Софтуерни технологии

Модели на СП

1. **Софтуерно инженерство** – дисциплина, която отговаря за създаването на качествен софтуер, като интегрира процеси, методи и средства за разработването му.

//приложение на систематичен, дисциплиниран, пределен подход за разработка, изпълнение и поддръжка на качествен софтуер.//  
**Софтуерно инженерство** (IEEE) – дисциплина, която предоставя методи и средства за разработване на качествен софтуер с ограничен бюджет, поствани срокове и с изменящи се изисквания.  
**Качество** – съвкупност от средства и характеристики на продукт или услуга, носители на способността му да отговори на явно или неявно указани нужди.  
Съществуват **модели**, поддържащи систематичния процес на разработка. Те са една от основните задачи на софтуерното инженерство.

1. **Жизнен цикъл** – процес за разработка на софтуер, който идентифицира всички фази и стадии на продуктите – развой, въвеждане в експлоатация, поддръжка, снемане от експлоатация и премахване.

* Статичен – ограничени във времето фази
* Динамичен – свързани помежду си във времето фази

1. **Моделите** – стратегия, която съдържа методи и основните фази за разработка. Моделите позволяват разработването на прототип в най-ранна фаза, правят процеса методологичен и помагат за по-добра работа на екипа.

**//**са дефиниция за разработка на софтуер: дейности, които трябва да се изпълнят; хората, които трябва да ги изпълнят; ред на изпълнение; продукти, които ще бъдат разработени и начин на оценка.//

* 1. **Последователни модели** – спазва се последователност на фазите при разработка на софтуер  
     **Недостатъци:** Трудно приложими за реален проект; няма възможност за връщане към предишна фаза; няма паралелност между фазите
* **Водопаден модел** – първият систематичен модел за разработка на софтуер. Използва се за основа на всички други.  
  **Предимства:** Удобен шаблон за разбиране на процеса  
  **Недостатъци:** Характерните недостатъци на последователните модели  
  ***Водопадният модел включва следните фази (дейности):***

1. Анализ и дефиниция – анализира се проблема и се дефинират изискванията
2. Проектиране – създава се софтуерна архитектура
3. Разработка – реализират се компонентите от архитектурата
4. Тестване – тестват се всички компоненти (поотделно и интегрирани в едно цяло)

* **V-модел (модел за тестване)** – тестването е разделено на повече отделни подфази. Тества се от възможно най-малка единица до тестване на цялото приложение.
* **Cleanroom-Engineering** – важно е предаването на междинни резултати на друг екип, който продължава работата
  1. **Непоследователни модели** – допускат връщане в предишна фаза, което обаче е скъпа дейност.
* **Прототипиране** – съществува заради факта, че в началото изискванията са непълни или неясни. Разработвайки част от приложението и използвайки го като прототип (демо), екипът изчиства грешки възможно най-рано в разработката.  
  **Прототипирането е два вида**: **вертикално** (избират се само някои функционалности, но се реализират напълно) и **хоризонтално** (цялата функционалност се реализира само от части)  
  **Подходите за прототипиране са два: бързо** (прототипите се изоставят след като изпълнят задачите си и приложението се разработва наново) и **еволюционно** (прототипите се използват като част от разработката на приложението)
* **Спирален модел** – дейностите не са насочени директно към разработка, а всяко „завъртане“ използва резултата от предишното
* **Фонтанен модел** – работи се с актуалното състояние на проекта като за целта всички резултати се събират на общо място
* **Whirpool**
* **Итеративен(инкрементален развой)**

Основни концепции на изгледи на системата

Функционален изглед

Целта е изискванията за софтуерния продукт да бъдат по-прецизни и ясни.

1. **Функционални дървета** – Функциите имат две значения: **задача на софтуера** (във фазата на анализ) и **метод** (в проектирането и разработката)  
   Функционалните дървета са **йерархии от функции**Разликата им спрямо спецификацията на изискванията е, че вторите имат линеен списък от функции
2. **Data Flow Diagrams (Диаграми на потока от данни)** – Най-използваното представяне на поток от данни е на Де Марко. Състои се от поток данни, функции, хранилища и външни обекти.  
   Основна идея: системата е работеща, т.е. компонентите (изброени по-горе) „комуникират“ един с друг чрез потоци данни.  
   Правила:

* Всяка DFD съдържа поне един външен обект, който може да се съдържа само веднъж
* Всеки поток от данни има име, освен тези идващи от хранилището
* Външните елементи и хранилищата не комуникират пряко едно с друго

**Предимства:** Лесно четими и създавани, лесни за разбиране от непрофесионалисти, съдържат повече информация от функционалните дървета  
**Недостатъци:** Възможно е да станат огромни, поддръжката им е трудна, означаването с имена не винаги е достатъчно

1. **Use-case диаграми** – диаграми на бизнес процеса – описват го като сценарии – функционални изисквания, връзките им едно с друго и с актьорите.  
   Use-case диаграмите описват функционални изисквания.  
   За описване на use-case се използват различни **техники**: **текстово**, **collaboration диаграми** (представя съобщения, които си разменят обектите), **sequence диаграми** (показват последователността на съобщенията), **activity диаграми** (представят алгоритъма на действие), **крайни автомати** (представят поведението на обект и състоянията му), **мрежа на Петри** (паралелни процеси).  
   Връзки между Use Cases: **директна асоциация**, **extend** (разширение на функционалност), **include** (включващо разширение), **generalize** (наследяване).  
   **Предимства:** Фокусира се върху функционалностите на системата; Фокус върху основния проблем, лесни за разбиране от клиенти;  
   **Недостатъци:** Лесно се претоварват с излишни детайли

Ориентиран към данни изглед

1. **Речници от данни DD** – каталози, съдържащи информация за структурата, свойствата и използването на данни. **Цел** – синтактична структура на оперативните данни от потребителска гледна точка. За описание се използва BNF.  
   **Предимства:** Могат да се представят формално и сбито.  
   **Недостатъци:** Не се представят графично и така ограничават разбирането им от страна на потребителя.
2. **Entity-Relationship модел** – използват се за моделиране на бази данни. Основната **цел** е описание на постоянните записи данни и връзките им.   
   Основни елементи:

* **Идентичности (Entities)** – една идентичност е един обект от реалния свят.
* **Множество от идентичности** – имат еднакви характеристики. Имат два вида атрибути: **идентифициращи** (уникални) и **описателни** (съществени свойства)
* Ключ-минимална комбинация от атрибути, идентифицират уникално всяка идентичност.
* **Асоциации** – семантични връзки между множества от идентичности
* **Кардиналност** – сложност на връзка между множество идентичности т.е. колко entities от едно множество са свързани към дадено множество. Колко идентичности от един вид се свързват с една от друг вид.
* **Агрегация** – специален вид асоциация. Интерпретира се като „е част от“
* **Роля** – функция на идентичност във връзка.

Entity-Relationship диаграмите са по-силни от речниците от данните заради връзките, които показват между идентичностите.

Ориентиран към правила изглед

Целта на ориентирания към правила изглед на системата е описание на **зависими от условия** действия, потоци на действия.

1. **Правила** – представят предикатна логика – **if** условие **else if** условие2 .. **then** действие. Имат **ясен синтаксис** **и семантика** като така спецификацията на изискванията става по-прецизна. Трябва да определи елементарните условия и действия от словесните изрази (т.е. от спецификацията на изискванията).  
   **Проблеми:** Трудно се постига яснота; Не могат да се обхванат всички условия
2. **Таблици на решенията** – Действия, зависещи от някакви условия, могат да бъдат дефинирани кратко и ясно. Състоят се от 4 квадранта: условия, действия, свързани условия и действия (х2).  
   **Проблем:** При нарастване на условията, нарастват комбинациите експоненциално.

**\*Ако има над 5 условия таблицата става не четима**. Решение е да се обединят правила с идентични условия. Друг начин чрез декомпозиране на данните в отделни таблици на решенията, които са свързани със съответните връзки.

1. **Дървета на решенията** **–** Алтернативно представяне чрез хоризонтална подредба на правила. Обхождат се отляво надясно. Всички възможности се представят явно.

Ориентиран към състояния

1. **Крайни автомати** – **Цел** – Моделиране на вътрешното състояние на системата. Могат да се представят като граф.  
   **Употреба**: динамичен изглед на обектите, спецификация на операциите на класа, спецификация на use-cases (промяна на състоянието при взаимодействие на потребител със системата).  
   Видове крайни автомати са Автоматите на **Mealy(изходните състояния се прикрепят към преходите), Moore(изходните състояния се прикрепят към вътрешните състояния) и Harel(хибриден, той се използва, когато трябва да се моделират сложни връзки)**. Първите два могат да бъдат трансформирани един в друг и са еквиваленти. Третият е хибриден – комбинация от Mealy и Moore.

Крайните автомати в обектно ориентиран свят:

* Използват се за моделиране на жизнения цикъл на обекти
* Всички обекти могат да се представят с краен автомат
* Всеки обект може да има индивидуално състояние
* Не е необходимо да се създава краен автомат за всеки един клас

1. **Activity диаграми –** вариант на крайните автомати. Описват алгоритми и/или бизнес процеси чрез състояния на действията.  
   ***Action state*** е стъпка (дейност) по време на изпълнението на алгоритъм

Ориентиран към сценарии

Описва динамична последователност от съобщения между актьори и обекти на системата. Collaboration и Sequence са два типа Interaction диаграми

1. **Collaboration диаграми** – При тях се набляга на взаимодействието между обекти в системата. Показват струпването на съобщения към даден обект. Използват се, когато се търси бързо действие.
2. **Sequence диаграми** – Акцентът е върху редът на предаване на съобщенията. Четат се от горе надолу. Няма нужда от номериране на съобщенията, заради това, че са подредени графично. Показва живота на обектите: създаването, времето през, което са активни и унищожаването. Един обект може да изпраща съобщения към себе си или друг обект.

Процес на разработка

1.Фаза на анализ и дефиниция

**Цели**: Анализ на проблема, който трябва да бъде решен; Дефиниция на изискванията към софтуерния продукт.  
Проучването може да е основа за договор, а дефиницията – за основа на проекта.

Разделя се на две подфази:

* 1. Фаза на планиране

**Резултати (документи):** речник, предварителна спецификация на изискванията, определяне на цена, план на проекта.   
Източник за тях са изискванията на клиента, „нанасяни“ върху шаблони, разработени по определен стандарт.

*Речник* – дефинира общата терминология; използва се за потребителския интерфейс и ръководството за потребителя;

*Предварителна спецификация на изискванията* – основната й задача е да представи основните функции и данни на проекта, производителността за продукта, основите на потребителския интерфейс, критериите за качество. **Предназначен е за четене от потребителя.** м/у клиент и разработчик

*Определяне на цената* – Целият процес за определяне на цената помага за избор на проект за реализация, определяне на екип и оценка на работата им. Критериите за определяне на цената според Боем са определеност, точност, обективност, детайлност, предсказуемост, икономичност, и др.

**Метод на Боем** (**COCOMO** – Constructive Cost Model)  
**Цел**: Да се определи цена и срок на изработка за всеки проект  
**Основна идея**: Да се използва броя редове първичен код  
**Същност**: Метриката при COCOMO е човекомесец

**Прилага се при условие че**: Не се броят редове – коментари, използвани стандартни програми (външни библиотеки например); Един човекомесец е 19 дни; Не се правят сериозни промени в спецификацията на изискванията  
Съществуват **три типа софтуерни продукти**: разпространен (разработва се в малка група в познати условия), полунезависим (междинно положение между другите два), вграден (работи се със свързана апаратура и външни процеси)  
**Проблем:** Няма точно дефиниция за „ред код“

**Метод на функционалните точки** – предложен от Олбрихт  
**Основна идея**: Да се определят функционалностите на проекта чрез функционални точки  
**Цели**: Да се използват външните характеристики на софтуера (функционалността му, а не кода);   
Да се прилага в ранен етап;   
Да е независим от редовете код;  
Как се определя фукц. точки: Основава се на **пет функционални типа** с по **три нива на сложност** – просто, средно, сложно:   
Външен входен тип – всеки вход от потребител;   
Външен изходен тип – всеки изход за потребителя (съобщения, отчети);   
Вътрешен логически файлов тип – логическа група от данни в приложението;   
Външен интерфейсен файлов тип – споделени файлове между повече от едно приложения;   
Външен справочен тип – всяка комбинация вход-изход без задържане на логика в програмата (пример: заявки);  
**Пресмятане**: Преброяват се елементите на всеки от петте типа, определят се нивата им на сложност, с данните се създава таблица за пресмятане, допълвана със стойности от 0 до 5 за различните характеристики

**Други модели – Doty, SPQR, ESTIMACS** (подобни на COCOMO)**, BANG** (подобен на метода на функционалните точки)

Оценяване при обектно ориентирано програмиране: модели, свързани с брой редове НЕ могат да се приложат; Метод на функционалните точки, COCOMO 2.0 и Метод на Moose са допустими

Приложният специалист е отговорен за предварителната спецификация на изискванията и речника, а ръководителят на проекта – за определяне на цената и плана на проекта. Клиента трябва да сътрудничи за да се разработи един качествен софтуер.

* 1. Фаза на дефиниция

**Резултати (документи):** спецификация на изискванията, модел на продукта, прототип на потребителски интерфейс, ръководство за потребителя.

**Цел:** Дефиниция на изискванията към софтуерния продукт.

Системният анализатор е отговорен за модела на продукта, а приложният специалист за всичко останало.

*Спецификация на изискванията* – описва две основни групи изисквания: функционални и нефункционални. **Предназначен е главно за разработчиците** и трябва да бъде написан в тяхната терминология. Използва се като изходна точка на проектирането или разработването.  
**Съдържа**: функционални изисквания, изисквания за средата на приложението, технически изисквания, изисквания за производителност, изисквания за валидност, качество и реализация  
IEEE определя следното съдържание: въведение, общо описание и горните изисквания.

*Модел на продукта* – задължително съответства на два типа анализ – структурен или обектно ориентиран(системния анализатор).

1. Видове анализ

**Структурен анализ** – приложим в структурното програмиране. Остарял е, но все още се използва в индустрията. Съсредоточен е върху **диаграмите на потока от данни**. Тъй като те могат да обхващат много страници при по-голяма система, структурният анализ предлага разделянето им на отделни нива (йерархични структури). Ако даден процес не може да бъде декомпозиран повече, той се представя чрез **мини-спецификации (псевдо код, таблици или дървета на решенията)**

**Обектно Ориентиран анализ –** метод, представящ изискванията към софтуерния продукт като класове и обекти, които да бъдат синтезирани от проблемната област. Като резултат се получава модел на продукта. Основната идея на ООА е декомпозиция на софтуера на обекти и класове.

Декомпозиране на домейна(проблемна област) на класове и обекти.  
За представянето на ООА се използват use case, class, sequence, collaboration, package, state, activity, component, object диаграми.  
Основни концепции: обект (съвкупност от състояния(данни) и поведения(операции), клас, атрибути, операции, съобщения, наследяване, полиморфизъм. Основната концепция е клас диаграмата (включва гореизброените).

**ООА представя изискванията от гледна точка на класове и обекти!**  
 **Подмодели в OOA:  
 - статичен –** описва класове, асоциации и кардиналност между тях. Използват се class, package и component диаграми **- динамичен –** представя се динамичното поведение на системата. Използват се Use case, Sequence, Collaboration, State-transition (крайни автомати), Activity диаграми

2.Фаза на проектиране

**Основна задача**: На основата на зададените изисквания към разработвания продукт да се създаде програмно-техническо решение. Изходна точка на фазата на проектиране са резултатите, получени в предишната фаза (на анализ и дефиниция), а резултатите от проектирането са основа за дейностите по разработка на системите.  
**Документи за изготвяне**: Проект на архитектурата =>софтуерна архитектура (високо ниво на абстракции); Проект на разработка =>спецификация на компонентите (зависи от езика за програмиране и използваната среда). За тях е отговорен проектанта.

1. **Софтуерна архитектура:** Състои се от компоненти и връзки. Два фактора оказват влияние – дефиниция на продукта и език и среда на разработка. Оказва се, че в някои случаи се появяват и други фактори – интернационалност, изисквания за качество и многократно използване. **Продуктът трябва да отговаря на изискванията, да е надежден, продуктивен и да е лесна промяната му**.
2. С**пецификацията на компонентите** - необходими са различни документи. Тъй като ни интересува външното поведение на системата, всяка част от нея се приема за черна кутия, която ще бъде запълнена в процеса на разработка.

Критерии за качество – Слабо свързване и силна съгласуваност

Софтуерните архитектури могат да имат неограничен брой класове, функции.

**Свързаността** се определя от връзката между компонентите. Колкото по-малко информация си обменят те, толкова по-слабо е свързването и те са по-лесни за разбиране и модифициране. Ако обменът на информация е твърде голям, компонентите трябва да бъдат декомпозирани на по-прости.

**Съгласуваността** е логическа връзка между елементи на компонентите. Целта е всички елементи в компонента да решават обща задача. Компонентите трябва да се разработват с идеята за решаване на конкретен проблем, т.е. да са съгласувани с останалата част от приложението.

Горните двата компонента са в съперничество!

**Многослойна архитектура** – прилага се когато има много компоненти, които се обособяват в логически групи. Предимства: лесна преносимост; тестване и поддръжка Недостатъци: губи се продуктивност, защото винаги се минава през всички нива на архитектурата.

**Проектирането** бива два вида:

1. **Структурно проектиране –** Има за цел да създаде архитектура, състояща се от йерархично подредени функционални модули.   
   **Недостатък –** основата на структурното проектиране – диаграмите на потока на данни – нямат достатъчно изразителна мощ за постигане на целите на структурното проектиране. Затова е необходимо да се въведе нова концепция – структурни диаграми (имат поток от данни и функционална йерархия)   
   **Структурни диаграми –** представят графично функционални модули и взаимодействието между тях (например извикване). Използват се за представяне на структурата на извикване на потока от данни.
2. **Обектно ориентирано проектиране –** разделя се на две – проект на архитектурата (вида на приложението, увеличаване на продуктивността, повторно използване) и проект на разработката (настройка към езика за програмиране).

Първо се определя архитектурата на системата: свързване към потребителския интерфейс и към базата данни като се разширява ООА с техни класове.  
**За повторното използване** се разширява ООА с повторно използваеми класове и се модифицира ООА модела като се приспособява към тези класове.  
Влияещи фактори са категория на приложението, връзка към потребителския интерфейс, продуктивност, рамки, проектни шаблони

3.Фаза на разработка

**Основни участници** са разработчиците. Използват се софтуерната архитектура и спецификацията на системните компоненти, за да се създаде сорс код, обектна програма и тестови план и протокол за тестове.

**Цел:** създаване на продукт, който е независим от хардуера.

**Документи:** source код, коментари, обектна програма, тестови план и протокол за тестови случаи.

**Принципи на разработване:**1.Постъпково усъвършенстване(#за инфо посетете: 24 слайд 12 презентация),   
2.пълна документация(кои са авторите на системата; версията и статус на системата(дали е завършена процес на тестване и разработка)  
3.Методология на програмиране: метод на постъпково усъвършенстване, използване на основните концепции, език за програмиране, оценка на грешките;  
4.Стил на програмиране – добре коментиран код; подреден код; спазване на всички конвенции на програмния език.  
5.Ръководство за форматиране

//За да бъде един стил на програмиране добър, кодът трябва да съдържа коментари, документация, добре избрани имена, добро форматиране, бързо работещ и т.н.  
Пътят към добрата програма включва(методология за програмиране): избор на програмен език, подредба на компонентите за разработка, метод за постъпково усъвършенстване, използване на основни концепции, оценка на грешките.//

**Основни концепции на ориентирания към алгоритми изглед на системата:** поддръжка на програмната разработка, псевдокод, бокс диаграми, диаграми на програмния поток.



4.Фаза на тестване

1. **Стратегия за тестване на софтуерни системи  
   Тестването** е множество от дейности, които могат да бъдат предвидени предварително и проведени систематично. Всяка стратегия трябва да съдържа планиране на тестовете, проект на тестовите случаи, изпълними тестове и резултатни данни, които се оценяват.

Тестването се провежда от разработчиците(когато екипа е малък) на софтуера или специален екип за тестване. То приключва когато завърши фазата определена в общия проект на системата.

Отговорността се носи от Project Manager-a.

**Unit тестове –** тества се най-малката единица на софтуера. Видове: независими пътища, интерфейси, групи от данни, гранични условия. (Разработчика) При тестване на зависими модули пишем overhead код(код, който симулира/преизпълнява работата на зависимия модул)

**Интеграционни тестове –** вземат се unit тествани компоненти и се изгражда структура, която е наложена от проекта на продукта. Причините за това тестване са, че между интерфейсите могат да се изгубят данни, даден модул може да повлияе нежелано на друг, комбинацията от подфункции да не работи както е очаквано и т.н.  
Съществуват два подхода за интеграционно тестване:

**Неинкрементална интеграция – big-bang.**

Инкрементална интеграция: (не са приложими при ОО)

**=>Top-down** – модулите се тестват от най-главния надолу в йерархията. Постъпково всички модули, използвани в главния се заменят със стъбове. При успешни резултати, стъбовете се заместват отново с реалния компонент.

**=>Bottom-up –** тестването започва от най-малките компоненти (тоест тези, които са най-ниско в архитектурата на програмата). Понеже те се интегрират от долу нагоре, не са необходими стъбове, а се създава драйвер, който да координира входа и изхода в по-горните нива на приложението. Компонентите на ниско ниво извършващи една и съща функция се комбинират в клъстери.

**Регресионни тестове** – повторно изпълнение на подмножество от вече изпълнени тестове. По този начин се осигурява, че направените промени не създават непредвидени странични ефекти.

**Валидационни тестове** – Установяват дали са изпълнени критерии, описани в „Спецификация на изискванията“ (създадена още в началото на целия процес).  
Критерии: Всички функционални изисквания и характеристики са изпълнени; Производствените изисквания са постигнати; Документацията е коректна; Изпълнени са допълнителни изисквания. Са:

**=>Alpha тестове –** Приложението се тества в „контролирана“ среда. Разработчикът присъства при тестването от страна на потребителя и го ръководи.

**=>Beta тестове** – Приложението се тества самостоятелно от потребителя (клиента). Този вид тестове са най-точни

**Системни тестове** – (Съпровождат интегрирането на софтуера в голяма система.) Приложението се тества в реалната среда, в което ще работи. Има 4 вида:

**=>Recovery тестове –** тестове, определящи за какво време системата ще възстанови работата си при неочакван срив.

**=>Security тестове –** тестове за сигурността на системата

**=>Stress тестове –** тестове за максимално натоварване

**=>Performance тестове** – тестове за производителността

1. Тестване на обектно ориентирани системи

Налага се промяна в стратегията на тестване.

Класическата система за тестване започва с тестване на малките компоненти и продължава към тестване на по-големи такива. Започва се с unit-testing, продължава се с integration тестове и накрая се завършва с validation и system testing.   
**Unit в ОО разработка е клас**.  
Интеграционни тестове в ООС:  
- thread-based (интегрират се множество класове, необходими за отговор на един вход или събитие към системата)  
- use-based-първо се тестват независимите класове, а след това зависимите.  
- cluster-testing (група от сътрудничещи си класове се анализира посредством тестови случаи).

Проектиране на тестови случаи за ОО: всеки случай трябва да бъде уникален и явно свързан с класа, който ще се тества; трябва да се знае целта на теста; всеки тест съдържа стъпки за изпълнение.

Валидационни тестове в ООС: за подпомагане на получаването на тези тестове, тестващите трябва да ползват use-cases, sequence, collaboration, за да разберат дали всички изисквания са изпълнени.  
Приложение на методите за разработване на тестови случаи:

=>White-box тестове – подходящи за тестване на операциите на класовете (тестваш вътрешността на кода)

=>Black-box тестове – use cases подходящи за разработване на тестови случаи (тестваш от потребителска гледна точка, за валидни данни)

=>Fault-based – тества се там, където архитектурата е най-слаба и има вероятност от възникване на грешки. (търси се къде могат да възникнат грешки като се използват аналитични и проектни модели.) Тестват се операциите и атрибутите на класовете

Тестване на повърхностни структури – интерфейсите се изследват за пропуснати сценарии.

Тестване на вътрешните структури – тестват се зависимостите между компонентите

Partition тестване на класовете – категоризират се входовете и изходите (и се правят тестове за всяка една категория)… от операциите на класовете като тестването може да бъде State based, Attribute-based, Category-based

Random тестване – произволна комбинация от различни методи на даден клас.

Общата цел на ОО тестване е да открие максимален брой грешки с минимум усилия. ОО тестването се разширява с тестване на аналитични и проектни модели и фокусът пада върху класовете, а не върху процедурни компоненти.

Модели за качество на софтуера

Надеждност на софтуера се определя от пълнота, точност и непротиворечивост.  
Разбираемостта на софтуера също се определя от непротиворечивост.

1. **Модел на Боем:** Йерархичен характер с 2 нива на полезност и възможност за лесно съпровождане. **Недостатък**: няма ясна структурираност, съсредоточава се върху качеството на кода, а не върху продукта, не е подходящ за ОО

Метрика – мярка за оценяване на конкретно свойство на софтуерен продукт, трябва да се определи обективно. Важно е да се определи важността на метриката и да се автоматизира процеса на оценяване.

1. **Типичен йерархичен модел –** Пет на брой нива (от 0 до 4)  
   **Предимства**: простота, конструктивност, автоматизация, крайния резултат е число между 0 и 1, лесен за четене и разбиране.  
   **Недостатъци**: субективност, трудоемкост.

Ниво 0 – **качеството** – разглежда се като йерархична структура

Ниво 1 – **фактори**, които представят аспект от качеството на софтуера

Ниво 2 – **критерии** – софтуерни ориентирани свойства, представящи характеристиките на продукт

Ниво 3 – **метрики** – софтуерно ориентирани детайли за определяне на оценъчни елементи

Ниво 4 – **оценъчни елементи** – елементи от най-ниско ниво, подлежащи на количествена оценка

**Фактори –** в зависимост от модела са от 6 до 16. При типичния йерархичен модел са 6 – гъвкавост, коректност, надеждност, съпровождаемост, удобство на използване, ефективност  
**Критерии –** структурираност, простота нагледност  
**Метрики –** коментари, оформяне на текста на програмата, възприета система за идентификация

Управление на качеството

1. **CMM(американски стандарт)  
   Цел:** Подобряване и оценяване на софтуерните процеси; Оценяването на способностите на потенциални изпълнители на проекта.  
   Оценката става на базата на 150 въпроса

Нива на зрелост:

Ниво 1 – начално – не осигурява стабилна среда за разработка и поддръжка

Ниво 2 – повторяемо – има политика за управление на софтуерния проект и процедури за спазването й

Ниво 3 – определено – процесите са стандартни и непротиворечиви(появява се възможност за обучение на персонала)

Ниво 4 – управляемо – организациите са предсказуеми, защото работят с измерими процеси и граници. Предлагат високо качество и възможност за коригиране

Ниво 5 – оптимизиращо – организацията е съсредоточена към непрекъснато подобряване на процесите. Процесът на разработка се нагажда спрямо използваната технология.

1. **BOOTSTRAP**Основава се на CMM и използва въпросника му. Отговаря на стандартите за качество ISO 9000, 9001, 9000-3  
   **Цели**: //Да се използват най-добрите софтуерни практики за създаване на средство за оценка на способността на организация за ефективен софтуерен процес.// Оценка на качеството на софтуерния продукт и определяне на неговите силни и слаби страни по време на разработката.  
   Подобно на CMM съществуват нива на зрелост (ниво 0 до ниво 5: 0-ниво-непълен процес; 1-ниво-изпълняван; 2-ниво-управляван; 3-ниво-установяван; 4-ниво-предсказуем;5-ниво-оптимизиращ).

Метрики

Съществуващи метрики за софтуер:

1. Редове код
2. Функционални точки

Метрики за софтуерния процес:

1. Усилия – изразходване на ресурси (човекодни или др. подобни)
2. Продължителност
3. Персонал = усилия / продължителност
4. Цена и приходи – определят се от усилията, които се влагат
5. Продуктивност – Размер на продукта / Усилия
6. Бързина – Размер / Продължителност
7. Ускорение – Предвидена продължителност / Наложена продължителност
8. Инфлация на усилията – Усилия в ускорен процес / Усилия в предвиден процес

Измерванията са основна част за софтуерното инженерство.

Метрики за качество на кода:  
-принцип на refactoring-промяна на модел на кода и запазване на неговата функционалност. Има шаблони за тая работа…Не трябва да има повтарящ се код и закоментиран(мъртъв) код.

Анализ и управление на риска

Софтуерен риск: Поредица от стъпки, които помагат на софтуерния тим да управлява несигурността при разработка.

Характеристики на риска: Несигурност (няма 100% вероятен риск); загуба (ако рискът стане реален се появяват нежелани последствия)

**Категории рискове:**   
проектни (застрашават проектния план, модела и цената);   
технически рискове (застрашават качеството и сроковете);  
бизнес рискове (застрашават приложимостта – пазарен, стратегически, управленски, бюджетен)  
Категоризация на Charette: познати (открити след преглед на плана), предсказуеми (възниквали в минали проекти), непредсказуеми (трудни за идентифициране).

Шаблонни рискове –такива, които винаги са налице при разработка (риск за цената).

Уникални рискове – уникални за проекта.

Рисковете трябва да се идентифицират (записват се в таблица, всеки риск има категория: катастрофален, критичен, несигурен, незначителен)   
Стратегия за работа с риск: рисковете трябва да се избягват, наблюдават и управляват. Project Manager-a е отговорен за анализирането на рисковете.

SCRUM

Използва се от : Microsoft, Yahoo, Google , Nokia

SCRUM е рамка за разработване на софтуер.

СКРЪМ се ползва за разработка на :

• Проекти с фиксирана цена

• Комерсиален софтуер

• Вътрешно фирмен софтуер

• Мобилни приложения и др.

Разработват се в поредица от спринтове(с продължителност от 2 до 4 седмици).

SCRUM Master- носи отговорност за спазването на правилата на СКРЪМ

Екипът разработва всичко наведнъж и постоянно. Всеки от екипа сам избира задачата, върху която да работи.

SCRUM STRUCTURE:

• Роли : собственици на продукта, екип, СКРЪМ МАСТЪР

• Срещи: планиране на спринт(Скръм мастъра следи да не продължава повече от 4 часа), ревю на спринта, ретроспективна среща(15-30 мин), дневна (15 мин);

• Артефакти: продуктов списък(product backlog) ,спринтов, диаграми