесъобразни шаблони за предварителната спецификация на продукта или за речника. Използваните стедства обиновено са различни типове хранилища и текстови редактори.

Таблицата на Фиг.4. показва ролите, артифактите и връзката между тях във фазата на планиране на продукта. Както вече беше споменато, основните роли в тази фаза са: потребител (клиент), ръководител проект и приложен специалист. Основните артефакти, които трябва да бъдат създадени са : предварителната спецификация на изискванията, речник, определяне на цената и план на проекта. В таблицата са обобщени отговорностите на отделните роли. Така напр. ръководителят на проекта е отговорен за създаването на план на проекта и за определяне на цената за проекта, като за тази цел той трябва да си сътрудничи с потребителя и приложния специалист. Приложният специалист от своя страна отговаря за създаването на предварителна спецификация на изискванията и речник на проекта, за която дейност той си сътрудничи с потребителя и ръководителя на проекта. Видно е също така, че потребителят няма никакви определени отговорности, като основната му задача е да сътрудничи на всички участници в тази фаза и по този начин да съдейства за разработването на коректен софтуер.

Тази фаза може да бъде разделена на две подфази (Фиг.2.):

* Фаза на планиране
* Фаза на дефиниция

|  |  |  |  |
| --- | --- | --- | --- |
| **Дейности** | **Клиент** | **Ръководител проект** | **Приложен специалист** |
| Предварителна спецификация на изисквания | с | c | o |
| Речник | с | c | o |
| Определяне на цената | с | o | c |
| План на проекта | с | o | c |

Легенда:

* с – сътрудничество
* о - отговорност

Фигура 4: Отговорности и сътрудничество на участниците във фазата на планиране

Речникът дефинира общата терминология, използвана по време на развойния процес. Той се използва за разработване на потребителския интерфейс, модулите за помощ (help) и ръководството за потребителя.

Определянето на цената на проектите се разглежда в следващата глава. Планирането на проектите е една по-специфична тематика, която би излязла извън рамките на учебника.

**Спецификация на изискванията**

Спецификацията на изискванията описва две основни групи изисквания:

* Функционални - описват задачите (функциите) на приложението. Това е основната част от спецификацията на изскванията.
* Нефункционални - други важни за крайния продукт изисквания описващи различни негови свойства.

Този документ е предназначен главно за разработчиците, но може да бъде използван също така от потребителите. Той трябва да бъде написан в терминологията на разработчиците (не на тези на потребителя). Спецификацията на изискванията се използва като изходна точка за проектирането или разработването на продуктите.

Документът съдържа обикновено следните данни:

* функционални изисквания - функционалност данни (логически изглед) потребителски интерфейс
* изисквания на средата на приложението - например потребителски профили
* технически изисквания - език за разработка операционна система хардуер
* изисквания за производителност - производителност количество данни
* изисквания за валидност - разработване на тестове тестови случаи
* изисквания за качество - удобен за използване интерфейс надеждност ...
* изисквания за реализацията - модел на процеса документация срокове цена

Така напр. стандартът IEEE SRS (1) (Software Requirements Specification) определя следното съдържание на документа ”Спецификация на изискванията”:.

* Въведение - описва целите областта дефинициите използваните акроними и съкращения референциите и общия преглед на разрабитваните продукти
* Общо описание - съдържащо изглед на продукта функции на продукта потребителски характеристики ограничения допускания и зависимости
* Специфични изисквания - могат да бъдат различно структурирани в зависимост от приложението но независимо от избраната структура да съдържа следната информация:
  + изисквания за външния интерфейс
  + функционални изисквания
  + изисквания за производителност
  + проектни ограничения
  + критерии за качество
  + други изисквания

**Предварителна спецификация на изискванията**

Основната задача на предварителната спецификация на изискванията е да се представят:

* основните функции на продукта
* основните данни
* общата производителност за продукта
* някои съществени аспекти от потребителския интерфейс
* критерии определящи качеството на продукта

Този документ е ориентиран към потребителя и трябва да бъде четен и разбран от него, т.е. той трябва да бъде написан в терминологията на потребителя (не в тези на разработчика).

Предварителната спецификация на изискванията е въведена като подмножество на спецификацията на изискванията. Този документ хронологично се разработва преди спецификацията на изискванията.

### Речник

Специфицирането по-точно на понятията, описани в началните изисквания. Тези основни понятия ще послужат за изясняване на това кой точно ще използва разработваното приложение. За целта ще трябва да се разработи една първа версия на документ, съдържащ основните понятия (речник)

### Модел на продукта

По време на фазата на дефиниция се разработва модел на продукта, който съответства на типа анализ – структурен или обектно-ориентиран. При структурен анализ се използва йерархия от DFD (диаграми на потока на данните), а при обектно-ориентиран анализ това са – клас диаграми, sequence и collaboration диаграми, activity, use case диаграми, които формират статичния и динамичен модел на приложението.

Повече информация за модела на продукта може да се види съответно в лекциите за структурен анализ и обектно-ориентиран анализ

**Раздел 8:**

**COCOMO**

1. Необходимост и цели

Всеки ръководител на софтуерен проект би желал във възможно най-ранен етап от жизнения цикъл на разработвания програмен продукт да знае колкото се може по-точно какви разходи ще трябва да се направят до завършването му и колко ще продължи целият процес.  
С този проблем теоретиците започват да се занимават още през 60-те години и в резултат на това се появява и първият метод за оценяване разходите по производството на даден програмен продукт  - SDC, 1965. Последват го още няколко метода със съответни модели, докато се стига до 1981, когато се появява COCOMO, предложен от Боем. Тази разработка се смята за фундаментална за практическото използване на такива методи и изследвания в областта на икономиката на разработването на софтуер.  
Един от много важните приноси на Боем в неговата разработка е, че той формулира множество от критерии, които следва да удовлетворява даден модел за установяване цената на разработване на софтуерни продукти.

1. Критерии

Критериите дефинирани от Боем са следните:

* + Определеност – този критерии означава точно и обективно определяне на началните данни и понятия, както и на крайните резултати и количествени характеристики. Много е важно да се знае кои са точно фазите на процеса и какво е тяхното съдържание , какви функции се изпълняват по време на всяка фаза и в какво точно се състои всяка от тях.
  + Точност – има се предвид съответствието между предсказаната от модела цена на разработване и реално получената накрая.
  + Обективност – както и при други оценяващи процедури е важно постигането на възможно най-голяма обективност, или иначе казано – максимално избягване на субективния фактор.
  + Детайлност – по принцип колкото един модел е по-подробен, толкова той е по-адекватен на реалните обекти и процеси и следва основаният на него метод да даде по-точни резултати. За съжаление по-дълбоката детайлност изисква повече ресурси ( време, хора, пари).
  + Устойчивост – определят се граници на приложение на метода;
  + Област на приложение – няма универсален модел за определяне на цената. Следователно за всеки модел трябва да се определи областта на приложение.
  + Конструктивност – всеки модел трябва да дава възможност да бъдат анализирани и разбрани получените резултати.
  + Простота на прилагане – този критерии определя степента на трудност на разбирането и получаването на входните данни, както и степента на трудност на изпълнение на процедурите по оценяване.
  + Предсказуемост -  този критерии засяга проблема за практичното използване на моделите. Важно е за входни данни на метода да се изберат точни и ранно предсказуеми фактори.
  + Икономичност  - от една страна желателно е входните данни за оценката да бъдат по-малко на брой и лесно измерими, от друга – да не са прекалено малко, така че да доведат до фатална неточност на крайните резултати.

1. Моделът на Боем COCOMO

COCOMO произлиза от Constructive Cost Model.  Основната му цел е за всеки планиран софтуерен проект да се оцени цената и срокът на разработване. Основополагащата му идея е използването на броя редове първичен код.

**Същност на модела**  
За изчисляване на усилията се използва формулата:  
ЧМ = 2.4 x ХРПК1.05  
ЧМ – брой човекомесеци  
ХРПК – хиляди реда първичен код  
За оценка на продължителността на разработване на софтуерния проект формулата е:  
В = 2.5 x ЧМ0.38,  
където В е срокът на разработване в месеци.  
Формулите се прилагат при следните предположения:

* + - Редовете първичен код се броят без коментираните редове; принадлежат на крайният продукт; не включват използваните стандартни програми;
    - Включват се фазите проектиране, програмиране и оценка включително усилията по управлението и документирането;
    - Смята се че един човекомесец е от 19 дни или 152 часа;
    - Не се правят сериозни промени в спецификацията на изискванията след одобрението и;

Пример:  
Нека имаме резултат от предварителна експертиза, че бъдещ софтуерен продукт ще има 32000 реда код =>  
ЧМ = 2.4 x 321.05 = 91 човекомесеца  
В = 2.5 x 910.38 = 14 месеца =>  
Производителността = 32000 / 91 = 352 LOC за човекомесец  
Екип = 91 / 14 = 6.5 човека  
Тълкуването на последната бройка е ясно – един или повече специалисти от екипа ще бъдат заети с проекта не през всичките 14 месеца на разработката.

Усъвършенстване на модела  
Първото разширение на модела на Боем е въвеждането на 3 типа софтуерни проекти – разпространен, полунезависим и вграден (organic, semidetached, embedded). За всеки от тях формулата е различна. В таблица 1 всеки тип се илюстрира кратко, илюстрира се с примери и му се съпоставя предложената от Боем формула.

**FPM**

4. Метод на функционалните точки

Методът на функционалните точки е предложен през 1979г. от Олбрихт. Основният мотив на автора са недостатъците на фактора брой редове първичен код. Като най-съществени недостатъци на COCOMO (основният представител на модели броящи редове код) са трудната му ранна определеност и липсата на единно виждане за ред първичен код. Поставяйки си за цел да намери по-добър фактор в това отношение, Олбрихт достига до понятието “функционална точка”. Главната му идея е, че усилията за разработката на даден софтуер се определят от неговата функционалност. Последната може да бъде измерена на основата на въпросните функционални точки. Доколкото функционалните точки могат да бъдат определени с помощта само на някои първоначални проектни документи, това означава и ранно получаване на желаните оценки.  
Всъщност Олбрихт сам формулира явно 5 цели, които се стреми да постигне със своето предложение за модел и метод:

* + Да се използват външните характеристики на софтуера;
  + Да се третират средства важни за крайния потребител;
  + Да може да се прилага в ранни фази на производствения процес;
  + Да може да се свърже лесно с икономическа оценка;
  + Да има независимост от редовете първичен код.

Същност на модела

Моделът  се основава на 5 функционални типа.  Това са 5 непресичащи се класа обекти с точно определени функции. Допълнително се определят по 3 нива на сложност за всеки тип – просто, средно и сложно. Тези типове са следните:

* Външен входен тип – всеки потребителски входен управляващ поток или поток от данни. Този тип може да бъде:
  + прост – съдържа малък брой типове данни и те се отразяват върху малък брой вътрешни логически типове;
  + междинен – който не може да се определи еднозначно като прост или сложен;
  + сложен – съдържа значителен брой типове данни и тези данни се отразяват върху голям брой вътрешни логически данни

Пример – входните екрани. Въвежданите чрез тях данни “навлизат” в приложението и засягат един или повече вътрешни логически типа.

* Външен изходен тип - всеки потребителски изходен управляващ поток или поток от данни. Примерни такива могат да бъдат различните видове съобщения до потребителя или изходни отчети. Класификацията е като при предходния тип.Допълнително отчетите се класифицират на:
  + прост – съдържа една или две колони, данните почти не се преобразуват при извеждането им;
  + междинен – отчетът съдържа много колони;
  + сложен – резултатите в отчета се получават след сложни преобразования.
* Вътрешен логически файлов тип – такава е всяка съществена група от потребителски данни или управляваща информация. Пример са логическите файлове и въобще всяка логическа група от данни. Класифицира се на три нива:
  + прост – малко типове записи, малък брой типове елементи от данни;
  + междинен – типът не може еднозначно да се определи като прост или сложен;
  + сложен – голям брой типове записи, типове елементи от данни.
* Външен интерфейсен файлов тип – това е файл, който се предава или се ползва съвместно от две или повече приложения. Класификацията се прави според дефиницията за вътрешния логически файлов тип.
* Външен справочен тип – всяка комбинация вход/изход, при която входът предизвиква незабавен изходен резултат. Като пример това са заявките. Класификацията се определя така:
  + входната част се класифицира според определенията за външния входен тип;
  + изходната част се класифицира според определенията за външния изходен тип;
  + сложността на външния справочен тип е равна на по-голямата от така определените две сложности.

Процедура по пресмятането

Преброяват се елементите на всеки от 5-те типа. След това за всеки елемент от всеки функционален тип се определя нивото на сложност. С получените данни се попълва таблицата за пресмятане (таблица 3).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип** | **Наименование** | **прост** | **междинен** | **сложен** | **Общо** |
| IT | Външен входен | Х3 | Х4 | Х6 | ... |
| OT | Външен изходен | Х4 | Х5 | Х7 | ... |
| FT | Вътрешен логически файлов | Х7 | Х10 | Х15 | ... |
| EI | Външен интерфейсен файлов | Х5 | Х7 | Х10 | ... |
| QT | Външен справочен | Х3 | Х4 | Х6 | ... |
| FC | Общо ненастроени функционални точки |  |  |  | ... |

Таблица 3: Таблица за пресмятането

Обозначението „X a“ в трите колони за сложностите означава, че получената за тази клетка бройка трябва да се умножи по теглото „а“.  
Като се сумират 5-те числа от най-дясната колона, се получава общият брой ненастроени функционални точки FC за изследваното приложение.   
Следващата стъпка е предизвикана от осъзнатата от авторите необходимост от съществени уточнения на така получения брой FC. Разглеждат се изброените по-долу 14 характеристики и за всяка от тях се определя степента и на влияние върху оценяваното приложение. Допустимите стойности за всяка характеристика са от 0 до 5 и имат следния смисъл:

* + 0 – не е налична или не влияе
  + 1 – незначително влияние
  + 2 – умерено влияние
  + 3 – средно влияние
  + 4 – значително влияние
  + 5 – силно влияние

Самите характеристики са следните:

1. Данните и управляващата информация се изпращат или получават по комуникационни линии;
2. Има разпределена обработка на данни;
3. Важно е достигането на висока ефективност;
4. Експлоатация върху силно натоварена операционна конфигурация – хардуер, софтуер
5. Интензивността на транзакциите е висока;
6. Наличен е интерактивен режим на въвеждане на данните;
7. Цели се ефективност от гледище на потребителя;
8. Наличен е интерактивен режим на актуализирането на данните;
9. Логиката на обработките е сложна;
10. Програмният код трябва да е reusable;
11. Цели се лесно инсталиране;
12. Цели се лесна експлоатация;
13. Може да се използва от разнообразни потребители;
14. Приложението е гъвкаво и лесно се модифицира.

Процедура по пресмятане

Определя се величината PC (processing complexity) като сума от стойностите на 14-те характеристики.  
На базата на PC се определя коригиращия коефициент PCA (processing complexity adjustment).  
PCA = 0.65 + (0.01 x PC)  
Крайният брой FP (функционални точки) се получава по формулата:  
FP = FC x PCA  
Лесно се вижда, че коригиращият коефициент PCA коригира първоначалния резултат с +/- 35%.

**Други модели**

5 Други модели

* Doty – наподобява COCOMO. Основава се на 2 формули:

ЧМ = 5.288 x ХРПК1.047 за ХРПК < 10  
ЧМ = 2.060 x ХРПК1.047 x f1 x f2 x…x f14 за ХРПК >=10  
Означенията са както при COCOMO.

* SPQR – подобен на COCOMO – Кейпърс Джоунс Основава се на 20 добре дефинирани и 25 недостатъчно добре дефинирани фактора, определящи цената на софтуера.  Не е добре документиран. Потребителят трябва да отговори на 100 въпроса свързани с проекта, чиито отговори служат за входни параметри. Създателят твърди че метода дава +/- 15% отклонение.
* ESTIMACS – съсредоточава се върху разработването – Рубин – 15% отклонение. Състои се от следните модули:
  + Оценител на усилията за разработване на системата – 25 въпроса за проекта;
  + Оценител на разходите за колектива разработчик – входни данни са получените от предната точка;
  + Оценител на хардуерната конфигурация – като вход се подават изискванията на приложението за ОС, очаквания брой и тип на транзакциите, а като резултат се получава оценка за необходимата хардуерна конфигурация;
  + Оценител на риска. – отговор на 60 въпроса.
* BANG – Де Марко. Принцип на функционалните точки. Цел – системен софтуер и научни приложения.Основните елементи които се оценяват и броят са:
  + Функционални примитиви
  + Модифицирани функционални примитиви
  + Елементи от данни
  + Входни елементи от данни
  + Изходни елементи от данни
  + Елементи от данни в паметта
  + Обекти
  + Връзки
  + Фактори свързани със спецификата на системата в реално време – преходи, състояния на чакане.
* Моделът MOOSE (Metric for object-oriented system environment)

**Раздел 9:**

**Структурен анализ**

**Основни принципи**

С понятието *структурен анализ*(SА, structured analysis) се обозначава един клас от методи, подходящи за структурно програмиранe. Въпреки, че тези методи се причисляват към остарелите, те са индустриален стандарт и са все още широко използвани. Така напр.:

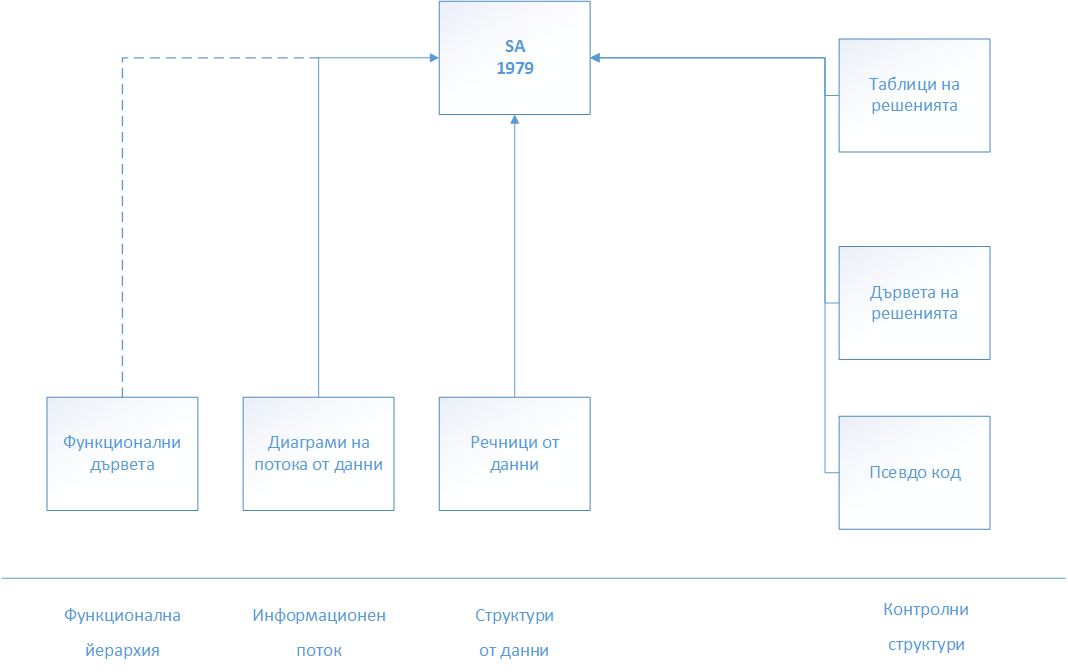
* В САЩ половината от системните аналитици (използващи по принцип някой от методите) използват SA
* В DaimlerChrysler през 2000 год. в 70% от всички съществуващи проекти се използва структурен анализ.

Една интересна обосновка за използване на старите методи може да се прочете в дадения на Фиг.10.1. цитат от препоръките на *Германската общност на информатиците* за ролята на остарелите методи – „Знанията за остарелите методи са важни за намиране на причини за въвеждане на нови модерни методи.“

Широкото разпространение на този метод се базира на процедурно-алгоритмично ориентирания поглед към системата, където:

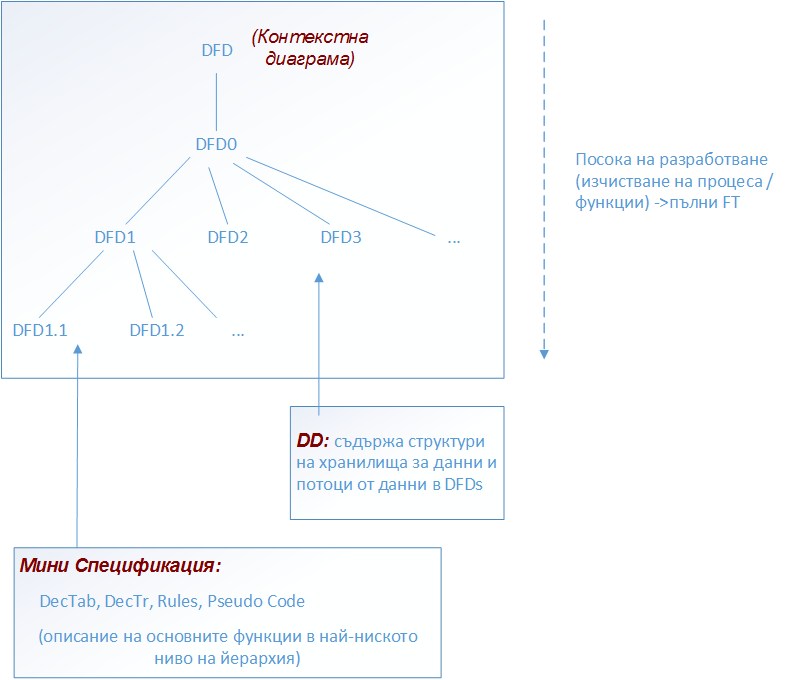
* Данните и функциите се моделират като самостоятелни единици
* Потоците от данни описват трансформиращото поведение на функциите
* Съсредоточен е около *диаграмите на потока от данни*.

Основната цел на SA е да се построи формализиран модел, който ще бъде част от дефиницията на разработвания софтуер. Структурният анализ не въвежда нови базови концепции. Новото в него е начинът, по който се комбинират известните базовите концепции (Фиг.1.), за постигане на предимствата при осигуряване качеството на продуктите. Особено съществена е възможността за предостяване на ефективна CASE (Computer Aided Software Engineering) поддръжка.



Фигура 1: Структурен анализ – комбинация от основни концепции

Един съществен проблем при използването на диаграмите на потоци от данни е, че при по-големите задачи те обхващат много страници и стават непригледни. Идеята на структурен анализ е тези диаграми да се детайлизират на отделни нива и да се представят като йерархични структури. Започвайки от една абстрактна диаграма на потока на данните, наречена *контекстна диаграма*, отделните процеси се детайлизират и се описват посредством собствени диаграми. Ако един процес не може да бъде повече декомпозиран на подпроцеси (лист в йерархията), тогава той може да се представи посредством така наречените *мини-спецификации* (MiniSpecs). Мини-спецификациите могат да бъдат някакъв псевдо-код, таблици на решения или дървета на решения. Така може да се структурира модела на продукта и на описанията на задачите, както и да се покаже необходимата трансформация на поведението на цялата система и на нейните подсистеми (подзадачи). Структурният анализ е един итеративен процес. Отделните диаграми се означават посредством идентификатори, започващи с DFD (съкращение от Data Flow Diagram), след което следва цяло число, показващо нивото на абстракция в йерархията (Фиг. 2.).



Фигура 2: Йерархия при разработване на DFD

**Контекстни диаграми**

Контексните диаграми описват интерфейсите на моделираната система с околната среда. Спрямо синтаксиса и семантиката тези диаграми заемат особено положение в структурния анализ. Основната идея на контекстните диаграми е да се моделира  трансформацията на цялостното поведение на разработваната система, като:

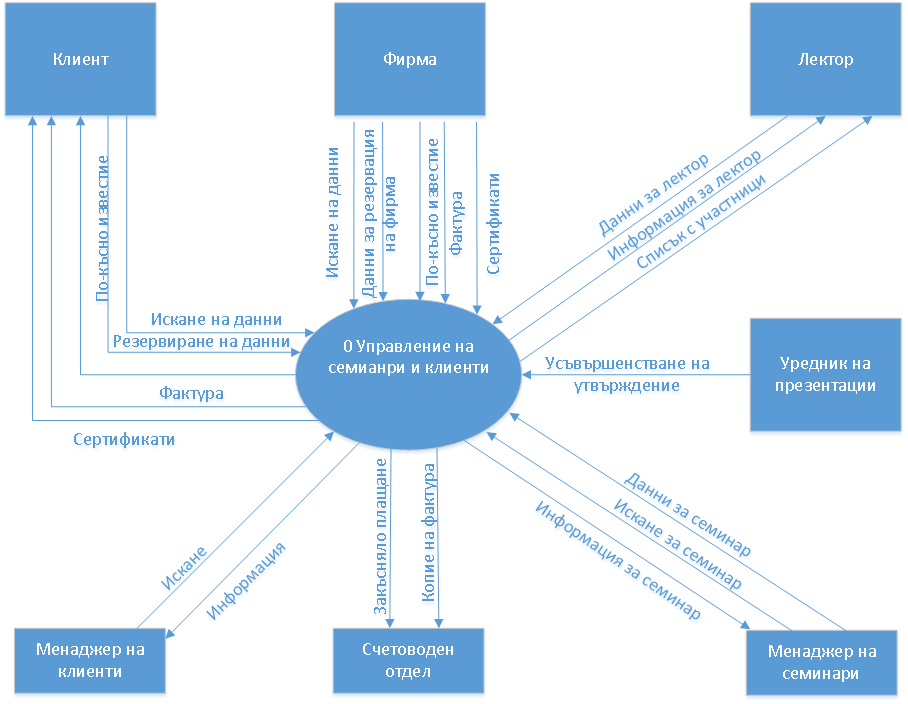
* Системата се представя като черна кутия
* Явно се представя входното/изходното поведение
* Представя се трансформацията на интерфейсната информация.

При създаване на конткстните диаграми трябва да се съблюдават следните синтактични правила (обикновено се проверяват от CASE средствата):

* Контексните диаграми трябва да представят само един процес, който моделира цялата система и има номер 0
* Диаграмите съдържат поне един интерфейс
* Между интерфейсите не съществуват потоци от данни
* Всеки интерфейс се представя само веднъж (по изключение, за подобряване на разбирането на диаграмите, може да се представи повече пъти)
* Диаграмите не съдържат хранилищата на данни.

Процесът 0 в една контексна диаграма (представляващ цялата система) може да се детайлизира, като се декомпозира на отделни подпроцеси и се представя в една нова диаграма (DFD 0)(fig.2).

На Фиг.3. е дадена диаграмата на основния процес (процес 0) за един пример от проекта “Организация на семинар”.



Фигура 3:

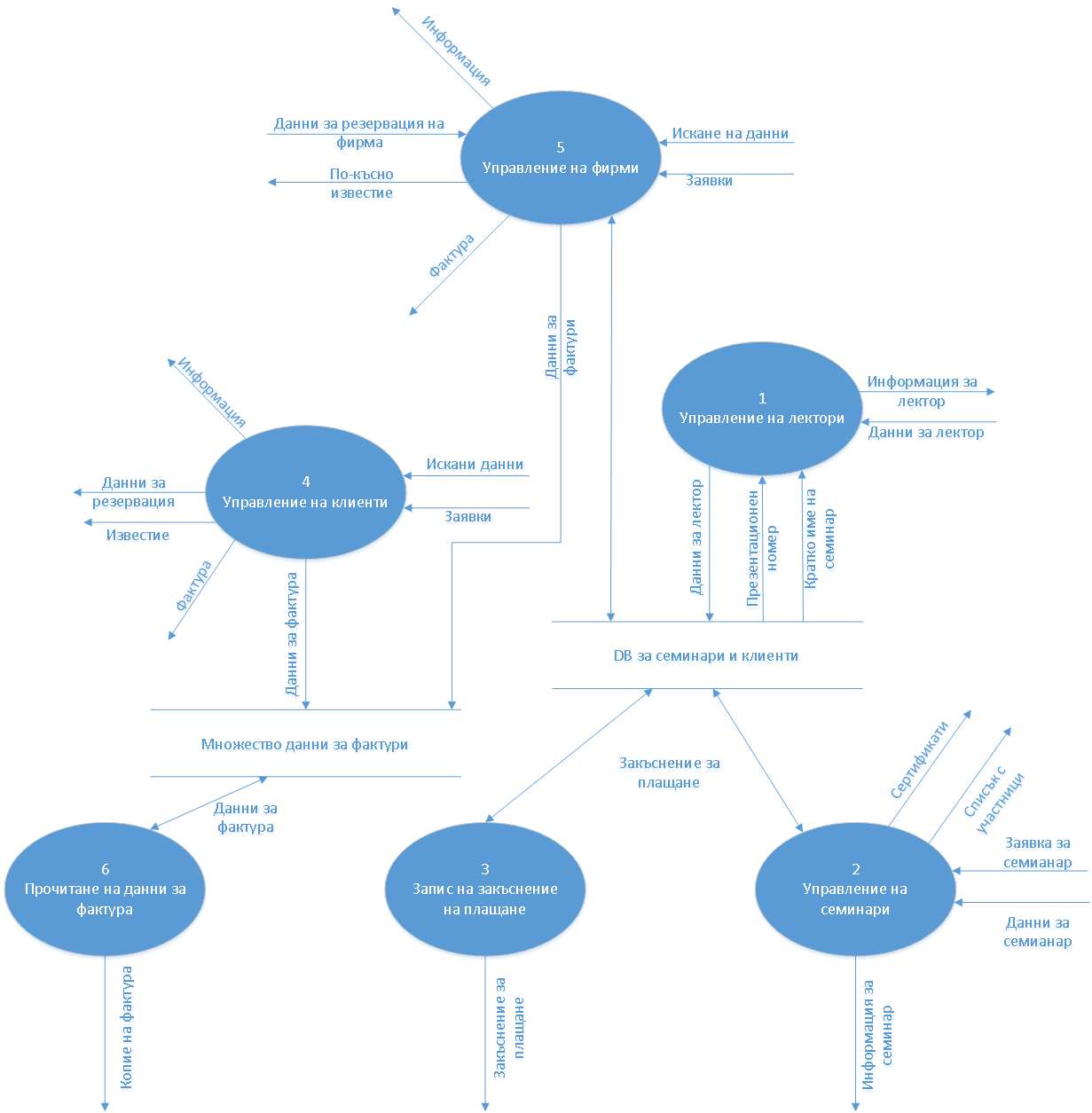
Основната функционалност “Управление на семинари и клиенти” се декомпозира на 6 подфункции в DFD0 (Фиг.4.):

* Управление на лектори
* Управление на семинари
* Запис на закъснение за плащане
* Управление на клиенти
* Управление на фирми
* Въвеждане данни за фактура.

Освен това са показани две от хранилищата на системата:

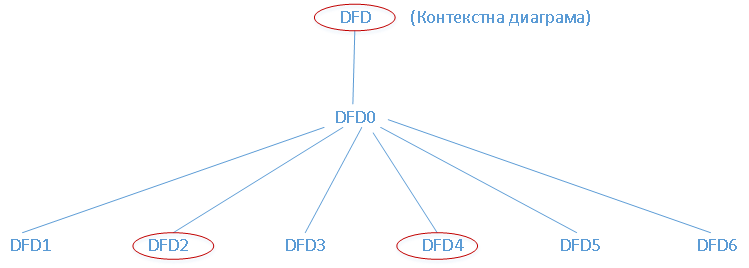
* База данни за семинари и клиенти
* Данни за фактури.

Представени са също така информационните потоци към и от отделните функции.



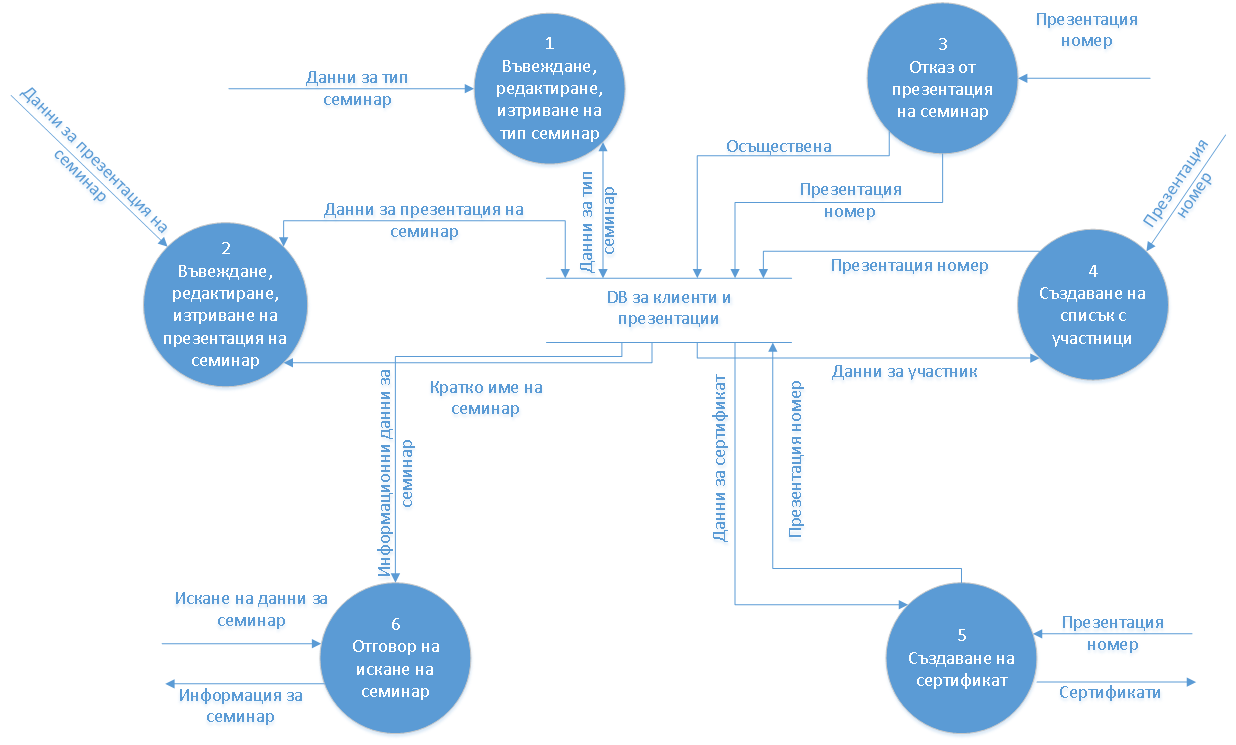
Фигура 4: DFD0: Управление на семинари и клиенти

Нека разгледаме по-подробно процеса на декомпозиция за този пример, който  следва маркираната на Фиг.5. последователност



Фигура 5: Декомпозиране на контекстната диаграма

Всяка една от DFD от 1 до 6 описва процес от DFD0. По-долу са представени DFD2 и DFD4.



Фигура 6: DFD2

Диаграмата DFD 2 (Фиг 6.) показва една възможна декомпозиция на функцията “Управление на семинари”. Тази функция се разделя на следните 6 подфункции:

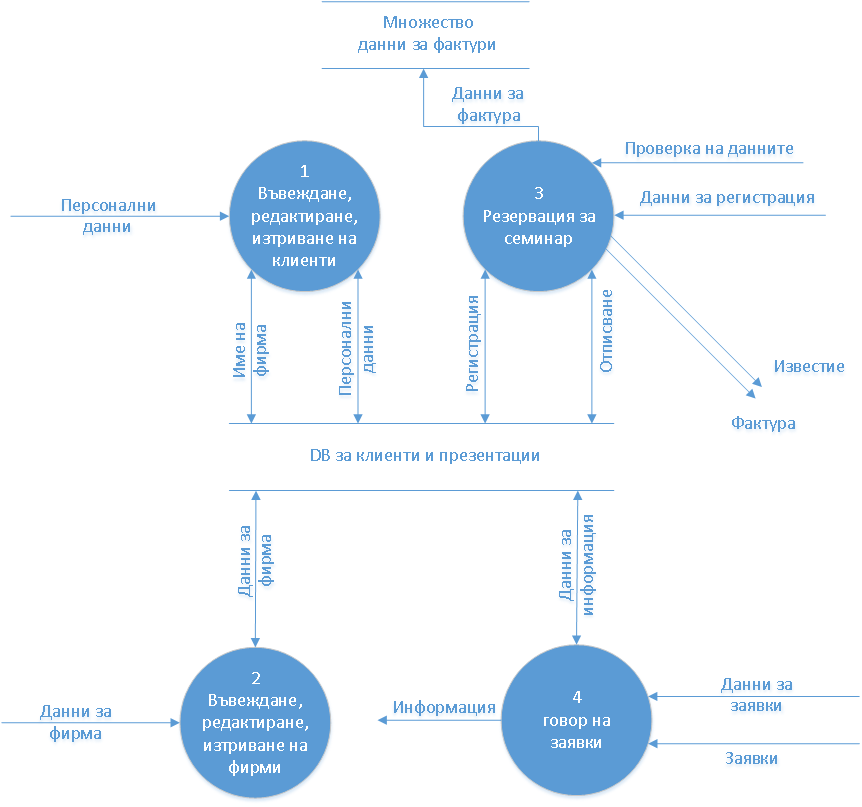
* Редактиране и изтриване на тип семинар
* Въвеждане, редактиране, изтриване на презентация на семинар
* Отказ от презентация на семинар
* Създаване на списък с участници
* Издаване на сертификат
* Отговор на искане за семинар.

В тази диаграма е представено и едно хранилище – база данни за клиенти и презентации, както и информационния поток в тази функционалност

Аналогична структура има и диаграмата DFD 4 (Фиг.7.), която представя декомпозицията на основната функция от DFD 0 “Управление на клиенти”. Тя се декомпозира в четири подфункции:

* Въвеждане, редактиране и изтриване на записи за клиенти
* Въвеждане, изтриване и редактиране на записи за фирми
* Резервация за семинари
* Отговори на заявки

Тя включва и двете хранилища – база данни с клиенти и презентации и данни за фактури.

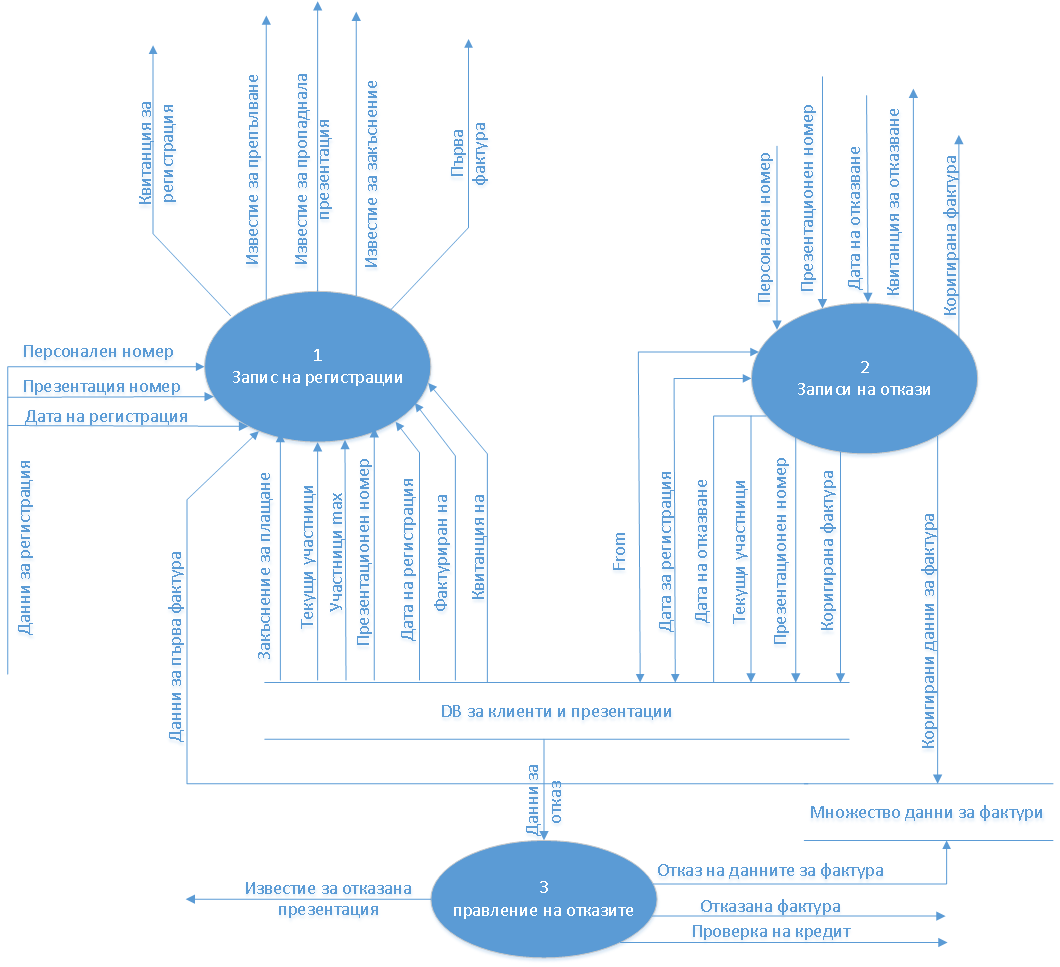


Фигура 7: DFD 7

След декомпозицията на DFD ще разгледаме диаграмите от по-ниско ниво. Като пример тук ще дискутираме усъвършенстването на диаграма DFD 4.3, която представя детайлизация на функционалността “Резервация за презентация” от диаграмата DFD 4.

DFD 4.3 (Фиг. 8.) се декомпозира на три подфункции:

* Запис на регистрации
* Запис на откази
* Управление на отказите



Фигура 8: DFD4.3

Като обобщение на това подробно разглеждане на примера можем да направим следните изводи:

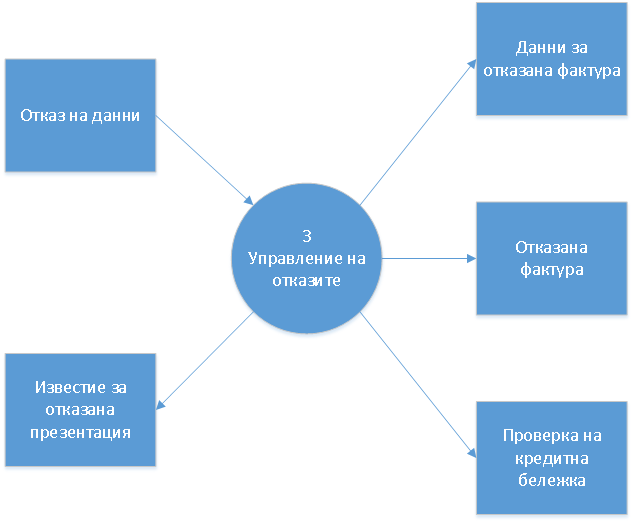
* Интерфейсите не могат да бъдат напълно изяснени
* Хранилищата също така не могат да се изяснят напълно, но те могат да бъдат повтаряни без да е необходимо да бъдат усъвършенствани на всяко отделно ниво
* Броят на процесив една диаграма не трябва да надвишава *7*
* Нивото на абстракция трябва да е едно и също за всички процеси и потоци от данни в една диаграма
* Процесите се номерират, започвайки от 1
* Всяка диаграма има номер, който показва позицията и в йерархията – така напр. DFD 4.3е детайлизация на процес 3от DFD *4*
* За уникална идентификация на DFD номерът й се поставя пред номера на процеса – напр. Процес 4.3.1означава Процес 1 от диаграма 4.3
* Системата за номериране обикновено се поддържа от комерсиалните CASE средства.

**Миниспецификации**

В определен момент последователната декомпозиция на процесите трябва да завърши, като  определени процеси (листата в дървото) останат без кореспондираща диаграма. Тези процеси могат да се опишат посредством мини-спецификации.  За мини-спецификациите са валидни следните правила:

* Всяка *MiniSpec* трябва да описва как входните данни се трансформират в изходни
* Една *MiniSpec* не трябва да съдържа правила за изпълнението
* *MiniSpecs* може да бъде псевдо код, правила, таблици на решенията или дървета на решенията

Една възможна мини-спецификацията за процеса 3 “Управление на отказите” от пример на Фиг.9. може да се представи посредством дадения на Фиг.10. псевдо-код.



Фигура 9: Спецификация на процес 3 от DFD4.3

**Read cancellation data from database;  
Send notification presentation canceled to all participants  
if participant already paid invoice  
then send canceled invoice and check credit note, too  
end if**

Фигура 10: Миниспецификация с псевдо код

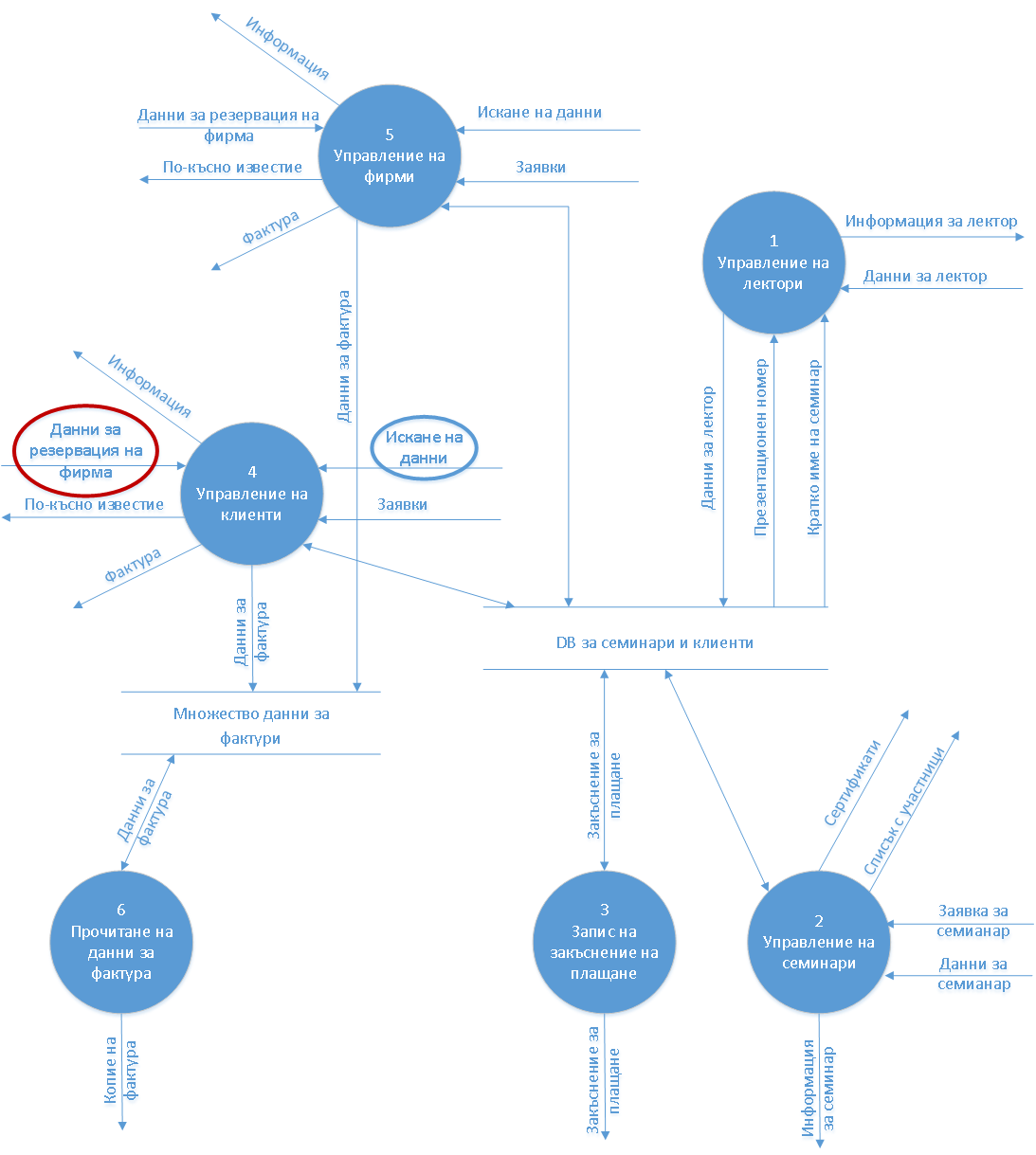
**Речници от данни и усъвършенстване на данните**

Функционалностите в една диаграма на потока от данни може да се детайлизира като за нея се специфицират подфункционалности или съответни мини спецификации. Паралелно с това трябва да се детайлизират и данните, свързани с тези функционалности. За тази цел трябва да се спазват следните правила:

* Всяка стрелка на потока от данни получава *име на потока от данни –*изключение е за данните, идващи от и отиващи към хранилищата и имащи достъп до цялото им съдържание
* Всяко име на поток от данни се дефинира в *речника на данните*
* Всяко хранилище има *име*
* Всяко име на хранилище се дефинира в *речника на данните*

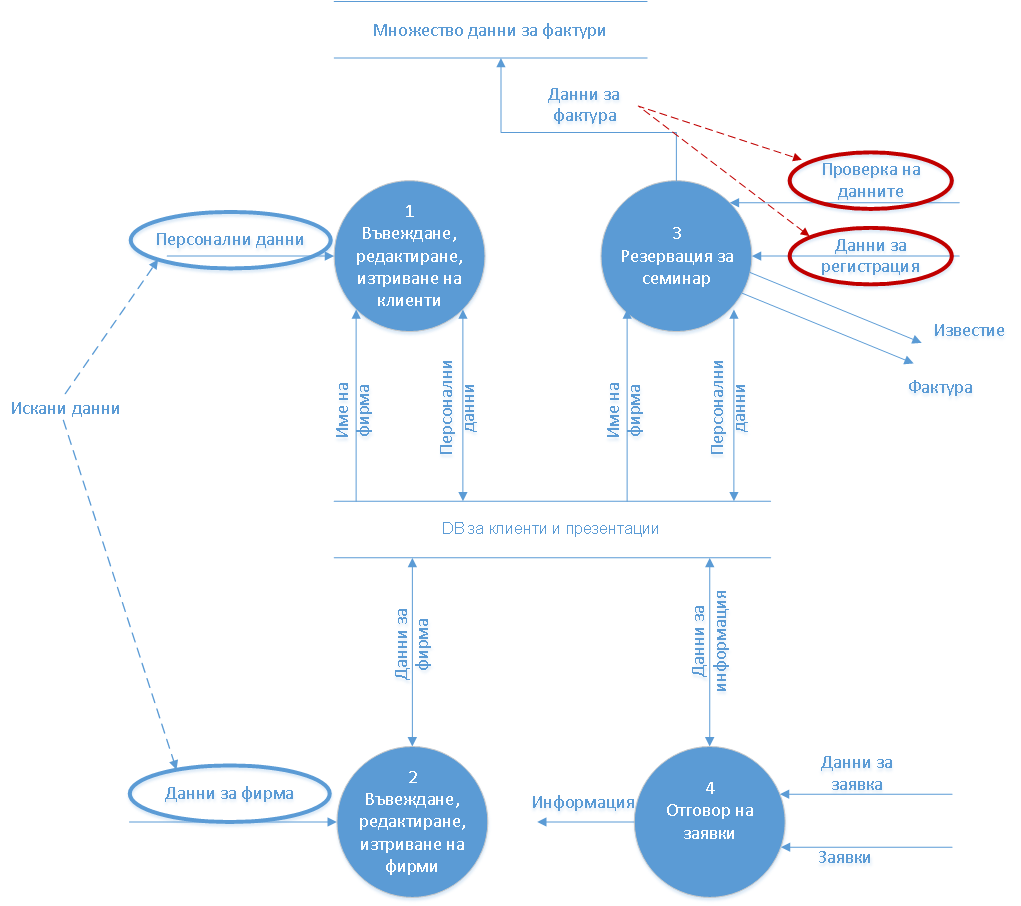
Примера долу е на DFD 0. Ще разгледаме усъвършенстването на потоците от данни от DFD 0 в DFD 4. Потока от данни “Искани данн” на фигурата е означен със синьо а “Данни за резервация” в червено. На следващата фигура ще видим как тези потоци са усъвършенствани.

**искани данни  = персонални данни + (данни за компанията)  
данни за резервация = данни за регистрация + проверка на данни**



Фигура 11: DFD0: Усъвършенстване на потока от данни в DFD4

Това е DFD 4 на примера за организация на семинар. Ясно се вижда горе споменатите потоци от данни как са усъвършенствани. Потока от данни “Искани данни” се е разделил на две – персонални данни и данни за фирма, а потока от данни “данни за резервация ” се е разделил на потоците проверка на данни и данни за регистрация.



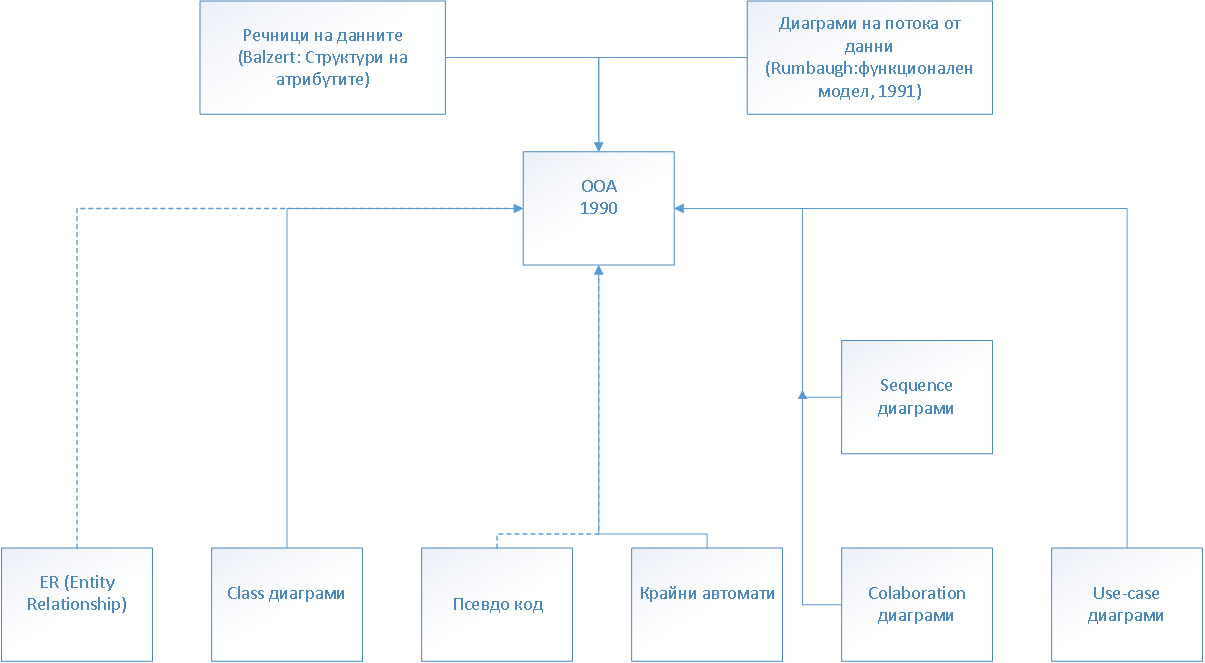
Фигура12: DFD4- усъвършенства на потоци от данни в DFD0

**Раздел 10:**

**Обектно-ориентиран анализ**

### Обща характеристика

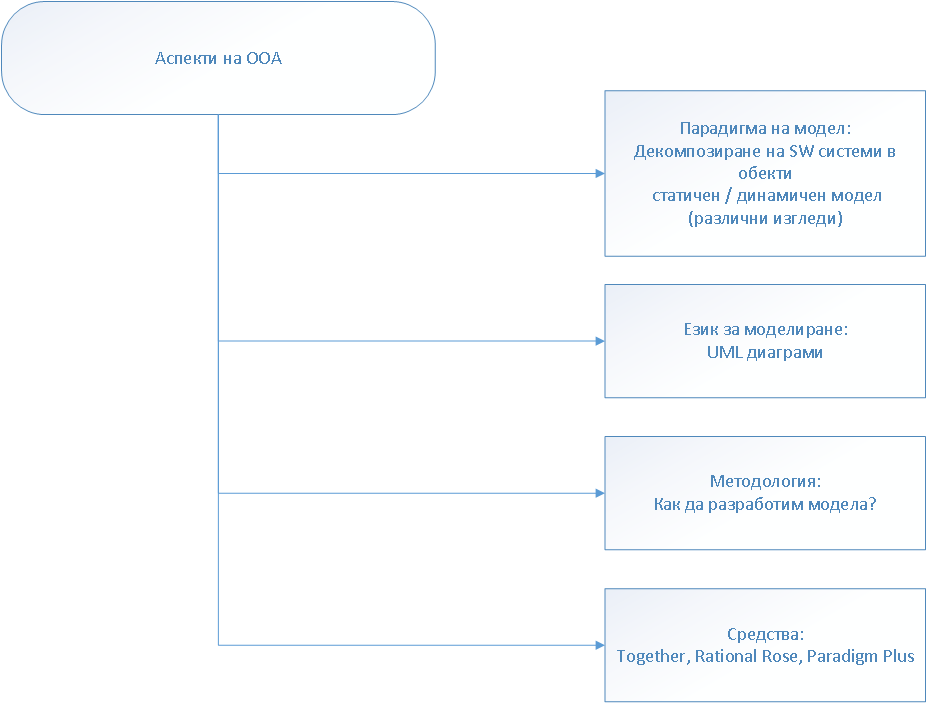
Както при структурния анализ целта на обектно-ориентирания анализ (ООА) е разработване на формализиран модел, кoйто е част от дефиницията на разработваната система. ООА моделирането се извършва с помощта на обектно-ориентирани средства. Обектно-ориентираният анализ е възникнал като комбинация от основните концепции, както е показано на Фиг 1.



Фигура 1: Основни концепции на обектно-ориентирания анализ

Една общоприета дефиниция определя обектно-ориентирания анализ като метод, представящ изискванията към софтуерния продукт като класове и обекти, които могат да се синтезират от проблемната област.

ООА може да бъде разгледан от четири различни аспекти (Фиг.2.).



Фигура 2: Аспекти на ООА

Първият аспект е парадигмата на модела, която се занимава с проблема за декомпозиране на софтуерната система на отделни обекти. В следствие на декомпозицията се получават два модела на системата, представящи две различни гледни точки. Това са статичният и динамичният модел на разработваната система.

Вторият аспект е езикът за моделиране. Най-често използваният език за моделиране на обектно-ориентирани системи в последните години е UML (Unified Modeling Language), предлагащ една богата и удобна за моделиране нотация.

Третият аспект на ООА е методологията, която изяснява начина за разработване на модела на системата.

Четвъртият аспект са средствата, които ще се използват за разработване на обектно-ориентирания модел.

## Oсновни понятия на обектно-ориентираната парадигма

Основната идея на обектно-ориентирания подход е декомпозиция на софтуерната система на обекти и класове. Базовите концепции и свързаният с тях понятиен апарат са възникнали първоначално на ниво програмиране (1970 - 1985). По-късно (от 1990) те се доразвиват и адаптират за целите на обектно-ориентирания анализ и обектно-ориентираното проектиране (OOD). Понеже тези концепции са основен предмет на други лекционни курсове тук ще направим само едно кратко обобщение

Основното понятия в обекно-ориентираната парадигма е обект. Обектите се използват във всички фази на развойния процес. Съществуват различни дефиниции за обекти, като най-адекватни на целите на обектно-ориентирания развоен процес са следните:

* Комбинация от две части: данни, където се съхраняват състоянията на обекта, и операции, доставящи механизмите за достъп и манипулация с тези състояния.
* Софтуерна единица, която съдържа множество от логически свързани помежду данни и операции, манипулиращи тези данни
* Комбинация от структури от данни и поведение
* Абстракция от данни с интерфейс към именовани операции и скрити локални състояния
* Структура, състояща се от някаква памет и множество операции
* Абстрактна единица, която има състояние, поведение и идентичност, като структурата и поведението на подобни обекти се дефинират в един общ клас.

Освен обектите, другите значими за ООА обектно-ориентираните концепции са следните:

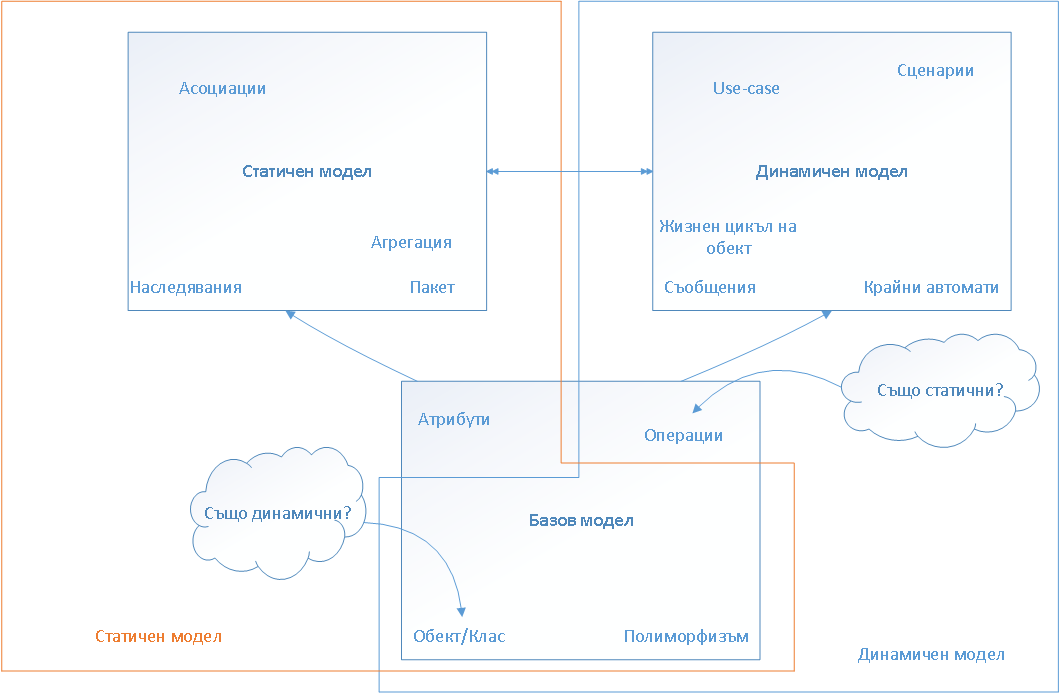
* Клас - съвкупност от подобни обекти
* Атрибути - състояние, данни на обект
* Операции - поведение
* Съобщение - изпращане на съобщения към обект
* Наследяване - йерархия на класове
* Полиморфизъм – препокриване на операции или различни реализации на една и съща функционалност.

Основно средство за моделиране при обектно-ориентирания развой са клас диаграмите, които визуално представят комбинации от гореизброените базови елементи.

## Модели в ООА

Един пълен модел на софтуерна система, създавана в рамките на обектно-ориентиран развоен процес, обикновено включва три взаимно допълващи се подмодела (Фиг.3.):

* Базов модел – представя основните изграждащи елементи на модела на системата, като класове, обекти, атрибути операции и полиморфизъм
* Статичен модел – описва системната архитектура, т.е. класовете, обектите и съществуващите между тях връзки, заедно с различните ограничения, които трябва да удовлетворяват
* Динамичен модел – описва поведението на системата.

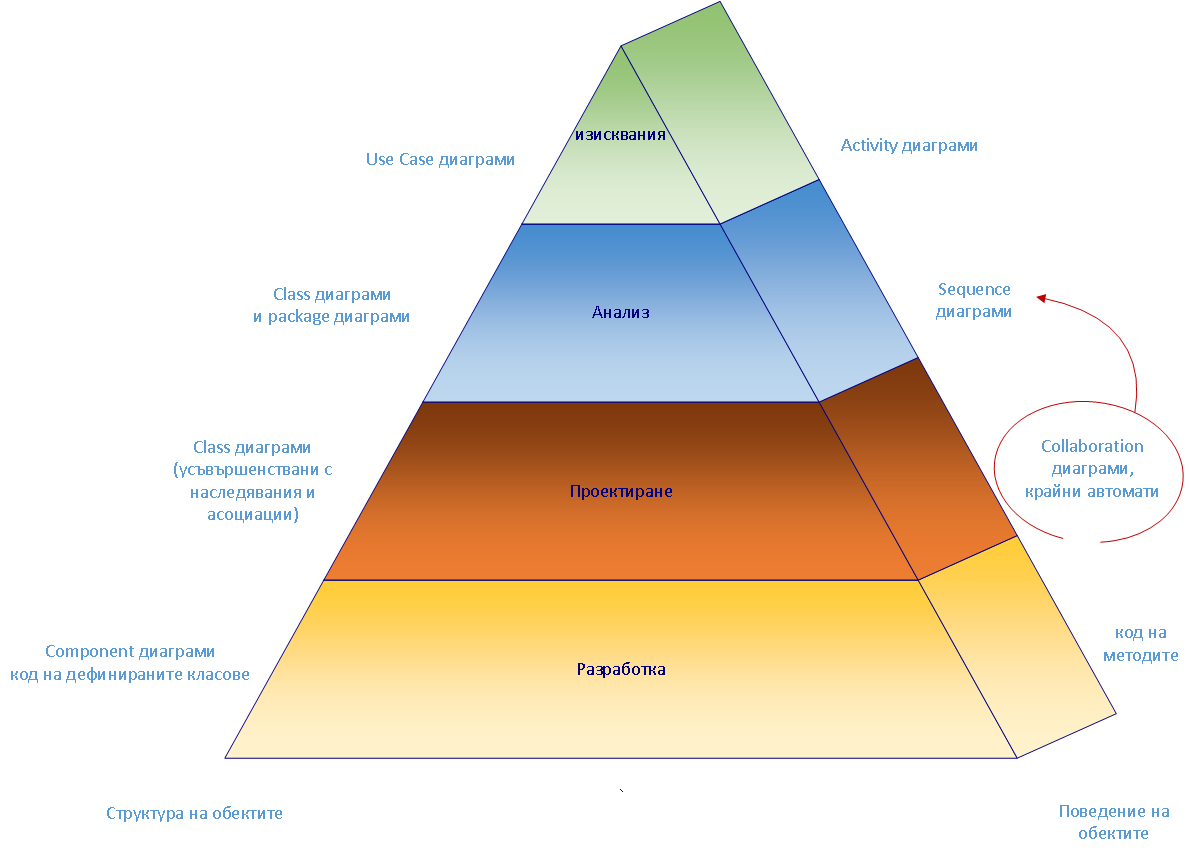


Фигура 3: Декомпозиране на ООА модели

За представянето на тези модели обикновено се използват UML диаграми. В езика UML са дефинирани слените видове диаграми:

* use case диаграми: взаимодействие на хората със системата
* class диаграми: класовете и техните връзки
* sequence диаграми: зависещ от времето поток от съобщения
* collaboration диаграми: също като sequence диаграмите
* package диаграми: модулизация
* state диаграми: динамично поведение на обектите
* activity диаграми: паралелни процеси
* component диаграми: единици за компилация, хардуерна структура
* object диаграми: обектите и техните връзки

На Фиг.4. е обобщено използването на UML диаграмите във фазите на софтуерен развой.

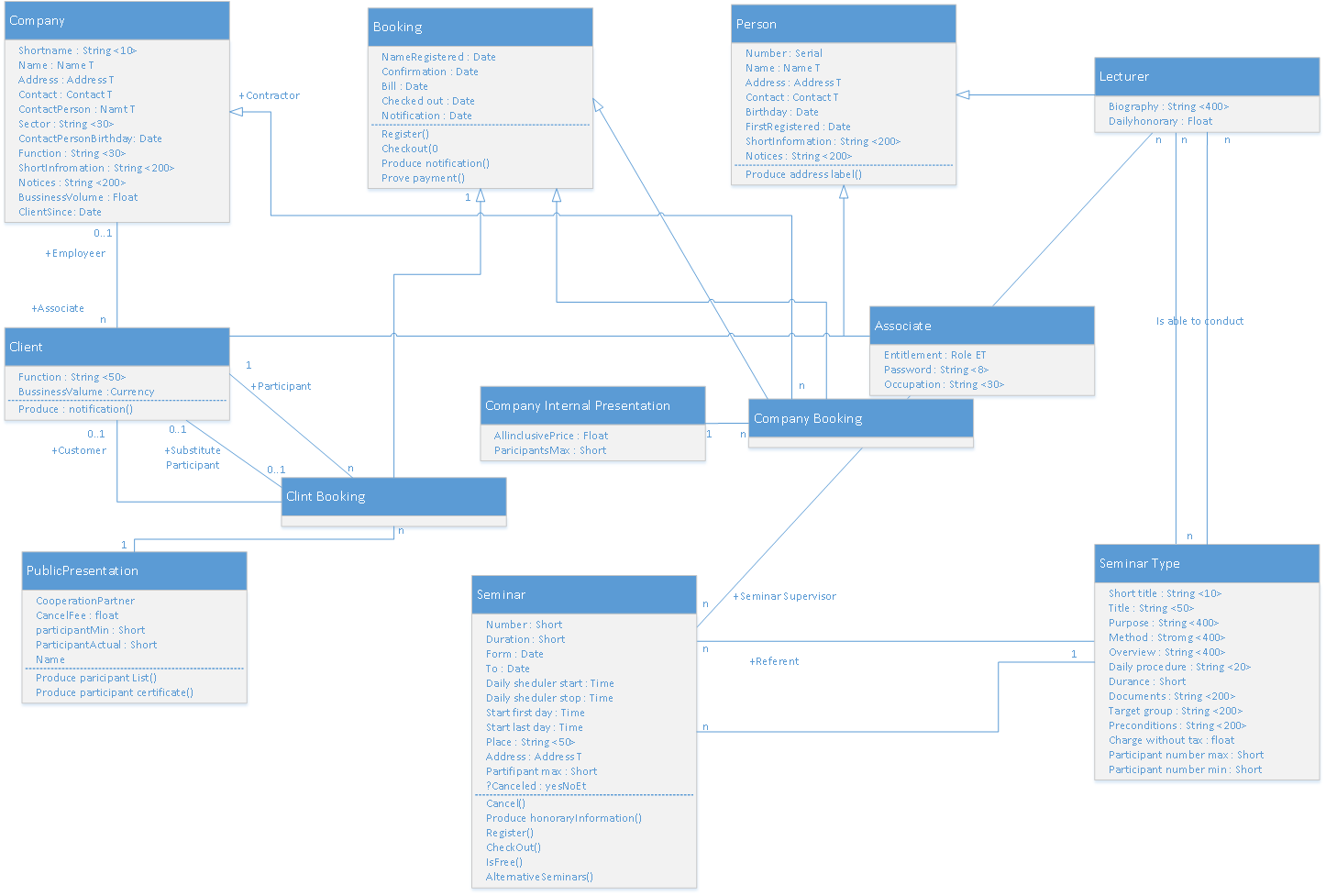


Фигура 4: Приложение на UML диаграмите във фазите на софтуерен развой

Collaboration диаграмите и крайните автомати освен във фазата на разработване могат да се използват също така и по време на фазата проектиране от процеса на разработка на софтуер.

### Статичен модел

Фигура 5 представят клас диаграмата на примера “Организация на семинар”. Целта на тези фигури е да се представи основната концепция в обектно-ориентирания анализ и обектно-ориентираното моделиране – клас диаграмите. Тя представя общ изглед на това какво представляват клас диаграмите и по-точно клас диаграмата на примера “Организация на семинар”.



Фигура 5: Клас диаграма на примера “Организация на семинар”

Основата на статичния модела е самия клас, които представя обектите. Всеки клас се специфицира чрез име, атрибути и операции. Между калсовете има връзки, които представят взаимодействието между обектите от съответните класове – асоциации, наследяване. От своя старана всяка от асоциациите се представя чрез сопствени характеристики като роли и кардиналност.

Има три основни атрибута на модела при обектно-ориентиран анализ, които могат да предизвикат някакви проблеми при разработката:

* Възможност за четене на модела – разбираем ли е, нотацията достатъчно ясна ли е;
* Възможност за оценка на модела – това е друг важен атрибут на модела свързан с това, дали модела е смислен, т.е. логически правилен ли е. Трябва да е възможно да се оцени дали са нужни допълнителни класове, атрибути или операции, които да бъдат добавени към него. И накрая трябва да се прецени ако в някои случаи имаме избор да създадем нови класове или атрибути и операции;
* Възможност за разработка на модела – последната и може би най-важната от характеристиките е това, че създадения модел трябва да може да бъде разработен.

## Характеристики на статичния модел

Съществена характеристика на обектно-ориентираните статични  модели е наследяването. Определянето на класовете, наследници и родители е една от основните задачи на обектно-ориентирания развой.

Асоциациите моделират отношения между обектите на класове от един и същ ранг. Асоциации се допускат и между обектите на един клас. За разлика от наследяването асоциациите свързват обекти (не класове). В системния анализ асоциациите обикновено са бидирекционални (между два обекта), при което не е задължително да бъдат реализирани в двете посоки. При тези асоциации в общия случай всеки един от двата обекта принадлежи към различен клас.

Асоциациите могат да бъдат сравнени с асоциациите от Entity-Relationship-моделите. Те представят не само една статична структура между класовете, но са също така  предпоставка за потока на съобщенията между обектите. Информацията за това, кой обект с кой друг е свързан се намира изключително в асоциациите. За разлика от релационните бази данни, при обектно-ориентираните модели не са необходими външни ключове или референциални атрибути.

В много случаи асоциациите се именуват. Обикновено името описва само едната посока на асоциацията. Когато значението на асоциацията е очевидно, тогава името може да липсва. Ако съществуват повече от една асоциации между два класа, тогава асоциациите трябва да бъдат именувани. Възможна алтернатива на имената на асоциациите са ролите. Асоциациите се изобразяват като линии между свързаните обекти.

Ролите описват функциите на обектите в една асоциация. Една бинарна асоциация напр., притежава максимум две роли. Имената на ролите обикновено се задават в края на асоциациите и по-точно към класа, значението на който те описват по-детайлно. Използването на ролите е опционално. Сполучливият избор на имената на ролите може да допринесе повече за разбирането отколкото самото име на асоциациите.

Кардиналността показва броя на обекти, с които даден обект от определен клас може или трябва да се намира в една конкретна връзка

Агрегациите са насочени между обектите асоциации. Те представляват частен случай на асоциациите и определеят връзки между обектите, които се интерпретират като”част\_от” или “състои\_се\_от”. С други думи, това са отношения между между цялото и неговите части.

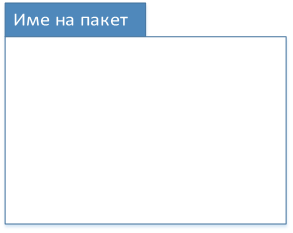
## Подсистеми (пакети)

Броят на класовете в един ООА-модел зависи от приложната област. Обикновено една средно голяма система се моделира с около 35 класа. За големите системи в общия случай са необходими повече от 100 класа. Ако всички тези класове трябва да се поставят в една диаграма, това ще затрудни значително читаемостта й. Подсистемите се използват за групиране на елементите на моделите в по-големи единици. Освен това те дават възможност за идентификация на подсистеми при големите проекти.

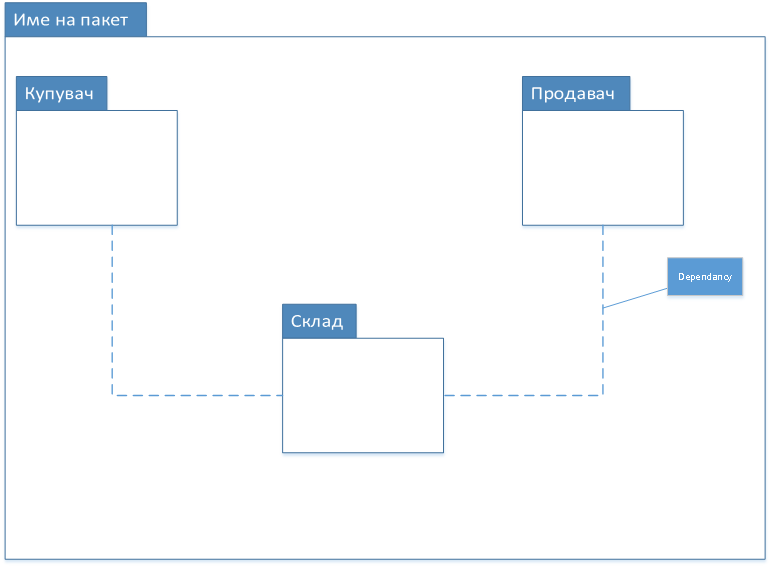
В UML подсистемите се представят с помощта на пакети, като:

* Пакетите съдържат елементи от модела (т.е. класове) и диаграми
* Пакетите могат да съдържат други пакети.

Нотацията, която се използва в UML за изобразяване на пакети е дадена на Фиг.6.



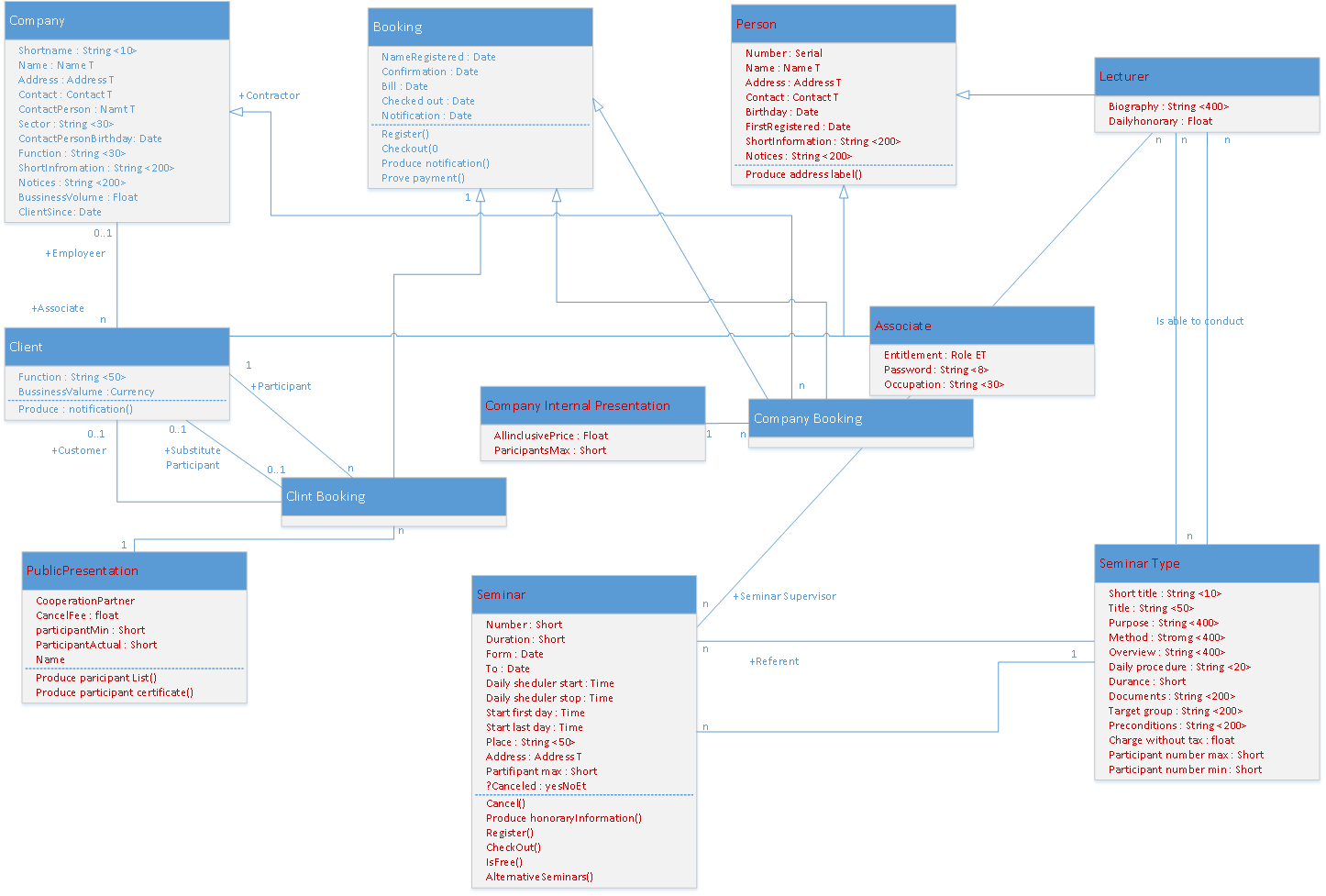
Фигура 6: Нотация за пакет



Фигура 7: Package диаграма на търговска система

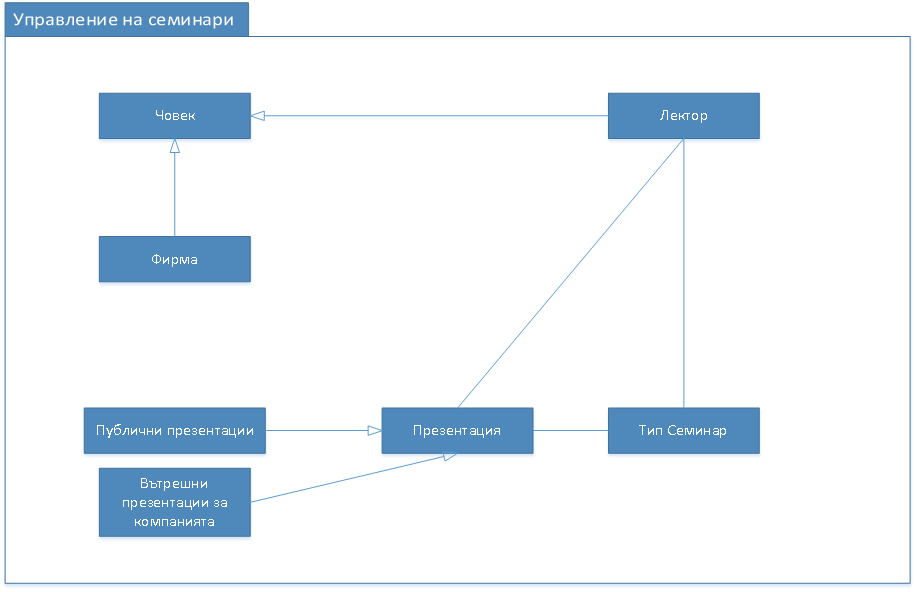
В примера на Фиг.7. е представена package диаграма на една търговска система, където разработчикът е решил (за по-добра четаемост) да групира определени класовете в отделни пакети. На диаграмата е показан един пакет “Склад”, свързан с другите два пакета – “Купувач” и “Продавач”.

В клас диаграмата на Фиг. 8. са маркирана класове, които не зависят от клиентите. Тези класове могат да се групират в един пакет, който ще наречем “Оферти на компанията – тук не може да се види име защото това е клас диаграма. Решава се кои класове ще участват в пакета и му се дава име. Името се вижда на следващата фигура която е package диаграма)



Фигура 8: Класове независещи от клиентите

Друг пример е package диаграмата на “Управление на семинари” (Фиг.9), представена в нотацията на UML.



Фигура 9: package диаграмата на “Управление на семинари

**Динамичен модел**

Динамичният модел при ООА включва във себе си следните диаграми от UML:

* Use case
* Sequence
* Collaboration
* State- transition (крайни автомати)
* Activity

Всички те представят динамично поведение на системата. Всяка една от тях е разгледана в съответния изглед на системата, който представя.

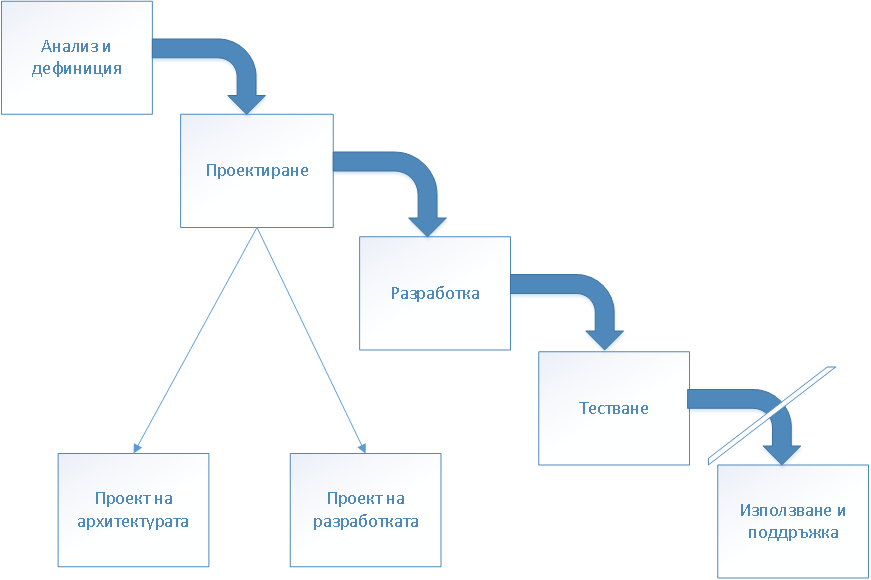
**Раздел 11:**

**Фаза на проектиране**

**Обща характеристика**

Основната задача на фазата проектиране е, на основата на зададените изисквания към разработвания софтуерен продукт да се разработи едно програмно-техническо решение. Изходна точка за тази фаза са резултатите, получени от фазата на анализ и дефиниране. Резултатите от проектирането от своя страна са основа за дейностите по разработка на системите. Двата основни документи (Фиг.1.), които трябва да се изготвят са следните:

* Проект на архитектурата – проект с високо ниво на абстаракци
* Проект на разработката – този проект зависи от езика за програмиране и използваните развойни средства и среди.



Фигура1: Подфази на фазата проектиране

Една от основните задачи, които трябва да се решат по време на проектирането, е разработване на софтуерната архитектура. В много информационни системи архитектурата е едно слабо звено. Тук сме избрали три типични примера, които потвърждават този извод.

Първият пример е оценка на студент за софтуерните архитектури, разработвани във фирмата, в която протича семестриалната му практика. Той ги оценя като катастрофални, като обосновава своето мнение с това, че те не могат да се поддържат и въобще не може да става и дума за многократното им използване.

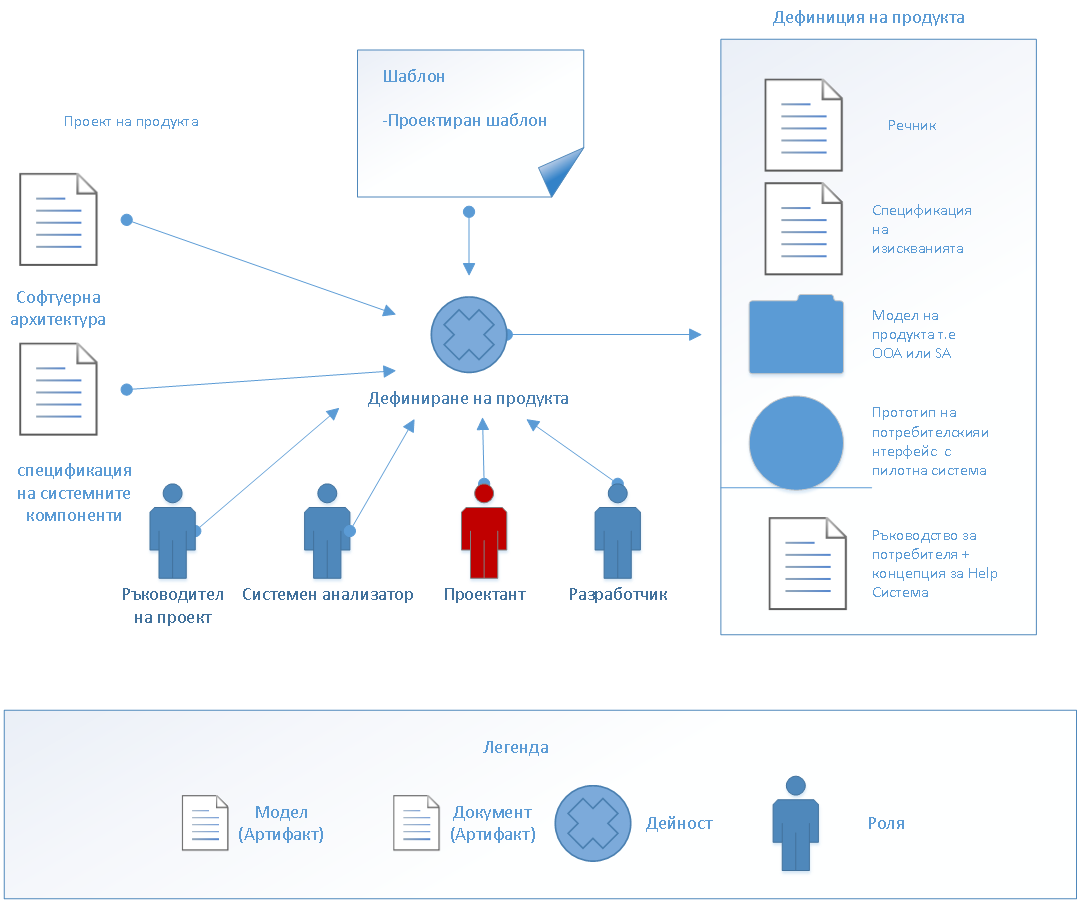
Вторият пример е за някои продукти на DaimlerChrysler, за които основните използвани в развоя софтуерни единици са функции и процедури, притежаващи обикновено над 2000 параметъра. Около 60-70% от тях са излишни. Оценка на излишните изисква огромни ресурси, включително и финансови. Тази ситуация се дължи на не добре обмислената  архитектура на продуктите.  
Обобщавайки горните примери можем да заключим, че доброто и прецизното  проектиране на архитектурата на системата се отплаща в по-късните фази на развоя - от разработката до поддръжката. Трябва да отбележим също така, че грешки, открити в по-късни етапи от разработката на софтуера, са много по-скъпи за отстраняване в сравнение с откритите в по-ранна фаза от разработката.

Софтуерната архитектура описва структурата на системата посредством системни компоненти и връзките между тях. Един системен компонент е една отделна част от софтуерната система, която се използва като градивен блок в нейната физическа структура. Системни компоненти могат да бъдат:

* Функции
* Процедури
* Абстрактни обекти от данни
* Абстрактни типове от данни
* Класове.

*Връзките* описват някакво възможно свързване между системните компоненти. Те могат да бъдат *динамични* или *статични*.

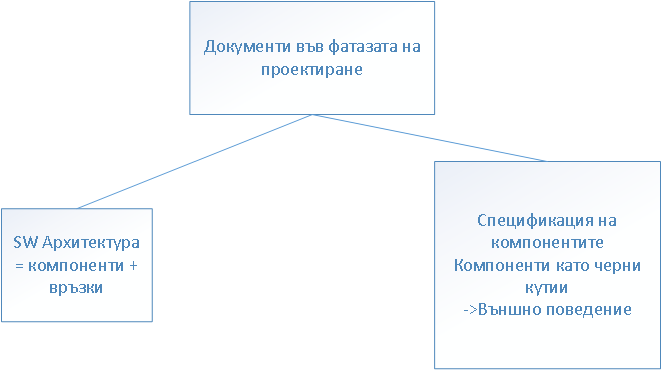
На Фиг.2. е представен процесът на проектиране в контекста на софтуерния развой.  В лявата старана на фигурата са дадени входните документи (речник, спецификация на изискванията, модел на продукта, прототип на потребителския интерфейс, ръководство за потребителя и концепция за help), които са основа за резултатите от тази фаза - документите в дясната страна на фигурата (софтуерна архитектура и спецификация на системните документи). Целта на проектирането са двата документа – софтуерна архитектура и спецификация на системните компоненти. За тяхната разработка могат да се използват различни софтуерни шаблони (patterns), които ще наричаме проектни шаблони. Освен това, от фигурата може да се види, че основната дейност в тази фаза се извършва от проектантите. От съществено значение е също така участието на персонала както от предишната фаза (ръководител на проекта, системни анализатори), така също персонала от следнващата фаза (разработчици, обикновено програмисти). Това е необходимо, защото те трябва да изпълняват ролята на консултанти и съветници при разработването на архитектурата на приложението. По-нататък ще разгледаме подробно двата основни резултата от тази фаза – софтуерна архитектура и спецификация на системните компоненти.



Фигура 2: Процес на проектиране

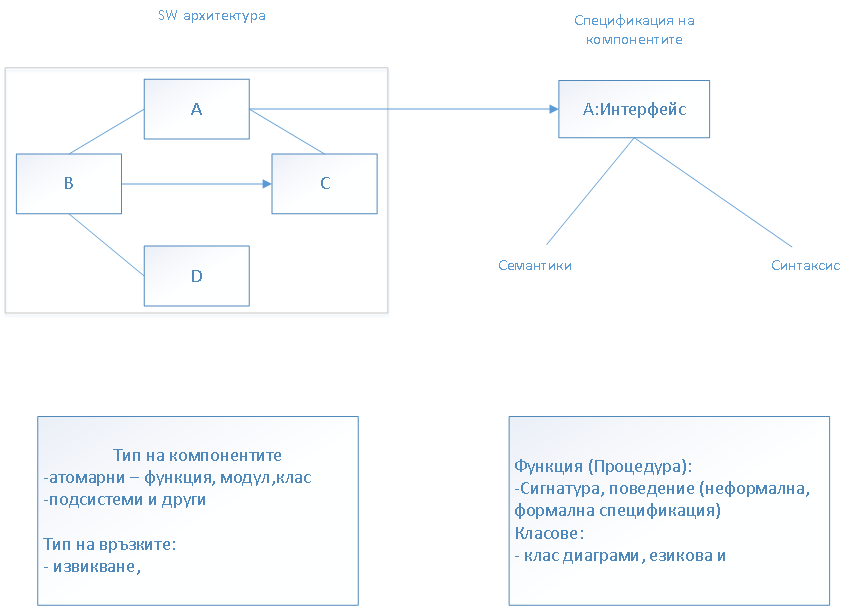
**Софтуерна архитектура**

Софтуерната архитектура се състои се от системни компоненти и връзки. За спецификацията на компонентите са необходими различни документи (Фиг.3., Фиг.4.).



Фигура 3: Документи във фазата на проектиране

Всеки компонент на софтуерната архитектура се разглежда като черна кутия, която трябва да се попълни по време на следващата фаза от развойния процес – разработване. В тази фаза основно ни интересува външното поведение на системата. Във фазата на анализ и дефиниция системата се разглежда също така като черна кутия, но от гледна точка на потребителя.



Фигура4: Спецификация на документите

На Фиг. 4. е показано съдържанието на една спецификация на компонент, която обикновено включва:

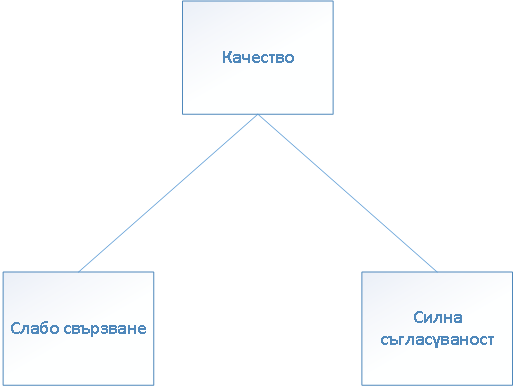
* Тип на компонента – функция, модул, клас, подсистема и др.
* Ако компонентът е функция или процедура, тогава той трябва да има сигнатура и поведение, което да бъде описано с формална или неформална спецификация
* Ако компонентът се състои от класове, то е необходимо да се разработи клас диаграма за него и да се направи езикова и формална спецификация
* Тип на връзките между компонентите – може да бъде извикване, агрегация, асоциация и т.н.

**Качество на софтуерната архитектура**

При оценяване на софтуерните архитектури трябва да търсим отговори на следните два втпроса:

* Какво разбираме под „добра“ софтуерна архитектура?
* По кои критерии могат да се оценяват архитектурите?

Софтуерните архитектури могат да съдържат произволен тип компоненти, като функции, класове, подсистеми, и др. Една архитектура е добра, ако между компонентите има слаба свързаност и силна съгласуваност (Фиг. 5.).



Фигура5: Характеристики, от които зависи софтуерната архитектура

Свързаността се определя от интерфейсната връзка между отделните компоненти на архитектурата. Колкото по-малко информация се обменя между компонентите (т.е компонентите са по-прости), толкова по-слаба е свързаността (с това и по-ефективна архитектура). В резултат на слабата свързаност компонентите са по-лесни за разбиране и за модифициране от разработчиците. Ако между два компонента има прекалено голям обмен на информация (т.е силно свързване) ще бъде целесъобразно те да бъдат декомпозирани на по-прости, което ще улесни работата и разбираемостта на приложението.

Съгласуваността представлява логическа връзка между вътрешните елементи на компонентите. Такива елементи могат да бъдат данни и операции върху тези данни. Основната задача на съгласуваността е всички елементи вътре в компонента да решават една обща задача. В резултат на това всеки компонент трябва да се разработва за решаване на конкретна функционалност, т.е. той има силна съгласуваност. Така се повишава разбираемостта на компонента. Ако един компонент решава повече от един проблем неговата съгласуваност намалява. Когато съгласуваността е слаба ще бъде целесъобразно да декомпозираме съответния компонент на по-малки, всеки от които ще решава отделен проблем.

Спрямо съгласуваността съществуват различни ситуации, които могат да предизвикат определени проблеми и трудности. Ще разгледаме два типични случая:

* В един компонент са вградени повече от една функции;
* Една функция е разпределена в повече компоненти

С нарастването броя на компонентите се увеличават и интерфейсните връзки – всеки компонент има свой отделен интерфейс. Така между свързването и съгласуваността съществува взаимовръзка, която създава също така определени проблеми. Двете изисквания (слабо свързване и силна съгласуваност), определящи добрите архитектури, могат да доведат до противоречие и необходимост от вземане на компромисни решения, които могат да се разгледат в следните два аспекта:

* При архитектурите с по-малко на брой компоненти свързването е по-слабо, което води и до по-малка съгласуваност - предполага се, че колкото по-малко са компонентите, толкова повече задачи ще извършва всеки един от тях. Това от своя страна трябва да намалява съгласуваността. Важното в такива случаи е да се намери общ баланс между съгласуваността и свързването. Обикновено това става чрез ясно определяне на семантиката на компонентите.
* При архитектури с твърде много компоненти, многото връзки и силната съгласуваност води до силна свързаност – при наличието на повече компоненти се предполага, че всеки един от тях решава определен проблем, което води до силна съгласуваност. От друга страна всеки компонент има свой интерфейс за връзка, което определя силна свързаност

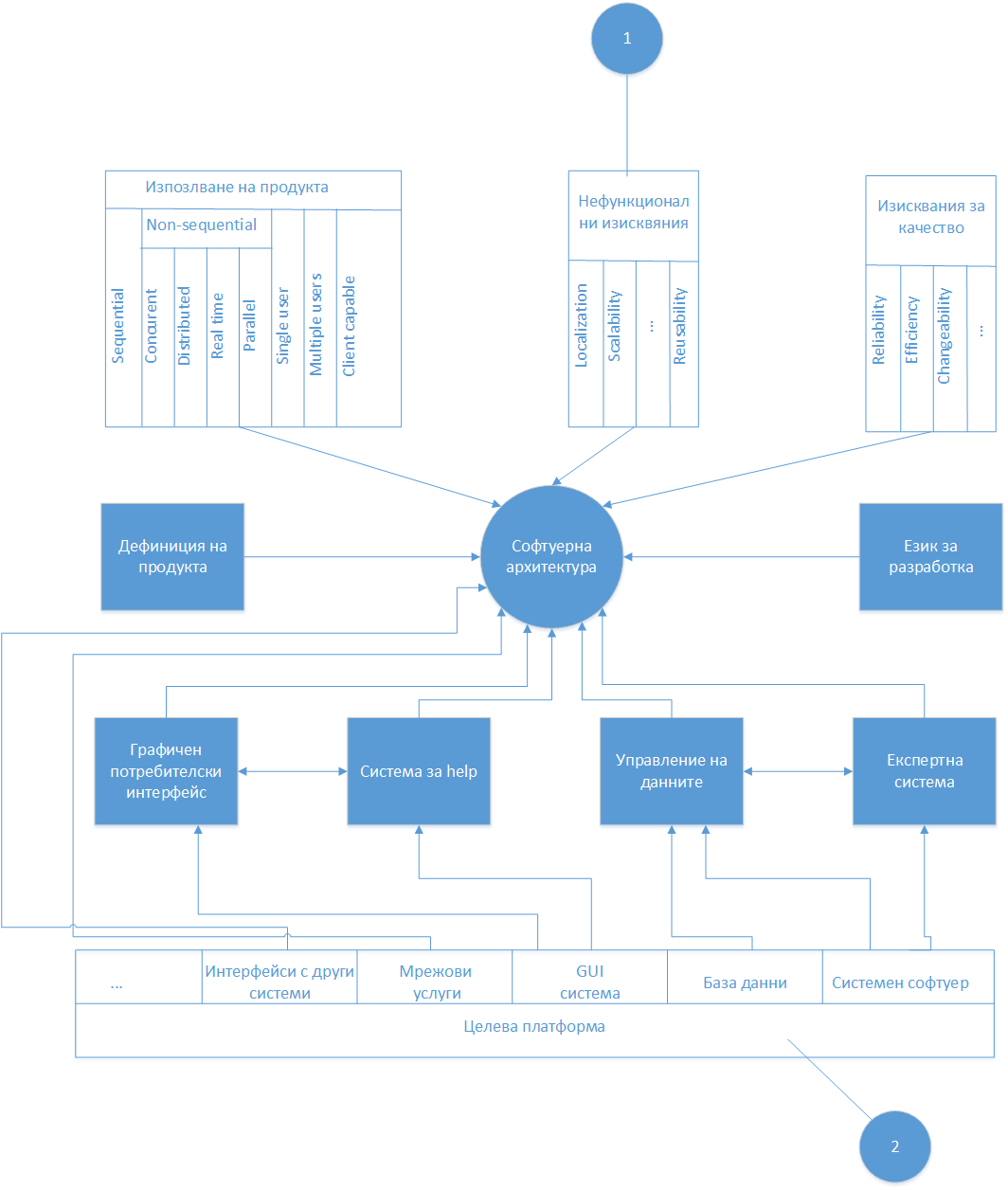
**Влияещи фактори в/у СА**

Два основни фактора оказват влияние върху софтуерната архитектура:

* Дефиницията на продукта
* Езика и средата за разработка

Освен представените по-горе два фактора в практиката съществуват още различни други специфични фактори, които не трябва да бъдат пренебрегвани. Допълнителните  фактори обикновено се групират в следните категории (Фиг. 6.):

* Използване на продукта
* Нефункционални изисквания
* Изисйквания за качество.



Фигура 6: Допълнителни фактори влияещи върху софтуерната архитектура

Ще разгледаме някои от допълнителните фактори по-подробно.

Основните нефункционални изисквания са:

* Интернационалност - поддръжка на много езици (ясно е че това трябва да се предвиди в архитектурата на приложението), както и използване на специфични за дадената страна функционални компоненти (напр. данъчна система)
* Мащабируемост – възможност за поддръжка на продукта от различни платформи, инсталация за различни платформи и използване на различни системи за графични потребителски интерфейси
* Многократно използване – възможност компоненти да се използват многократно в различни приложения, както и поддръжка на библиотеки за многократно използване и проектни шаблони.

Качеството на продукта силно зависи от следните фактори:

* Надеждност – ако архитектурата е добра, тогава и самият продукт ще бъде устойчив на грешки и проблеми
* Продуктивност – ако компонентите на архитектурата са добре измислени и задачите са добре разпределени между тях, тогава и продуктивността на приложението ще е добра
* Възможност за промяна – ако архитектурата е добре измислена, тогава лесно ще могат да бъда правени промени в приложението.

Софтуерният продукт трябва да бъде интегриран в платформената архитектура  на целевата система. Целевите платформи предлагат голямо многообразие от интерфейси, услуги и продукти с голяма вътрешна зависимост. Платформените услуги помагат да се изолира софтуерната архитектура на разработвания софтуерен продукт  от основната функционалност на системата. Така софтуерните продукти могат да бъдат преносими и независими от производителя. Освен това използването на готови платформени услуги спестява значителни усилия и разходи.

Към основната функционалност, описана в спецификацията на изискванията, могат да се добавят някои допълнителни подсистеми като напр.:

* Графичен потребителски интерфейс
* Система за помощ (help)
* Системи за управление на данните
* Експертни системи.

Тези допълнителни подсистеми напълно зависят от тази целева платформа и по-точно от предлагани от нея услугите, като напр. системни услуги, мрежови услуги, бази данни и др. Добавянето на такива услуги влияят на разработката и архитектурата на тези подсистеми.

**Многослойни архитектури**

Ако една софтуерна система се състои от много компоненти, тогава е целесъобразно прилагането на едно по-силно структуриране. Обикновено се използва групиране на отделните компоненти в различни слоеве (съответно архитектурата се нарича многослойна). Съществуват два основни начина за структуриране на такива архитектури:

* Обособяване на компонентите в пакети – подсистеми от логически свързани компоненти
* Обособяване на компонентите в слоеве – определяне на абстрактни нива в архитектурата (многослойни архитектури).

На Фиг. 7. е дадена многослойна архитектура състояща се от три нива:

* Слой 0 - Управление на данните
* Слой 1 - Реално приложение
* Слой 2 - Потребителски интерфейс

Между отделните слоеве съществува определени взаимовръзки.



Фигура 7: Многослойни архитектури: йерархично построени

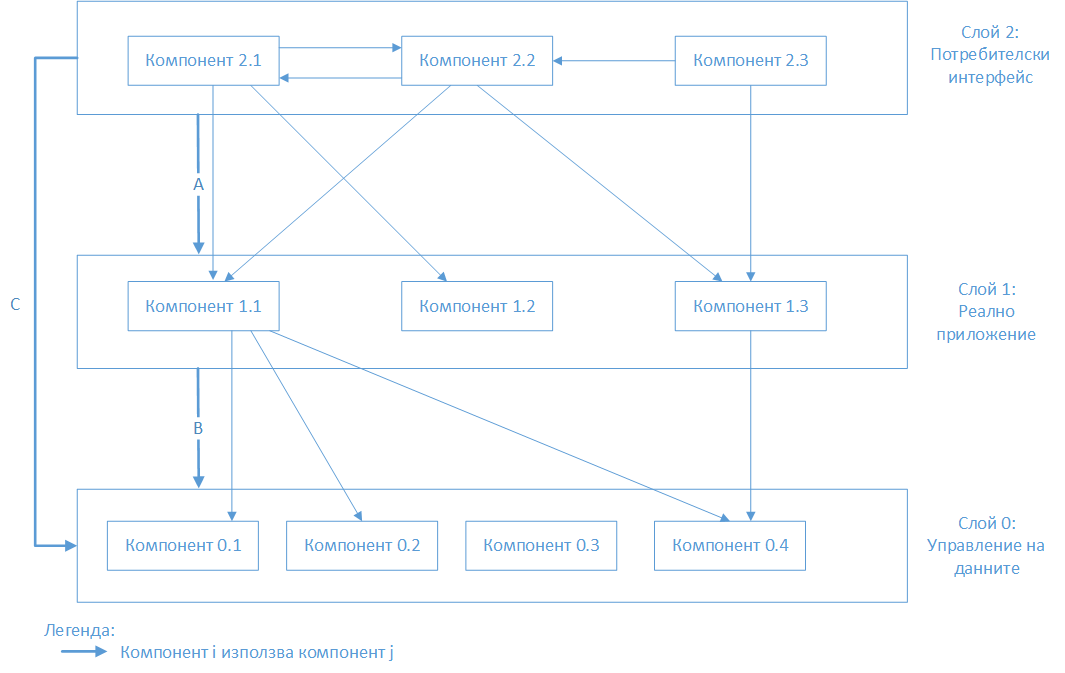
Съществуват следните видове многослойните архитектури:

* Слоеве с линейна подредба – допуска се достъп само до следващия по-долен слой
* Слоеве със строга подредба – разрешен е достъп до всички слоеве на по-долно ниво
* Слоеве с дървовидна подредба – не се допуска комуникация между слоеве на едно и също ниво

Обикновено в рамките на един слой се допуска произволен достъп между съставящите го компоненти.

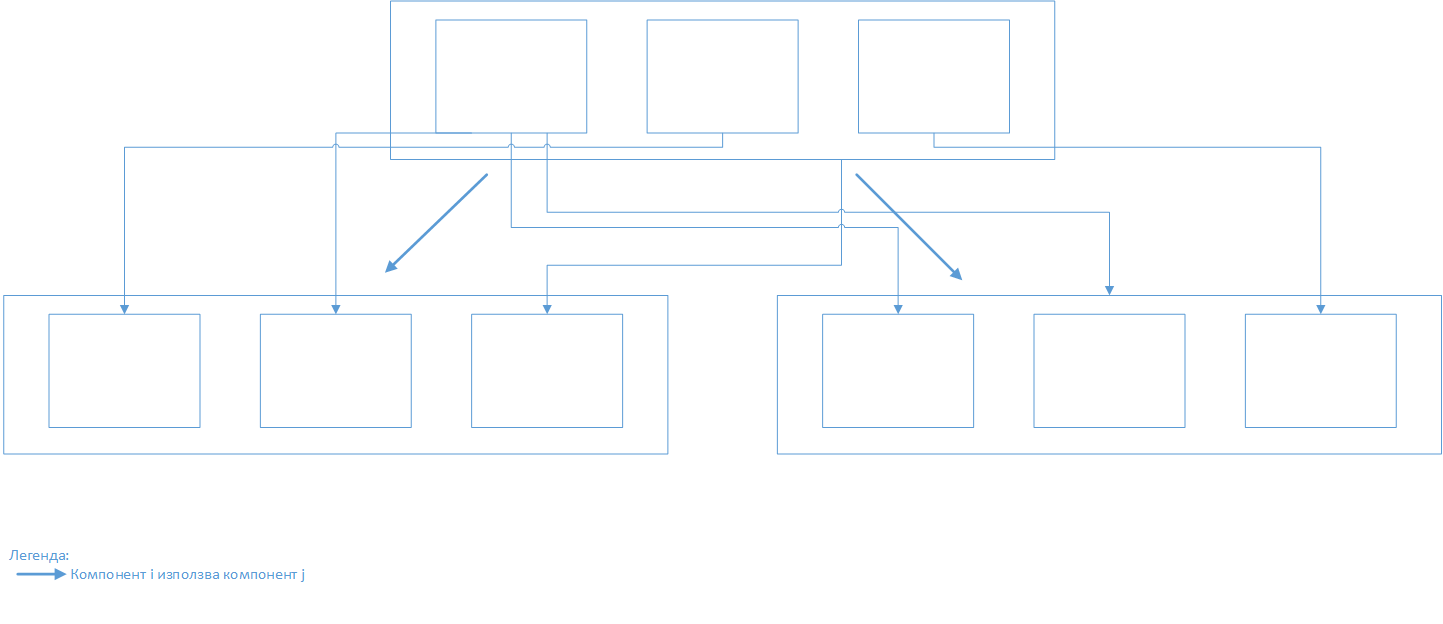
Ще разгледаме два примера за многослойни архитектури. Първият пример (Фиг. 8.) е за трислойна архитектура, в която се поддържат две подредби:

* Линейна подредба (a,b)
* Строга подредба (a,b,c).



Фигура 8: Трислойна архитектура

Вторият пример (Фиг. 9.) представя една дървовидна архитектура, при която няма комуникация между слоеве на едно и също ниво.



Фигура 9: Дървовидна архитектура

Многослойни архитектури обикновено се прилагат когато услугите на слоя са на едно и също абстрактно ниво. Слоевете са подредени според тяхното абстрактно ниво, като едно ниво се нуждае само от услуги на по-долно ниво.

Предимства на многослойните архитектури са следните:

* Ясна и добре подредена структура
* Строга йерархия и възможност за свободна структура вътре в слоевете
* Поддръжка на повторно използване,изменяемост, лесна поддръжка, преносимост, възможност за лесно тестване.

Основни техни недостатъци:

* Загуба на продуктивност - всички данни трябва да бъдат транспортирани през слоевете (в линейна подредба)
* Не винаги е възможно ясно разделяне на абстрактните нива.

Многослойните архитектури са предпоставка за Client/Server и Web архитектури.

**Раздел 12:**

**Структурно проектиране**

**Общ преглед**

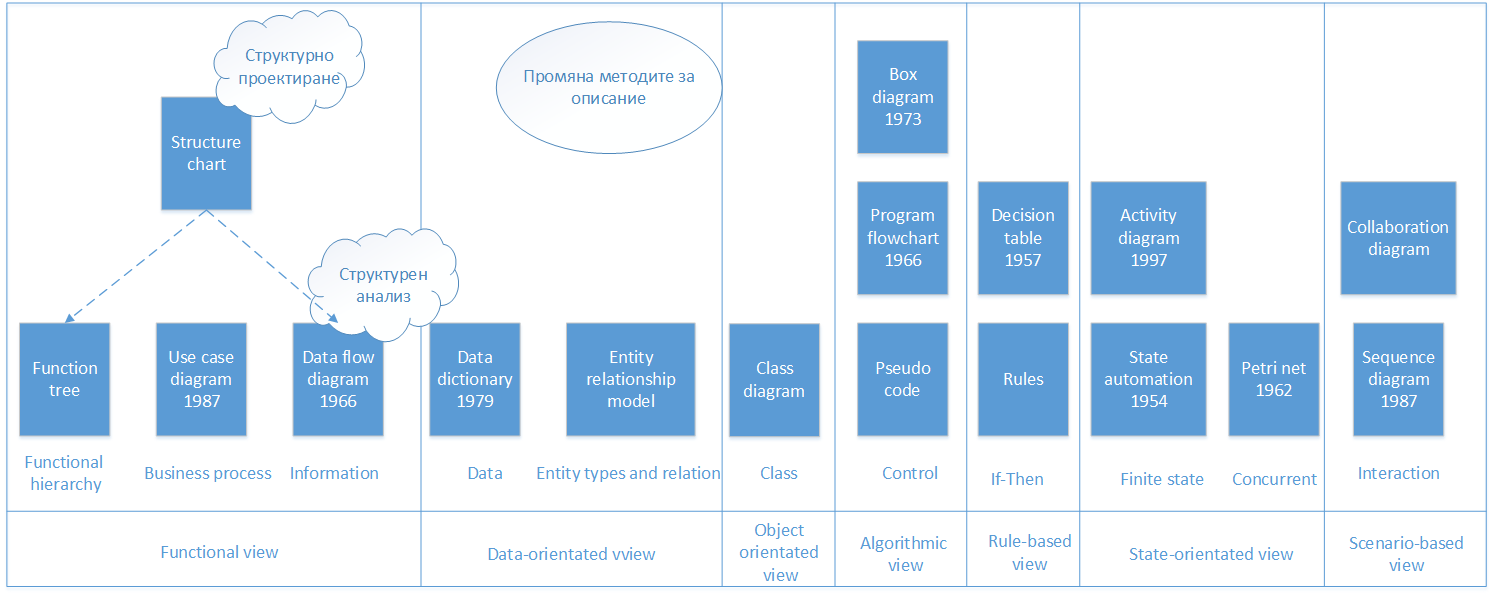
Основната цел на структурното проектиране (structured design) е създаване на архитектура на разработваната система, състояща се от йерархично подредени функционални модули (функции, процедури, подпрограми). За описание на проектирането се използват структурни диаграми и спецификация на модулите.

Сложността на разработваната система може да се редуцира в процеса на проектиране посредством избора на подходящи абстракции. Исторически първата възникнала форма на абстракция е функционалната абстракция. Една функционална абстракция представлява определена функционалност под формата на абстрактна функция, абстрактна процедура или абстрактна операция (често се нарича също така операционна или процедурална абстракция). Реализацията на функционалните абстракции ще наричаме функционални модули, с което искаме да подчертаем следните три аспекта:

* Модулите са активни или ориентирани към извършване на действия (т.е. те “правят” нещо)
* Те притежават някакво трансформационно поведение (т.е. входните данни се трансформират в изходни данни)
* Идентични входни данни водят винаги до идентични изходни данни (т.е. модулите не притежават памет).

Основната концепция за структурния анализ - диаграмите на потока от данни (DFD) – нямат достатъчно изразителна мощ за постигане на целите на структурното проектиране. По тази причина е необходимо въвеждане на нова концепция – структурни диаграми, които са смесица от функционална йерархия и поток от данни (Фиг. 1.). Трябва да отбележим, че при структурния подход методите за описание на софтуера се променят при прехода от анализ към проектиране, което е най-големият недостатък на този подход. Друг недостатък на подхода е, че архитектурите, базиращи се на функционални модули са подходящи за една сравнително ограничена група приложни области. Типични приложения за такъв вид архитектури са:

* Системно програмиране – напр. многопасови компилатори
* Числени алгоритми.

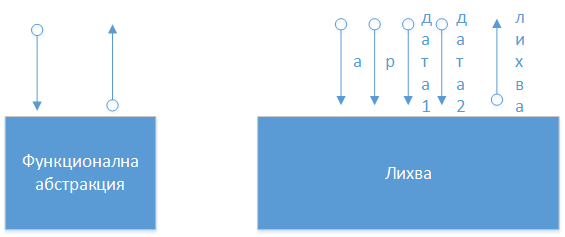


Фигура 1: Основни концепции в софтуерната разработка

### Структурни диаграми

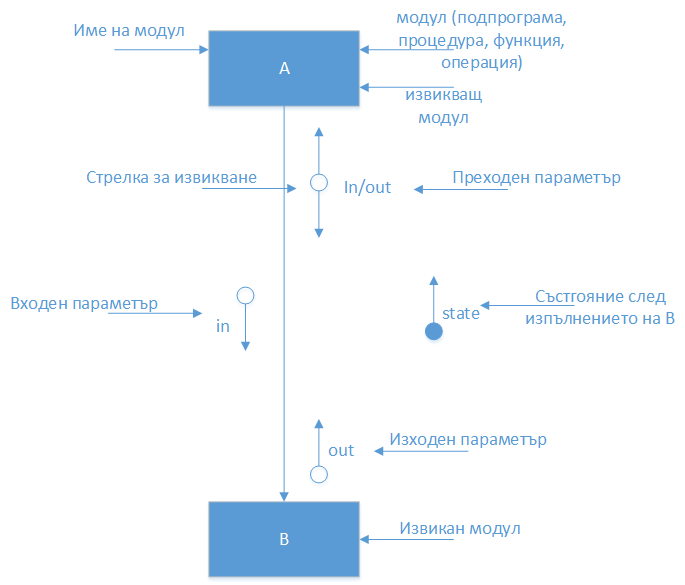
Структурните диаграми представят графично функционални модули и взаимодействието между тях, които могат да бъдат извиквания (активиране) на определен функционален модул и предаване на параметри.

Структурната диаграма, представяща един пример за предаване на параметри за функционалния модул  „Пресмятане на лихва“ (Фиг.2.) съдържа най-простите структурни елементи. В лявата част на фигурата е даден общия вид на използваната функционална абстракция с един входен и един изходен параметър. Дясната част е приложение на абстракцията за конкретната функция (“Лихва”), където входни параметри са сума за олихвяване (a), лихвен процент (p) и две дати (дата1, дата2), между които функцията трябва да изчисли лихвата. Получената лихва е изходен параметър.



Фигура 2: Пример за предаване на параметри за функцията "Пресмятане на лихва"

Структурните диаграми обикновено се използват за представяне структурата на извикване и потока от данни между отделните функционални модули, както е показано на Фиг.16.3. Визуално функционалните модули се представят като правоъгълници. Връзките на извикване на функционалните модули са представени също така на фигурата. В примера функционалният модул “A” извиква функционалния  модул “B”. Останалите стрелки в диаграмата представят входните и изходни параметри на функционалните модули.



Фигура 2: Пример за предаване на параметри за функцията "Пресмятане на лихва"

**Раздел 13:**

**Обектно-ориентиран анализ**

**Обща характеристика**

Целта на проектирането е, изхождайки от дефиницията на разработвания продукт да се създаде проект, който реализира изискванията към продукта и който предлага архитектурно решение за интегриране на системата в съответната приложна среда. Обектно-ориентираното проектиране е продължение на обектно-ориентирания анализ.

Основна концепция на обектно-ориентираното проектиране (както и на обектно-ориентирания анализ) са клас диаграмите. Освен клас диаграмите съществуват още две концепции, които се използват за описание на класовете на разработваната система – крайни автомати и activity диаграмите. Основното предимство на обектно-ориентирания подход е това, че за анализа и за проектирането на приложенията се използват едни и същи методи за описание.

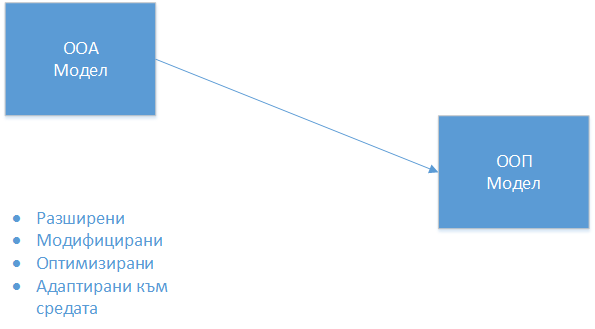
Обектно-ориентираното проектиране (object oriented design) може да бъде разделено на следните две стъпки:

* Проектиране на архитектурата
* Проектиране на разработката.

Първоначално се определя архитектурата на системата, като се представят:

* Свързването към потребителския интерфейс
* Свързването към системата за съхранение на данните

Разпределението на приложението върху различни свързани компютърни системи.

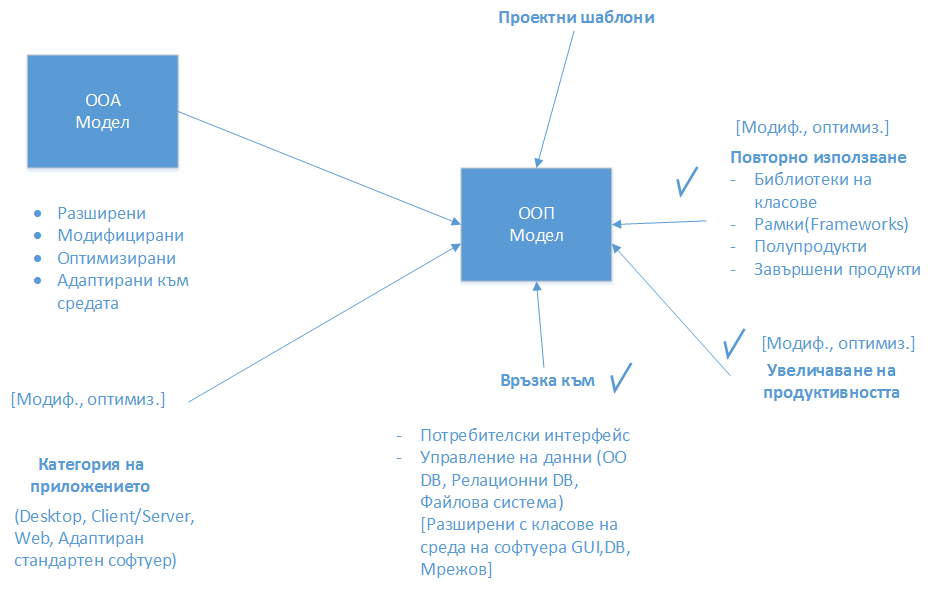


Фигура 1: Адаптация на ООА към ООП

При разработване на архитектурата моделите, получени по време на обектно-ориентирания анализ, трябва да бъдат разширени, модифицирани, оптимизирани и адаптирани към средата (Фиг. 1).

По време на създаване на проекта трябва да се решат следните две задачи:

* Детайлизиране на архитектурата на системата
* Настройка към езика за програмиране.



Фигура 2:Фактори влияещи върху проекта на архитектурата

Основните фактори, влияещи върху проекта на архитектурата на приложението са следните (Фиг. 2):

* Категория на приложението
* Връзка към потребителския интерфейс и управлението на данните
* Продуктивност
* Многократно използване на библиотеки от класове
* Рамки и проектни шаблони.

Някои от тях ще бъдат разгледани в по-големи детайли по-нататък в главата.

**Проект на архитектурата**

За разработване на архитектурата на разработваната софтуерна система се използват два основни източника:

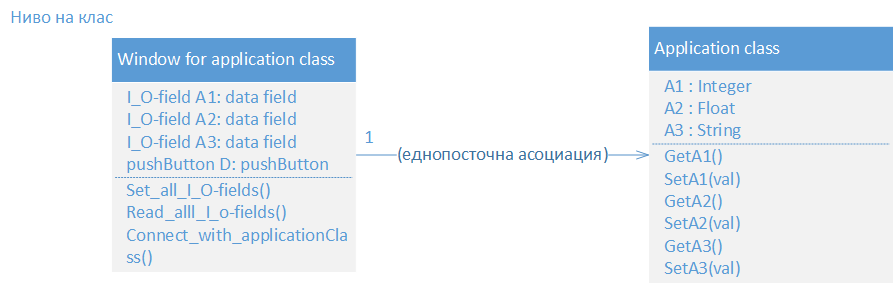
* Обектно-ориентиран модел от анализа на системата (създаден при дефиниране на системата)
* Интерфейси към околната среда.

Цялостната архитектура трябва да се обмисли така, че да осигурява оптимално изпълнение на функционалността на приложението, като се съблюдават изискванията на околната среда. Освен това искаме да обърнем внимание на това, че адаптирането на обектно-ориентирания модел от анализа не може да се прави отделно за всяка заобикаляща система и независимо от останалите системи. Също така, ако данните ще се съхраняват в обекно-ориентирана база данни, свързването на модела към избрания потребителски интерфейс не може да става независимо от конкретната СУБД.

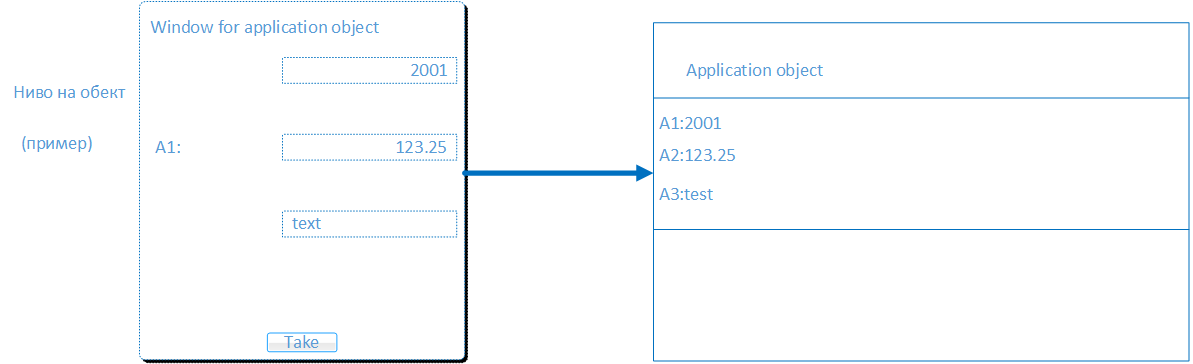
**Връзка към потребителския интерфейс**

При обектно-ориентираното проектиране като потребителска итерфейсна система трябва да се избира такава, която притежава някакъв обектно-ориентиран модул (интерфейс) за връзка с външни системи. В противен случай възниква “прекъсване” между потребителската система и обектно-ориентирания модел, което трябва да се преодолее посредством допълнителни усилия.

Връзката към потребителския интерфейс обикновено се реализира като разширение на обектно-ориентирания модел от анализа с класовете на потребителския интерфейс. Класовете на потребителския интерфейс се разглеждат като принадлежащи към класовете на самото приложение.



Фиг.3: GUI клас



Фигура 4: GUI обект

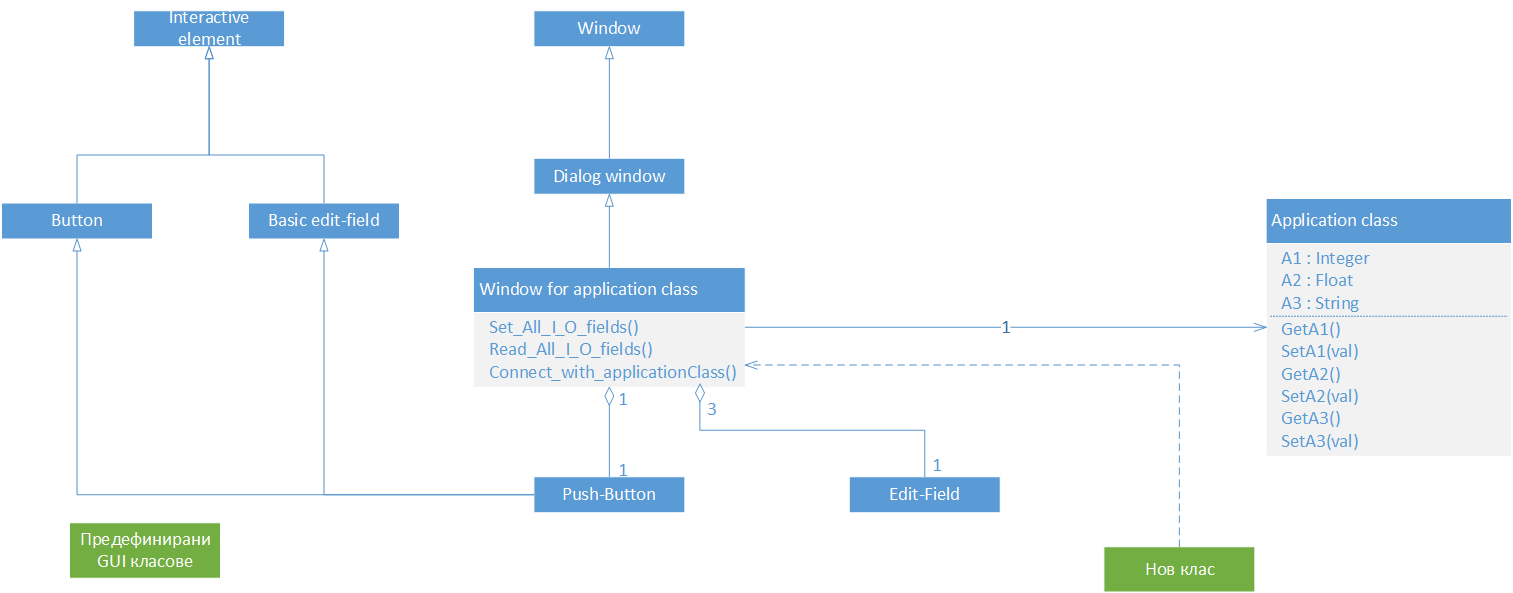
Връзката към потребителския интерфейс се разглежда от две гледни точки:

* На ниво класове
* На ниво обекти.

За да покажем тези два аспекта ще разгледаме един малък пример за потребителски интерфейс. На ниво класове са специфицирани два класа – Window for application class и Application class (Фиг.3.). Първият клас обслужва потребителския интерфейс. Негови атрибути са три входно/изходни полета  и един операционен бутон. Методите (операции) са set, get и методът за връзка с приложението Connect\_with\_application\_class(), като нито един от тях реализира бизнес-логика на приложението. Това става в класа Application class.

На ниво обекти (Фиг.4) графичният интерфейс на приложението се разглежда като  обект от тип Window for application class. Освен това се дава и начинът за неговото свързване с кореспондиращия обект от класа Application class (приема въведените данни за по-нататъшна обработка).

Следващата стъпка, която трябва да се направи, е интеграция на класовете на интерфейса в съществуващата вече йерархия на класовете за приложението. Разширена с интерфейса диаграма на класовете за примера е показана на Фиг. 5. Червената линия отделя класовете на приложението от интерфейсните класове, които са част от средата за разработка. Синята линия показва, че класовете на интерфейса трябва да са отделени от класовете на приложението.



Фигура 4: Отделяне на класовете от проектния модел и тези от аналитичния и разработката

Обобщавайки принципите, които разгледахме в примера, можем да дадем една подходяща дефиниция за обектно-ориентирания проектен модел (OOП модел). ООП моделът е OOA модел + класове на потребителския интерфейс  (GUI) + класове, специфициращи другата функционалност на разработваното приложение.

**Връзка към системата за съхраняване на данни**

Разработването на интерфейси между обектно-ориентираните приложения и използвните системи за съхраняване на големи обеми данни е  съществен проблем за много софтуерни проекти. Поради спецификата му, тук ще разгледаме накратко само някои негови аспекти.

Свързването към модулите за съхраняване на данни зависи предимно от начина, по който ще се съхраняват данните, който може да бъде:

* Обектно-ориентирана база данни
* Релационна база данни
* Файлова система.

Ако се използва обектно-ориентирана база данни, свързването на модела със съхраняването на данните е сравнително просто. Могат да се използва разширяване на обектно-ориентираните модели с класове, които поддържат персистентността на обектите.

Свързването с релационна база данни е значително по-сложно. Принципно съществуват три възможности:

* Ръчна реазлизация на свързването
* Използване на подходяща библиотека от стандартизираани класове
* Автоматично свързване посредством генератори.

**Разпределение на приложната система**

В случаите, когато се налага разпределение на разработваната система върху повече компютри, обектно-ориентираният проектен модел може да се декомпозира на два отделни модела:

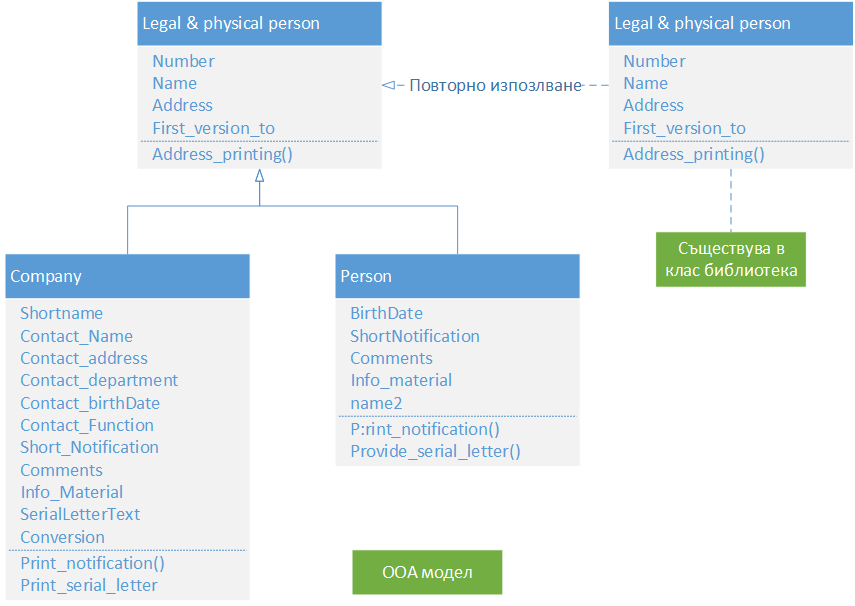
* Клиентски проектен модел
* Сървърен проектен модел.

Клиентският проектен модел обикновено се разширява с класовете, които реализират потребителския интерфейс.

Сървърният модел може да бъде разширен по различен начин, в зависимост от конкретните обстоятелства (напр. ще използваме ли middleware софтуер, като CORBA).

**Многократно използване**

Обектно-ориентираните модели могат да бъдат сравнително лесно модифицирани за многократно използване. Те могат да бъдат разширени с класове, които са многократно използваеми (reusable). Модификация на моделите може да се направи и посредством приспособяване на отделни класове за многократното им използване. Така дефиницията за OOП модел може да бъде разширена както следва: ООП модел е: модифициран OOA модел + класовете на потребителския интерфейс (GUI) + повторно използваеми класове + класове, специфициращи другата функционалност на разработваното приложение.



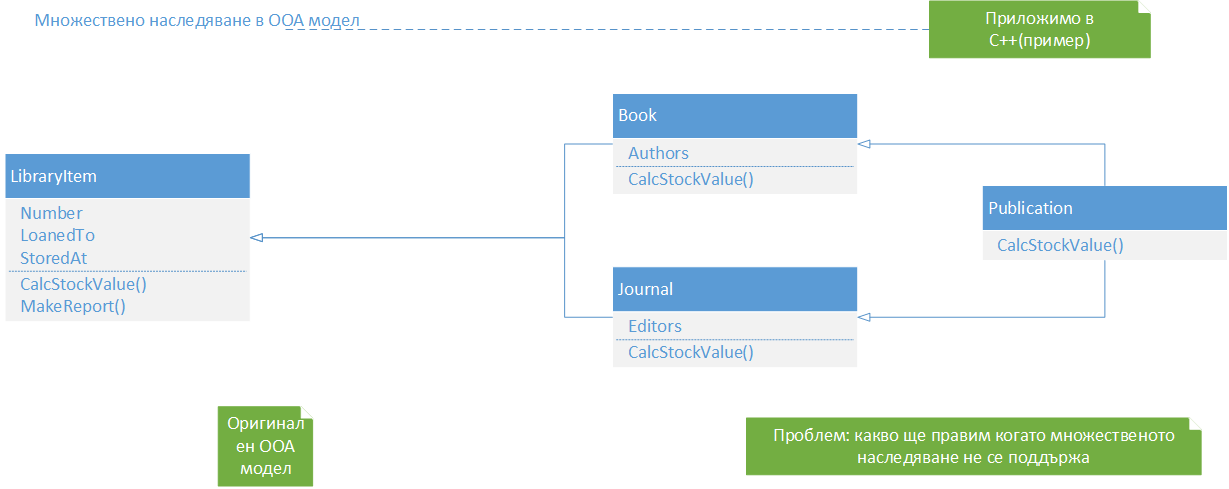
Фигура 5: Пример за използване на налични библиотеки

На Фиг.5. е дадена една възможност за използване на налични библиотеки с класове за многократно използване. Класовете Company и Person са класове, създадени по време на обектно-ориентиран анализ. Legal & physical person е клас за многократно използване. За интеграция на тези класове трябва да се изтрият някои от атрибутите и операциите на класовете Company и Person, като те стават наследници на повторно използвания клас.

**Проект на разработката**

Проектът на разработката детайлизира предложената системна архитектура, като се съблюдава спецификата на използвания език за програмиране. Най-голямо влияние върху този процес има въпросът, дали езика за програмиране поддържа механизми за наследяване. Така проектът на разработката:

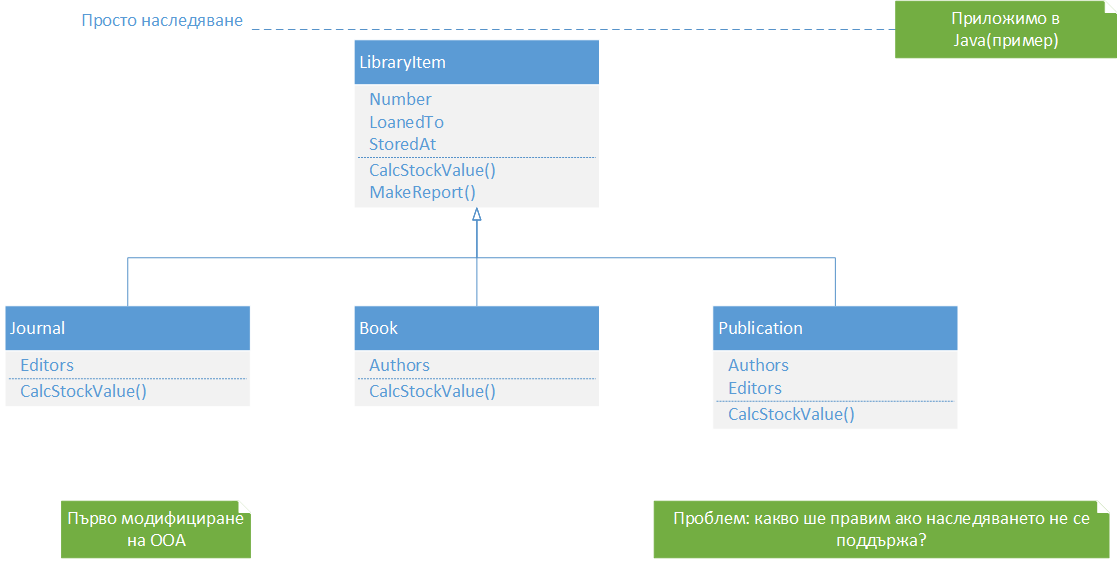
* Модифицира OOA модела
* Взема под внимание липсващите концепции в езика за разработка - напр. не се поддържа наследяване или многократно наследяване
* Съобразява се със специалните конструкции на езика за разработка - напр. правила за настройка на C, Java ...



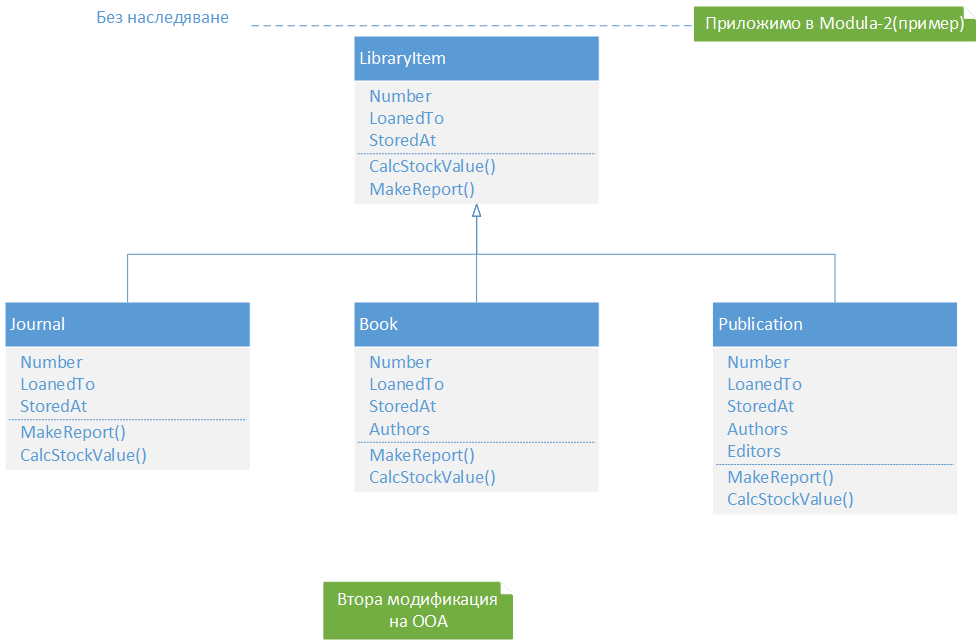
Фигура 6: Множествено наследяване в C++

Ще разгледаме един пример за решаване на проблем, възникващ от това, че езикът за програмиране не поддържа множествено наследяване. В примера (Фиг. 6.) клас Library е наследен от класовете Book и Journal, които от своя страна са наследени от класа Publication. Такова наследяване е възможно за някои езици за програмиране (напр. C++), но какво ще стане, ако езикът за програмиране не поддържа множествено наследяване (напр. Java)? Едно възможно решение е промяна на съответния аналитичен модел. За примера наследяването се променя като клас Library е наследен от класовете Book, Journal и Publication (Фиг 7.), т.е. множествено наследяване се заменя с просто наследяване. Такъв тип наследяване вече е приложимо за Java.

Остава проблемът обаче как да променим аналитичния модел, ако езика за програмиране, който ще бъде използван за разработката, не поддържа наследяване? На Фиг. 8. е представена една втора модификация на аналитичния модел, която може да се приложи напр. за езика Modula2 (не поддържа наследяване). Новият модел се състои от три класа – Book, Publication и Journal. Всички атрибути и методи на класа Library сега се намират в класа Book.



Фигура7: Наследяване в Java



Фигура 8: ООП в Modula-2

Обикновено рамка (скелет) на кода на избрания обектно-ориентиран език се генерира автоматично от диаграмата на класовете на приложението чрез използване на подходящо средство . Повечето програмни средства, поддържащи ООА и ООП модели могат да генерират рамка на програмния код от UML диаграми на класовете.

**Рамки (Frameworks)**

За решаване на различни проблеми по време на проектирането могат ефективно да се използват обектно-ориентирани стандартизирани библиотеки от класове, организирани като рамки (frameworks). Такива проблеми могат да бъдат напр. свързвания на обектно-ориентирания модел към потребителския интерфейс или към системата за съхраняване на данни (особено ефективни са при свързвания към релационните бази данни). Основната цел на разработването и използването на рамки е улесняване на многократното приложение на готови софтуерни модули. Не съществуват ограничения към многократно използваемия код (програми). Усилията са насочени към разработване на софтуерните рамки за всички фази от софтуерната разработка, като за конкретните приложни област се обособява съответстващ общ софтуер.

Тук ще дадем някои от най-популярните дефиниции за рамки:

* Система от многократно използваеми и приспособими класове и приспособими библиотеки с класове (възможност за настройка: нови подкласове, предефиниране и/или разработване на празни операции)
* Множество от взаимодействащи си класове, които могат да се използват многократно в специфичен тип софтуер. Рамките определят архитектурата на приложението, като обхващат проектни решения, които са общи за даден приложен домейн
* Общи софтуерни архитектури с общи части за изпълнение (библиотеки от класове) за конкретен клас от проблеми
* Многократно използваеми основни спецификация, проект, код и тестови шаблони за част от едно приложение.

Софтуерна архитектура, разширена с приспособими и многократно използваеми библиотеки от класове може да бъде ориентирана към определена приложна област.

Една рамка за компилатор напр., обикновено трябва да включва:

* В определена степен отворен първичен код и спецификация на целевия език
* Предварително разработени части с определени езикови конструкции, като напр.:
  + Лексикален анализ на идентификатори за Pascal, C, ...
  + Символни таблици разработени за конкретен език: блокове, наследяване, и т.н.

**Раздел 14:**

**Тестване на софтуера**

**Въведение**

Стратегията за тестване на софтуер обединява тестови случаи, проектирани в серии от стъпки, които резултират в успешни конструкции на софтуер. Стратегията представя карта, която описва стъпките, които трябва да бъдат изпълнени като част от тестването, кога тези стъпки са планирани да се изпълнят и колко усилия, време и ресурси ще са необходими. Следователно всяка стратегия за тестване трябва да съдържа планиране на тестовете, проект на тестовите случаи, изпълними тестове и резултатни данни, които се събират и оценяват.

**Стратегически подход за тестване на софтуер.**

Тестването е множество от дейности, които могат да бъдат планирани предварително и проведени систематично. Поради тази причина шаблон за тестване на софтуер – множество от стъпки, в които може да се поставят специфични проектирани тестови случаи, техники и тестови методи – трябва да бъде дефиниран за софтуерния процес.

Има много различни стратегии за тестване, но всички дават на разработчиците на софтуер шаблон за тестване и всички имат следните характеристики:

* Тестването започва на компонентно ниво
* Различните техники за тестване са подходящи в различни точки във времето
* Тестовете се провеждат от разработчиците на софтуера и (за големи проекти) от независима група за тестване
* Тестването и дебъгването са различни дейности, но дебъгването трябва да бъде приспособено към стратегията за тестване.

Стратегията за софтуерно тестване трябва да включва високо ниво тестове, които са необходими както за верифицирането на коректното имплементиране на малки сегменти от сорс код, така и тестове от по-високо ниво, които да валидират основните системни функции.

**Организация на софтуерното тестване**

Тестването се провежда от разработчиците на софтуера, но за по-голяма сигурност при големи проекти се включва и специална тестваща група.   
Разработчиците на софтуера винаги отговарят за индивидуалните модули от програмата, осигурявайки, че всеки от тях изпълнява функцията, за която е проектиран. В много случаи разработчиците провеждат интеграционни тестове – тестове, които водят до построяване и тестване на завършената програмна структура. Само когато софтуерната архитектура е завършена се намесва независимата тестваща група.

Ролята на тази независима тестваща група е да отстрани останалите проблеми след тестването на разработчиците. Независимата група за тестване отстранява конфликти на интереси, които могат да се появят. Разработчиците и независимата тестваща група работят заедно през целия проект, за да са сигурни, че правилните тестове ще бъдат проведени. Докато се провежда тестването на системата, разработчиците трябва да бъдат на разположение, за да коригират грешките. Групата за тестване на софтуера е част от екипа за разработване на софтуер.

**Стратегия за тестване на софтуер.**

Стратегията за тестване също може да бъде представена като спирала фигура 1. Единичните тестове (Unit testing) започват във вътрешността на спиралата и се концентрират върху всяка единица (компонент) на софтуера. Движейки се навън по спиралата се преминава към интеграционните тестове (Integration testing), които се фокусират върху проектирането и изграждането на софтуерната архитектура. Правейки още един кръг навън по спиралата се стига до валидационните тестове (Validation testing), където изискванията са установени като част от софтуерните изисквания получени по време на анализа и се валидират със софтуера, който е разработен. Накрая се стига до системните тестове (System testing), където софтуера и други системни елементи се тестват като цяло.

**Критерии за приключване на тестването**

Класическия въпрос тук е: Кога да приключи тестването? – как да е сигурно, че  тестването е било достатъчно. За съжаление няма дефиниран отговор на този въпрос, но има няколко прагматични отговора.

Първия отговор на горния въпрос е “Никога не приключвате тестването, тежестта пада от вас на потребителя.” Друг отговор е “Приключвате тестването, когато изразходите времето си или парите за проекта.”  
Използвайки статистическо моделиране и теорията за надеждност на софтуера, модела за софтуерен неуспех (непокрити грешки по време на тестването) може да се представи като функция от времето на изпълнение. Версия на този модел има следната форма:

1. f(t) = (1/p) ln [ l0 pt + 1],

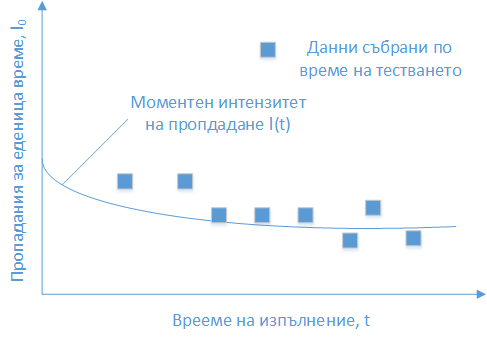
където

f(t) – кумулативния (натрупващ се) брой на пропадания, които се очаква да се срещнат .  
l0 – начален софтуерен интензитет на пропаданията (пропаданията за единица време) до започването на тестването.  
p – експоненциа  
лно намаляване на интензитета на пропаданията, като грешките са открити и поправките са направени.

Моментния интензитет на пропадане l(t) може да бъде получен като производна на f(t)

1. l(t) = l0 / (l0 pt + 1)

Използвайки връзката в уравнение 2 , тестващите могат да предскажат премахването на грешки като прогрес на тестването. Актуалния интензитет на грешките може да бъде сравнен с предсказаната крива на фигура 1:



Фигура 1: Интензитета на пропадане като функция на t

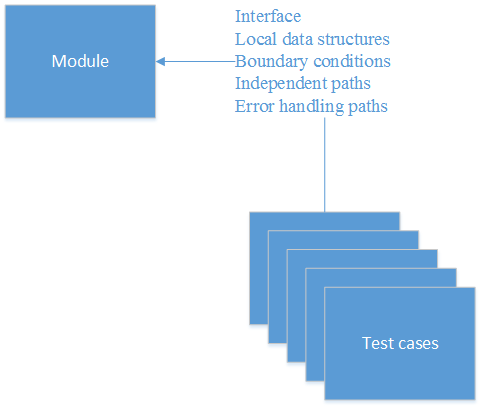
Чрез събиране на метрики по време на тестването на системата и използването на съществуващия модел за сигурност на софтуера, е възможно да разработим ръководство за отговор на въпроса “Кога да приключим с тестването?”.

**Единични тестове (Unit testing).**

Единичните (Unit) тестове съсредоточават усилията си върху най‑малките единици на софтуерния проект – софтуерните компоненти или модули.

**Характеристика на unit тестовете**

Тестовете, които се провеждат като части на този вид тестване са дадени схематично на Фигура 2:



Фигура 2: Unit test

* Module interface – за осигуряване на правилното движение на информацията в и от тестваната програмна единица.
* Local data structure – временно съхранените данни поддържат цялостността си по време на изпълнение на всички стъпки на алгоритъма.
* Boundary conditions – програмната единица оперира правилно на границите, установени да лимитират или ограничат обработката.
* Independent paths – всички изложения в програмната единица са поне веднъж изпълнени.
* Error handling paths.

Тестването на потока от данни през интерфейса на модула се изисква преди да се инициира кой да е друг вид тест. Ако данните не влизат или излизат коректно, тогава всички други тестове са спорни.

По време на unit тестването се изследват локалните структури от данни. Освен това евентуално може да се установи и локалното въздействие върху глобалните данни.

Една съществена задача е избирателното тестване на изпълнимите пътища. Трябва да се разработят тестови случаи за откриване на грешки, дължащи се на грешни изчисления, некоректни сравнения или неверни контролни потоци. Базов път (basis path) и тестване на цикли (loop testing) са ефективни техники за откриване на широка област от грешки, свързани с пътищата.

Най-често срещаните грешки при изчисленията са следните:

* грешно разбрани или некоректни аритметични приоритети;
* смесване на типове оператори;
* некоректни инициализации;
* неприцизност;
* некоректни символни представяния на изрази.

Сравнянията и контролните потоци са тясно свързани – често се налага смяна на контролния поток след сравнение.

Тестовите случаи трябва да откриват следните типове грешки:

* сравнения на различни типове данни;
* некоректни логически оператори или приоритети;
* очакване на еквивалетност, когато грешка в прецизността прави сравнението невярно;
* некоректно сравняване на променливи;
* неподходящо или несъществуващо завършване на цикли;
* грешки при напускане на несходни итерации;
* неподходяща промяна на променливи, управляващи цикли.

Един добър развой предполага, че условията за грешки са очаквани и се инициализират пътища за обработка на грешки (error-handling paths) за премаршрутизиране или чисто завършване на обработката, когато се появи определена грешка. Този подход се нарича antibugging. За съжаление често срещана практика е да се интегрират такива елементи в програмите, но да не се тестват изобщо или недостатъчно.

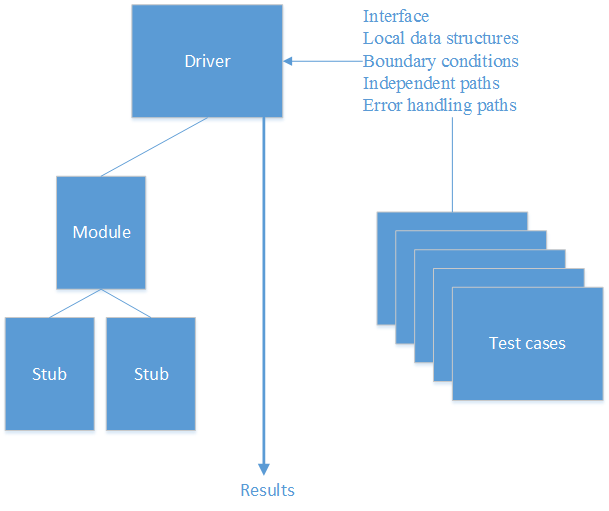
Някои от потенциалните грешки, които трябва да се тестват при error handling са следните:

* смислено описание на грешките;
* отбелязаните грешки не кореспондират на срещнатите грешки;
* грешно условие пречинява системна интервенция преди обработка на грешката;
* обработката на изключение е некоректно;
* описание на грешка не доставя достатъчно информация за подпомагане на определянето на причина за грешка..

Boundary testing – последната (и евентуално най-важната) задача на unit тестването. Софтуерът често е грешен при граничните стойности.

**Процес за провеждане на единични тестове (unit test).**

Понеже един компонент не е самостоятелна програма, за всеки unit тест трябва да се разработи определена тестова среда (Фигура 3).



Фигура 3: Среда за провеждане на unit тестове.

В повечето приложения драйверът не е нищо повече освен една “главна програма” (main program), която:

* акцептира данни на тестовия случай;
* предава тези данни на компонента, който се тества;
* отпечатва съществените резултати.

Stubs служат за заместване на модулите, които са подчинени на тествания компонент. Един stub или “dummy subprogram”:

* използва подчинения модулен интерфейс;
* може да прави минимални манипулации с данни;
* печата верификации на входни точки;
* връща контрола към подлежащия на тестване модул.

 Драйверите и стубовете са носители на оверхед (overhead), т.е. те трябва да бъдат програмирани, но не се включват в крайния софтуерен продукт. Ако те се оставят прости, действителният оверхед е малък. За нещастие много компоненти не могат да бъдат адекватно тествани с малък оверхед. В такива случаи цялостнито тестване може да бъде отложено за интеграционното тестване (където също така се използват драйвери и стубове).

Unit тестването се опростява когато се разработват компоненти с висока съгласуваност. Когато в един компонент се разглежда само една функция, тогава броят на тестовите случаи се редуцира, което от своя страна води до лесно предсказване и откриване на грешките.

**Интеграционни тестове (Integration testing)**

Интеграционното тестване е една техника за конструиране на програмни структури, която същевременно управлява тестове за откриване на грешки, свързани с интерфейсите. Целта е да се вземат unit тествани компоненти и да се изгради една програмна структура, която е наложена от развоя.

Причини за интеграционно тестване са:

* между интерфейсите могат да се губят данни;
* определен модул може да има непреднамерено нежелано въздействие върху другите модули;
* когато се комбинират подфункции (subfunctions) може да не се получи желаната главна функция (main function);
* индивидуално акцептирана неточност може да бъде усилена до неприемливо ниво;
* глобални структури от данни могат да предизвикат проблеми.

Този списък може да бъде продължен.

Съществува тенденция да се опитва неинкрементална интеграция (non-incremental integration), т.е. да се конструират програми, използвайки “big bang” подход. При него всички компоненти се комбинират предварително, след което цялата програма се тества като едно цяло. Обикновено това резултира в един хаос. Голямо множество от грешки остава неоткрито. Корекции се правят много трудно, понеже изолирането на причините се усложнява от бързото нарастване на програмата. Много често процесът влиза в безкраен цикъл.

Инкременталната интеграция е антитеза на “big bang” подхода. Програмите се конструират и тестват с малки нараствания (инкременти - increments), където:

* лесно се изолират и коригират грешките;
* интерфейсите могат по-подходящо да се тестват напълно;
* може да се приложи систематичен подход за тестване.

**Top-Down интеграция.**

Top-down integration testing е инкрементален подход за конструктуиране на програмни структури. Модулите се интегрират посредством преместване надолу през контролната йерархия, като се започне от главния контролен модул – главната програма (main control module – main program). Модулите, подчинени на главния контролен модул, се интегрират в структурата по един от двата начина:

* Първо в дълбочина (depth-first)
* Първо в широчина (breadth-first).

Интеграционният процес се извършва като последователност от следните пет стъпки:

* Главният контролен модул се използва като тестови драйвер, а всички компоненти, директно подчинени на него, се заместват временно със стубове (stubs).
* В зависимост от избрания подход за интеграция заместващите стубове (subs) се заместват последователно и поотделно от съответните актуални компоненти.
* Тестовете се направляват, когато всеки компонент е интегриран.
* При завършване на всяко множество от тестове, следващ стуб (stub) се замества от реален компонент.
* Регресионното тестване може да се използва за осигуряване, че няма да се въведат нови грешки.
* Процесът продължава във втора стъпка, докато се създаде цялата програма.

При top-down интеграционната стратегия се верифицира първо главният контрол или точките за решение (decision points). В една добре декомпозирана програмна структура, вземането на решение става на по-горните нива в йерархията. Ако съществуват проблеми с главния контрол, тогава ранното им разпознаване е съществено.

Top down стратегията изглежда сравнително проста, но на практика могат да възникнат някои логистични проблеми. Най-съществените от тях се появяват, когато за адекватно тестване на по-горните нива се изисква обработка на ниските нива. Стубове заместват модулите на ниски нива в началото на top down тестването. Следователно несъществени данни могат да протичат в посока нагоре в програмната структура. Тестващите са конфронтирани с три възможности за избор:

* Забавя много тестове докато стубовете (stubs) се заместят с актуални модули
* Разработват се стубове (stubs), които извършват ограничени функции, симулиращи актуалните модули
* Интегриране на софтуер от по-долните слоеве на йерархията

**Bottom-Up интеграция.**

Bottom-Up integration testing започва конструирането и тестването с атомарни модули (atomic modules), т.е. компоненти на най-ниско ниво в структурата на програмата. Понеже компонентите се интегрират от долу нагоре, обработката, необходима за субкоординация на компонентите на едно дадено ниво е винаги налична и не са необходими стубове(stubs).

Една bottom-up интеграционна стратегия може да бъде имплементирана на следните стъпки:

* Компонентите на ниско ниво се комбинират в клъстери (наричани също така builds), които извършват една специфична софтуерна подфункция.
* Един драйвер (контролна програма за тестване) се създава за координиране на входа и изхода на тестовите случаи.
* Тества се клъстерът
* Драйверите се отстраняват и клъстерите се комбинират, премествайки се нагоре в програмната структура.

**Регресионни тестове (Regression Testing).**

Всеки път, когато се добави един нов модул като част на интеграционния тест, софтуерът се променя. Появяват се нови пътища на потоците данни, могат да се осъществят нови В/И операции, както и да се извика нова контролна логика. Тези промени могат да причинят проблеми с функциите, които преди са функционирали безупречно.

В контекста на една интеграционна стратегия за тестване regression testing е повторно изпълнение на едно подмножество от вече изпълнени тестове, целта на което е да се осигури, че промените няма да разпространят непредвидени странични ефекти.

В един по общ контекст, успешните тестове резултират в намиране на различни грешки, които трябва да бъдат коригирани. Всеки път, когато се коригира софтуерът се променят някои аспекти на конфигурацията му (програма, документация, данни). Регресионното тестване е дейността, която помага да се осигури, че тези промени не са въвели непредвидено поведение или допълнителни грешки.

Регресионното тестване може да бъде управлявано ръчно (повторно изпълнение на някои вече проведени тестове) или посредством използване на автоматизирани средства (capture-playback tools). Тези средства позволяват на системните инженери да обхващат тестови случаи и резултати от последователен плайбек и да правят сравнения.

Сюитата на регресионния тест (regression test suite) (подмножество от тестове, които ще се изпълнят) може да съдържа три различни класове от тестови случаи:

* Представителен образец от тестове, които ще изпълнят всички софтуерни функции.
* Допълнителни тестове, фокусиращи се върху функциите, които вероятно са били засегнати от промените.
* Тестове, фокусиращи се върху софтуерните компоненти, които са променени.

С напредване на регресионното тестване броят на регресионните тестове може да нарасне значително. Следователно тестовата сюита трябва да бъде така разработена, че да включва само тези тестове, които адресират един или повече класове грешки във всяка основна програмна функция. Не е практично и е неефективно да се изпълнява повторно всеки тест за всяка функция.

**Валидационни тестове (Validation Testing).**

В кулминацията на интеграционното тестване софтуерът се асемблира напълно като един пакет. Интерфейсните грешки са разкрити и коригирани. Сега може да започне една финална серия от тестове – валидационни тестове (validation testing).

Валидирането (validation) може да бъде дефинирано по различен начин, но една сравнително проста дефиниция е, че валидирането успява когато функциите се държат по начин, логично очакван от потребителите. Кой е арбитърът за логичните очаквания?  
Логичните очаквания се дефинират в документа Software Requirements Specification, където са описани видимите за потребителите атрибути на софтуера. Този документ съдържа раздел, наречен Validation Criteria. Информацията от този раздел формира базата за валидационното тестване.

**Критерии за Validation Test**

Валидирането на софтуера се извършва посредством поредица от black-box тестове, които демонстрират съответствието с изискванията. Тестови план очертава класовете на тестовете, които ще се управляват. Тестова процедура дефинира специфични тестови случаи, които ще се използват за демонстриране на съответствието с изискванията. Планът и процедурата се разработват за осигуряване, че:

* всички функционални изисквания са удовлетворени;
* всички поведенчески характеристики са постигнати;
* всички производствени изисквания са постигнати;
* документацията е коректна;
* изпълнени са допълнителни изисквания (преносимост, съвместимост, ...).

След извършване на всеки валидиращ тестови случай, съществува едно от двете възможни условия:

* Функционалните или производствените характеристики съответстват на спецификацията и се приемат.
* Разкрито е отклонение от спецификацията и се създава един списък с недостатъците (deficiency list).

За отстраняване на грешките често се налага споразумение с потребителите.

**Конфигурационен преглед**

Съществен елемент на валидационния процес е конфигурационния преглед. Целта на прегледа е да осигури , че всички елементи на софтуерната конфигурация са разработени правилно, каталогизирани са и имат необходимите детайли да подкрепят поддържащата фаза на жизнения цикъл.

**Alpha and Beta тестове.**

За разработчиците практически е невъзможно да предвидят как потребителите реално ще използват една програма:

* могат грешно да бъдат интерпретирани инструкции;
* редовно могат да се използват непривични комбинации от данни;
* изход, който изглежда ясен за тестващите, може да бъде неясен за потребителите.

Когато се разработва софтуер за определен потребител е необходимо да се извършат множество от тестове за приемане (acceptance tests), за да се даде възможност на потребителите да валидират всички свои изисквания. Тези тестове се управляват по-скоро от крайните потребители, отколкото от софтуерните инженери. Те могат да се разпростират от неформални “test drive” до една планирана и систематично изпълнена последователност от тестове. Продължителността на тези тестове може да бъде от седмици до месеци.

Ако софтуерът се разработва за повече потребители, не е ефективно да се провеждат формални тестове за приемане за всеки отделен потребител. Вместо това е целесъобразно да се извършат алфа (alpha) и бета (beta) тестове.

Алфа тестовете се извършват от потребителите на мястото на разработването. Тестовете се извършват по принцип от разработчиците в присъствието на потребители (“looking over the shoulder”). Алфа тестовете се извършват в контролирана среда.

Бета тестовете се извършват от крайните потребители на едно или повече потребителски места. За разлика от алфа тестовете в общия случай разработчиците не присъстват. Следователно бета тестовете представляват тестване на “живата” система в среда, която не може да бъде контролирана от разработчиците. Потребителите записват откритите грешки в специални репорти, които се предоставят на разработчиците. На основата на тези репорти се коригират откритите грешки и се подготвят нови реализации/версии (releases) на продукта.

### Системни тестове (System Testing)

Софтуерът е само един елемент от една голяма компютърна система. В крайна сметка софтуерът се интегрира с другите системни компоненти (хардуер, персонал, информация), при което е необходимо да се извърши една поредица от системни интеграционни и валидационни тестове. Тези тестове надхвърлят рамката на развойния софтуерен процес и не се извършват единствено от софтуерните инженери. Обаче, извършването на някои дейности по време на разработването и тестването на софтуера, могат в значителна степен да подобрят вероятността за успешно интегриране на софтуера в една голяма система.

### Възстановителни тестове (Recovery testing)

Много компютърни системи трябва да се възстановяват от сривове и да продължат обработката в рамките на ограничено време. В някои случаи системата трябва да бъде устойчива на проблеми т.е. обработката на грешките не трябва да блокира цялата система. В други случаи срив в системата трябва да бъде бързо отстранен в рамките на предварително зададен период от време, в противен случай могат да възникнат големи икономически загуби.   
Recovery тестовете са системни тестове, които предизвикват срив на системата по различни начини и проверяват дали възстановяването се извършва правилно. Ако възстановяването е автоматично (т.е. извършва се от самата система), трябва да се провери всяка стъпка от процеса за възстановяване за коректност по отделно. Ако възстановяването изисква намесата на човек, то общото време за възстановяване, трябва да бъде проверено дали е в приемливи граници.

### Защитни тестове (Security testing)

Това са тестове, които се опитват да верифицират защитните механизми на системата. По време на тези тестове, тестващите играят ролята на някой, който иска да проникне в системата. Тестващите могат да се опитата да достигнат до паролата през външни потребители, могат да атакуват системата с клиент ски софтуер като разбият защитата, която е била разработена и други подобни.

Тук ролята на проектантите на системата е да определят цената за проникване в системата на по-висока от тази на информацията, с която ще се сдобият евентуалните хакери на системата.

### Стресови тестове (Stress testing)

По време на ранните стъпки на тестване на софтуера white-box и black-box техниките за тестване резултират в оценка на нормалните програмни функционалности и производителност. Stress тестовете се разработват, за да конфронтират програмата с анормални ситуации.  
Stress тестовете изпълняват програмата в случаи, които изискват ресурси в анормално количество, честота или сила. Тези тестове се наричат още sensitivity тестове. Те се опитват да открият комбинации от данни, които са валидни, но могат да предизвикат нестабилност или неправилна обработка.

### Производителни тестове (Performance testing)

Това са тестове, които тестват производителността на системата в контекста на интегрираната система. Тези тестове се провеждат при всяка стъпка от стратегията за тестване, дори при unit тестовете. Производителността на индивидуалния модул може да бъде оценена при провеждане на white-box тестове. Но трябва да се има предвид, че реалната производителност на системата не може да бъде оценена, докато не се интегрират всички модули.

Много често тестовете за производителност се комбинират с strеss тестовете и обикновено засягат използвания хардуер и софтуер.

### Заключение:

Целта на тестването на софтуер е да се открият допуснатите грешки при разработката му. За да бъде постигната тази цел се планират и изпълняват серия от тестови стъпки – unit, integration, validation и system тестове. Unit и integration тестовете се занимават с функционалната верификация на модулите и обединението им в програмни структури. Validation тестовете проследяват софтуерните изисквания, а системните тестове валидират софтуера как се интегрира в големи системи.  
Всяка стъпка на тестването е съпроводена от серия систематични тестови техники, които участват в проектирането на тестови случаи. С всяка стъпка на тестване, нивото на абстракция на софтуера се разширява.

**Раздел 15:**

**Въведение**

Въпреки, че при обектно-ориентираното тестване целта остава същата, естеството на обектно-ориентираните програми налага промяна на стратегията и подходите за тестване.

Тестването на обектно-ориентирани система предоставя нови предизвикателства към софтуерните инженери. Дефиницията за тестване трябва да бъде разширена така, че да включва техники за откриване на грешки, приложими за обектно-ориентираните модели. По време на развоя трябва да бъдат оценени пълнотата и целостността на обектно-ориентираните представяния. Unit testing губи много от неговото значение, а стратегиите за интеграция се променят значително.

**Разширяване на виждането за тестването**

Конструирането на обектно-ориентиран софтуер започва със създаване на аналитични (analysis models) и проектни модели (design models). Поради еволюционната природа на обектно-ориентираната парадигма за развой на софтуер, тези модели започват развитието си като едно сравнително неформално представяне на изискванията към системата и се развиват в детайлни модели на класове, връзки и релации между класове, системно проектиране и разпределяне, проектиране на обекти. На всяка отделна стъпка моделите могат да бъдат тествани, с цел ранното откриване на грешките и неразпространяването им към следващите итерации.

Нека разгледаме например клас, в който са дефинирани няколко атрибута по време на първата итерация на обектно-ориентиран анализ. Външен атрибут е добавен към класа поради недостатъчното разбиране на проблемния домейн. Следователно са дефинирани две операции, които манипулират този атрибут. Експертите са провели преглед на аналитичния модел и са определили, че в модела има грешка. Чрез изключване на външния атрибут от модела на този етап от разработката на системата, следните проблеми и ненужни усилия могат да бъдат избегнати по време на анализа:

* Специален подклас може да е бил генериран за разполагане на ненужния атрибут или изключения за него. Избягва се работата за създаване на ненужни подкласове.
* Неправилното интерпретиране на дефинициите за класа може да доведе до некоректни или външни връзки между класовете.
* Поведението на системата или на нейните класове може да бъде неправилно определено за разположението на външния атрибут.

Ако грешката не се открие по време на анализа и се разпространи в по нататъшната работа, следните проблеми могат да възникнат по време на проектирането:

* Неправилно разпределяне на класа към подсистема и/или задача, може да възникне по време на системния анализ.
* Ще се наложи допълнителна ненужна проектна работа за процедурно проектиране на операциите, които се обръщат към външния атрибут.
* Модела на съобщенията (messaging model) ще бъде некоректен (защото съобщенията трябва да бъдат проектирани за външни операции).

Ако грешката не бъде открита по време на проектирането и бъде пропусната във етапа на кодиране, значителни усилия ще бъдат хвърлени за разработване на ненужен атрибут, две ненужни операции, съобщения, които направляват комуникацията между обектите и други свързани последствия. Освен това тестването на класовете ще отнеме повече от необходимото време. След като проблема бъде открит, модифицирането на системата ще доведе до странични ефекти предизвикани от промените.

Всички обектно-ориентирани модели трябва да бъдат тествани за коректност, цялостност и консистентност в контекста на синтаксиса, семантиката и прагматиката им.

**Тестване на обектно-ориентираните аналитични и проектни модели**

Аналитичните и проектните модели не могат да се тестват в обичайния смисъл, понеже те не могат да бъдат обработени. За изследване на коректността и целостността на тази модели се използват предимно формални технически прегледи (technical reviews).

**Коректност на обектно-ориентираните аналитични и проектни модели.**

Нотацията и синтаксисът, които се използват за представяне на аналитичните и проектните модели, трябва да бъдат свързани със специфичния метод, избран за разработване на проекта. Понеже за синтактичната коректност (syntactic correctness) се съди по характерната система от символи, тогава всеки модел трябва да осигури характерните конвенции за моделиране.

По време на анализа и проектирането семантичната коректност (semantic correctness) се оценява на основата на съответствието на модела с реалната проблемна област. Ако моделът отразява коректно реалния свят (на определената степен на детайлизация), тогава той е семантично коректен. За да се определи дали моделът действително отразява реалния свят, той трябва да бъде предоставен на експерти от проблемната област, които да изследват:

* Дефинициите на класовете и йерархията за пропуски и двусмислия
* Връзките между класовете дали правилно отразяват връзките между реалните класове.

**Консистентност на обектно-ориентираните аналитични и проектни модели.**

Целостността на обектно-ориентираните аналитични и проектни модели може да се оцени посредством разглеждане на връзките между идентичностите в модела. Нецялостният модел има представяния в една част, които не се отразяват коректно в другите части на модела.

За да се оцени целостността, всеки клас и неговите връзки към другите класове трябва да бъдат изследвани. За подпомагане на тази дейност могат да се използват:

* Class-responsibility-collaboration (CRC) модел
* Object-relationship диаграмите.

За оценяване на модела на класовете (class model) се препоръчват следните стъпки:

1. Ревизия на CRC модела и на object-relationship модела
2. Преглеждане на описанието на всяка CRC индексна карта за определяне дали делегирана отговорност е част на дефиницията за сътрудничество (collaborator’s definition)
3. Обръщане на връзката за осигуряване, че на всеки сътрудник, който иска услуга, се изпращат заявки от отговорния източник
4. Използване на инвертираните връзки (изследвани в 3), за да се установи, дали други класове могат да се изискват или дали отговорностите са присъщо групирани между класовете.
5. Определяне дали широко изисквани отговорности могат да се комбинират в една единична отговорност
6. Стъпките 1-5 се прилагат итеративно към всеки клас и през всяка еволюция на обектно-ориентирания аналитичен модел.

След като веднъж е разработен обектно-ориентирания проектен модел, тогава могат да се правят прегледи на системния проект и на обектния проект. Системния проект изобразява:

* Общата архитектура на системата
* Подсистемите, на които е декомпозирана
* Начина, по който подсистемите се разполагат върху машините
* Разполагането на класовете в подсистемите
* Проектирането на потребителския интерфейс.

Обектният модел представя:

* Детайлите на всеки клас
* Messaging activities – необходими за реализация на сътрудничеството между класовете.

Системното проектиране се преглежда посредством изследване на модела на поведението на обектите (object-behaviour model), разработен по време на обектно-ориентирания анализ и съпоставяне на изискваното поведение на системата на фона на подсистемите, реализиращи това поведение. В контекста на системното поведение се разглеждат също така конкуретността (concurrency) и разполагането на задачите (task allocation). Състоянията на поведението на системата се оценяват за определяне, кои от тях съществуват едновременно (concurrently). За изследване на проектирането на потребителския интерфейс се използват use-case сценарии.

Обектният модел трябва да се тества на фона на object-relationship network за осигуряване, че всички проектирани обекти съдържат необходимите атрибути и операции за реализиране на сътрудничеството, дефинирано за всяка CRC индексна карта. Освен това се разглежда подробната спецификация на детайлите на операциите (т.е. алгоритъмът, който имплементира операциите), използвайки конвенционални техники за инспекция.

**Обектно-ориентирани стратегии за тестване**

Класическата стратегия за тестване започва с “тестване в малкото” (testing in the small) и продължава към “тестване в голямото” (testing in the large). Т.е. :

* Започваме с unit testing
* Продължаваме с интеграционно тестване
* Завършваме с валидационни и системни тестове.

**Интеграционни тестове (Integration Testing) в контекста на обектно-ориентирания развой.**

Понеже обектно-ориентираният софтуер няма йерархични контролни структури, конвенционалните top-down и bottom-up стратегии за интеграция нямат съществено значение. Освен това последователното интегриране на отделните операции в един клас често е невъзможно.

Съществуват две основни стратегии за интегрирано тестване на обектно-ориентираните системи:

* Thread-based testing – интегрира множеството от класове, които са необходими за отговор на един вход или събитие към системата. Всяка нишка (thread) се интегрира и тества индивидуално. За да се осигури, че не се появяват странични ефекти, може да се приложи регресивно тестване.
* Use-based testing – изграждането на системата започва чрез тестване на тези класове наречени независими класове (independent classes), които използват много малко (ако въобще има такива) сървърни класове. След тестването на независимите класове, следващият слой класове, които се тестват са зависимите класове (dependent classes), които използват тестваните вече независими класове. Последователно се тестват следващи слоеве от зависими класове докато се конструира цялата система.

Cluster testing е една стъпка в тестването на обектно-ориентиран софтуер. При нея един клъстер от сътрудничещи си класове (определени чрез изследване на CRC и object-relationship модела) се анализира посредством тестови случаи, които се опитват да открият грешки в сътрудничеството.

**Интеграционни тестове (Integration Testing) в контекста на обектно-ориентирания развой.**

Понеже обектно-ориентираният софтуер няма йерархични контролни структури, конвенционалните top-down и bottom-up стратегии за интеграция нямат съществено значение. Освен това последователното интегриране на отделните операции в един клас често е невъзможно.

Съществуват две основни стратегии за интегрирано тестване на обектно-ориентираните системи:

* Thread-based testing – интегрира множеството от класове, които са необходими за отговор на един вход или събитие към системата. Всяка нишка (thread) се интегрира и тества индивидуално. За да се осигури, че не се появяват странични ефекти, може да се приложи регресивно тестване.
* Use-based testing – изграждането на системата започва чрез тестване на тези класове наречени независими класове (independent classes), които използват много малко (ако въобще има такива) сървърни класове. След тестването на независимите класове, следващият слой класове, които се тестват са зависимите класове (dependent classes), които използват тестваните вече независими класове. Последователно се тестват следващи слоеве от зависими класове докато се конструира цялата система.

Cluster testing е една стъпка в тестването на обектно-ориентиран софтуер. При нея един клъстер от сътрудничещи си класове (определени чрез изследване на CRC и object-relationship модела) се анализира посредством тестови случаи, които се опитват да открият грешки в сътрудничеството.

### Валидационни тестове (Validation Testing) в контекста на обектно-ориентирания развой

На валидационно или системно ниво изчезват детайлите за връзките (connections) между  класовете. Подобно на конвенционалното валидиране, валидирането се фокусира върху видимите за потребителите действия и разпознаваеми изходи на системата. За подпомагане на получаването на валидационни тестове, тестващите трябва да ползва use-cases, които са част от аналитичния модел. Те доставят един сценарий, който помага за откриването на неоткрити грешки в изискванията за потребителско взаимодействие.

Могат да се използват black-box методи за тестване. Освен това могат да се извличат тестови случаи от object-behaviour модела и от event flow диаграмите, създадени като част на обектно-ориентираните аналитични модели.

**Проектиране на тестови случаи за обектно-ориентиран софтуер**

Методите за разработване на тестови случаи за обектно-ориентирани системи все още се развиват. Един общ подход е следният:

* Всеки тестови случай трябва да бъде уникално идентифицируем и трябва да бъде явно свързан с класа, който ще се тества
* Трябва да бъде обявена целта на теста
* За всеки тест трябва да се създаде един списък от тестови стъпки, който трябва да съдържа:
  + Списък на специфицирани състояния за обекта, който ще се тества
  + Списък от съобщения и операции, които ще бъдат изследвани като резултат от теста
  + Списък от изключения, които могат да възникнат при тестване на обекта
  + Списък от външни условия (т.е. промени в средата, външна за софтуера, който трябва да съществува за правилното управление на теста)
  + Допълнителна информация, която ще подпомогне разбирането и изпълнението на теста.

За разлика от създаването на конвенционални тестови случаи, които се направляват от аспекта на вход-процес-изход (input-process-output) или алгоритмичните детайли на отделните модули, обектно-ориентираното тестване се фокусира върху създаването на подходящи последователности от операции за изследване на състоянията на един клас.

**Изводи за създаване на тестови случаи за ОО концепции**

ОО класове са целта на създаването на тестови случаи. Понеже атрибутите и операциите са капсулирани, тестването извън класовете в общия случай не е продуктивно. Въпреки, че капсулирането е една съществена развойна концепция за ОО подход, тя може да създаде определени пречки за тестването. Тестването изисква докладване за конкретните и абстрактните състояния на обектите. Капсулирането обаче, може да затрудни получаването на тази информация. Освен когато built-in операции са предоставени за съобщаване на стойностите на атрибутите на класовете, моментална снимка на състоянието на обектите може да се получи трудно.

Наследяването води също така до допълнителни предизвикателства  за тестващите. Както вече беше споменато, всеки нов контекст на използване изисква повторно тестване, дори при многократно използваеми компоненти. Многократното наследяване допълнително усложнява тестването чрез увеличаване броя на контекстите. Когато инстанцираните от един суперклас подкласове се използват в същата проблемна област, тогава е възможно множеството от тестови случаи, изведени за суперкласа да се използва за тестване на подкласовете. Ако обаче, подкласовете се използват в съвсем различен контекст, тестовите случаи за суперкласовете са слабо приложими. В такива ситуации, трябва да се разработват нови тестови случаи.

**Приложение на конвенционалните методи за разработване на тестови случаи**

White-box методите за тестване могат да се прилагат за операциите, дефинирани в един клас. Базови пътища, тестване на цикли (loop testing) или техники за потоци данни могат да помогнат при осигуряване, че всеки израз в един оператор ще бъде тестван. Сбитата структура на повече класови операции води до твърдения, че е по-добре да се пренасочат усилията, необходими за white-box тестването, към тестове на ниво класове.

Black-box методите за тестване са подходящи за ОО системи. Use-cases могат да доставят полезен вход при разработването на black-box и state-based тестове.

**Fault-Based Testing**

Целта на fault-based тестването в една ОО система е да се разработят тестове, които да имат голямо подобие на неоткритите правдоподобни грешки. Понеже системите трябва да отговарят на изискванията на потребителите, предварителното планиране, необходимо за извършване на fault-based тестване започва  с аналитичните модели. Тестващите търсят правдоподобни грешки (т.е. аспекти на реализацията, които могат да резултират в грешки). За да се определи дали тези грешки съществуват, тестовите случаи се разработват за изследване на проектирането или кодирането.

Ефективността на тези техники зависи от това, как тестващите схващат “правдоподобните тестове”. Ако реалните грешки в една ОО система се възприемат като неправдоподобни, тогава този подход реално не е по-добър от коя да е вероятностна техника за тестване. Ако обаче аналитичните и проектните модели могат да доставят проницателност за това, което вероятно ще бъде грешно, тогава този вид тестване може да намери значителен брой грешки със сравнително ниски разходи.

Интеграционното  тестване търси правдоподобни грешки в извикването на операции или съобщителни връзки. В този контекст могат да се открият три вида грешки:

* Неочаквани резултати
* Грешни операции/съобщения
* Некоректни извиквания.

За определяне на правдоподобни грешки при извикване на функции (операции), трябва да се изследва поведението на операциите.   
Интеграционното тестване може да се прилага както към атрибутите, така също и към операциите. “Поведенията” на един обект са дефинирани посредством стойностите, присвоени на неговите атрибути. Тестването ще изследва атрибутите за определяне дали се появяват характерни стойности за различни типове на поведението на обектите.

Съществено е да се подчертае, че интеграционното тестване се опитва да намери грешки в клиентските обекти, а не в сървъра. Фокусът на интеграционните тестове е определяне дали съществуват грешки в извикващия код, а не в извиквания. Извикването на операции се използва като следа (път) за намиране на тестови изисквания, които изследват извикващия код.

**Влиянието на ОО програмиране върху тестването**

Съществуват няколко начина, по които ОО програмиране може да оказва въздействие върху тестването. В зависимост от подхода за ООП:

* Някои видове грешки стават по-неправдоподобни
* Някои видове грешки стават по-правдоподобни
* Появяват се нови видове грешки.

Когато се извика една операция може да се окаже много трудно да се предвиди кодът, който трябва да се изследва. Т.е. операцията може да принадлежи към един от многото класове. Също така е трудно да се определи точният тип/клас на един параметър. Когато кодът го прави достъпен, той може да получи една неочаквана стойност.

Наследяването не премахва необходимостта от пълно тестване на всички наследени класове. В действителност то може да усложни процеса на тестване.

**Проектиране на тестове базирани на сценарии (Scenario-Based Test Design)**

Fault-based тестването пропуска два основни вида грешки:

* Некоректни спецификации
* Взаимодействия между подсистеми.

Когато се появят грешки, свързани с некоректна спецификация, системата не прави това, което иска потребителят. Тя може да прави грешни неща или да пропуска съществена функционалност. И в двата случая страда качеството.

Грешки, свързани с взаимодействията между подсистемите, се появят когато поведението на една подсистема създава обстоятелства (събития, потоци данни), които причиняват грешки в други подсистеми.

Scenario-based тестването се концентрира върху това, което прави потребителят, а не върху това, което прави системата. Това означава улавяне на действията (чрез use-cases), които трябва да извърши потребителя, след което прилагане на тях самите и техни варианти като тестове.

Сценариите откриват грешка на взаимодействието. За да се осъществи това тестовете, тестовете трябва да бъдат по-комплексни и по-реалистични в сравнение с fault-based тестовете. Scenario-based тестването е насочено към изследване на повече подсистеми в един единичен тест (потребителите не трябва да се ограничават към използване на една подсистема в един определен момент).

**Тестване на повърхностните и вътрешни структури (Testing Surface Structure and Deep Structure)**

Повърхностна структура (surface structure) се отнася към явно обозримата структура на една ОО програма. Т.е. структурата, която непосредствено е очевидна за един краен потребител. Независимо какъв е интерфейсът на системата, тестовете се базират на действията на потребителите. Обхващането на тези действия означава разбиране, изчакване и провеждане на разговори с представители на потребителите.

За една конвенционална система с командно-ориентиран интерфейс, потребителите могат да използват списъка на всички команди като тестови списък (checklist). Ако за отделните команди не съществуват сценарии, тогава  тестването навярно ще пропусне някое потребителско действие. При обектно-ориентираните интерфейси тестващият може да използва списъка на всички обекти като списък (checklist) за тестването.

Независимо от вида на интерфейса, при разработването на тестовите случаи, които изследват повърхностната структура обектите и операциите се използват като следи (нишки), водещи към пропуснати действия.

Deep structure се отнася към вътрешните технически детайли на една обектно-ориентирана програма. Т.е. структурата, която се разбира посредством изследване на проекта и/или кода. Deep structure тестването се извършва, за да се изпълнят механизмите на:

* зависимостите
* поведенията
* комуникациите

които са установени като част на системния и обектния развой на ОО софтуер.

Като основа на този вид тестване се използват аналитичните и проектните модели. Така напр. object-relationship diagram или subsystem collaboration diagram изобразяват сътрудничеството между обектите и подсистемите, което може да не бъде явно видимо. Тогава пред разработчиците на тестови случаи стоят следните въпроси: обхваната  ли е (като един тест) някоя дейност, която упражнява сътрудничеството, дадено в object-relationship diagram или в subsystem collaboration diagram? Ако не, защо?

Проектните представяния на йерархията на класовете доставя вникване в структурата на наследяване. Тази структура се използва при fault-based тестването.

**Методи за тестване, приложими за класове**

Както вече беше споменато тестването започва “в малкото” и бавно напредва към тестване “в голямото”. Тестването “в малкотоl” се занимава с еди отделен клас и методите, капсулирани чрез него. Random testing и partitioning са методи, които могат да се използват за изпълнение на един клас по време на ОО тестване.

**Random testing for OO Classes**

За да се разбере този метод нека разгледаме следния пример: Нека имаме клас X, който има следните операции: О1, О2, О3, О4, О5,О6. Всяка от тези операции може да бъде приложена за класа X, но с някакви ограничения зависещи от същността на проблема (например О2 не може да бъде изпълнена преди О1). Въпреки тези ограничения съществуват множество пермутации за последователността на изпълнение на тези операции. Минималното поведение на инстанция на класа X включва например следните операции:

О1.О2.О4.О6, което описва минималната комбинация за тестване на последователността. Но както е ясно съществуват още множество вариации –  например O1.O2.O3.O5    
Вариациите от различни последователности на операции трябва да се генерират случайно (randomly), откъдето и името на тези тестове.

**Partition Testing на ниво клас**

Този вид тестване редуцира броя на тестовите случаи, необходими за изпълнение на класа. Входовете и изходите се категоризират и тестовите случаи се разработват за изпълнение на всяка категория. Как обаче се получават partitioning категориите? В зависимост от отговора на този въпрос различаваме следните видове partitioning:

* State-based partitioning – категоризира операциите на класовете на основата на способността им да променят състоянието на класа.
* Attribute-based partitioning – категоризира операциите на класа на основата на атрибутите, които използва.
* Category-based partitioning – категоризира операциите на класовете на основата на използваните генетични функции.

**Проектиране на тестови случаи за вътрешни класове (Inter-Class Test Case Design)**

Разработването на тестови случаи става все по-сложно, когато започне интеграцията на една ОО система. На този етап трябва да започне тестването на сътрудничеството между класовете.

Подобно на тестването на индивидуалните класове, тестването на сътрудничеството между класовете може да се осъществи посредством прилагане на:

* Random методи
* Partitioning методи
* Scenario-based тестване
* Behavioural тестване.

**Multiple Class Testing**

За генериране на случайни (random) тестови случаи за multiple class може да се използва следната последователност от стъпки:

* За всеки клиентски клас – използва се списъкът на операторите на класа за генериране на серии от random test последователности. Операторите ще изпращат съобщения към други сървърни класове.
* За всяко генерирано съобщение – определя се сътрудничещият клас (collaborator class) и кореспондиращият оператор в сървърния обект.
* За всеки оператор в сървърния обект (който се извиква чрез съобщенията, изпратени от клиентския обект) – определят се съобщенията, които той предава.
* За всяко едно съобщение – определя се следващото ниво на операторите, които  извикват и включват тези съобщения в тестовата последователност.

Подходът на multiple class partition тестването е подобен на подхода, използван за partition testing за индивидуалните класове. Тестовата последователност се разширява така, че да включва тези оператори, които се извикват посредством съобщенията на сътрудничещите класове.

**Заключение**

Общата цел на ОО тестване е да открие максимален брой грешки с минимум усилия – това е идентично с целта на конвенционалното тестване. Но стратегията за ОО тестване по смисъл се различава. Тя се разширява с тестване на аналитичните и проектни модели и фокуса се премества от процедурните компоненти при конвенционалното тестване, към класовете.