**Въведение**

В средата на 60-те години се отбелязват две неща, които на пръв поглед не са толкова съществени, но които в последствие стават основни движещи тенденции в областта на софтуерното инженерство:

* За първи път започва да се говори за софтуерната криза
* Цената на софтуера става по-голяма от цената на хардуера

Като едно следствие от това на две последователни конференции на НАТО през 1968 год. и през 1969 год. се дефинира понятието софтуерно инженерство (software engineering), което спомогна за активизиране на научно-изследователските дейности и привличане на сериозни инвестиции. Така напр. вложените в тази област средства от около 140 милиарда долара през 1985 год. нарастнаха на 435 милиарда през 1995 (за сравнение - бюджета на Германия за 1992 е бил около 430 милиарда DM). Въпреки това и в момента продължава да се говори за софтуерна криза.

За разлика от някои информатични дисциплини (като напр. техники на компилиране, бази данни), където научно-изследователската област е сравнително ясно определена и в значителна степен формализирана, областта на софтуерното инженерство може да бъде доста неопределена и обширна.

От специализираната литература, учебниците, научно-изследователските проекти и конференции могат да се обобщят следните области (някои от тях се застъпват частично), които могат да бъдат причислени към софтуерното инженерство:

* Модели на жизнения цикъл (life cycle models)
* Софтуерни процеси (CMM, ISO 9000)
* Управление на конфигурацията (configuration management)
* Оценка на разходите (effort estimation)
* Управление на проекти (project management)
* Управление на качеството (quality management)
* Реинженеринг (reverse engineering)
* Софтуерни метрики (software metrics)
* Софтуерни архитектури (software architecture)
* Поддръжка на софтуера (software maintenance)
* Софтуерни спецификации (software specification)
* Многократно използване (reuse)
* Верификации на програми (program verification)
* Документиране на софтуера (software documentation)
* Базирано на знания софтуерно инженерство (knowledge-based SE)
* Системно тестване (systematic testing)
* Прототипиране (prototyping)
* Средства, подпомагащи софтуерното инженерство (CASE tools)
* Програмни езици за софтуерното инженерство (programming languages for SE)
* Обекно-ориентирана парадигма (object-orientation - analysis, design, programming, test)
* Екстремно програмиране (extreme programming) и др

Най-често използваните дефиниции на софтуерното инженерство са:

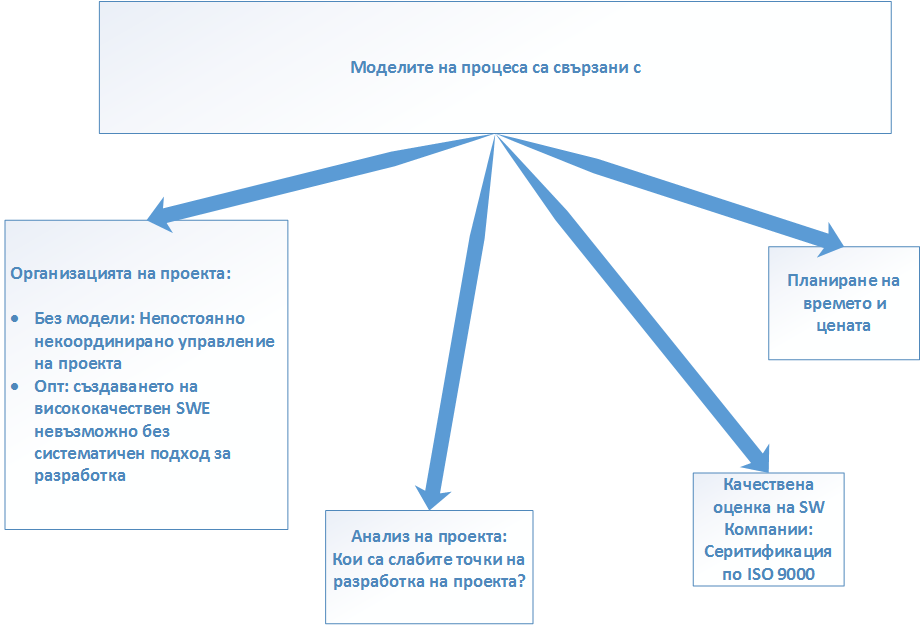
* Софтуерното инженерство цели икономически обоснована разработка на високо качествен софтуер
* Софтуерното инженерство е инженерна дисциплина занимаваща се с практически проблеми при разработката на големи софтуерни системи
* Софтуерното инженерство е прилагане на систематичен, дисциплиниран подход за разработване, експлоатация и поддръжка на софтуер, което е приложение на инженерството за софтуер
* Софтуерното инженерство е един постъпков процес, който улеснява спецификацията, проектирането, кодирането и тестването на софтуер с възможно най-бързия и евтин подход.

Софтуерното инженерство не е една специализирана област от компютърните науки. По-скоро то възниква и се развива като интегрална област между различни специализирани области.

**Модели на СП**

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.



Фиг. 1: Дейности по моделиране на СП

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.

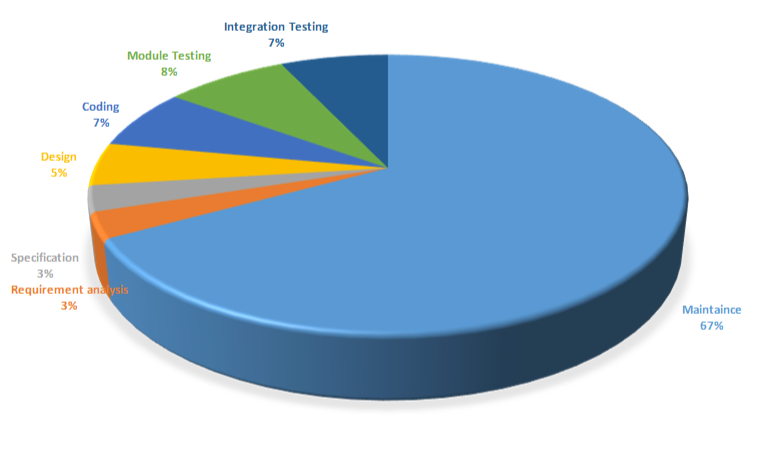
Основният поставян тогава въпрос е какво точно става в тази “черна кутия”. Днес вече съществуват много и различни модели за разработка на софтуер. Въпреки това все още съществуват фирми за развой на софтуер с хаотичен и неопределен процес на разработка.

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**

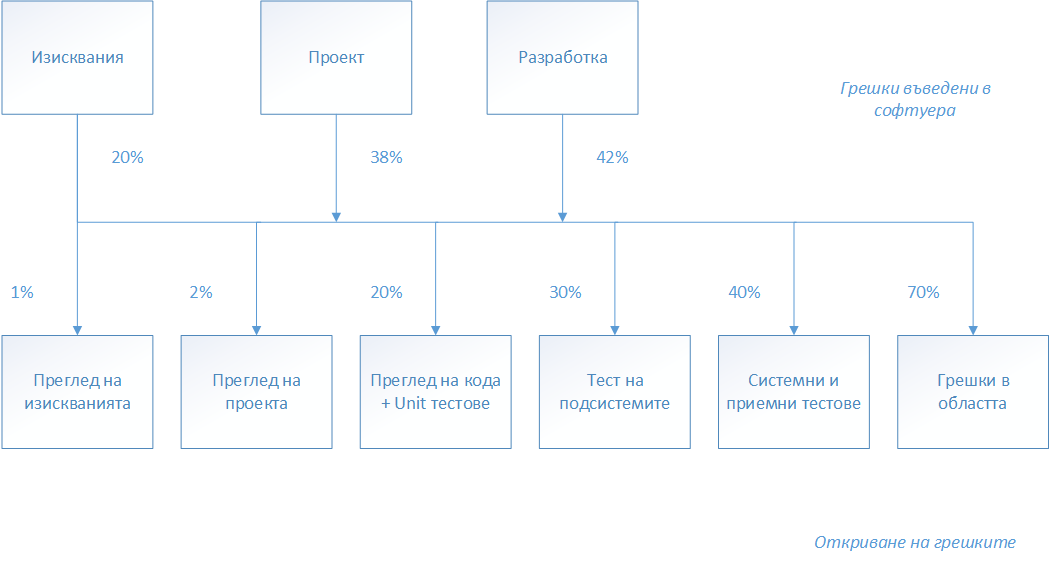
На Фиг. 2 е дадено едно типично разпределение на цената на софтуера според фазите на класическата разработка на софтуер. Повече от половината цена отива в поддръжка на софтуера – 67%. Фазата на анализ и дефиниция е разделена в две отделни дейности: анализ, за която дейност отиват около 3% от средствата 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.



Фиг. 3: Допускане на грешки при разработката

**Водопаден модел**

Разгледаните фази на жизнения цикъл на софтуерния развой могат да бъдат свързвани по различен начин в съответствие с избрания модел за развой. Съществуват две големи групи модели:

* *Последователни модели* (sequential life-cycle models)  - при тези модели се спазва една сравнително строга последователност на дейностите (фазите) при разработване на софтуер
* *Инеративни модели* (nonsequential life-cycle models)  -  тези модели (наричат се още *циклични модели*) допускат връщане (feedbacks) към предишни дейности (фази).

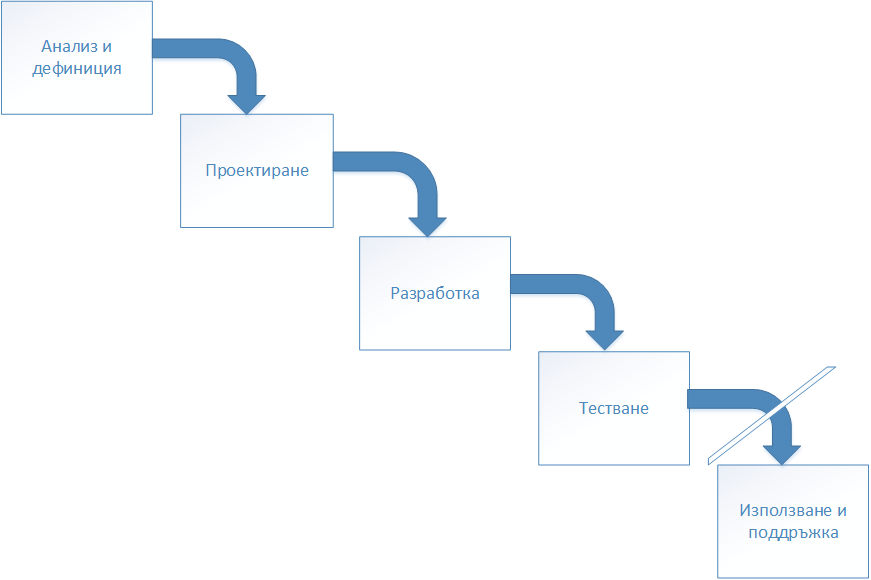
Ще разгледаме три типични представители на последователните модели:

* Водопаден модел (waterfall model)
* V-модел (V model)
* Cleanroom-Engineering.

**Водопаден Модел**

В началото на седемдесетте години възниква първият модел за систематичен развой на софтуер, който е наречен класически водопаден модел. Водопадният модел е типичен пример за последователен модел, където различните дейности на развойния процес са обособени като отделни фази, които се изпълняват в една строга последователност (Фиг.4.).

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

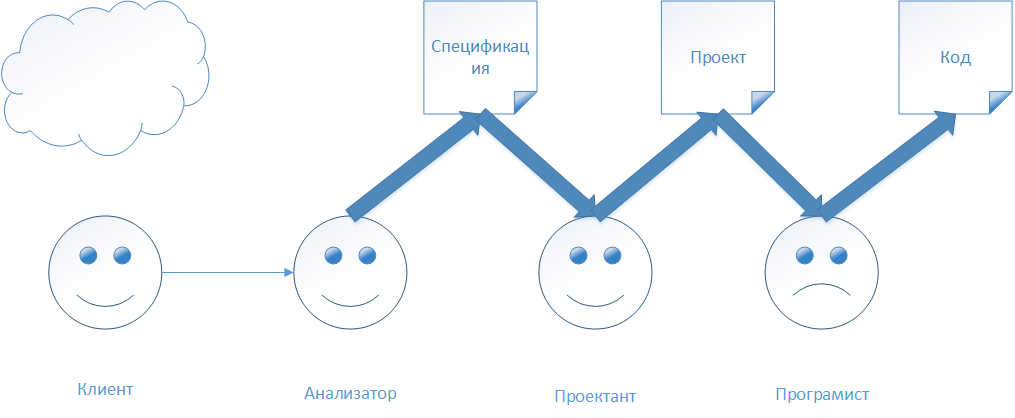


Фигура 4: Водопаден модел

Водопадният модел е основата за всички останали модели на софтуерния процес. Въпреки това той има някои основни недостатъци. Схемата на модела е доста стриктна, поради което е затруднено използването й в реалните разработки на софтуер. Тази строга последователност на фазите може да служи като удобен еталон за принципно изучаване и разбиране на процеса на разработка на софтуер. Схемата на водопадния модел е стриктна по следните причини:

* Няма обратна връзка между отделните фази – една фаза следва веднага след приключилата предишна. В реалността е необходимо да съществува възможност за връщане назад. Например по време на разработката (кодирането), програмистът може да открие грешка или някаква неточност в проекта, получен по време на фазата на проектиране. Той трябва да има възможност да се върне във фазата на проектиране и да промени проекта
* Няма паралелност между фазите – фазите в практиката често протичат паралелно. Така напр., разработката на практически приложението и тестването непрекъснато си взаимодействат (тестването изисква промяна на кода).

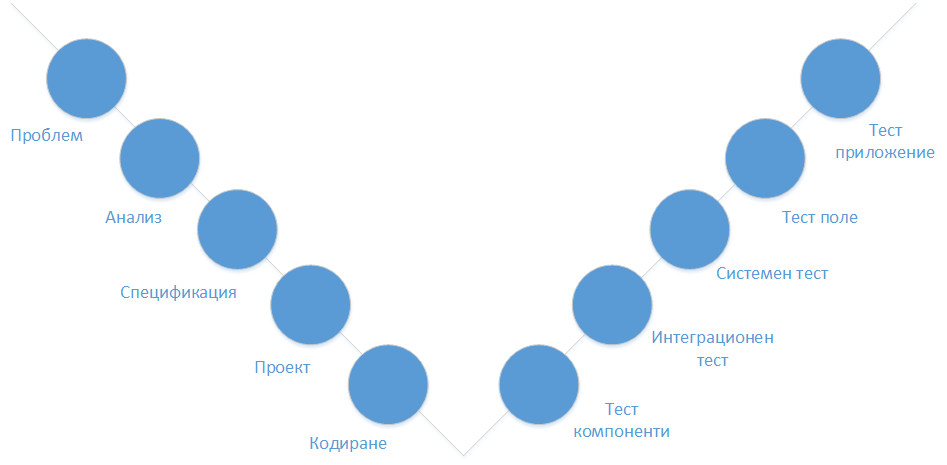
На Фиг. 5. е дадена една неформална илюстрация на процеса на разработка на софтуер. От нея може да се види, че сътрудничеството между актьорите, участващи в процеса на разработка, е една основна задача. Клиентът представя своите неформални изисквания на анализатора, който трябва да създаде от тях съответна спецификация. Анализаторът и проектантът си сътрудничат по време на създаването на спецификацията на изискванията. На базата на тези изисквания проектантът трябва да създаде проект на приложението. Програмистът чете проекта и реализира (програмира) приложението. По време на разработката на софтуера могат да възникнат комуникационни проблеми.



Фигура5: Документи на SW разработка

### V-модел

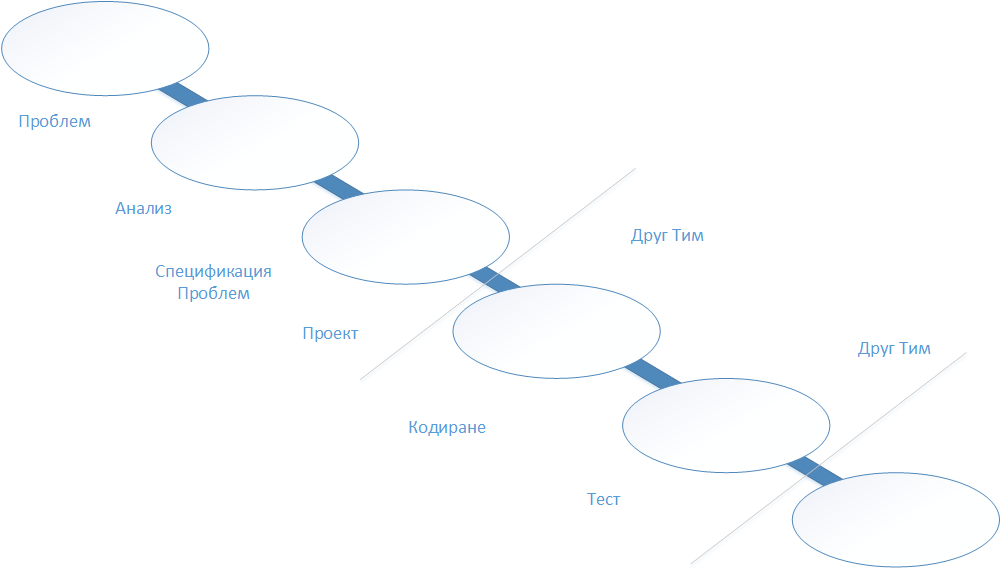
При този модел тестването е разделено на повече отделни фази (Фиг. 6). Това произлиза от разбирането, че разработването на софтуер първоначално протича като декомпозиция на цялата система, а по време на тестването се извършва по обратен ред интегриране на вече тестваните по-малки структурни единици.



Фиг.6: V-модел

### Cleanroom Engineering

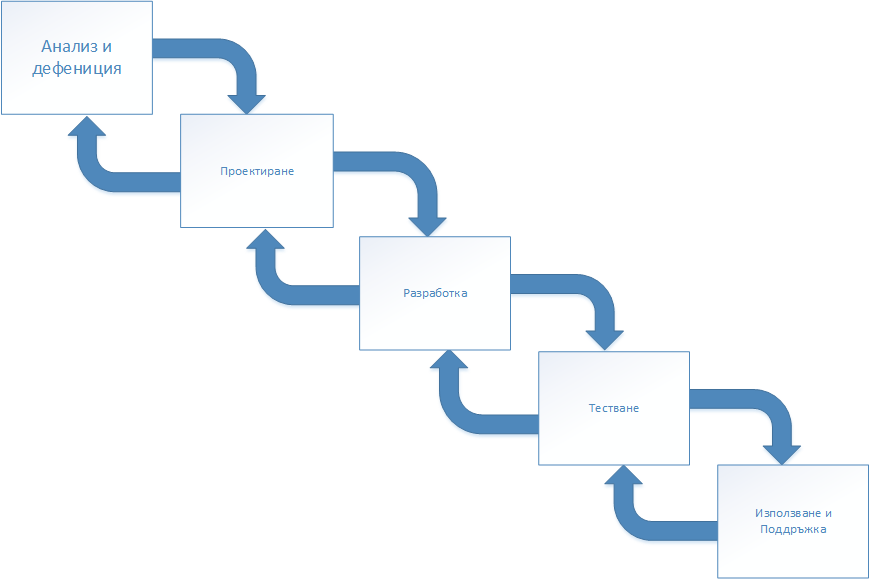
При Cleanroom-Engineering се подчертава, че за разработването на надежден софтуер от съществено значение е последователното предаване на междинните резултати на друг екип, който да продължи работата по проекта (Фиг.7).



Фиг.7: Cleanroom Engineering

**Итеративен модел**

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.



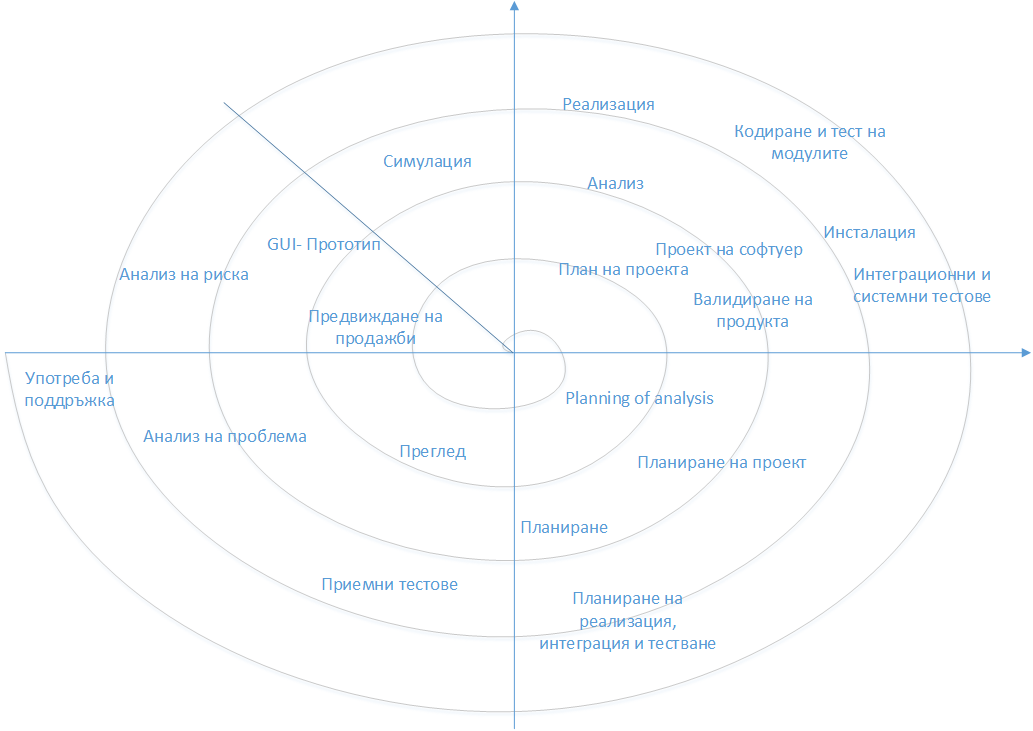
Фиг.8: Итеративен модел

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.

**Спирален Модел**

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. 9: Спирален модел

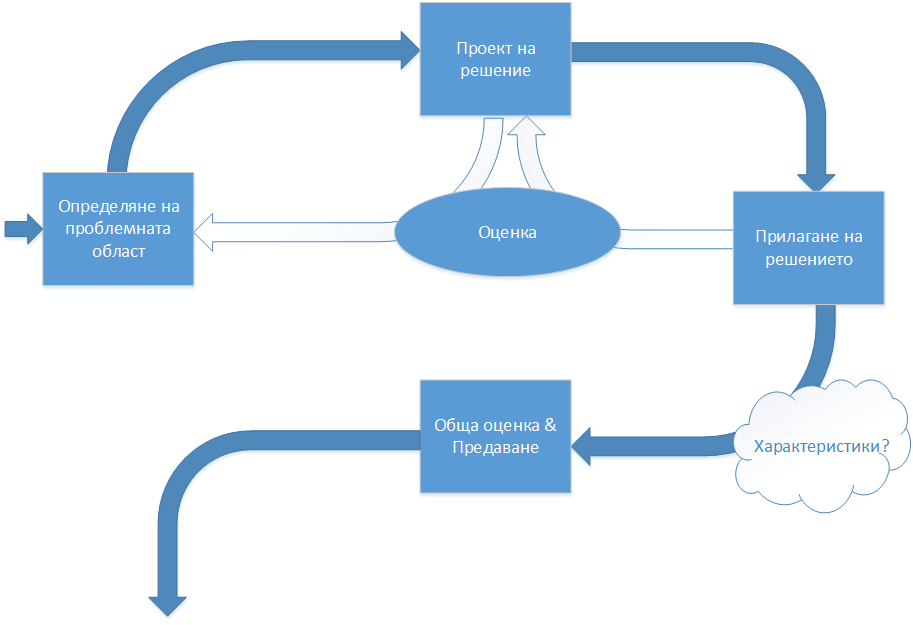
Спиралният модел въвежда пет нови фази, които се различават от тези на водопадния модел (Фиг. 9). Той може да бъде характеризиран както следва:

* Абстрактен модел – фазите са абстрактни. Може да се интерпретира като един мета модел, който е абстракция от други модели за развой на софтуер. Така напр. една итерация (цикъл) може да бъде една от обичайните за водопадния модел фаза или разработка на следващ прототип на системата
* Ориентиран към подобни дейности на всяка стъпка - дейностите във фазите не са насочени директно към разработка на софтуер. Така напр. анализът на риска може да се приложи и за класически инженерни области (напр. производство на кораби или куфари). При производството на софтуер анализът на риска може да се използва във всяка една от обичайните фази. Анализът на риска може да включва сравнение на възможни алтернативи, като напр. избор на програмен език, метод за анализ и разработване, развоен екип, използване на помощни средства
* Гъвкав - обратната стрелка в модела индикира някакъв вид цикъл в процеса. Този цикъл може да бъде представен във формата на спирала. Така напр. първото завъртане произвежда прототип, второто завъртане си кореспондира с фазата анализ, следващото с фазата проектиране и т.н.

**Прототипиране**

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 – разработеният 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.

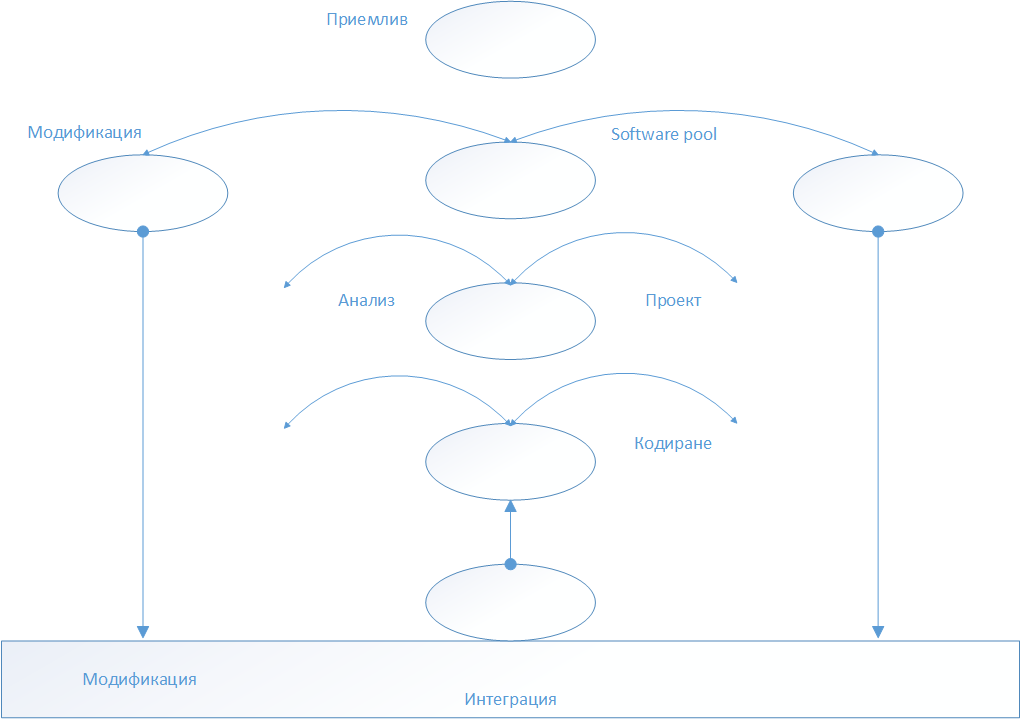
Във фазата на проектен прототип се усъвършенства функционалния прототип.

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.

### Фонтанен модел

Тази модел (Фиг.11) обобщава опита за събиране на всички възможни резултати (включително и междинните) на софтуерния продукт в един общ пул (software pool). За поддържане на софтуера в актуално състояние (фонтана в действие) са необходими непрекъснати усилия за реализация на промените.



Фиг. 11: Фонтанен модел

### 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**

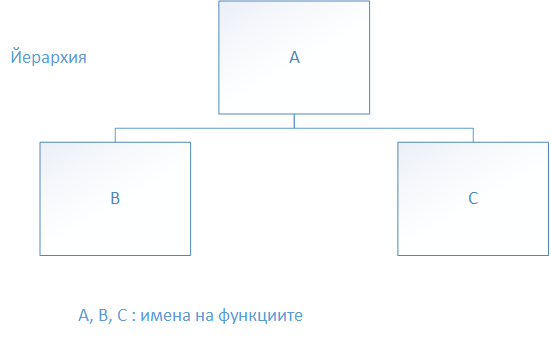
Функциите обикновено описват определени дейности или ясно очертани задачи в рамките на една проблемна област. От гледна точка на софтуерните технологии една функция:

* 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 “*A се състои отB  иC*”, т.е. така наречена *зададена йерархия*
* Във фаза на проектиране интерпретацията може да бъде “*A извикваB иC*”, 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 функции

**DFD**

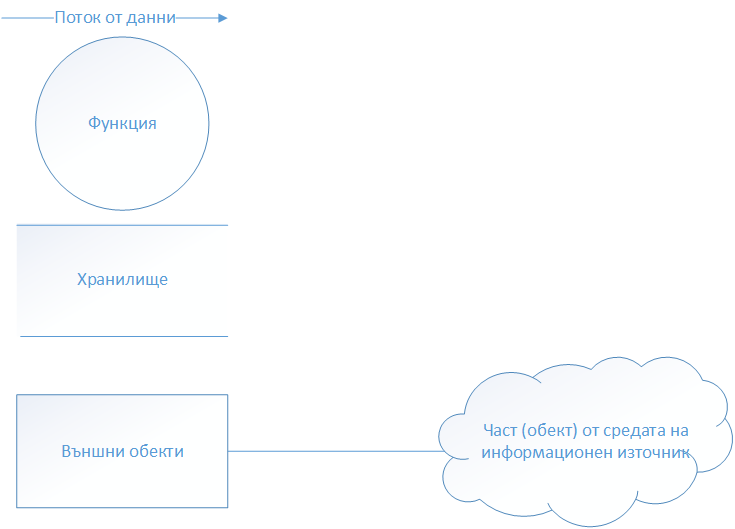
Основната идея на този вид представяне (Data Flow Diagram – DFD) е, че разработваната система е един информационен поток от данни между:

* 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.

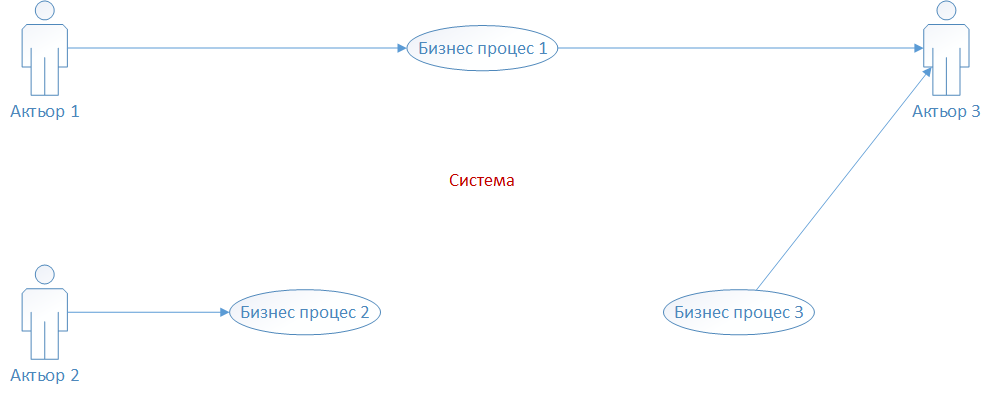
Първо се идентифицират актьорите на системата – това са субектите на предлаганата функционалност, които могат да бъдат определена група потребители или различни видове устройства, външни за разработваното приложение. Актьорите всъщност представят определени роли, които потребителите (или устройства) 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.

Фигура 3: Система представена чрез сценарии и актьори

**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.

**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.

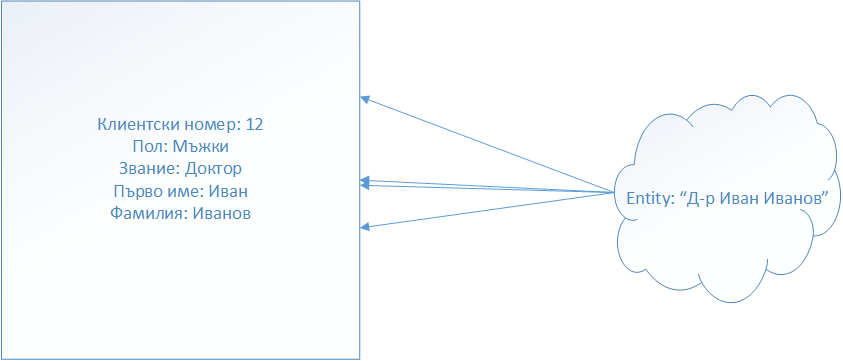
**Entity-Relationship (ER) модел**

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”).



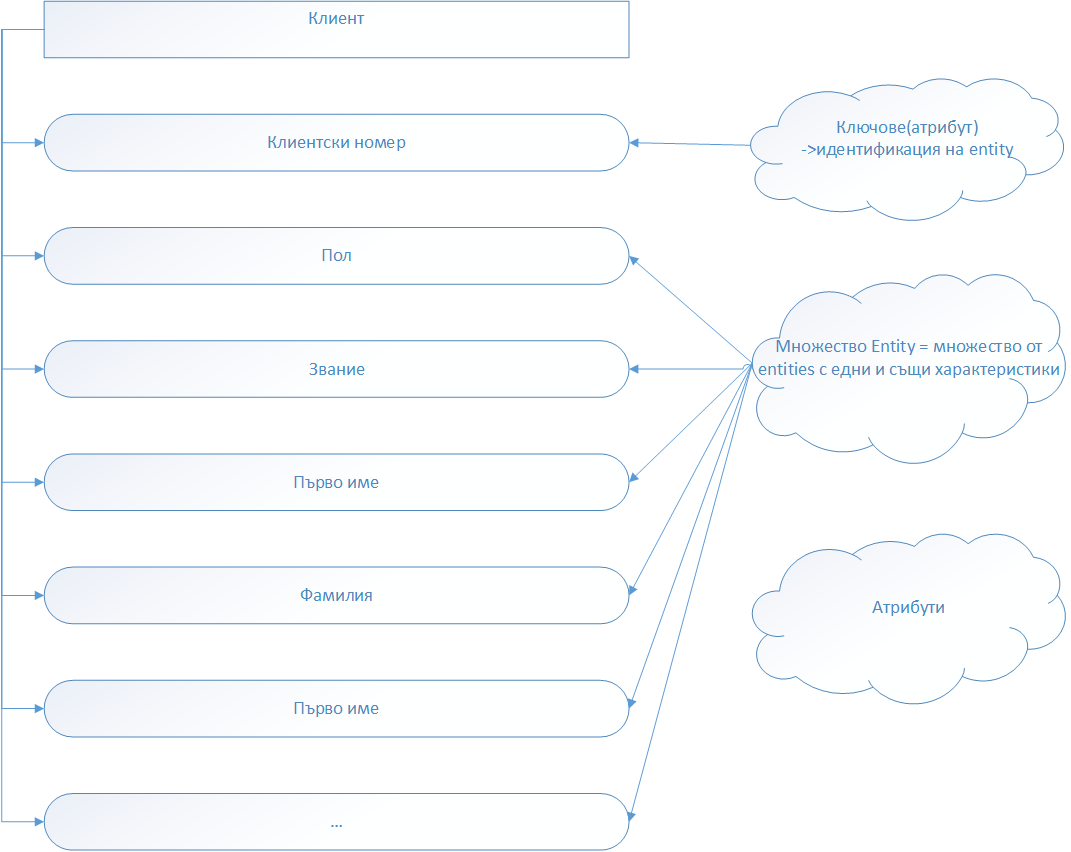
Фигура 1: Обект от реалния свят

**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”.

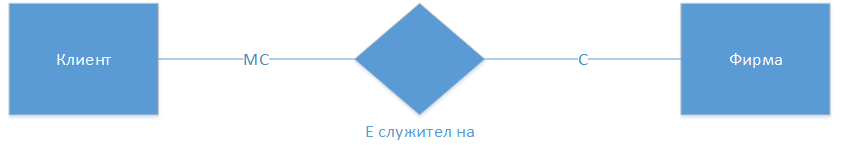
Възможни са повече от един възможни идентифициращи атрибути. Един *ключ* K е винаги една *минимална* комбинация от атрибути, с която може да бъде еднозначно идентифицирана една идентичност. Всяко супермножество на тази комбинация се нарича *ключ-кандидат*. В моделите атрибутите, представляващи ключове,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

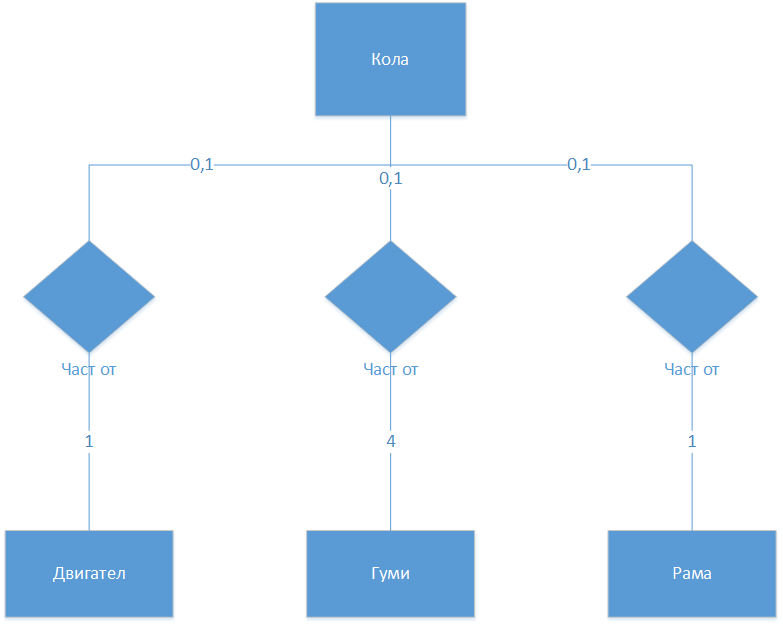
За представяне на кардиналността се използват спецификатори, които могат да бъдат:

* 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:

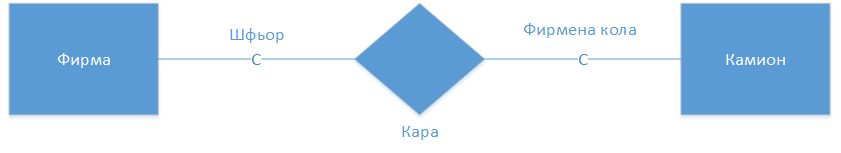
* един двигател може да не бъде част или може да бъде част от една кола
* една кола може да съдържа точно един двигател
* четири гуми могат да не са част от една кола или могат да бъдат част от една кола
* една кола може да има точно четири гуми
* една рама може да не е част или да е част от една кола.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