**Лекция 1 - Въведение в софтуерните технологии.**

**Софтуерен продукт** = програми + данни + документация

**Характеристика на софтуера:**

1. Той се разработва, а не се произвежда
2. Абстрактност
3. Мултидисциплинираност
4. Надеждност
5. Рискове
6. Не се “износва”
7. Компонентно базирано разработване

**Software Engineering** – дисциплина, която предоставя методи и средства за разработване на качествен софтуер с ограничен бюджет, поставени срокове и постоянно променящи се изисквания.

**Software еngineering** – подход за разработване, изпълнение и поддръжка на качествен софтуер.

**Софтуерен развоен процес** – цялостния процес по разработка на софтуера, включително използваните помощни средства, методи и необходимия персонал.

**Софтуерен процес** – характеризира се с 3 фази:

1. **Фаза на дефиниция:**
2. Оценка на информацията
3. Планиране на софтуерния продукт
4. Анализ на изискванията
5. **Фаза на разработването**
6. Софтуерно проектиране
7. Генериране на кода
8. Тестване на софтуера
9. **Фаза на поддръжка**

**Модели на софтуерния процес:**

1. **Линейни последователни** - при тези модели се спазва една сравнително строга последователност на дейностите (фазите) при разработване на софтуер.
2. **Водопаден модел** – този модел не се препоръчва, тъй като е трудно да се следва последователността на модела. Също така клиента трябва да постави своите изисквания още в началото и няма възможност да ги променя в последствие. Друг голям недостатък на модела е, че не можем да видим работеща версия (прототип) на софтуерния продукт преди неговото финализиране (за разлика от гъвкавите модели за разработка на софтуерен продукт).
3. **V-модел** - при този модел тестването е разделено на повече отделни фази. 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.
4. **Cleanroom-engineering** – тук 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.
5. **Линейни непоследователни** – тези модели допускат връщане към предишни дейности (фази).
6. Прототипиране
7. Инкрементален развой
8. Спирален модел
9. Whirlpool модел
10. Фонтанен модел
11. **Гъвкави модели (Agile Software Process)** – характеризира се с предсказуемост на процеса, както и с висока успеваемост на разработката на софтуерния продукт. Едно от най-важните предимства на модела е постоянната комуникация с клиента, която е ключов фактор за успеха на софтуерния продукт. Предоставянето на прототипи през кратък период от време (най-често седмица или месец) спомага за успеха на софтуера. Такъв тип проекти се поемат от екипи с голям опит, при които основната цел е качеството на софтуера.

**Лекция 3 - Резултати от фазата на “анализ и дефиниция”**

1. **Цели на фазата “анализ и дефиниция”:**

* Анализ на проблема, който трябва да бъде решен
* Дефиниция на изискванията към софтуерния продукт

1. **Защо разделяме фазата "Анализ и дефиниция" на две части?** -защото във фазата на анализа определяме дали софтуерния продукт е рентабилен за разработка, т.е. дали имаме ресурсите за разработката на този софтуер, тъй като цената е основен фактор за един разработката на един софтуерен продукт.
2. **Документи на фазата “Анализ и дефиниция”:**
3. Фаза на планиране:

* Разучаване
* Речник
* Предварителна спецификация на изискванията
* Определяне на цената
* План на проекта

1. Фаза на дефиниция:

* Дефиниция на продукта
* Спецификация на изискванията
* Модел на продукта(основни методи, формализирано описание)
* Потребителски интерфейс (концепция и/или прототип)
* Ръководство за потребителя (предварително)

1. **Предварителна спецификация и изисквания:**

* Основни функции
* Основни данни
* Обща производителност
* Важни аспекти на потребителския интерфейс
* Важни критерии за качество

**Лекция 4 - Цена на софтуера**

1. **Първият модел за оценка на цената на софтуер** – SDC 1965г.
2. **През 1981г. Боем създава COCOMO (Constructive Cost Model).**
3. **Критерии на моделите за установяване на цената според Боем:**

* Определеност
* Точност
* Обективност
* Детайлност
* Устойчивост
* Област на приложение
* Конструктивност
* Простота на прилагане
* Предсказуемост
* Икономичност

1. **Моделът на Боем – COCOMO:**

* Цел на модела – да се оцени цената и срокът за разработване на софтуерния продукт. Основната идея на Боем е използването на броя редове първичен код. Формулите на Боем за изчисляване се прилагат при следните предпоставки:
  + Редовете първичен код се броят без коментарните редове.
  + 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.

1. **Метод на функционалните точки:**

* Предложен е от Олбрихт – 1979г.
* Основа на модела е понятието “функционална точка”. Идеята му е, че усилията в разработката на даден софтуер (следователно и цената) се определят от неговата функционалност, която се измерва с горе споменатите “функционални точки”.

1. **Цели на метода:**

* Да се използват външните характеристики на софтуера.
* 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.

1. **Типове връзки:**

* **Extend** – не се изпълнява задължително, за разлика от **include** връзката.
* **Generalize** – възможност за изпълнение на 2 use case-a (като merge на activity diagram).

**Лекция 8 - Основни концепции на ориентирания към състояния изглед на системата**

1. **Крайни автомати** – цел и употреба:

* **Цел** - моделиране на поведението на системата, което зависи от вътрешното състояние на системата.
* **Употреба:**
* Динамичен изглед на обектите в класа: жизнен цикъл на обекта.
* Спецификацията на операциите на класа: промяна на състоянието, докато операцията се изпълни.
* Спецификация на use cases: промяна в състоянието по време на взаимодействието между потребител и система.

1. **Автомати на Mealy и Moore:**

* Винаги могат да бъдат трансформирани един в друг.
* Предпоставка за автомата на Moore:
* В съответно състояние има точно един изход и/или действие.
* Автомата на Moore може да бъде трансформиран в еквивалентен автомат на Mealy:
* Изход генериран от състояние в автомата на Moore се поставя към прехода, който води към това състояние.
* Автомата на Mealy може да бъде трансформиран в еквивалентен автомат на Moore:
* Целевите състояния с различни изходи се представят чрез няколко състояния в резултатния автомат на Moore.

1. **Разлика между автоматите на Mealy и Moore** – при изхода автомата на Moore е към състояние, а на Mealy е към прехода.
2. **Автомат на Harel: хибриден краен автомат** - комбинация от автоматите на Mealy и Moore.

**Лекция 10 – Обектно-ориентиран анализ (ООА)**

1. **Аспекти на ООА:**
2. **Парадигма на модел** - декомпозиране на SW системи в обекти (статичен/динамичен модел).
3. **Език на моделиране** – UML диаграми
4. **Методология** – как да разработим модела
5. **Средства** – Together, Rational Rose, Paradigm Plus
6. **Комбинация от основни концепции в обектно-ориентирания анализ –** ER (entity relationship – таблиците в БД), class диаграми, псевдо код, крайни автомати, collaboration, sequence и use case диаграми.
7. **Обектно-ориентирани парадигми:**
8. **Основна идея** - декомпозиция на SW система в обекти и класове.
9. **Основно понятие** – обект
10. **ООА според Booch (1994)** - Обектно-ориентирания анализ е метод на анализ, който представя изискванията от гледна точка на класове и обекти, открити в речника на проблемния домейн.
11. **Статичен и динамичен модел - използвани UML диаграми:**
12. **Статичен модел** – class, package и component диаграми.
13. **Динамичен модел** – крайни автомати, activity, sequence и collaboration диаграми.
14. **Диаграми в UML:**
15. **Use case диаграми** – взаимодействие на хората със системата
16. **Class диаграми** – класовете и техните връзки
17. **Sequence диаграми** – зависещ от времето поток на съобщения
18. **Collaboration диаграми** – също като sequence диаграмите
19. **State диаграми** – динамично поведение на обектите
20. **Activity диаграми** – паралелни процеси
21. **Component диаграми** – единици за компилация
22. **Object диаграми** – обектите и техните връзки

**Лекция 11,13 – ООА и ООД**

1. **Top-down или Bottom-up:**
2. **Bottom-up:**

* Започва се от съществуващи класове, проучва се кои класове са логически свързани.
* Проверка на характеристиките за съгласуваност и свързване на потенциална под-система.

1. **Top-down:**

* Първо се разделя цялата система на под-системи и ако е необходимо разделяме подсистемите по-нататък.
* Проверка на характеристиките за съгласуваност и свързване на потенциална под-система

1. **Рамка (Framework):**
2. **Според Balzert** - Система от повторно използваеми и приспособими класове и приспособими клас библиотеки (възможност за настройка: нови под-класове, предефиниране и/или разработване на празни операции)
3. **Според Gamma** - Множество от взаимодействащи си класове, които изграждат проект за повторно използване за специфичен тип софтуер… Рамката (framework) определя архитектурата на приложението… обхваща проектни решения, които са общи за даден приложен домейн.
4. **Основни проектни принципи ООП (SOLID):**
5. **S**ingle Responsibility Principle (SRP) - класът трябва да има една единствена отговорност.
6. **O**pen-Closed Principle (OCP) – модулът трябва да е отворен за разширения, но затворен за модификации.
7. **L**iskov Substitution Principle (LSP) - подкласовете трябва да заменят техните базови класове.
8. **I**nterface Segregation Principle (ISP) – много интерфейси, а не само един
9. **D**ependency Inversion Principle (DIP) - зависи от абстракции, а не от конкретики.

**Лекция 15 - Стратегия за тестване на софтуерни системи**

1. **Unit тестове** - съсредоточават се върху най-малката единица на софтуера – софтуерните компоненти или модули.
2. **Интеграционни тестове** - целта е да се вземат Unit тествани компоненти и да се изгради една програмна структура, която е наложена от развоя. Разделят се на 2 типа:
3. **Инкрементална интеграция:**

* **Top-down интеграция** - модулите се интегрират посредством преместване надолу през контролната йерархия, като се започне от главния контролен модул – главната програма. 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. При завършване на всяко множество от тестове, следващ стуб (stub) се замества от реален компонент.
* **Bottom-up интеграция** - започва конструирането и тестването с компоненти на най-ниско ниво в структурата на програмата. 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.

1. **Неинкрементална интеграция** – “big-bang” подход - при него всички компоненти на даден софтуер се интегрират наведнъж. Основен недостатък, е че при възникване на грешка тя трудно се открива.
2. **Регресионни тестове** - повторно изпълнение на едно подмножество от вече изпълнени тестове.
3. **Валидационни тестове** - установяват дали са изпълнени валидационните критерии описани в документа “Спецификация на изискванията”.
4. **Системни тестове** - това са тестове, които съпровождат интегрирането на разработения софтуер в една голяма система – хардуер, други системи, персонал, информация и др.

**Лекция 16 – Тестване на обектно-ориентирани системи**

1. **Интеграционни тестове в контекста на ОО развой:**

* **Thread-based testing** – интегрира класове, които са необходими за отговор на един вход или събитие към системата.
* **Use-based testing** - тестване на класове, които използват много малко сървърни класове. След това се тестват зависимите класове, които използват тестваните вече независими класове.
* **Cluster testing** - един клъстер от сътрудничещи си класове се анализира посредством тестови случаи, които се опитват да открият грешки.

1. **Валидационни тестове** – използват се use-cases, които са част от аналитичния модел. Те доставят един сценарий, който помага за откриването на грешки. Разделят се на два вида – Alpha и Beta тестове.
2. **Приложение на конвенционалните методи за разработване на тестови случаи:**

* **White-box тестове** – подходящи за тестване на операциите на класовете.
* **Black-box тестове** – use cases подходящи за разработване на тестови случаи.
* **Fault-Based тестове** – тестват се операциите и атрибутите на класовете.

1. **Тестване на повърхностните и вътрешните структури:**

* **Тестване на повърхностните структури** – интерфейсите се изследват за пропуснати сценарии.
* **Тестване на вътрешните структури** – тестват се механизмите на зависимостите, поведенията и комуникациите.

1. **Partition тестване на класове:**

* **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.

**Лекция 18 – Управление на качеството**

1. **Оценяване на софтуерния процес:**
2. **Методологията SEI CMM (Capability maturity model):**

* **Цели:**
* Подобряване на софтуерните процеси
* Оценяване на софтуерните процеси от специално подготвени експерти
* Оценяване на способността на потенциални изпълнители на даден софтуерен проект от специално подготвени експерти.
* Оценяването – на базата на 150 въпроса.
* Подобряването – прави се на базата точно и подробно определени действия в планирането, технологията и управлението на разработването и поддръжката на софтуера.

1. **Методологията BOOTSTRAP** - разработена в рамките на европейската програма ESPRIT през 90-те.

* **Цели:**
* На основата на най-добрите софтуерни практики да се създаде средство за оценяване способността на дадена организация за ефективен софтуерен процес.
* Да се отразят признатите технологични софтуерни стандарти.
* Да се гарантира повторяемостта и устойчивостта на оценката.
* Във всяка оценявана организация да се идентифицират слабите и силните страни на софтуерните процеси.
* Да се подпомогне увеличаването на ефективността на процесите при прилагането на стандартни изисквания.

1. **Стандарти от серията ISO 9000** - появяват се през 1987г. Някои от тях свързани със софтуера са:

* ISO 9001 – отнася се за организации, които проектират, разработват , произвеждат, инсталират и обслужват софтуерни продукти. Състои се от 20 групи изисквания за качество.
* ISO 9004-2 част 2 – предоставя указания за качество на услугите, които са приложими към обслужването на софтуерни продукти.
* ISO 9000-3 – предоставя указания за приложението на ISO 9001 по отношение на разработването, доставката и поддръжката на софтуерни продукти.

**Лекция 19 – Анализ и управление на риска**

1. **Същност** - поредица от стъпки, които помагат на софтуерния тим да управлява несигурността при разработка на един софтуерен проект.
2. **Категории рискове**:

* Познати рискове – които са открити след внимателно изследване на проектния план.
* Предсказуеми рискове – възниквали в минали проекти.
* Непредсказуеми рискове – тези които са трудни за идентифициране.