ВОВЕД

Општествата се социјални групи што се разликуваат според стратегии за егзистенција, начини на кои луѓето ги користат технологија за да обезбедат потреби за себе.

Земјоделско општество - општество во кое основното средство за егзистенција е одгледување на земјоделски култури со употреба на мешавина од човекот и нечовечки средства (т.е. животни и / или машини).

Индустриско општество - општество каде индустријата е основно средство за егзистенција.

Информациско или пост-индрустриско општество - општество каде услужно-ориентираната работа е основно средство за егзистенција.

Информатичкото општество е општество во кое создавањето, дистрибуцијата, дифузијата, употребата, интеграцијата и манипулацијата со информации е значајна економска, политичка и културна активност.

Економијата на знаење е нејзин партнер при што се создава богатство преку економска експлоатација на разбирање.

Карактеристики на информатичкото општество:

- Информацијата е најважната работа
- Информациските технологии стануваат широко распостранети
- Се потпира на складирање на податоци, процесирачка моќ и комуникациски мрежи
- Флексибилност и можност за реконфигурација
- Високи брзини

Кој пишува софтвер? – Бизнисмени, научници, инженери, хобисти, професионалците.

Професионалниот софтвер:

- е наменет за луѓето (не за основачот)
- е креиран од тимови, не од самостојни индивидуи
- е одржуван и менуван во текот на неговиот живот
- е најчесто повеќе од една програма

Тој се состои од:

- повеќе програми

- конфигурациски датотеки за тие програми
- системска документација (ја опишува структурата на системот)
- корисничка документација (опишува како да се користи системот)
- веб сајтови, каде корисниците можат да спуштаат различни работи

Негови карактеристики се:

- нема физички трошоци при неговото креирање (нема ерозија)
- стареење
- долготрајно користење
- лесен за копирање
- тежок за мерење
- комплексен

Терминот софтверско инженерство за прв пат е употребен во 1968 во конференција одржана од НАТО за "софтверската криза" (тешкотиите во креирањето на голем софтвер во 60тите години). На таа конференција било предложено да се направи адаптација на инженерски пристап во креирањето на софтвер, како решение на големите трошоци и изградбата на надежен софтвер.

Софтверското инженерство се занимава со теории, модели и алатки за изработка на професионален софтвер. Тоа вклучува разни техники за програмска спецификација, дизајн и еволуција.

1965-1968: Софтверска криза

1985 – 1989: Зголемено користење на објектно-ориентирано програмирање и CASE алатки

1990 – 1999: Доминацијата и важноста на интернетот

2000 – денес: Лесни методологии, ефтини софтверски решенија, поедноставни и брзи методологии за креирање софтвер.

Знаењето за развој на софтвер има 3 годишен полу-живот: половина од она што треба да го знаеме денес ќе биде застарено во рок од 3 години.

Но, постои и друг вид развој на софтверско знаење - „принципи за софтверско инженерство" – каде не постои тригодишен полуживот. Овие софтверски инженеринг принципи би му служеле на професионален програмер во текот на неговата/нејзината кариера.

[1] Софтверското инженерство (СИ) е примена на систематски, дисциплиниран, квантитативен пристап кон развојот, работењето и одржувањето на софтверот и проучување на овие пристапи. Тоа е всушност таа примена на инженеринг на

софтвер бидејќи вклучува математика, компјутерско инженерство и други практики кои имаат потекло од инженерството.

[2] Друга дефиниција, софтверското инженерство претставува системски пристап до анализа, дизајн, проценка, имплементација, тест, одржување и реинженеринг на софтвер, односно вклучувањето на инженерингот во креирањето на софтвер.

Софтверски инженери се професионалци што развиваат и одржуваат софтверски апликации, користејќи принципи од:

- компјутерско инженерство
- проектен менаџмент
- инжинерство
- апликациски домен
- и др.

Изградбата на софтвер вклучува големи трошоци.

Прашања поврзани со изградба на софтвер:

1. Што е софтвер? – Повеќе програми во еден пакет со соодветна документација, развиен за одредена личност или цел маркет.
2. Атрибути на добар софтвер? – Функционалност, добри перфоманси, корисност, можност за одржување, висока надежност.
3. Фундаментални активности при СИ? - Спецификација на софтвер, развој на софтвер, софтверска валидација и еволуција на софтвер
4. Разлики меѓу СИ и КИ? – КИ се фокусира на теорија и основи. СИ се фокусира на изградба на доверлив софтвер.
5. Разлики меѓу СИ и Системско инженерство? – Системското инж. се фокусира на изградба на компјутерски-базирани системи вклучувајќи хардвер, софтвер и процес инжинерство. СИ подразбира креирање на голема програма која му кажува на компјутерот (изграден од системскиот инженер) да изврши одредена задача.
6. Предизвици на СИ? – Диверзитет, пократко време на достава на софтвер, изградба на доверлив софтвер и др. Најважни се: хетерогеност (техники за изградба на софтвер за истиот да биде компатибилен на повеќе хетерогени платформи и околини), достава (техники за брзината на доставување) и доверба(техники кои се користат за да им се покаже на купувачите дека може да веруваат во софтверот и да се чувствуваат сигурно).
7. Трошоци на СИ? – 60% за изградба, 40% за тестирање.
8. Кое влијание го има интернетот на СИ? – Поголема достапност на софтвер и можности за изградба на квалитетни дистрибуирани сервисно-базирани

системи. Главни карактеристики во ова поле се: користење на веќе креиран софтвер (software reuse), инкрементална изработка на веб базирани системи, ограничувања на корисничките интерфејси од страна на способностите на прелистувачите.

Атрибути на добар софтвер:

- Одржливост
- Сигурност
- Ефикасност
- Прифатливост

Софтверските продукти можат да бидат:

Генерички – за повеќе купувачи (спецификациите за софтверот се одредени од креаторот на тој софтвер и сите промени и одлуки поврзани за тој софтвер се направени од негова страна).

Персонални – за еден купувач (спецификациите за софтверот се одредени од купувачот и сите промени и одлуки поврзани за тој софтвер се направени од негова страна.)

Софтверско инженерство – сите аспекти на изградба на софтвер.

Софтверските инженери треба да користат систематски и организиран пристап во нивната работа.

А што е инженерство? - Инженерството подразбира добивање резултати од потребниот квалитет во рамките на распоредот и буџетот.

Која е работата на инженерот? - Дизајн (чекорите пред финалната конструкција на решението што вклучува формулирање на идеи, скицирање и моделирање, развој на прототипи и тестирање на дизајни).

Главна цел на софтверот – да ја трансформира информацијата.

Тој:

- создава
- трансформира
- менаџира
- собира
- прикажува
- пренесува

ИНФОРМАЦИЈА.

Класично инженерство – изработка на материјален ентитет кој мора да почитува физички правила, за разлика од софтверскиот систем, кој се потпира на логика и токму затоа може да достигне високи нивоа на комплексност.

Има два типа на софтверско стареење: недостаток на движење и игнорантна операција.

Недостаток на движење – стареење поттикнато од неуспехот на креаторите да го модифицираат софтверот за да ги исполни барањата за промена.

Игнорантна операција – стареење поттикнато од направени промени во софтверот.

Намалување на сигурност (“Weight Gain”) – како што старее софтверот, станува поголем и бара повеќе системска меморија.

Како што софтверот е одржуван со текот на времето, се прават грешки.

Системски пристап (или софтверски процес) – множество активности кои се преземаат со цел изработка и еволуција на софтверот.

Генерички активности во сите софтверски процеси се:

- спецификација (што треба системот да прави)
- изработка
- валидација (проверка дека софтверот го исполнува она што клиентот го бара)
- еволуција (промена на софтверот заедно со промената на барањата)

Модел на софтверски процес – симплифицирана презентација на софтверскиот процес, презентирана од специфична перспектива.

Примери за процесни перспективи:

- Workflow перспектива (секвенца активности)
- Data-flow перспектива (тек на информации)
- Role/Action перспектива (кој што прави)

Генерички процесни модели:

- Waterfall
- Итеративна изработка
- Компонентно-базирано софтверско инженерство

СИ методи - Структурирани пристапи за развој на софтвер кои вклучуваат системски модели(графички модели), нотации, правила(ограничувања), совети за дизајн и водење на процеси(кои активности да се следат).

CASE (Computer-Aided Software Engineering) - Софтверски системи што имаат за цел да обезбедат автоматска поддршка за активности на софтверски процеси:

- Upper-CASE : алатки за поддршка на рани процесни активности на барања и дизајн
- Lower-CASE : алатки за поддршка на активности кои се јавуваат подоцна како програмирање, дебагирање и тестирање

Структура на цената на софтверот:

- Up-front fee (едно плаќање, 1990 – 70% од приходот кај софтверските компании, 2000 – 50%)
- Одржување (вообичаено се прави еднаш годишно)
- Друго (инсталација, интеграција, тренинг...)

Апликациски типови:

- Stand-alone апликации (се извршуваат на локалниот компјутер на корисникот)
- Interactive transaction-based апликации (се извршуваат на некој сервер и се пристапуваат од страна на локалните компјутери на корисниците)
- Embedded контролни системи (најзастапени софтверски контролни системи, се користат за контролирање и менаџирање хардверски уреди)
- Batch процесирачки системи (бизнис системи за процесирање податоци во големи количини)
- Entertainment системи (системи за лична употреба и забава на корисниците)
- Системи за моделирање и симулација (системи создадени од научници и инженери за моделирање физички процеси или ситуации)
- Data collection системи (системи кои собираат податоци од нивната околина користејќи сензори и кои ги праќаат тие податоци до други системи за да бидат процесирани)
- Системи од системи (системи изградени од повеќе индивидуални софтверски системи)

2. КВАЛИТЕТ НА СОФТВЕРОТ

1. Софтверски bugs (бубачки) – е било кој проблем кој предизвикува една програма да “падне” (crash) или да проиведе невалиден излез. Овој проблем се предизвикува од недоволна или погрешна логика.

   -bug може да биде грешка, дефект што може да предизвика неуспех или отстапување од очекуваните резултати. Најголем дел од баговите се резултат на човечки грешки во изворниот код или неговиот дизајн.
2. Озогласни софтверски bugs:

   1.1962: Грешка во софтверот за летање на Вселенското летало Маринер 1, што предизвикало промена на патеката по неочекуван пат. Оваа грешка е една од најскапите грешки (158 милиони $). Неправилно кодирана punch картичка.

   2.1985 - 1987: Грешка во кодот за контролирање на машината наречена Терак – 25, која се користела за терапии со зрачење и резултирала со смрт кај пациентите. Оваа машина масивно предозирала 6 лица

   3.1990тите: Грешка во софтверот за контрола на AT&T #4ESS кај прекинувачите на далечина што предизвика да се срушат многу компјутери. Проценета загуба од $120 милион. 9 часа работно време и во просек околу 75 милион пропуштени повици и проценета загуба од 200 000 резервации за лет. Помал механички проблем испратил сигнал за застој порака до сите прекинувачи.

   4.1991: Патриотска ракета не успеала да ги следи и пресретне ракетите на Ирак. Скуд (ракета на Ирак) погодила касарна на американската армија, убивајќи 28 војници и повредувајќи околу 100 други луѓе. Причината за оваа грешка била заради компјутерски аритметички грешки.

   5. 4 Јун 1996: Експлозијата на беспилотната ракета Ariane 5 - уништена неколку секунди по лансирањето поради грешка во компјутерската програма. Прво патување на ракета по деценија на развој. 64 битен број со подвижна точка била претворена во 16 битен цел број со знак.

   6. 2000: Y2K (милениумска грешка) грешка поради која настанал хаос во компутерите и компјутерските мрежи. Грешка која може да создавала проблеми кога се користеле датуми поголеми од 31ви декември 1999. Кога програмите биле првично програмирани, инжинерите ставале простој само за две бројки, оставајќи ги 19 фиксни.
3. Професионална и етичка одговорност:

   -Софтверското инженерство вклучува поголема одговорност од само примена на технички вештини.

   -Инженерите мора да е однесуваат искрено и на етички одговорен начин ако сакаат да бидат почитувани како професионалци.

   -Етичкото однесување е повеќе од само едноставно почитување на законот.
4. Прашања од професионална одговорност:

   -ДОВЕРЛИВОСТ: без разлика дали договорот за доверливост е формално потпишан или не, инженерите мораат формално да ја почитуваат доверливоста на нивните клиенти

   -КОМПЕТЕНТНОСТ: да не се претставува погрешно нивото на надлежност, тие не треба свесно да прифаќаат работа која е вон нивна надлежност.

   -ПРАВАТА НА ИНТЕЛЕКТУАЛНА СОПСТВЕНОСТ: треба да се свесни за локалните закони како што се патенти, авторски права итн. Тие треба да бидат

внимателни и да се осигураат дека интелектуалната сопственост на работодавците и клиентите се заштитени.

-ЗЛОУПОТРЕБА НА КОМПЈУТЕР: Да не се користат техничките вештини што ги поседуваат за злоупотреба на компјутерите на другите луѓе. Оваа злоупотреба може да биде релативно тривијална (играње на играта на работодавачот), до крајно сериозна (дисеминација на вируси).

5. Првите етички кодекси – во 1997 Дон Паркер дефинирал речиси 50 хипотетички примери за злоупотреба на компјутер.

   -Don Parker, Deborah Johnson и ACM: прв акредитиран етички кодекс на компјутерската етика.

   -IEEE објавува свој кодекс на компјутерска етика

   -Кодексот на честа на MASIT

6. ACM/IEEE принципи на етичкиот кодекс:

   1. ЈАВНОСТ: Инженерите треба да се однесуваат конзистентно според јавниот интерес

   2.КЛИЕНТИ И РАБОТОДАВЦИ: да работи во нивен најдобар интерес

   3.ПРОИЗВОД: да биде со највисоки можни стандарди

   4.ПРЕСУДА: одржуваат интегритет и независност во нивното професионална проценка

   5.УПРАВУВАЊЕ: етички пристап кон управувањето со софтверскиот развој и одржување

   6.ПРОФЕСИЈА: унапредување на интегритетот и угледот на професијата во согласност со јавниот интерес

   7.КОЛЕГИ: фер и поддржливи

   8.СЕБЕ: доживотно учење во практикувањето на својата професија и да се промовира етички пристап кон практикувањето на професијата.

7. ACM/IEEE етички кодекс - преамбула: сумаризира аспирации на високо ниво, понатаму се детализираат; без аспирации деталите се легалистички и мачни, без детали аспирациите се гласни, но празни.

   1.Придонес на општеството и човековата благосостојба

   2.Избегнување штета на другите

   3.Бидете искрени и доверливи

   4.Бидете фер и превземете акција да нема дискримнација

   5.Почитувајте ги правата на сопственост, вклучително и авторските права и патент.

   6.Почитувајте ја приватноста на другите

   7.Почитувајте ја доверливоста

8. Стандардизација – процес на развивање и имплементирање на технички стандарди. Процесот поставува заеднички договор за инженерски критериуми, термини, принципи, практики, материјали, предмети, процеси и делови и компоненти на опрема.

9. Корист од стандардизацијата поради:

   1.Овозможува масовно производство

   2.Овозможува прилагодување

   3.Ја подобрува координацијата на снабдувачот

   4.Подобрува квалитетот

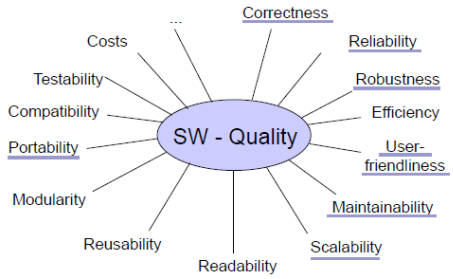
   5.Поставува поедноставување

6.Овозможува одложена диференцијација

7.Намалува инвентар (залихи)

10. Критериуми на квалитет:

Quality criteria for software products



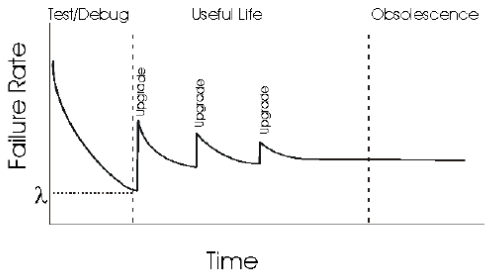
1.КОРЕКТНОСТ: Придржување кон спецификации кои одредуваат како корисниците можат да комуницираат со софтверот и како софтверот треба да се однесува кога се користи правилно.

*Проблем: Софтверскиот производ е точен, но не успева да го постигне она што клиентот го замислил.

2.ДОВЕРЛИВОСТ: Софтверската доверливост е веројатноста дека софтверот ќе извршува за одреден временски период без неуспех. Моделите за доверливост: проценува број на неуспеси во софтверот по развојот според неуспеси во тестирањето.

*Проблем: Софтверската доверливост не е функција на операциското време

Curve for software reliability



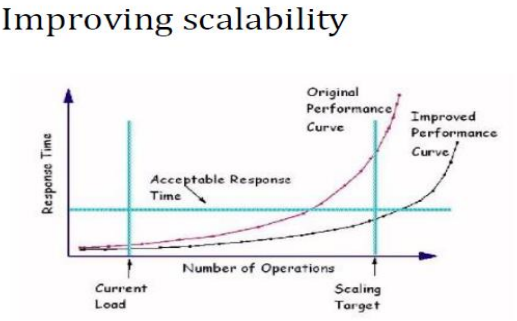
3.РОБУСНОСТ/СТАБИЛНОСТ: Стабилноста претставува способноста на компјутерскиот систем да се справува со грешки при извршување и да се справува со грешен влез. Софтверот е стабилен ако може да толерира грешки како што се предвидени настани, невалидни влезови, корумпирани внатрешно складирани настани, недостапни бази на податоци итн.

*Проблем: оштетување на опремата, загуба на енергијата, паѓања на софтвер итн.

4.ЛЕСЕН ЗА КОРИСНИЦИТЕ: лесни за употреба, едноставен, чист, интиутивен, доверлив

5.ПРИЛАГОДЛИВОСТ/СКАЛАБИЛНОСТ: ако се зголемува бројот на корисници на системот, софтверот ќе продолжи да функционира со споредливи времиња на одговор.

*Проблем: кога една апликација ќе се користи од повеќе корисници, backend ќе почне да се распаѓа



6.ПРЕНОСЛИВОТ: Софтверот да може да се пренесува од една софтверска платформа на друга.

-Облак преносливост на апликациите:

1.Платформа на слоеви како услуга

2.Области на предизвик во преносливоста на апликациите:

-програмски јазик и рамка

-специфични услуги на платформа

-скалдирање на податоци

-специфични датотеки за конфигурацијата на платформата

3.Индустриска перспектива во преносливост на облак

7.ОДРЖЛИВОСТ: Степен до кој апликацијата е разбрана, поправена или подобрена. Одржливоста на софтверот им овозможува на организациите да идентификуваат области за подобрување како и да утврдат вредноста од тековните апликации или за време на развој.

*Главни проблеми со одржливоста:

1.Лош квалитет на код

2.Дефекти на изворниот код

3.Неоткриени слаби страни

4.Преголема техничка сложеност

5.Големи системи

6.Слабо документирани системи

7.Преголем мртов код

11. Класификација на критериумот за квалитет:

1.ВНАТРЕШЕН КВАЛИТЕТ: Овие фактори за квалитет може да бидат согледани само од компјутерски професионалци. Ја одредува способноста да можеме да напредуваме со некој проект.

2.НАДВОРЕШЕН КВАЛИТЕТ:Овие фактори се главно релевантни и се согледуваат од страна на корисниците. Одредува исполнуавњето на барањата на засегнати страни.

*Надворешните фактори на квалитет зависат од внатрешните фактори

Внатрешни карактеристики:	Надворешни карактеристики:
1.Одржливост	1.Коректност
2.Флексибилност	2.Употребливост
3.Преносливост	3.Ефикасност
4.Реупотребливост	4.Приспособливост
5.Читливост	5.Сигурност
6.Тестабилност	6.Интегритет
7.Разбирливост	7.Адаптилност
	8.Точност
	9.Робусност

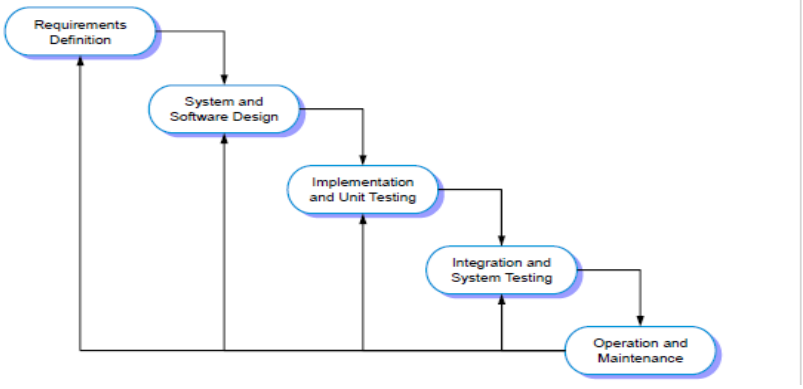
12. Обезбедување на квалитет на софтвер (SQA) – процес што гарантира дека сите софтверско инженерски процеси, методи, активности и работни предмети се следат и се во согласност со дефинираните стандарди.
    -ISO9000, ISO90003, ISO15504, ISO25010
13. Maturity models – модели на зрелост?
    1.OPM3 (Organizational Project Management Maturity Model Third Edition) – глобален стандард кој ги обезбедува алатките според кои организациите треба да ја мерат нивната зрелост. Овој стандард опфаќа итеративен циклус од 5 чекори што нагласува проценка и континуирано подобрување.
    2. Prince maturity model (P2MM) – стандард кој обезбедува рамка со која организациите можат да го проценат нивното тековно усвојување на PRINCE2 project management методот и фокусирање на планови за подобрување со мерливи исходи врз основа на најдобрите практики во индустријата. Обезбедува потврден доказ за зрелост на три нивоа.
    3. Trillium Model – средства за континуирано подобрување. Подржува способност на процесот, самооценување, во преддоговорни преговори
    4. Capability Maturity Model Integration (CMMI) – процес и модел на однесување што помага организациите да се насочат кон подобрување на процесот и поттикнување на продуктивни и ефикасни однесувања кои ги намалуваат ризиците на софтверот, производот, услугата и развојот.
    -Нивоа на зрелост:
    1. Првична: Работата е завршена, но со доцнење и над договрениот буџет
    2.Управувана: Проектите се планираат, извршуваат, мерат и контролираат, но има сепак некои проблеми кои треба да се разгледуваат.
    3. Дефинирани: Низа на организациски стандарди за да обезбедат насоки.
    4.Квантитативно управуавње: Организацијата работи на квантитативни податоци за да утврди предвидлици процеси што се усогласуваат со потребите на засегнатите страни.
    5.Оптимизирање: Постојано подобрување и одговор на промени или други можности.
    -Иднината на CMMI V2.0 – фокус на изведба, интегрирана агилност, додадени вредности, поедноставно за користење и употреба.
14. ISO9000 фамилија – аспекти на управување со квалитет
    -ISO стандардите обезбедуваат насоки и алатки за компаниите и организациите

кои сакаат осигурување дека нивните производи и услуги постојано ги исполнуваат барањата на клиентот и тој квалитет постојано се подобрува

15. ISO/IEC 90003: 2014 – дава насоки на организациите за примената на ISO 9001:2008 за стекнување, снабдување, развој, работење и одржување на компјутерски софтвер и сродна поддршка на услуги.
    -Примена на ISO/IEC 90003: 2014 соодветен за:
        1.Дел од трговски договор со друга организација
        2.Производ достапен за пазарниот сектор.
        3.Се користи како поддршка на процесите во една организација
        4. Вграден во хардверски производ
        5. Поврзани со софтверски услуги.
16. ISO/IEC 155045:2012 одговорен за SPICE, збир на документи за технички стандарди за развојот на софтвер и поврзани бизнис и менаџмент функции
    -Збир на индикатори што треба да се земат предвид при толкувањето на намерата на моделот за референца на процесите.
17. ISO 25010 одговорен за:
        1. Системско и софтверско инженерство
        2. Systems and Software Quality Requirements and Evaluation (SquaRE)
        3. Модели за квалитет на систем и софтвер
18. Критични системи:
    -Многу активности се заменети со компјутерски системи. Нивната неправилна функција може да резултира со некои негативни резултати.
    -Мора да бидат сигурни и доверливи
    -Се одликува со последици поврзани со системска или функциска неуспешност
        1. Fail-operational – типично е потребно да не се работи само во очекувани услови туку и во деградирани ситуации каде некои делови не работат правилно (авиони)
        2. Fail-safe – мора безбедно да се прекине/исклучи во случај на неуспех(воз)
        3. Safety critical - справуавње со сценарија кои можат да доведат до губење на животот, сериозни лични повреди или оштетување на природната средина
        4. Mission critical – избегнува неможност да се заврши целокупниот проект, целите на проектот или една од целите за кои е дизајниран системот.
        5. Business critical - избегнување значителни материјални или нематеријални економски трошоци(губење на деловна активност или оштетување на угледот)
        6. Security critical – справување со губење на чувствителни податоци (кражба или случајно губење)

3. СОФТВЕРСКИ МОДЕЛИ

1. Што е модел на процеси?
   -Општо: план на разој кој го специфицира општиот процес на развојот на производот
   -Поточно: Дефиниција што наведува кои активности треба да се извршат, од кого,постапувајќи во која улога, по кој редослед ќе бидат извршени активностите, кои производи ќе се развиваат и како да се оценуваат.
   -Улоги: членови на тимот, кои извршуваат одредени активности (test engineer, system analysy, project leader, software architect, programmer)
2. Процес на софтвер: структуриран пакет на активности кои се потребни за разој на софтверски систем.
   -Постојат различни софтверски процеси но сите вклучуваат:
   1.СПЕЦИФИКАЦИЈА: дефинира што системот треа да прави
   2.ДИЗАЈН И ИМПЛЕМЕНТАЦИЈА: дефинирање на организиација на системот и негово спроведување
   3.ВАЛИДАЦИЈА: проверка дека се прави тоа што го сака клиентот.
   4.ЕВОЛУЦИЈА: промена на системот како одговор на промените на потребите на клиентите
   -Моделот на софтверски процеси е апстрактно претставување на еден процес. Претставуа опис на еден процес од одредена перспектива.
3. Опис на софтверски процеси – кога опишуваме процеси, зборите за активнсотите, моделот на податоци, корисничкиот интерфејс како и редоследот на овие активнсоти.
   -Може да вклучува и:
   1.ПРОИЗВОД: резултат на процесна активнсот
   2.УЛОГИ: ги одразува одговрностите на луѓето вклучени во процесот
   3.ПРЕД- И ПОСТУСЛОВИ
4. Планирани и агилни процеси:
   -Планирани процеси: процесните активности се планираат однапред и напредокот се мери според овој план.
   -Агилни процеси: Постепено планирање, полесно променување ос цел да се одрази на промената на барањата на плановите
   -Практичните планови се и агилни и планирани. Не постои точен или погрешен софтверски процес.
5. Софтверски процесни модели:
   -МОДЕЛ НА ВОДОПАД (The waterfall model): планиран модел со одделни и разлилни фази за спецификација и развој
   -Има посебни идентифирани фази во моделот на водопад:



*Главен недостаток: тешкотии при промени кога процесот е во тек. Нефлексибилна

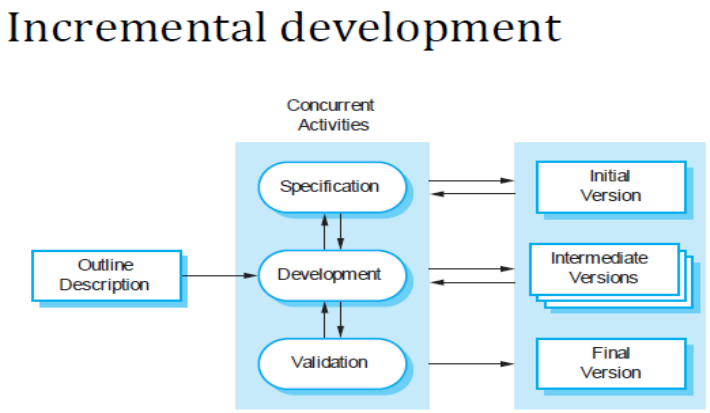
поделба на проектот (затоа најдобар е за планови со добро разбрани барања)

-Најмногу се користи за големи системски инженерски проекти каде системот се разавива на неколку места.

-ИНКРЕМЕНТАРЕН РАЗВОЈ (Incremental development): Спецификацијата, развојот и валидацијата се меѓусебно порзани. Може да биде плански развиван или агилен.

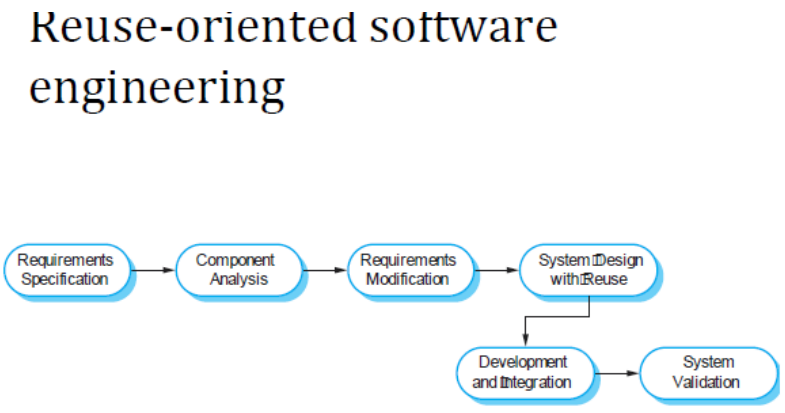
-Итеративен развој: развој во чекори, градиално подобрување

-Инкрементарен развој: одделни делови од производот се развиваат одделно.



Придобивки	Проблеми
1.Намалување на трошоците од промена на барањата на клиентите	1.Процесот не е видлив – потребна е редовна достава за мерење на напредок
2.Полесно е да се добие feedback	2.Деградација на системската структура при додавање на нови податоци
3.Побрза испорака и развој на корисен софтвер до клиентите.	

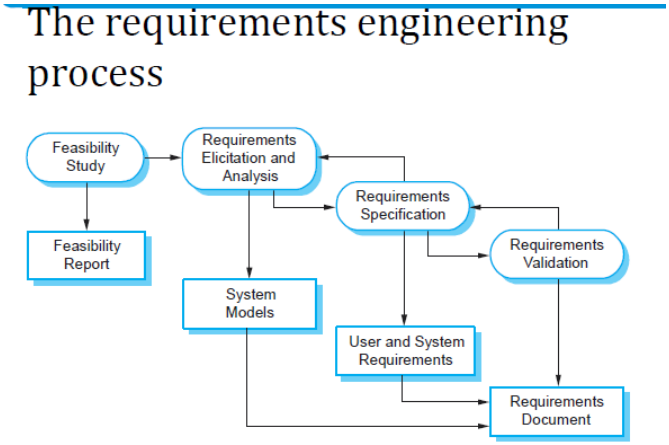
-СОФТВЕРСКО ИНЖЕНЕРСТВО ОРЕНТИРАНО КОН ПОВТОРНА УПОТРЕБА (Reuse-oriented software engineering): Систем составен од постојни компоненти (plan-driven or agile)



*Стандарден пристап за градење на многу видови бизнис системи.

6. Видови на софтверски компоненти – веб услуги што се развиваат според стандарди за услгуги и кои се достапни за далечинско управување. Колекции на предмети што се развиваат како пакет за да бидат интегрирани со компонентна рамка како што се NET or J2EE.

7. Процесни активност: меѓусебни секвенци на технички, колаборативни и менаџерски активности.
-СОФТВЕРСКА СПЕЦИФИКАЦИЈА: процесот на утврдување на тоа кои услуги се потребни и ограничувања во работењето и развојот на системот.



-ДИЗАЈН И ИМПЛЕМЕНТАЦИЈА НА СОФТВЕРОТ(РАЗВОЈ): процесот на конвертирање на системските сецификации во извршлив систем
(Софтверски дизајн- дизајнирај софтверска структура која ја отсликува спецфикацијата; Имплементација – преведи ја струкурата во извршлива програма)

ДИЗАЈН АКТИВНОСТИ:

1.Дизајн на архитектура: идентификација на целокупната структура, главните компоненти, нивните односи и начинот на распределување

2.Дизајн на интерфејс: дефинирање на интерфејсите помеѓу компонентите

3.Дизајн на компоненти: секоја системска компонента се дизајнира како ќе работи

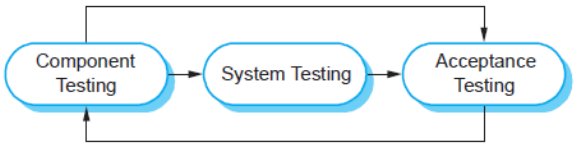
4.Дизајн на база на податоци: дизајн на системските структури на податоци и како тие треба да бидат претставени во базата на податоци

-ВАЛИДАЦИЈА НА СОФТВЕРОТ(Проверка на софтверот) – Верификација и валидација (V&V) има за цел да покаже дека системот одговара на својата спецификација и ги исполнува барањата на клиентот на системот.

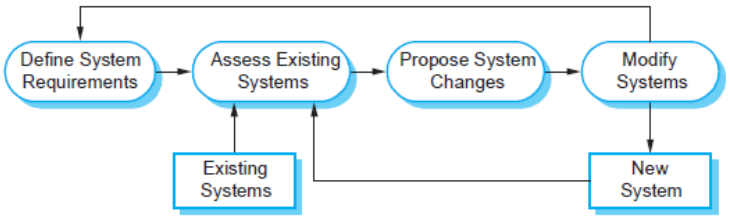
1.Вклучува тестирање на процеси и систем за проверка

2. Системското тестирање вклучува извршување на системот со тест случаеви

Фази на тестирање:



-ЕВОЛУЦИЈА НА СОФТВЕРОТ



8. Справување со промена – промената е неизбежна во сите проекти. Промените во бизнисот водат кон нови и променети системски барања, новите технологии отвараат нови можности. Промената води до преработка, па цената на промената е преработката и цената за внесување на новата функционалност.
-Намалување на трошокот за преработка: променете го избегнувањето, каде софтверскиот процес вклучувајќи активности кои можат да предвидат можна промена пред да е потребна значителна преработка. Променете ја толерантноста, каде што процесот е дизајниран да можат промените да се направат по релативен мал трошок.

9.Прототипирање на софтвер:
-Прототип: почетна верзија на системот, кој се користи за демонстрација на концепти и испробување на опции за дизајн.
Прототипот може да се користи кај: инженерство на барања, процесите на дизајн, процесите на тестирање.
-Придобивки од прототипирањето:
1.Подобрување на употребливоста на системот.
2.Подобро поклопување до вистинските потреби на корисникот
3.Подобар квалитет на дизајнот
4.Подобрена одржливост
5.Намалени напори за развој.

10. Throw-away прототипови – прототипите треба да се отфрлат по развојот бидејќи не се добра основна за системот на производство.

11. Дополнителна достава (Incremental delivery) – наместо системот да се испорача еднократно, развојот и испораката се расчленети со зголемувања со секој прирачник што испорачува дел од потребната функционалност.

Предности	Негативности
1.Функционалноста на системот е достапна порано	1.Распоред на прирачник за употре
2.Навремените зголемување делуваат како прототип за да помогнат во извлекувањето на барањата во подоцнежни зголемувања	2.Суштината на итеративните процеси е дека спецификацијата е развиена заедно со софтверот
3.Помал ризик од целосен неуспех на проектот	
4.Системите со најголем приоритет на системот имаат тенденција да добијат најмногу тестирања	

12.Спирален модел на Боем – процесот е претставен како спирала наместо низа од активнсоти каде што секоја јамка во спиралата е фаза во процесот.
Нема фиксни фази како спецификација или дизајн, сегментите во спиралата се избриаат во зависност од тоа што се бара.
Ризиците се експлицитно оценети и решени во текот на целиот процес.

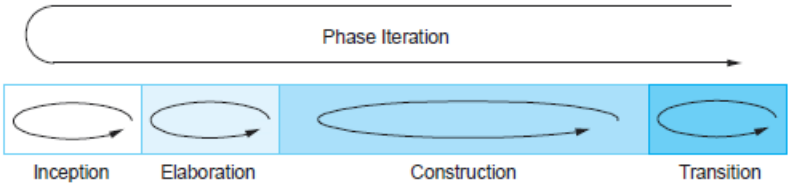
-Сектори на спирални модели:

1.Поставување на целта
2.Проценка и намалување на ризикот
3.Развој и валидација
4.Планирање

13.Рационален Унифициран процес - современ генерички процес произлезен од работата на UML и придружниот процес. Соединува аспекти на 3 генерички модели на процеси кои претходни дискутирани.
-Нормално опишани од 3 гледишта:

1.Динамчка перспектива: фази над време
2.Статичка перспектива: активностите на процесот
3.Практична перспектива: сугерира добра пракса

-Фази:



-Итерација(повторување):

1. Во фазно повторување: секоја фаза е повторлива со развиените резултати постепено
2.Кросфазно повторување: целиот пакет на фази може да се донесе постепено

-Добра практика:

1.Итеративно развивајте софтвер
2.Упраувајте со барањата
3.Користете архитектури базирани на компоненти
4.Визулено моделирјте софтвер
5.Потврдете го квалитетот на софтверот
6.Конторлирајте ги промените на софтверот

4. АГИЛЕН РАЗВОЈ

1. Брз развој на софтвер: често најважен услов за софтверските системи.
-Спецификацијата, дизајнот и имплементацијата се меѓусебни
-Системот се развива како серија верзии со засегнати страни вклучени во евалуација на верзиите
-Корисничките интерфејси често се развиваат со IDE и графички алатки

2. Агилни методи (’80-’90) – овие методи се со фокус на кодот наместо на дизајнот, базирани на итеративен пристап кон развој на софтверот, наменети за брза испорака на работен софтвер.
-Цел: намалување на трошоците во софтверскиот процес и брз одговор на промената на барањата.

3.Агилен манифест – “Откривање на подобри начини за развој на софтвер и со тоа им помагаме на другите да го сторат истото. Со оваа работа ја достигнавме вредноста: ”
-Поединци и интеракции над процеси и алатки
-Работлив софтвер пред сеопфатна документација
-Соработка со клиенти пред преговорите за договор
-Одговарање на промена пред следење на планот.

4. Принципи на агилните методи:
-Вклученост на клиентите
-Дополнителна достава
-Луѓе не процеси
-Прифаќање на промените
-Одржување на едноставност

5. Применливост на агилниот метод
-Развој на производи каде софтверската компанија развива мал или среден производ за продажба
-Развој на сопствен систем во рамките на една организација, каде постои јасна заложба на клиентот да се вклучи во процесот на развој и кога нема многу надворешни правил и регулативи кои влијаат на софтверот.

6. Проблеми со агилните методи
-Прилагодување на големи системи
-Тешко задржување на интересот на клиентот вклучен во процесот
-Членовите на тимот може да се непогодни за интензивно вклучување
-Тешка е приоретизацијата на методите
-Одржувањето на еднсотавноста може да бара повеќе работа
-Договорите може да се проблем

7.Одржување на агилост
-Клучни проблеми:
1.Недостаток на документација за производите
2.Одржување на вклученоста на клиентите во процесот
3.Одржување континуитет на тимот за развој
-Развојот на агилноста се потпира на тиммот за развој. За системтите со долг живот ова е проблем бидејќи оригиналните инженери нема секогаш да работат на системот.

-Технички, човечки, организациски проблеми:

1. Доколку има потреба од добар план – базирани на план
2.Дополнително доставување – агилен
3.Големи системи – базирано на план
4.Тип на системи – доколку бара многу анализирања подобро базирани на план
5. Очекуван животен век – доколку се со подолг животен век подобро базирани на план
6.Потпирање на добри алатки за водење сметка за развојот на дизајнот – агилен
7.Традиционалните инженерски организации имаат кулутра на развој базирана на план
8.Со агилниот метод се потребни повешти дизајнери и програмери
9. Ако системот треба да биде потврден од надворешни регулатори – подобро да се базирани на план

8. Екстремно програмирање (XP) – најпознат и најкористен агилен метод, користи екстремен пристап кон итеративен развој.

-Новите верзии може да се градат неколку пати на ден
-Зголемувањата се испорачуваат на клиентите на секои две недели

9. XP и агилни принципи:

-Дополнителниот развој е подржан од мали, чести системски изданија
-Вклученоста на клиентот подразбира 100% ангажираност на клиентот и тимот
-People not process through pair programming, collective ownership and a process that avoids long working hours.
-Промената е поддржана од редовните изданија на системот.
-Одржување на едноставноста преку постојано рефакторирање на кодот.

10.XP практики:

-Инкрементално планирање
-Мали изданија
-Едноставен дизајн
-Test-first развој
-Рефакторирање
-Pair programming – работење во парови, при што се врши меѓусебна проверка на работата и се обезбедува поддршка
-Колективна сопственост – поради работата во парови никој не зема сам одговорност
-Континуирана интеграција – само што ќе се заврши една задача се интегрира во системот
-Одржливо темпо
-Клиент на лице место – самиот клиент е член од тимот за да може да биде ефективно

11. XP

-Наместо промени, постојано подобрување на кодот.
-Подобрување и на места на кои нема реална потреба за нив
-Тестирање:
    1.Прв тест развој – пишување на тестови пред код

2.Зголемен тест на развој од сценарија

3.Вклучување на клиентите во тестирање и валидација

4. Автоматизираните прицврстувачи за тестирање се користат за извршување на сите тестови на компонентите на секој пат кога има ново издание.

-Потешкотии со XP тестирање

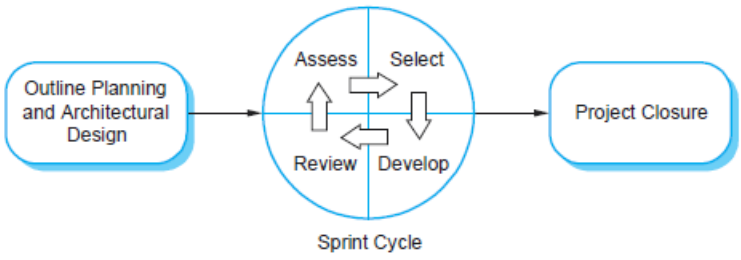
1. Програмерот да напише непотполн тест
2. Тешко пишување тестови
3. Тешко е да се процени комплетноста на тестовите

12.Агилно управување со проекти – потребно е различен пристап прилагоден на инкременталниот развој и на посебните јаки страни на агилните методи.

13. Scrum – агилен метод кој се фокусира на управување со итеративен развој, наместо конкретни агилни практики.

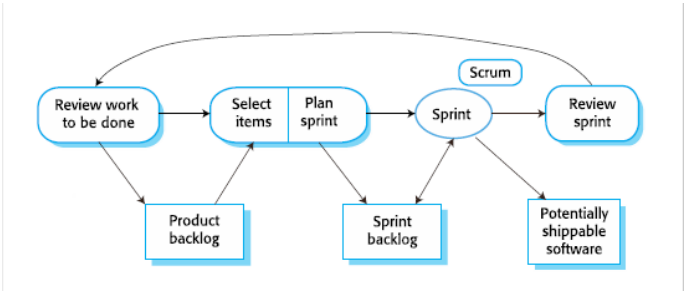
-ФАЗИ:

1. Планирање на прегледот, каде што се утврдуваат општите цели за проектот и се дизајнира архитектурата на софтверот
2. Низа циклуси на спринтови, каде што секој циклус развива прираст на системот
3. Фаза на затворање на проектот, ја комплетира потребната документација како системски рамки за помош и упатства за корисници и ги оценува научените лекции од проектот.



-ТЕРМИНОЛОГИЈА

-Scrum спринт циклус



Овие спринт циклуси се со фиксно времетраење обично од 2 – 4 недели. Почетна точка е заостанатоста на производот. Фазата на селекција – бираат карактеристики и функционалности што треба да се развијат за време на овој циклус.

-Придобивки:

1. Производот е поделен во збир на разбирливи делови
2. Нестабилните барања не напредуваат
3. Целиот тим има пристап до се
4. Навремена достава на инкрементот
5. Доверба меѓу клиент развивачите

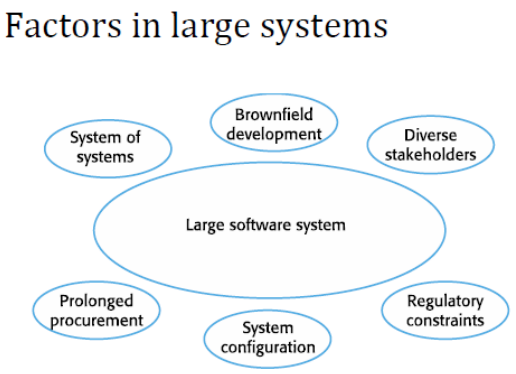
-Проблеми:

1. Ноформалноста на агилниот развојј некомпатибилна со правниот пристап
2. Најсоодветна за развој на нов софтвер, отколку одржување на постоечки
3. Дизајнирано за мали тимови, но сега има многу софтверски развој со тимови дистрибуирани низ целиот свет

14. Multi-team Scrum

-Репликација на улогата – секој тим има сопственик на производи и ScrumMaster
-Архитекти на производи – дизајн и развој на целокупната системска архитектура
-Усогласување на изадвање – датумите на издавање на производите се усогласени
-Scrum of Scrums – претставници од секој тим разговараат за напредокот и работата што треба да се направи

15. Scaling кај агилните методи



-Развој на големи системи – ограничени со надоврешни правила
-Scaling up – употреба на агилни методи за развој на големи софтверски системи што не може да ги развива мал тим
-Scaling out – воведување агилни методи низ голема огранизација со долгогодишно искуство во развој на софтвер

16. Crystal: збир на пристапи заосновани на идејата дека секој проект има потреба од различен збир на политички конвенции и методологии.
- Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Orange Web, Crystal Red, Crystal Maroon, Crystal Diamond, Crystal Sapphire

17. DSDM (Dynamic Systems Development Method)
–Главна идеја: Фиксирајте ги времето и ресурсите (временска рамка), соодветно прилагодете ја функционалноста

-Фази:

1. Feasibility
2. Business study
3. Functional Model Iteration
4. Design and Build Iteration
5. Implementation

18. Kanban – систем за закажување за лесно производство и just-in-time производство. Систем за контрола на залихите со цел да се конторлирање ланецот на снабдување.
-Обезбедува визуелен систем за управување со процеси што помага во донесувањето одлуки за тоа што да се произведе, кога да се произведува и колку да се произведува.
-Работен тек од три чекори:

1. Да се направи
2. Во тек
3. Завршено

5. ИНЖЕНЕРСТВО НА БАРАЊА

1.Инженерство на барања - процесот на поставување на услуги кои ги бара клиентот од еден систем и ограничувања под кои тој се извршува и се развива.
Барањата на системот се објаснувања за системскиот сервис и ограничувањата кои се генерираат за време на процесот на инженеринг барања.

2.Што е барање?
-Може да варира од високо ниво на апстрактно тврдење на системот или од системски ограничувања до детални математички функционални спецификации.
-Ова е неизбежно бидејќи барањата можат да имаат двојна функција
1.Основа за понуда за договор – затоа мора да биде отворен за интерпретации
2.Основа за самиот договор – затоа мора да бид објаснет во детали
3.Двата овие искази можат да се наречат барања

3.Видови барања:
- Барања на клиентите(корисниците) – исказите во природен јазик плус дијаграми на услугите кои системот ги обезбедува како и неговите операциски ограничувања. Напишано за клиенти.
Пр: Mentcare системот треба да генерира месечни менаџерски (управувачки) извештаи во кои се покажува трошокот на лековите кои се препишани од секоја клиника за време на тој месец.
- Системски барања – структуриран документ со детални описи за функциите на системот, услугите и операциските ограничувања. Опишува што треба да биде имплементирано, па може да биде дел од договор меѓу клиентите и изведувачот.
Пр: Последен ден од месецот, summary на препишаните лекови, трошоците и клиниките до кои (за кои) е препишано... ++

4.Читачи на различните типови на барањата:
-User requirements - > менаџери на клиентите, крајните корисници на системите, клиентските инженери, менаџер на изведувачи, системски архитекти
-System requirements -> крајни корисници на системите, клиентски инженери, системски архитекти, создавачи на софтвер.

5. Стеикхолдер (stakeholders) - > било која личност или организација која е погодена од системот на некој начин и тој кој има легитимен интерес.
Типови на стеикхолдери: - Крајни корисници
- Системски управувачи
- Сопственици на системот
- Надворешни чинители (stakeholders)

6. Стеикхолдери во Mentcare system:
-Пациенти чија информација е внесена во системот
-Доктори кои се одговорни за примање и лекување на пациенти
-Медицински сестри кои ги координираат консултациите со докторите и администрираат некои лекувања.
-Медицински рецепционисти кои управува со закажувањата / термините на пациентите.
-ИТ персонал кој е одговорен за инсталирање и одржување на системот.

-Менаџер на медицинска етика кој мора да се осигура дека системот се совпаѓа со сегашните етички упатства за негата на пациентите.
-Менаџерите на здравствениот систем кој обезбедува информација за системот
-Персонал кој работи со медицински досијеа кои се одговорни за осигурување дека системската инфомација може да биде одржувана и зачувана, и дека процедурите за водење на евиденција се правилно споделени

7. Агилни методи и барања:
-Многу агилни методи тврдат дека продуцирање на детални системски барања е трошење на време бидејќи барањата се менуваат премногу брзо. Затоа документите за барањата се секогаш застарени
-Агилните методи вообичаено користат incremental requirements engineering (зголемувачки инженерски барања) и може да ги изразуваат барањата како кориснички приказни
-Ова е практично за бизнис системите но проблематично за системите кои бараат анализи за пред-испорака (пр. Критични системи) или системи развиени од страна на неколку тимови.

8. Функционални и не-функционални барања:
-Функционални барања – искази за услугите кои системите треба да ги обезбедуваат, како системите треба да реагираат на посебни влезови и како системот треба да се однесува во посебни ситуации. Може да има што системот не треба да прави.
-Нефункционални барања – ограничувања на услугите или функциите кои се понудени од страна на системот како временски ограничувања, ограничувања на равојот на процесот, стандарди итн. Често се применуваат на системите како целина наместо на индивидуални услуги или карактеристики.
-Барања за домен – ограничувања на системот од страна на доменот на операцијата.

9. Функционални барања:
-Опиши функционалност или системски услуги
-Зависи од типот на софтверот, очекуваните корисници или типовите на системот каде софтверот треба да бид користен..
-Функционални кориснички барања може да бидат искази на високо ниво за тоа што треба системот да прави
-Функционални системски барања треба да ги опишат детално системските услуги.

10. Mentcare system: функционални барања
-Кориснкот треба да биде способен да пребарува низ листи за термини на сите клиники.
-Системот треба да се генерира секој ден, за секоја клиника, листа на пациенти кои се очуваат да дојдат на терминот тој ден.
-Секој вработен кој го користи системот треба да биде уникатно идентифициран со неговиот осумцифрен број на вработен.

11. Непрецизност на барањата
-Проблемите настануваат кога функционалните барања не се прецизно поставени
-Двосмислените барања може да бидат разбрани на различни начини од страна на создавачите и корисниците.

12. Комплетност на барањата и конзистентност - Во принцип, барањата треба да биде конзистенти и комплетни.
-Комплетност - > треба да вклучат опис за сите барани објекти (facilities)
-Конзистентост - > треба да нема конфликти или контрадикции во описот на системски објекти
-Во пракса, поради системска и околинска комплекност, не е возможно да се произведе комплетно и конзистенто барање на документ.

13. Нефункционални барања -дефинираат системски својства и ограничувања како доверливост, време на одговор и барања за складирање. Ограничувањата се И/О способнот, системски репрезентации и сл.
-Процесните барања можат да бидат специфицирани за посебни IDE, програмски јазик или развоен метод.
-Нефункционалните барања може да бидат покритични од функционалните барања. Ако овие не се соедниат, системот може да биде бескорисен.
-Типови нефункционални барања: производни барања, организацики барања, надворешни барања
-Имплементација: Овие барања може да влијаат на целокупната архитектура на системот наместа на индивидуални компоненти.
-Едно нефункционално барање, како барање за безбедност може да генерира гоелм број на поврзани функциоанлни барања кои ги дефинираат системските услуги кои се барани

14. Класификација на нефункционални барања.
1.Продуктни барања – барања кои специфицираат дека доставениот производ мора да се однесува на одреден начин (брзина на извршување , надежност..)
2.Организациски барања – барања кои се последица на организациски полиси и процедури како на пример стандардите кои се користени за време на процесот, имплементациски барања итн.
3.Надворешни барања – барања кои настануваат од факотри кои се надворешни за системот и неговите равојни процеси како на пример барањата за интероперабилност, барања за легислативата и така натаму .

15. Цели и барања
-Нефункционалните барања може да бидат многу тешки за да се изјаснат прецизно и непрецизните барања може да се тешки да се потврдат
-ЦЕЛ; Генерална намера на клиентот како на пример леснотија при користење
-Потврдено нефункционлано барање – исказ со користење на мерки кои може да бидат објективно тестирани
-Целите се од помош за создавачите бидејќи тие ги пренесува намерите на системските корисници

16. Барања за корисност(usability requirements) - системот треба да биде едноставен да се користи од страна на медицински лица и треба да биде организиран на тој начин да се минимализираат грешките

17. Процеси на инженерски барања – процесите кои се користат за овој тип на барања може да се разликуваат во зависност од доменот на апликацијата, луѓето кои се вклулени и орагизацијата која ги развива овие барања.
-Активности кои се исти за сите процеси:

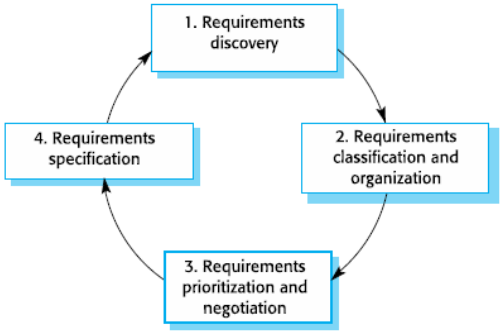
1. Елицитација на барањата
2. Анализа на барањата
3. Валидација на барањата
4. Управување со барањата

-Во пракса, РЕ е итеративна активност во која овие процеси се меѓусебни

18. Елицитација на барањата – наречено и откривање на барањата; вклучува технички вработени лица кои работат со клиентите за да дознаат повеќе за доменот на апликацијата, услугата која треба да се обезбеди и системски операциски ограничувања. Може да вклучи крајни корисници, менаџери, инженери вклучени во одржување, експерти за домени, -- стеикхолдери
-Проблеми:

1. Стеикхолдерите не знаат точно што сакаат
2. Стеикхолдерите изразуваат барања на нивни начин
3. Различни стеикхолдери може да имаат конфликтни барања
4. Организацискиот и политичкиот факотр може да влијаат на системот на барања
5. Барањата се менуваат за време на процесот на анализа.

19. Процесот на елицитација и анализа на барањата



-Откривање на барањата: процес на собирање на информации за потребните и постоечки системи и филтрирање на барањата на корисникот и системот од овие информации.
-Формалните и неформалните интервјуа со стеикхолдерите се дел од најголем број на процесите на елицитација на барањата.

-Видови интервјуа:

1. Затворено интервју – базирани на однапред одлучена листа од прашања
2. Отворено интервју – каде што се разгледуваат различни проблеми со стиекхолдерите.

-Интервјуата во пракса се најчесто комбинација од отворено и затворено интервју. Тие се добри за да може да се разбере што сакаат стеикхолдерите. Би требало да се предложуваат барања наместо да се прашува што сакаат стеикхолдерите.

-Проблеми со интервјуата - одредени специјалци може да користат терминологија која што не е едноставна за инжинерите на барања да ја разберат.

20. Етнографските истражувања покажуваат дека работата е всушност поскапа и покомплексна од она што е предложено во едноставните системски модели.
-Барања кои се изведени од начинот на кој луѓето работат наместо начинот кој е предложен според дефиницијата на процесот.
-Барања кои се изведени од кооперација и свесност за активноста на другите луѓе.
-Етнографијата е ефективна за разбирањето на постоечки процеси, но не може да идентификува нови каркатеристики кои можат да бидат додадени во системот.

21. Фокусирана етнографија – развиена во проект за контрола на процесот на воздушниот сообраќај. Комбинира етнографија со прототипирање.

22. Приказни и сценарија – се приказни од вистинскиот живот а тоа како еден систем може да се употреби. Тие се опис за тоа како еден систем може да биде користен за одредена задача.
-Сценарија – структурирана форма од приказна на корисникот
-Сценаријата треба да вклучуваат:
1. Опис на почетната ситуација
2. Опис на нормалниот тек на настаните
3. Опис на она што може да биде погрешно
4.Информација за останатите активности
5. Опис на состојбата кога ќе биде завршено сценариото

23. Спецификација на барањата
-Процесот на пишување на корисничките и системските барања во документот за барањата.
-Кориснички барања – треба да се разбирливи за крајните корисници и клиенти.
-Системски барања – подетални барања и може да вклучуваат повеќе технички информации.
-Барањата може да бидат дел од договор за системскиот развој, затоа е важно да што е можно покомплетни.

24. Начини на пишување на спецификација на системски барања:
-Природен јазик
-Структуриран природен јазик
-Јазици за опис на дизајн – го користи јазикот како јазик за програмирање со повеќе апстрактни карактеристики.
-Графички нотации
-Математички спецификации

25. Барања и дизајн – барањата треба да содржат што треба системот да прави и дизјанот треба да опишува како го прави тоа.
-Барањата и дизајнот се неразделиви.

26. Спецификација на природниот јазик – барањата се пишуваат со реченици на природен јазик подржан со дијаграми и табели. Се користи за пишани барања бидејќи треба да е разбирлив и за клиентите и за корисниците (експресивни, интиутивни, универзални).

-Насоки за пишување на барања

1. Измислете стандарден формат и користете го за сите барања
2.Користете го јазик на конзистетнте начин.
3. Користете нагласување на текстот да идентификувате клучните делови на барањата.
4. Избегнувајте да корситите компјутерски жаргон
5. Користете објаснување (рационално) за тоа зашто барањето е неопходно.

-Проблеми со природните јазици:

1. Недостаток на јасност
2. Конфузност на барањата
3. Спојување на барањата

27. Структурирани спецификации – пристап на пишување барања каде што слободата на авторот на барањата е ограничен и барањата се напишани на стандарден начин. Ова е добро за некои видови на барања, но понекогаш може да е премногу строго за пишување на системски бизнис барања.

28. Табеларни спецификации – се користи за дополнување на природниот јазик. Корисно кога треба да се дефинира број на можни алтернативни акции.

29. Use-case – сценарија кои се користат во UML. Тие идентификуваат актерите во интеракцијата и ја објаснуваат самата интеракција.
-Графичките модели на високо ниво се надополнуваат од подетални табеларни описи.
-UML дијаграмите може да се

30. Документ за софтверски барања – официјалната изјава за тоа што е потребно од страна на оние кои го развиваат системот. Треба да вклучува дефиниција за кориснички барања и спецификација на системските барања. Не е документ за дизајн.
-Варијабилност на документот за барања:

-Информацијата во документот за барања зависи од типот на систем и од пристапот за развој кој се користи.

*Системите кои се развиваат инкрементално, обично имаат помалку детали во документот на барања
IEEE стандардот (стандрад на документ за барања) е употреблив најчесто за барања на големи системско-инженерски проекти.

31. Валидација на барањата – одговрни за демонстрација дека барањата го дефинираат системот кој клиентот навистина го сака.
-Трошоците за грешките на барањата се многу високи, што ја прави валидацијата многу важна.
-Проверка на барањата:

1. Валидација – дали системот ги овозможува барањата кои најдобро ги задоволуваат потребите на клиентот.
2. Конзистентост – дали има некои конфликтни барања?
3. Потполност – дали сите функции побарани од клиентот се вклучени
4. Реалност – Дали барањата можат да се исполнат според дадениот буџет и технологија
5. Веродостојност – Дали барањата можат да се проверат?

-Техники за валидација на барањата:

1. Преглед на барањата – системска мануелна анализа на барањата. Оваа анализа треба да се прави додека се дефинираат барањата. Треба да се анализира од страна на клиентите и преговарачот.

-Може да бидат формални и неформални.

-Проверка:

1.Веродостојност

2. Разбирливост

3.Следливост

4.Прилагодливост

2. Прототипирање

3. Генерирање на тест-случаеви – развој на тестови за барањата за да се провери тестабилноста.

32. Промена на барања – бизнис и техничката околина на системот секогаш се менува по инсталацијата.

-Луѓето кои платиле за тој систем и корисниците на системот ретко се истите луѓе.

-Големите системи обично имаат разновидна заедница на корисници, каде што многу корисници имаат различни барања и приоритети кои можат да бидат конфликтни или контрадикторни.

33. Управување со барањата – процес на управување со промената на барањата за време на процесот на инженерство на барањата и развојот на системот.

-Одлуки на управување со барањата:

1. Идентификација на барањата
2. Управување со процес на промена
3. Политики за следливост
4. Алатки поддржуваат алатки

-Одлучување дали одредена промена треба да биде прифатена:

1. Анализа на проблемот и спецификација на промената
2. Анализа на промената и трошок
3. Имплементација на промената

6. МОДЕЛИРАЊЕ НА БАРАЊАТА

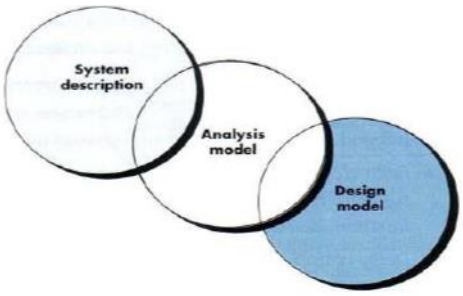
1. Анализа на барањата:
   -Ги специфицира оперативните карактеристики на софтверот
   -Посочува софтверски интерфејс со други системски елементи
   -Поставува ограничувања кои софтверот мора да ги исполнува.

-При моделирање на анализа, фокусот е на ШТО, а не на КАКО
-Моделот на анализа и спецификација на барањата им овозможува средства за проценка на квалитет откако ќе се изгради софтверот

2. Принципи на моделирање на анализа:
   -Информацискиот домен на проблемот мора да бид претставен и разбран (служи како компас на текот на податоците во системот, надвор од системот и склад на податоци)
   -Функциите на софтверот мора да бидат дефинирани (претставуваат директен бенефит на крајните корисници, и се внатрешна поддршка на одлуките што не се видливи за корисникот)
   -Однесувањето на софтверот мора да биде претставено (предводено од интеракција со надворешната околина)
   -Моделите кои покажуваат информација, функција и однесување мора да бидат поделени да откриваат детали на слоевит начин (клучна стратегија:се дели комплексниот проблем во подпроблеми се додека не станат лесни за разбирање – partitioning)
   -Анализирањето треба да се движи од неопходна информација кон детал за имплементација (суштината на проблемот е пишана без разгледување како решението ќе биде имплементирано; моделот за дизајн одлучува за неговата имплементација)

3. Три примарни цели:
   -Опиши што бара корисникот
   -Основај база за креирање на софтверски дизајн
   -Креирај сет барања кои можат да бидат валидирани откако ќе биде изграден софтверот

-Овие цели ја пополнуваат дупката помеѓу описот на системско ниво кои ја опишува целосната системска фукционалност и софтверскиот дизајн

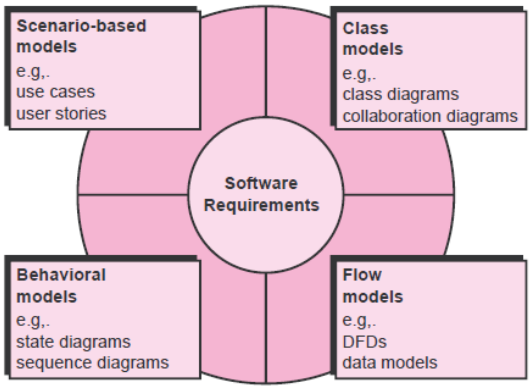


4. Правила за анализа на палец (Analysis Rules of Thumb) – фокусот се поставува на видливи барања на повлемот или бизнис доменот, каде што нивото на апстракција би требало да биде високо без премногу детали.
-Секој елемент треба да дава придонес за целокупното разбирање на софтверските барања и да даде увид во информацискиот домен, функција и однесување на системот.
-Минимизација на спојување низ системот, ако нивото за поврзување е високо треба да се гледа да се намали.
-Треба да нуди вредност на сите стеикхолдери
-Одржување на моделот на што е можно поедноставен

5. Елементи на моделот на анализа:
-Два пристапи:

1. Струкутирана анализа – податочните објекти се моделирани на начин што ги дефинира нивните атрибути и врски. Процесите покажуваат како податочните објекти ги трансформираат податоците во податочни објекти низ текот на системот.

2. Објектно-ориентирана анализа – фокус на дефиниција на класи и начин на нивна меѓусебна колаборација (UML)



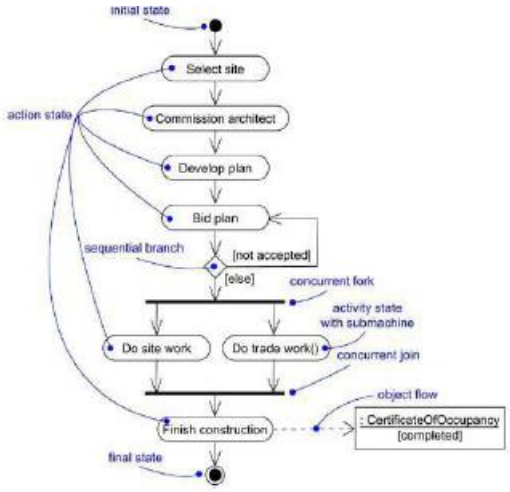
6. Дијаграми базирани на сценарија:
-Елементи:

1. Use-case: како надворешни соучесници комуницираат со системот
2. Functional: како софтверските функции се процесираат во системот
3. Activity: претставена на повеќе различни нивоа на апстракција

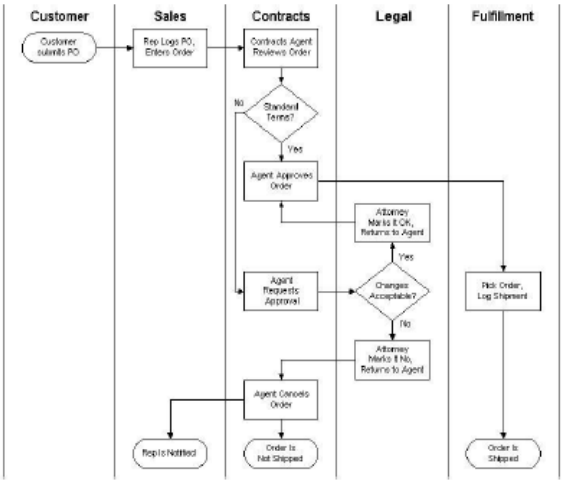
7. Формален use-case:

- Use case:
- Iteration:
- Primary actor:
- Goal in context:
- Preconditions:
- Trigger:
- Scenario:
- Exceptions:
- Priority:
- When available:
- Frequency of use:
- Channel to actor:
- Secondary actors:
  - Channels to secondary actors:
- Open issues:

8. Activity Diagram



9. Swimlane Diagram



10. Класен дијаграм за сензор
-Елементи базирани на класи: разни системски објекти (од сценарија) со нивни атрибути и функции (класен дијаграм)

11. Бихејвориален елемент – како системот се однесува како одговор на различни настани (state diagram)

12. Flow-oriented elements – како информацијата се менува како што минува низ системот. Системот прифаќа влез во разни форми, применува функции за го трансформира и прозведува излез во разни форми.

13. Моделирање на податоци
-Моделот на анализа често започнува со моделирање на податоците
-Моделот на податоци се состои од три меѓусебно поврзани парчиња информации:
1. Објект на податоци (data object)
2. Атрибутите што го објаснуваат предметот на податоци
3. Односите што ги поврзуваат предметите на податоци еден со друг

14. Објект на податоци (Data Object) – репрезентација на било која сложена информација која треба да биде разбрана од страна на софтверот.
-Сложена информација – повеќе разни својства или атрибути
-Пример: надворешен ентитет (продуцира/конзумира информација), нешто (извештај/дисплеј), настан (аларм), улога(продавач), организациска единица (сметководство), место (магацин), струкутра (датотека).

15. Атрибути на податоци (Data Attributes) – ги дефинираат својствата на објектот на податоци.
-Именуваат објект на податоци, ги опишуваат карактеристиките и во некои случаи упатуваат кондруги објекти. Барем еден атрибут мора да биде идентификатор.

16. Односи – поврзување на објекти на податоци. За да може да се одреди односот помеѓу објектите на податоците.
-Се дефинираат сет на објекти/врска парови кои дефинираат релевантни врски. (book – bookstore)

17. Кардиналност – колку појави од првиот објект се поврзани со колку појави од вториот објект на податоци. (Елемент на моделирање на податоци)
-Број на појави на објекти во една врска.
-Кај кардиналноста вообичаено го користиме изразот еден или повеќе ('one' or 'many')
*1:1 –еден објект може да се поврзе (се однесува) само на еден друг предмет.
*1:M – еден објект може да се однесува на повеќе објекти
*M:N – одреден број на појави кај еден објект се однесуваат на број на појави кај друг објект

18. Модалитет – кардиналноста не дава индикација дали одреден податочен објект мора да учествува во врската.
-Модалитетот на врската е 0: не мора, по избор
-Модалитетот на врската е 1: мора, задолжително

19. Моделирање на функција и тек на информација – при текот на информацијата во компјутерските базирани системи, таа се менува.
-Структурната анализа – техника за моделирање на текот на информациите.
-Елементи:
-правоаголник – надворешен ентитет(софтвер, хардвер, личност)

-круг – за процес или трансформација
-стрелка – еден или повеќе податочен објект и треба да биде означена
-двојна линија – зачувана информација што ја користи софтверот
-First data flow model (DFD) – го претставува целиот систем.

20. Креирање на модел за тек на податоци (Data Flow Model)
-Овозможува софтверските инженери да можат да развиваат модели на информатички и функционален домен во исто време.
-Дијаграмот за проток на податоци може да се користи за да претставува систем или софтвер на кое било ниво на апстракција.
-Бидејќи DFD е рафинирана во поголеми нивоа на детали, аналитичарот врши имплицитно функционално распаѓање на системот
-DFD ниво 0 (фундаментален модел на системот/модел на контекст) – го претставува целиот софтверски елемент како едно топче со В/И податочи означени со дојдовни и појдовни стрелки.
-Рафинирање DFD ниво 0 во DFD ниво 1 – со сите релевантни процеси до системот
-Сите DFD процеси на ниво 1 може да се рафинираат во DFD ниво 2.
-Рафинирањето на DFD продолжува се додека секое топче не извршува едноставна функција.

21. DFD упатства
-Прикажи го системот како едно топче на нулто ниво.
-Примарниот влез и излез треба внимателно да се забележани
-Рафинирајте со изолирање на процесите на кандидатот и нивните поврзани објекти за податоци и продавници за податоци
-Сите стрелки и кругчиња треба да биде означени со значајни имиња
-Континуитетот на текот на информациите мора да се одржува од ниво на ниво
-Едно по едно кругче да се рафинираат.

22. Control flow model – апликација што содржи колекција на класи зависни од настани, а не од податоци и продуцира контролна информација наместо извештаи или прикази.
-Ваквите апликации бараат употреба на моделирање на контролен тек во прилог на моделирање на текот на податоците.
-Упатства:
1. Наведете ги сите процеси што ги изведува софтверот
2. Наведете ги сите услови на прекинување
3. Наведете ги сите активности што ги извршува операторот или актерот
4. Наведете ги сите услови за податоци
5. Прегледајте ги сите „контролни артикли“ што се можни за В/И контролни текови.
6. Опишете го однесувањето на системот со идентификување на неговите состојби; идентификувајте како се достигнува секоја состојба; ги дефинираат транзициите помеѓу состојбите
7. Фокус на потенцијална емисија – честа грешка во CS

23. Control Specification (CSPEC) – го претставува однесувањето на системот на два различни начини
1. State diagram – секвенцијална спецификација на однесување (се одредува однесувањето на системот и да се откриe дали има дупки во одредено однесување)
2. Activation table – комбинаторна спецификација на однесување
-Го опишува однесувањето на системот, но не дава информации за внатрешното работење на процесите

24. Process Specification (PSPEC) – за опис на сите flow model процеси на динално ниво на рафинирање.
-Вклучува наративен текст, program design language, опис на алогритамот на процесот, математички равенки, табели, дијаграми, графикони.
-Се создава мини спецификација што може да послужи како водич за дизајнирање на софтверската компонента што ќе го спроведе процесот.

25. Моделирање базирано на класи
-ИДЕНТИФИКАЦИЈА НА АНАЛИЗИРАЧКИ КЛАСИ – идентификација на класите преку испитување на изјавата за проблемот или со Grammatical Parse на use – case или преку процесирање на наративи за системот.
-Манифестирање:
1. Надворешни субјкети кои произведуваат или трошат информации
2. Работи кои се дел од доменот на информации за проблемот
3. Појави или настан кои се случуваат во контекстот на системско работење
4. Улоги (луѓе кои комуницираат со системот)
5. Организациски единици кои се релевантни за апликацијата
6. Места
7. Структури кои опишуваат класа на предмети или сродни класи
-Селекција на критериуми:
1. Задржана информација – потенцијална класа мора да се запомни за да може да функционира системот
2. Потребни услуги – збир на операции кои можат да ја променат вредноста на атрибутите
3. Повеќе атрибури
4. Чести атрибути

-СПЕЦИФИКАЦИЈА НА АТРИБУТИ – Атрибути: збир на објекти со податоци што целосно ја дефинираат класата во контекстот на провлемот.
-ДЕФИНИРАЊЕ ОПЕРАЦИИ – операциите го дефинираат однесувањето на објектот
-Категории:
1. Операции кои манипулираат со податоци на некој начин
2. Операции што вршат пресметка
3. Операции кои се распрашуваат за состојбата на објектот
4. Операции кои набудуваат предмет за појава на контролен настан
-CRC МОДЕЛИРАЊЕ (Class Responsibility collaborator (CRC)) – збирка на стандардни индекс картички кои претставуваат класи.
-Картите се поделени на три дела:
1. Врв на картичка – име на класа
2. Во тело - ги листа одговорностите лево
3. Соработник на десно
- Едноставни средства за идентификување и организирање на класите што се релевантни за системот или на производот
-Три типови класи:
1. Entity Classes /Класи на субјекти – работи што треба да се чувааат во бази и да опстојуваат во текот на целото времетраење на апликацијата.
2. Boundary class – за создавање интерфејс, како објектите се претставуваат на корисникот.
3. Controller Class: менаџирање на работни единици од почеток до крај.
-Создавање или ажурирање на предметите на ентитетот
-Презентација на граничните објекти

-Комплексна комуникација помеѓу множества на објекти
-Валидација на податоците

26. Бихејвиорално моделирање – кажува како софтверот ќе одговара на надворешни настани
-Да се креира еден модел:
1. Евалуација на сите use case за да ја разберете низата на интеракција во рамките на системот
2. Идентификувајте ги настаните
3. Создадете низа за секој use case
4. Изградете state дијаграм за системот.
5. Прегледајте го бихејвиоралниот модел за да ја проверите точноста или конзистентноста

27. Идентификување на настани со use case
-Use case - низа активности што ги вклучуваат актерите и системот.
-Настан се појавува секогаш кога системот и актерот разменуваат информации. Еден настан не е информацијата што се разменила, туку фактот дека информациите се разменети.
- Треба да се идентификува актер за секој настан.
- Треба да се забележуваат информации што се разменуваат
- Треба да се наведат какви било услови или ограничувања

28. Репрезентација на состојба (State Representation):
1. Пасивна состојба - тековниот статус на сите атрибути на објектот.
2. Активна состојба - моментална состојба на објектот бидејќи е подложна на постојана трансформација или обработка
- Мора да се случи настан за да се присили некој предмет да изврши премин од една активна состојба во друга

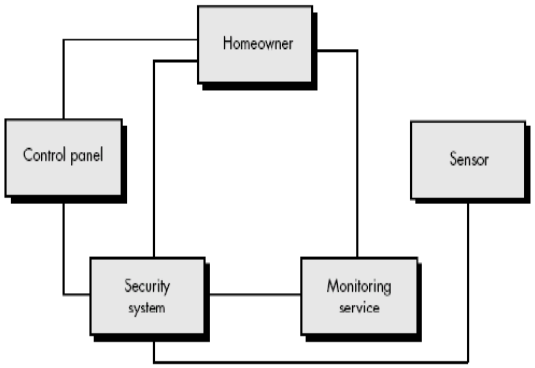
29. Дијаграм на состојба за анализирачки класи (State diagram for analysis classes) – UML дијаграм на состојба што претставуав активна состојба за секоја класа и настани што предизвикуваат промени меѓу овие активни состојби.
- Акција се случува истовремено со промената на состојбата или како низа од истата и општо вклучува една или повеќе операции на објектот.

30. Sequence diagram - означува како настаните предизвикуваат промени од објект на објект. Откако настанот ќе се идентификува, се креира дијаграм на секвенци.
- Претставува како настаните предизвикуваат проток од еден до друг предмет како функција од времето.
-Кратка верзија што репрезентира клучни класи и настани кои предизвикуваат премин од класа до класа.
-Системски објекти и настани помагаат во креирање на ефективен дизајн.

31. Механика на структурирана анализа:
1. Entity relationship diagram (ERD)
2. Data flow diagram (DFD)
3. State transition diagram (STD)

32. ERD

1. Клиентите прават листа на нештата кои ги посочува апликацискиот или бизнис процесот
2. Аналистот и клиентот дефинираат конекција меѓу објекти
3. Каде постои конекција се прават објект/врска парови
4. За секој пар се проверуваат кардналноста и модалитетот.
5. 2-4 се повторуваат за сите парови
6. Дефинирање на атрибути за секој ентитет
7. Дијаграм за односи со ентитетот се формализира и разгледува
8. 1-7 се повторуваат додека не заврши моделирањето на податоците.



Safe-Home Security System:

- Step 1: Identified things
  - homeowner
  - control panel
  - sensors
  - security system
  - monitoring service
- Step 2: one or more object/relationship pairs are identified for each connection.
- Step 3: **security system** *monitors* **sensor**
  - **security system** *enables/disables* **sensor**
  - **security system** *tests* **sensor**
  - **security system** *programs* **sensor**
- Step 4: Cardinality and modality
  - The cardinality between **security system** and **sensor** is one to many.
  - Modality of **security system** (mandatory) and **sensor**(mandatory)
- Step 5: repeat 2 to 4 for all objects.
- Step 6: Each object is studied to determine its attributes.
  - For example: **sensor** -- sensor type, internal identification number, zone location, and alarm level.

33. Речник на податоци (Data dictionary) – организиран список на сите елементи на податоци што се релевантни за системот, со прецизни дефиниции така што и корисникот и системскиот аналитичат ќе можат да имаат заедничка разбирање за влезовите, излезите, компонентите на складиштата и посредни пресметки.
-секогаш е имплементиран како дел од CASE "structured analysis and design tool.“
-Повеќето содржат:

1. Име – примарно име на податочни или контролни ставки
2. Алијас – други имиња користени за првиот влез
3. Каде/како користен – листа процеси кои ги користат податоците и како ги користат
4. Опис на содржина – нотација за содржината
5. Додатна информација – тип на податок, вредност на проект, ограничувања

The notation used to develop a content description is noted in the following table:

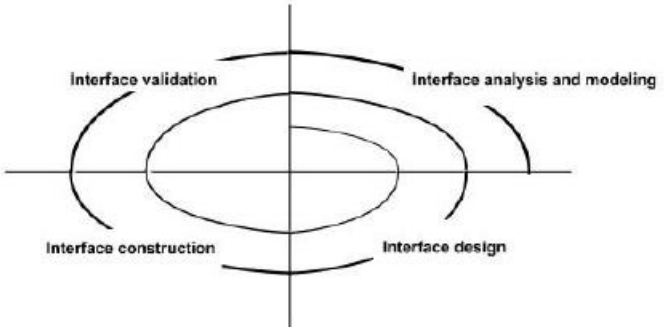
Data	Construct Notation	Meaning
	=	is composed of
Sequence	+	and
Selection	[|]	either-or
Repetition	$\{\}^n$	*n* repetitions of
	( )	optional data
	* ... *	delimits comments

9. ДИЗАЈН НА КОРИСНИЧКИ ИНТЕРФЕЈСИ

1. Типични грешки во дизајнот
   -недостиг на конзистентност
   -премногу меморизација
   -нема упатства/помош
   -нема контекстна чувствителност
   -слаб одговор
   -лачно/непријателски

2. Златни правила:
   -Постави го корисникот под контрола
      1. Дефинирајте интерактивни модули на начин кој не го тера корисникот на неопходни или несакани акции
      2. Обезбедете флексибилна интеракција
      3. Интеракцијата на корисникот да може да биде прекината
      4. Рационализирајте ја интеракцијата на нивоа како напредуваат вештините и овозможете прилагодување на интеракцијата
      5. Прикријте ги техничките работи од надворешните корисници
      6. Дизајн за директна интеракција со предмети што се појавуваат на екранот
   -Намалете го оптоварувањето на меморијата на корисникот
      1. Намалете ја побарувачката за краткорочна меморија
      2. Воспоставете значајни стандардни вредности
      3. Дефинирајте интиутивни кратенки
      4. Визуелниот распоред на интерфејсот треба да се заоснова на метафора во реалниот свет
      5. Откријте ги информациите на прогресивен начин
   -Направете го интерфејсот конзистентен
      1. Дозволи им на корисниците да постават тековна задача во значаен контекст
      2. Одржувајте конзистентност низ фамилија апликации
      3. Ако минатите модели ги исполнуваат очекувањата на корисниците, не правете промена доколку нема неопходна промена за тоа

3. Модели на дизајн на кориснички интерфејси
-Кориснички модел – профил на сите крајни корисници на системот
-Дизајн модел – реализација на дизајнот на корисничкиот модел
-Ментален модел (системска перцепција) – менталната замисла на корисникот за што е тоа интерфејс
-Имплементациски модел – интерфејсот look and feel заедно со информација за поддршка која ја опишува синатаксата и семантиката на интерфејсот



4. Интерфејс анализа значи разбирање на:

-Крајните корисници кои ќе комуницираат со системот преку интерфејсот

-Задачите кои корисниците мора да ги извршат за да ја завршат нивната работа

-Содржината која е презентирана како дел од интерфејсот

-Околината во која овие задачи ќе бидат спроведени

5. Чекори на дизајн на интерфејс – користејќи ја информацијата која е развиена за време на анализата на интерфејсот, се дефинираат операциите на интерфејсот.

- Дефинирај настани кои ќе предизвикаат да се промени состојбата на корисничкиот интерфејс. Моделирај го ова однесување

-Опишете ја секоја состојба на интерфејсот како што ќе му излгеда на корисникот

-Наведете како корисникот ја толкува состојбата на системот од информациите дадени преку интерфејсот

6. Дизајнерски проблеми

-Време на одговор

-Објекти за помош

-Справување со грешки

-Мени и обележување на командите

-Достапност на апликацијата

-Интернационализација

7. Interface Design Workflow-I

-Преглед на информациите содржани во моделот за анализа и рафинирајте како што е потребно

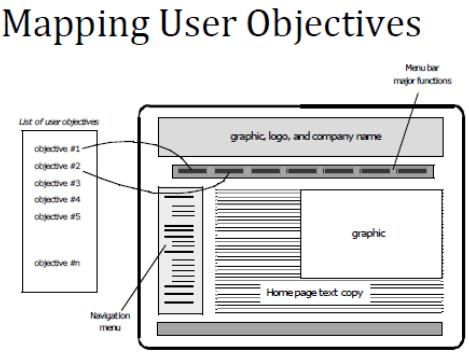
-Развивање на груба скица на изгледот на интерфејсот

-Мапирајте ги целите на корисниците во специфични активности на интерфејс.

-Дефинирање збир задачи на корисниците кои се поврзани со секоја акција.

-Слики за секоја интерфејс акција

-Рафинирајте го изгледот на интерфејсот и storyborads користејќи влез од естетски дизајн.



8. Interface Design Workflow-II

-Идентификувајте ги потребните објекти за кориснички интерфејс за спроведување на интерфејсот

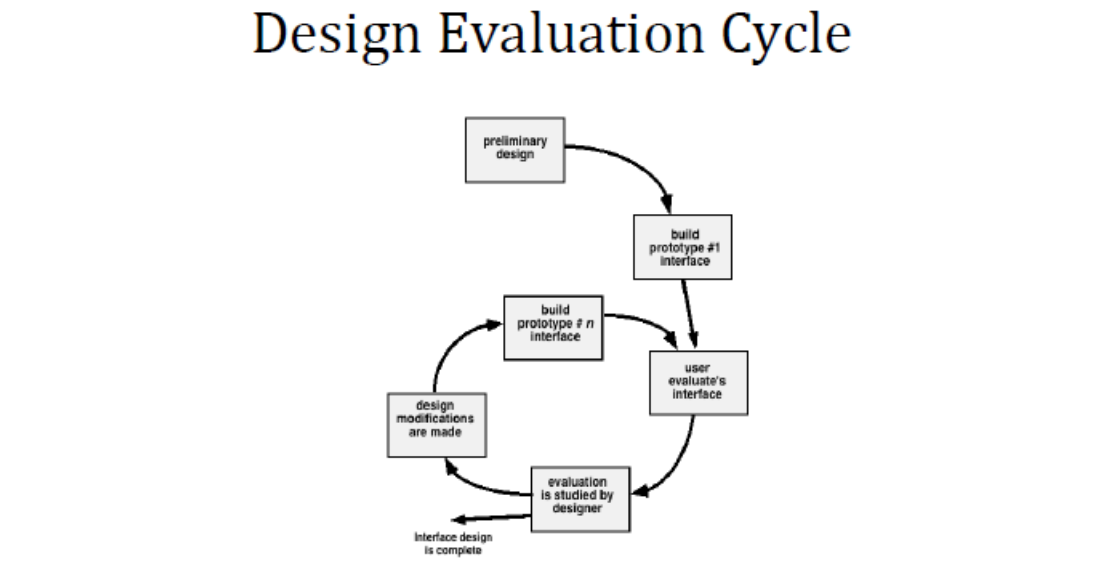
-Развијте процесно претставување на корисничката интеракција со интерфејсот

-Развијте бихејвориална престава за интерфејсот

-Опишете го изгледот на интерфејсот за секоја состојба

-Рафинирајте го и разгледајте го моделот на дизајн на интерфејсот

9.Естетски дизајн
-Не се плаши од белиот простор
-Нагласи ја содржината
-Организирај ги елементите од горе лево кон долу десно
-Групна навигација, содржина и функција географски на содржината
-Не ги продолжувајте неподвижните елементи со лента за движење
-Разгледајте ја резолуцијата и големината на прозорецот на прелистувачот при дизајнирање на изгледот.

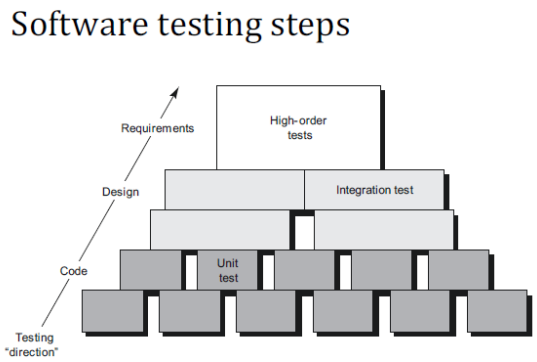


11. СТРАТЕГИИ ЗА ТЕСТИРАЊЕ НА СОФТВЕР

1. Софтверско тестирање – процес на извршување на програма со специфична намена да се најдат грешки пред доставата до крајниот корисник. Тестирањето има за цел за да покаже дека програмата го прави тоа што е наменета.
-Работата на осигурување на квалитетот на софтверот е да се осигура дека тестирањето е правилно испланирано и ефикасно спроведено така што има најголема веројатност за постигнување на својата примарна цел.
-Тестирањето честопати вклучува повеќе напори во проектот отколку било која друга акција на софтверското инженерство

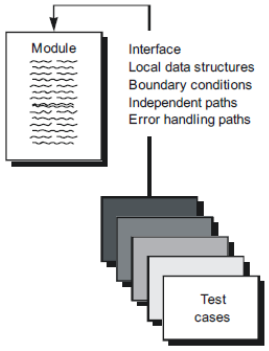
2. Стратегиски пристап – за ефективно тестирање потребно е ефективно спроведување на технички предлефи.
-Тестирањето започнува на ниво на компонента и работи нандвор кон интеграција на целиот компјутерски базиран систем.
-Тестирањето го спроведува развивачот на софтверот и независна тест група
-Тестирање и дебагирање е различно, но секоја стратегија за тестирање содржи дебагирање.

3. Верификација – збир на активности кој осигуруваат дека софтверот коректно имплементира специфична функција.
Дали го градиме правилно производот?
-Валидација – збир задачи што осигурува дека изградениот софтвер може да се следи според барањата на клиентите.
Дали го градиме правилниот производ?

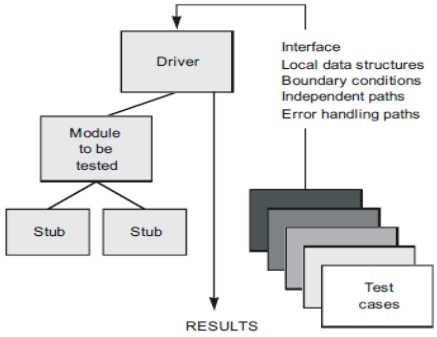


4. Стратешки проблеми
-Наведете ги барањата на производот на квантитативен начин долго пред да започне тестирањето.
-Разбирање на корисниците на софтверот и да се развие профил за секоја категорија на корисници.
-Развијте план за тестирање кој го потенцира “rapid cycle testing.”
-Изгредете стабилен софтвер дизајниран да се тестира
-Користете ефективни технички прегледи како филтер пред тестирање
-Спроведување на технички прегледи за проценка на стратегијата за тестирање како и самите тест случаеви
-Развијте пристап за континуирано подобрување на процесот на тестирање.

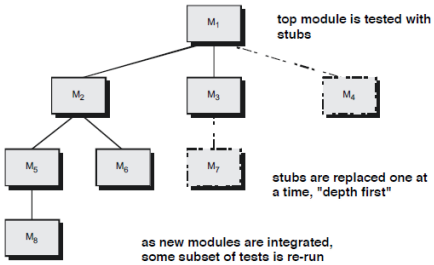
Unit Testing



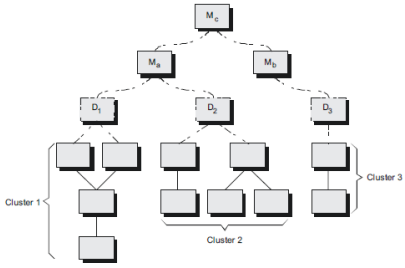
Unit Test Environment



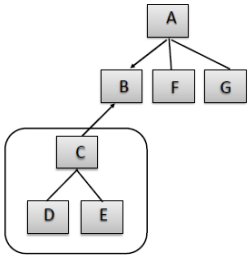
Top Down Integration



Bottom-up Integration



Sandwich Testing



5. Регресивно тестирање – повторно извршување на подмножество тестови што се веќе спроведени за да се обезбеди дека промените не предизвикале несакани ефекти.
-Ересивното тестирање помага да се осигураме дека промените не воведуваат ненамерно однесување или пак дополнителни грешки
-Може да се спроведе мануелно, со повторно извршување на подможеството на сите тест случаеви или пак автоматско со користење на алатки за снимање/репродукција.

6. Smoke Testing – чест пристап за создавање на “daily builds” за софтверски производ.
-Чекори:

1.Софтверки компоненти кои се преведени во код се интегрирани во build (сите податочни датотеки, библиотеки, модули, инженерски компоненти)

2. Низа од тестирања се дизајнирани да ги открие грешките кои ќе овозможат правилно извршување на функцијата на build-от

3. Build-от е интегриран со други builds и целиот производ секојдневно извршува smoke testing. Пристапот за интеграција може да биде од горе надолу или од доле нагоре.

-Придобивки:

1. Ризикот за интеграција е минимизиран
2. Квалитетот на крајниот производ е подобрен
3. Дијагностицирањето и корекцијата на грешки се поедноставни
4. Напредокот е полесен за проценка.

7. Документација за тест на интеграција – општ план за интеграција на софтверот и опис на специфични тестови е документирано во тест спеифкација.
-Овој производ вклучува план за тестирање и постапка за тестирање и станува дел од конфигурацијата на софтверот.
-Критериуми на тест фази:

1. Интегритет на интерфејс
2. Функционална валидност
3. Информативна содржина
4. Изведба

8. Објектно-ориентирано тестирање – започнува со проценка на исправноста и конзистентост на моделите за анализа и дизајн.
-Промена на стратегијата за тестирање
-Дизајн на случај на тестови се базира на конвенционални методи, но опфаќа и посебни карактеристики.
-Стратегија на тестирање:

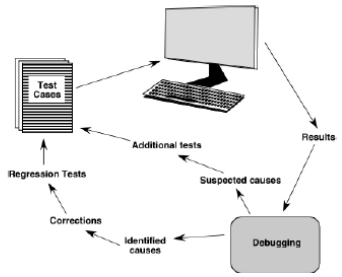
1. Класното тестирање е еквивалентно на тестирање на единци
2. Интеграцијата применува три различни стратегии:
    1. Тестирање базирано на нишка - го интегрира множеството на класи потребни за да одговорат за еден влез или настан
    2. Тестирање базирано на корисност – интегрира класи потребни да се одговори на use case.
    3. Тестирање на кластери – интегрира класи потребни да демонстрираан одредена соработка.

9. Тестирање на валидација – почнува при кулминацијата на интеграцијата на тестирање.
-Валидацијата е успешна кога софтверот функционира на начин очекуван од страна на клиентот (фокус на акции видливи за корисникот и излез разбирлив за корисникот)
-Алфа и бета тестирање

10. Тестирање на системот

1. Тестирање на системот – фокус на системската интеграција
2. Recovery testing – го тера системот да падне на различни начини и потоа потврдува дека обновувањето е правилно извршено
3. Security testing
4. Stress testing – тестирање што бара ненормална количина ресурси, фрекфенција или волумен.
5. Performance Testing
6. Deployment Testing

The Debugging Process



11. Симптоми и причини – може да бидат географски одвоени
- Симпотмите може да исчезнат откако ќе се реши одреден проблем
-Причината може да биде поради комбинација на не грешки
-Може да е тешко точно да се репродуцираат состојби на велз
-Симптомите може да се предивикани од човечки грешки кои не можат лесно да се откријат.
-Симптомите може да се резултат на временски проблеми наместо процесирачки проблеми.

12. Техники на дебагирање – brute force/testing, backtracking, induction, deduction

13. Поправање на грешки
-Дали причината за грешката е репродуцирана од друг дел на програмата?
-Која следна грешка може да се појави со корекцијата која ја правам
-Што можевме да направиме да ја спречиме појавата на грешката однапред

12. ТЕХНИКИ ЗА ТЕСТИРАЊЕ НА СОФТВЕР

1. Тесирање – софтверот мора да биде тестиран за да открие и поправи што е можно повеќе грешки пред доставувањето на клиентот.
-Цел: Да се дизајнираат низа на тест случаеви кои имаат голема веројатност да пронајдат грешки

2. Кој го прави тестирањето
-Софтверските инженери – за време на раните фази на тестирање, ги изведуваат сите тестови
-Програмери – кодот
-Специјалисти – при напредокот на процесот на тестирање
-Клиент – секој пат кога програмата е извршена, клиентот го тестира

3. Со цел да се најдат што поголем број на грешки за минимален напор и време, тестовите мора да се спроведуваат систематски и тест случаевите мора да се бидат дизајнирани со користење на дисциплинирани техники.

4. Софтверот треба да се тестира од две различни перспективи;
-“white box” тест случаеви – внатрешната логика на програмата
-“black box” – софтверските барања

5. Секундарна корист – тестирањето покажува дека функциите на софтверот работат според дадената спецификација. Тестирањето не може да покаже отсуство на грешки и дефекти, може да го покаже само дека тие се присутни

6. Принципи на тестирање
1. Сите тестови треба да бидат следливи за барањата на клиентот
2. Тестовите треба да се планираат долго пред да започне тестирањето
3. Парето принципот се применува за тестирање на софтвер
4. Да се започне тестирање на мало, а потоа да се зголемува кон тестирање на големо
5. Исцрпно тестирање не е можно

7. Тестабилност – колку лесно може да се тестира една компјутерска програма
1. Оперативност – колку подобро работи, поефикасно може да се тестира. Кога се дизајнира системот се размислува на квалитетот.
2. Набљудливост – она што се гледа е она што се тестира. Состојбата на системот и променливите се видливи за време на извршување. Основниот код е достапен
*Грешните излези лесно може да се идентификуваат.
*Внатрешните грешки автоматски се детектираат и пријавуваат
3. Контрола – колку подобро може да го конторлираме софтверот, толку повеќе тестирањето може да се автоматизира и оптиматизира.
*Сите можни излези да може да се генерираат преку одредени влезни комбинации, В/И формати се конзистентни и структурирани
*Тестовите можат лесно да се одредат, автоматизираат и репродуцираат
4. Разложливост – со конторлирање на обемот на тестирање, можеме побрзо да си изолираме проблемите и да изведеме попаметно ново тестирање.
*Софтверскиот систем е изграден од независни модули што можат да се тестираат независно.

5. Едноставност – колку помалку има за тестирање, толку побрзо може да го тестираме.
*Функционална едноставност – збир на минимум карактеристики кои се неопходни за исполнување на барањата
*Структурна едноставност – архитектурата е модализирана да го ограничи размножувањето на грешките
*Едноставност на код – усвоен стандард за кодирање за леснотија на инспекција и одржување

6. Стабилност – колку помалку промени, помалку пречки во тестирањето
*Софтверот добро се опоравува од неуспеси

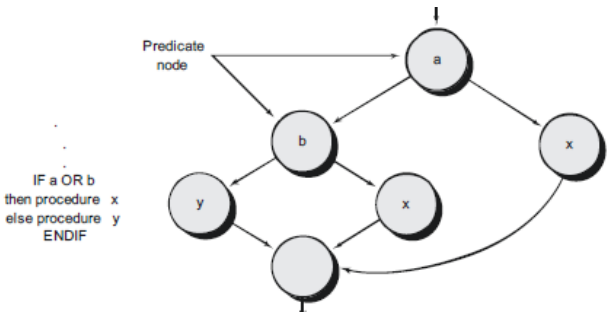
7. Разбирливост – колку повеќе информации имаме, толку попаметно ќе тестираме
*Дизајнот на архитектурата и зависностите меѓу внатрешни, надворешни и споделените компоненти се добро разбрани.
*Техничката документација веднаш достапна, добро организирана, специфична, детална и точна

8. Добар тест – висока веројатност за наоѓање на грешка, не треба да биде ниту премногу едноставен ниту премногу сложен

9. Внатрешни и надворешни погледи – секој инженерски производ може да биде тестиран на два начини:
- Ако се знае специфичната функција за која е дизајниран произоводот, тестирањето може да се направи така да се проверли дали секоја функција е целосно оперативна
- Ако се знаат внатрешните работи на функцијата, тестовите може да се спроведат за да се провери дали внатрешните операции работат според дадените спецификации

10. Basis path testing е white-box техника за тестирање – изведување на мера за логичка комплексност на процедуралниот дизајн. Користење на истата во дизајнирањето на оснвоното множество на извршни патеки кои гарантираат извршување на секоја наредба во програмата најмалку еднаш при тестирањето

11. Flow Graph го прикажува контролниот логичкиот тек на програмта во форма на граф. Секоја структура во процедуралниот опис се заменува со соодветен симбол.
-Сложена состојба – еден или повеќе булеан оператори се присутни во условна изјава



12. Цикломатска сложеност – софтверска метрика што обезбедува квантитативна мерка на логичната сложеност на програмата.
*Од аспект на тестирањето на основните патеки, вредноста на цикломатската комплексност го дефинира бројот на независни патеки во програмта и ја дава горната граница на тестови кои треа да се изведат за да се обезбеди секоја наредба да биде

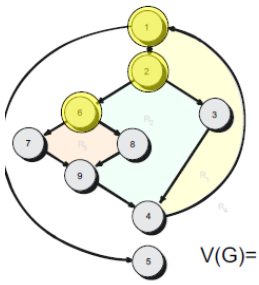
извршена барем еднаш.
*Независна патека е произволната патека на извршување низ програмата која воведува извршување на барем еден нов процесирачки израз или условно гранење (поминува низ барем една нова врска од графот)

13. Пресметување на цикломатската сложеност – колку е поголема, толку поголема можност за грешки

1. Бројот на регионите на графот на тек соодветствува на цикломатската комплексност

2. Цикломатската комплексност V(G) на графот G е дефинирана како:
V(G) = E–N+ 2, каде E е бројот на врски (рабови) во графот, а N број на јазли

3. V(G) = P + 1, каде P е бројот на предикатни јазли во графот



1. Number of regions
2. $V(G) = E - N + 2$
3. $V(G) = P + 1$

Number of regions = 4

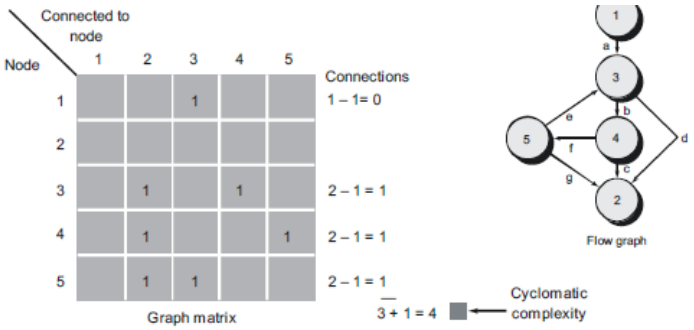
V(G)= 11 edges – 9 nodes + 2 = 4

V(G)= 3 predicate nodes + 1 = 4

14. Deriving Test Cases – користејќи го дизајнот или кодот како основа да се нацрта соодветниот flow graph -> одредување на цикломатска комплексност на графот -> одредеување на основен сет на независни патеки -> подготовка на тест случаеви кои ќе принудат извршување на секоја патека од основното множество.

*Примери: слајд 34 - 38

15. Матрица на граф – квадратна матрица со онолку редови колку што има јазли во градот. Келиите соодвествуваат на врски меѓу јазлите.
-Со воведување на врски меѓу јазлите, матрицата на граф може да се стане моќна алтка за проценка на програмската контролна структура за време на тестирањето.
-1 – постои врска меѓу јазлите
-0 – не постои врска.



16. Condition Testing – метод за дизај на тестови кои ги испитуваат логичките изрази во програмски модул – булова променлива или релациски израз

E1<relational-operator> E2,

каде E1и E2се аритметички изрази, а операторот е еден од: <, ≤, =, ≠,> или ≥

-Сложен израз е составен од два или повеќе изрази, загради и булови операции (AND, OR, NOT)

-Ако условот е погрешен, тогаш барем една негова компонента е погрешна

17. Branch Testing – ја извршува секоја гранка присутна во модулот на програмата барем еднаш за откривање на сите грешки присутни во branch-от.

18. Domain Testing – изведување на 3 до 4 теста за релационен израз E1<relational-operator> E2

-Потребни се три теста за да се покријат случаите кога E1<, > или = со E2

-За Булови изрази со n променливи потребни се 2^n теста

19. Branch and Relational Operator (BRO) Testing – ги тестира гранките присутни во модулот на програмта со користење на ограничувања на состојбите.

20. Data Flow Testing – избор на патеки за тестирање на основа на локациите на дефинициите и употребата на променливите во програмата.

-За изразот со реден број S:

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

- Дефиницијата на променливата X во изразот S е жива во изразот S’ ако постои патека од изразот S до S’ во која нема друга дефиниција на X

-Definition-use (DU) chain на променливата X е во форма [X, S, S’] каде S и S’се броеви на изрази, X е во DEF(S) и во USE(S’) и дефиницијата на X е жива во S’.

```
proc x
   B1;
    do while C1
    if C2
       then
         if C4
              then B4;
              else B5;
         endif;
       else
         if C3
              then B2;
              else B3;
         endif;
       endif;
    enddo;
    B6;
end proc;
```

нека променливата X е дефинирана во последниот израз во блоковите B1, B2, B3, B4 и B5,

и нека е употребена во првиот израз во блоковите B2, B3, B4, B5 и B6.

DU – извршување на најкусата патека од секој B_i, $0 < i \leq 5$ до секој B_j, $1 < j \leq 6$.

(ваквите патеки ја покриваат и употребата на X во условите C1, C2, C3 и C4)

Иако постојат 25 можни DU вериги, потребни се само 5 патеки за да се покријат сите вериги (ако се повторува, циклусот ги опфаќа)

21. Loop Testing – техника за тестирање која се фокусира само на валидноста на циклусите.
-Класи на циклуси:

1. Обични (simple loops)

ТЕСТОВИ:

1. Прескокни го целосно циклусот
2. Едно изминување низ циклусот
3. Две изминувања
4. m изминувања низ циклусот (m<n)
5. n-1, n, n+1 изминувања низ циклусот

2. Нанижани (concatenated loops)

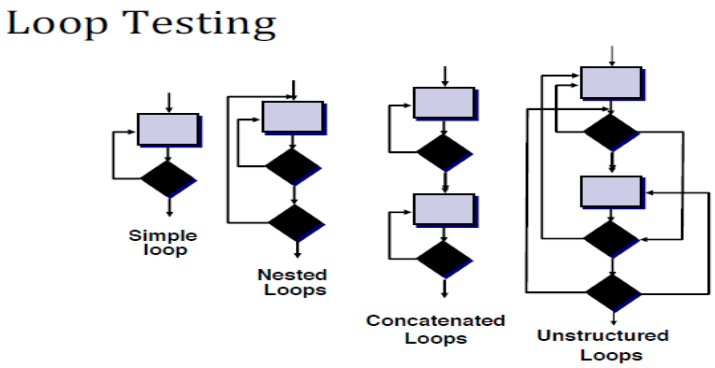
-Тестирање како обични ако се независни
-Тестирање како вгнездени ако се зависни

3. Вгнездени (nested loops)

ТЕСТОВИ:

1. Почни од највнатрешниот циклус, постави ги сите други на минимална вредност
2. Изведи ги тестовите за едноставен циклус за највнатрешниот, додека надворешните циклуси стојат на минималната вредност. Додади дополнителни тестови за надвор од опсег и исклучителни вредности
3. Движи се нанадвор, тестирајќи го лседниот надворешен циклус
4. Продолжи додека сите циклуси не се тестирани

4. Неструктурирани (unstructured loops) – секогаш кога е можно овие циклуси треба да се преработат во структурен дизајн



22. Black Box Testing (behavioral testing) – се фокусира на функционалните барања на софтверот.
-Изведување на сите множества на влезови кои целосно би ги провериле функционалните барања на програмата
-Комплементарен приод во однос на White box testing

23. Black Box Testing се обидува да пронајде грешки кај следниве категории

1. Погрешна или функција која недостасува
2. Грешки во интерфејсот
3. Грешки во податочните структури или базите на податоци
4. Грешки во однесување или перформансите
5. Грешки при иницијализација и терминирање

24. Со примена на Black Box техниките за тестирање се изведуваат тестови кои го намалуваат за повеќе од еден бројот на дополнителни тествои кои треба да ги направиме и ни кажуваат нешто за присуство или отсуство на комплетни класи на грешки.

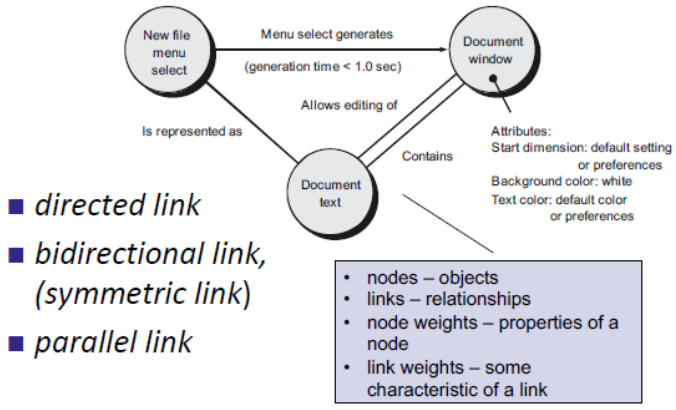
25. Graph-Based Testing Methods

-Прв чекор е да се разберат објектите кои се моделирани во софтверот и релациите кои ги поврзуваат овие објекти.

-Следен чекор е дефинирање на низа од тестови кои потврдуваат дека сите предмети имаат очекувана врска еден со друг

-Тестирањето на софтверот почнува со креирање на граф на важни објектни и нивните врски

-Изготвување на серија тестови што ќе го опфата графот така што секој објет и врска ќе се остварува а грешките ќе се откријат.



Граф на објекти

26. Equivalence Partitioning – го дели влезниот домен на програмта на класи на податоци од кои се изведуваат тестовите

-Идеалниот тест открива цела класа на грешки

-Евалуација на еквивалентни класи за влезни услови

-Секоја еквивалентна класа претставува множество валидни или невалидни влезови

27. Equivalence класи

1. Ако влезниот услов дефинира опсег, се дефинираат една валидна и две невалидни еквивалентни класи

2.Ако влезниот услов претставува специфична вредност, се дефинираат една валидна и две невалидни еквивалентни класи

3.Ако влезниот услов дефинира елемент од множество, се дефинираат една валидна и една невалидна еквивалентна класа

4.Ако влезниот услов е булова логичка вредност, се дефинираат една валидна и една невалидна еквивалентна класа

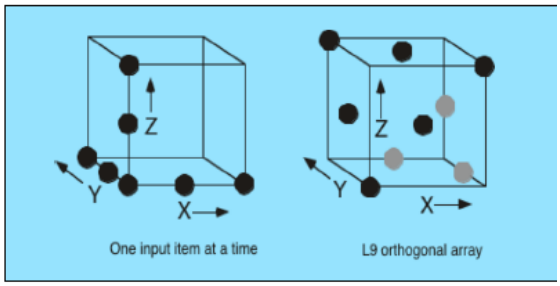
28. Boundary Value Analysis – техника за тестирање комплементарна на еквивалентно партиционирање.
-BVA води кон креирање на тестови кои оперираат на работ на доменот.

- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

29. Comparison Testing – во ситуации во кои веродостојноста на софтверот е апсолутно критичен.
-СИ тимови развиваат независни верзии на апликација со користење на иста спецификација.
-Секоја верзија може да се тестира со истиот тест податоци за да се обезбеди дека сите обезбедуваат идентичен излез.
-Потоа сите верзии се извршуваат паралелно со споредба во реално време на резултати за да се обезбеди конзистентност.

30. Orthogonal Array Testing (OAT) - се користи кога бројот на влезните параметри е мал и вредностите што секоја од параметрите што можат да ги преземат се јасно ограничени.



31. One input item at a time – една влезна ставка во единица време може да се разликуваат во низа по секоја влезна оска.
-Релативно ограничено покривање на влезниот домен

32. L9 orthogonal array
- Откријте и изолирајте ги сите единечни грешки
-Откријте ги сите двојни грешки во модемот.
-Мултимодни грешки.

33. Model-Based Testing чекори:

1. Анализирајте постоечки модел
2. Наведете ги влезовите што ќе го натераат софтверот да направи премин од една состојба во друга.
3. Забележете ги очекуваните исходи како софтверот прави премин од состојба во состојба.
4. Изврши тест случаеви
5. Споредете ги реалните и очекуваните резултати и преземети корективни активности како што се бара.

34. Тестирање за специјализирани опкружувања, архитектури и апликации

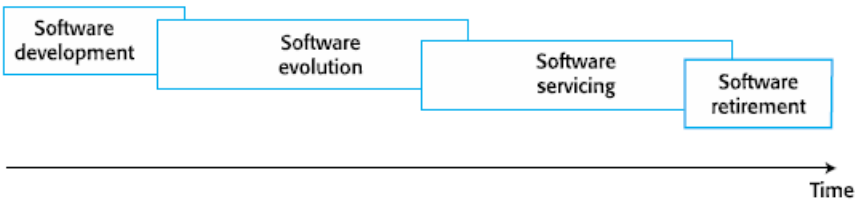
- Testing GUIs
- Testing of Client-Server Architectures
- Testing Documentation and Help Facilities
- Testing for Real-Time Systems
    1. Task testing
    2. Behavioral testing
    3. Intertask testing
    4. System testing

13. ЕВОЛУЦИЈА НА СОФТВЕРОТ

1. Промена на софтверот – неизбежна
-Нови барања се појавуваат кога се користи софтверот
-Бизнис околината се менува
-Грешките мора да се поправат
-Нови компјутери и опрема
-Перформансот или доверливоста може да се подобрат
*Клучен проблем за сите организација е имплементација и управување со промените на постојните софтверски системи.

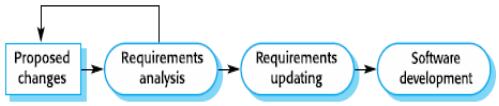
2. Важноста на еволуцијата – организациите прават огромни инвестиции во нивните софтверски системи. За да може да се одржи вредноста на овие средства во бизнисот тие мора да се променуваат и ажурираат.
-Поголемиот дел од софтверскиот буџет на компаниите е наменет за промена и развој на постојниот софтвер, наместо за развој на нов.

3. Еволуција и сервисирање
-Еволуција – во неговата оперативна употреба и се развива како со добивање и имплементирање на нови барања во системот
-Сервисирање – единствените измени се они што се потребни да се одржи оперативен (поправа на грешки), не се додаваат нови функционалности
-Phase-out – софтверот се уште може да се користи но нема нови промени направени.



4. Процесите на еволуција зависат од:
    1. Типот на софтверот кој се одржува
    2. Процесот на развој кој се корсити
    3. Вештините и искуството на вклучените луѓе
-Клучни за системската еволуција се барањата за промена.
-Идентификацијата и еволуцијата на промените продолжува во текот на целиот животен век на системот.

5. Имплементација на промените
-Итерација на процесот на равој каде ревизиите на системот се дизајнирани, имплементирани и тестирани.
-Во фаза на разбирање на програмта, треба да се разбере како е структурирана програмата, како се обезбедува функционалноста и како промената влијае врз програмата.

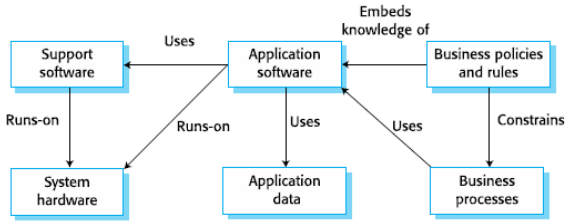


6. Барања за итни промени – овие промени може да се имплементираат без минување на сите чекори на процесот на софтверско инженерство
Пр. Сериозна системска грешка која мора да биде поправена.
Промена во околината на системот предизвикува несакани ефекти
Бизнис промени кои бараат брз одговор

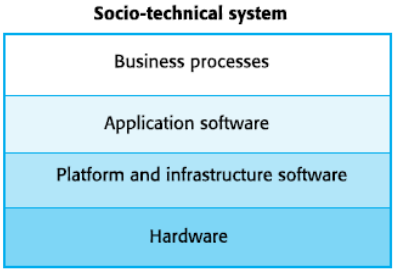
7. Агилни методи и еволуција -
-Агилните методи се заоснованат на постепен развој, така што преминот од развој во развој е непречен
-Еволуцијата е продолжение на процесот на развој заосновано на чести изданија на системот

8. Handover problems
1. Тимот за развој користел агилен пристап, но еволуцискиот тим не е запознаен со агилните методи и преферираат пристап базиран на план. Затоа, тимот за еволуција може да очекува детална документација за да ја поддржи еволуцијата, но таа не е вклучена во агилните процеси.
2. Тимпот за развој користел пристап базиран на план, но еволуцискиот тим користел агилен методи. Еволуцискиот тим може да треба да почне од 0 со кодот, и системот може да не е толку едноставен како со агилниот развој.

9. Наследни системи – постари системи кои се потпираат на јазиците и технологијата кои веќе не се користат за развој на нови системи.
-Наследниот софтвер може да зависи од постариот хардвер
-Овие системи не се само софтверски системи, туку пошироки социо-технички системи кои вклучуваат хардвер, софтвер, библиотеки и други придружни софтверски и деловни процеси.



10. Компоненти на наследни системи
-Системски хардвер – овие системи може да се напишани за хардвер кој може не е веќе достапен
-Софтвер за поддршка – се потпира на голем број на софтвер за поддршка, кој може да е застарен или неподдржан
-Апликациски софтвер
-Апликациски податоци – може да бидат неконзистентни, двојни или да се чуваат во различни бази на податоци.
-Бизнис процеси – за да се постигне одредена бизнис цел
-Бизнис политики и правила–Правила за спроведување на бизнисот и ограничувања



11. Промена на наследни системи – може да е ризично и скапо, затоа бизнисите продолжуваат да ги користат системите.
-Нема конзистентен стил на програмирање
-Користење на застарени програмски јазици
-Несоодветна системска документација
-Деградација на системската стурктура

12. Управување со наследни системи – организациите кои се потпираат на наследните системи мора да одберат стратегија за развој на ваквите системи
-Целосно отстранување на системот и модификација на деловните процеси,
-константно одржување на системот,
-трансформирајте го системот со реинженираење со цел да се подобри неговата одржливост или
-заменување на еден систем со друг.
*Која од овие стратегии ќе се одбере зависи од квалитетот на системот и неговата деловна вредност.

13. Категории на наследните системи
1. Лош квалитет, мала бизнис вредност – отстранување на системот
2. Лош квалитет, голема бизнис вредност – Реинженирање или замена на системот
3. Добар квалитет, мала бизнис вредност – замена со COTS, целосно замена или одржување
4. Добар квалитет, голема бизнис вредност – продолжување на операциите со нормално одржување на системот.

14. Проценка на бизнис вредноста – проценката треба да земе предвид различни гледишта од страна на крајните корисници, клиенитите, линиските менаџери, ИТ менаџери, раководители, интервју со стеикхолдери итн.
-Issues:
1. Искористеноста на системот – ако системите се користат повремено, или од мал број на луѓе, тие имаат мала бизнис вредност
2. Поддржаните бизнис процеси – мала бизнис вредност, ако принудува употреба на неефикасни бизнис процеси
3. Сигурност на системот – ако системот не е сигурен и проблемите директно влијаат на клиентите, мала бизнис вредност
4. Излезите на системот – ако бизнисот зависи од аутпутите на системот, тогаш тој систем има висока бизнис вредност.

15. Проценка на квалитетот на системот
1. Проценка на бизнис процесот – колку бизнис процесот ги подржува целите на бизнисот
2. Проценка на околината – колку ефективна е околината на системот и колку е скапа таа да се оддржи?
3. Проценка на апликацијата – каков е квалитетот на апликацискиот софтверски систем

-Користете пристап ориентиран кон гледишта (viewpoints) и побарајте одговор од системските стеикхолдери

1. Дали е дефиниран модел на процеси и дали тој се следи?
2. Дали различни делови на организацијатња користат различни процеси за иста функција?
3. Како се прилагодуваат процесите?
4. Кои се односите со други бизнис процеси и дали тие се неопходни?
5. Дали процесот е ефективно поддржан од софтверот за апликации со наследство?

16. Мерење на системот – може да се собираат квантитативни податоци за да се направи проценка на квалитетот на апликацискиот систем
-Колку повеќе барања за промена на системот – поголема акумулирана вредност, помал квалитет на системот
-Колку повеќе интерфејси се користат поверојатно е дека ќе има недоследности
-Колку повеќе се зголемува обемот на бази на податоци, се згоемуваат недоследностите и грешките во тие податоци
-Чистење стари податоци е скап и процес кој трае долго време.

17. Одржување на софтверот – модификација на програмата откако таа ќе биде пуштена во употреба. Терминот најмногу се корсти за промена на custom софтвер, додека генеричките софтверски производи еволуираат со нови верзии.
-Промените не вклучуваат огромни промени во архитектурата на системот, туку модификација на постоечките компоненти и додавање на нови компоненти на системот.

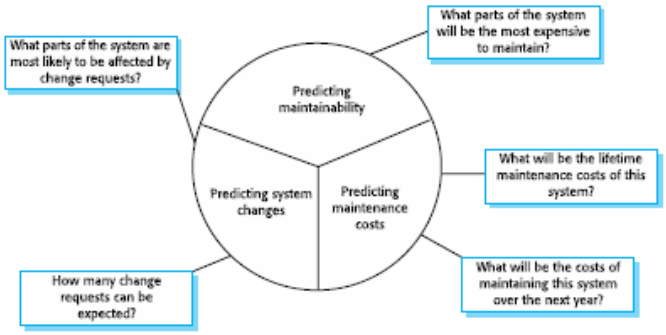
18. Видови на одржувања (ISO/IEC 14764:2006)
-Корективно одржување: реактивна модификација на софтверскиот продукт извршен откако е пуштен во употреба, за да се поправат откриените проблеми.
-Дистрибуција на напор за одржување - околу 21%
12.4% дебагирање во итни случаеви
9.3% рутинско дебагирање
-Адаптивно одржување: модификација по пуштање на продуктот во употреба за да се задржи неговата употребливост во сменетата или при промена на околината.
-Дистрибуција на напор за одржување – околу 25%
17.3% адаптација на податочната околина
6.2% хардверски промени или промени во оперативниот систем
-Перфективно одржување: измена по пуштање во употреба за подобрување на перформансите или на одржливоста
-Дистрибуција на напор за одржување – околу 50%
41.7% додатоци за корисниците
5.5% за подобрување на документацијата
3.4% друго
-Превентивно одржување: измена по пуштање во употреба за детекција и корекција на латентни грешки во софтверот, пред да станат ефективни грешки.
-Дистрибуција на напор за одржување - 4% за подобрување на ефективност на код

19. Видови на одржувања
-Поправка на грешки – промена на системот за да се отстранат грешки/слабости и корекција на недостатоци на начин што ги задоволува барањата на системот (24%)
-Адаптација на околината – одржување за адаптација на софтверот во друга извршна околина (19%)
-Дополнување на функционалноста и модификација – модификација на системот за задоволување на нови барања (58%)

20. Трошоци за одржување – често се поголеми од трошоците за развивање на софтвер (од 2 до 100 пати поголеми трошоци, во зависност од апликацијата)
-Предизвикани од технички и нетехнички фактори
-Како се одржвуа софтверот, овие трошоци се зголемуваат
-Стареењето на софтверот може да има високи support трошоци – стари јазици
-Трошоците за додавање нови карактеристики во системот за време на одржувањето се поголеми, споредебено со додавање на истите карактеристики при развивањето.

1. Нов тим треба да како програмите се одржуваат
2. Двоењето на одржувањето и развојот значи дека нема стимул тимот да напише одржлив софтвер
3. Програмата за одржување често е непопуларна
4. Персоналот за одржување често нема доволно искуство и има ограничено знаење за доменот.

21. Предвидувања за одржување – се занимаваат со проценката кои делови од системот можат да предизвикаат проблеми и имаат големи трошоци за одржување.
-Прифаќањето на промената зависи од одржливоста на компонентите погодени од компонентата.
-Спроведувањето на промени го деградира системот и ја намалува неговата одржливост.
-Трошоците за одржување зависат од бројот на промени, а трошоците за промена зависат од одржливоста.



22. Предвидување на промените – бара разбирање на односите меѓу системот и неговата околина.
-Coupled системите бараат промени секогаш кога се менува околината.
-Фактори кои влијаат на оваа врска се:

1. Бројот и комплексноста на интерфејсите во системот
2. Бројот на инхерентни непостојани барања
3. Бизнис процесите каде се користи тој систем.

23. Метрики на сложеност
-Предвидувањата за одржливост можат да се направат со проценка на комплексноста на компонентите на системот.
-Сложеноста зависи од:

1. Сложеноста на контролните структури
2. Сложеноста на податочните структури
3. Објекти методи и големина на модулите

24. Метрики на процеси – се користат за да се процени одржливоста

1. Број на барања за корективно одржување
2. Просечно време за анализа на влијанието
3. Просечно време за да се имплементира барањето за промена
4. Број на outstanding барања за промена.

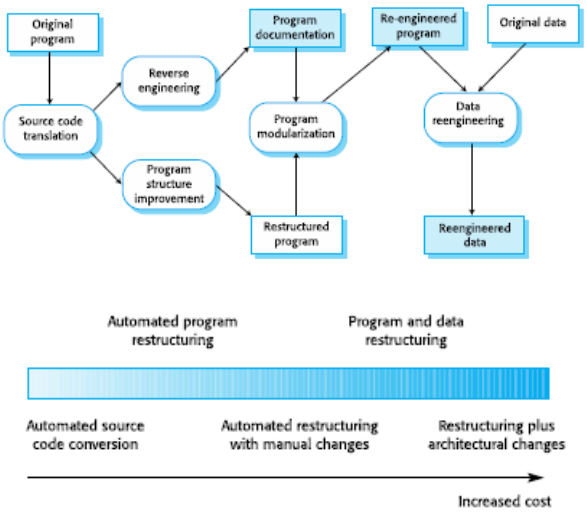
-Ако некои од овие се зголемува, може да укажи на опаѓање на одржливоста.

25. Реинженирање на софтверот – реструктурирање или повторно пишување на дел или сите наследни системи без да се промени функционалноста.
-Се применува кога некои, но не сите подсистеми на еден поголем систем, често бараат одржување
-Реинженерството вклучува додавања напор за да можат полесно да се одржат. Системот може да се реструктурира и да редокументира.
-Придобивки:

1. Намален ризик – развивањето на целосен нов софтвер носи големи ризици.
2. Намалени трошоци

26. Активности на процесот на реинженирање

1. Преведување на изворниот код – во нов јазик
2. Обратно инженерство – анализирање на програмата за да може да се разбере
3. Подбрување на структурата на програмата
4. Модуларизација на програмта – реорганизиација на програмската структура
5. Реинженирање на податоците – чистење и реструктурирање на системските податоци.



27. Reengineering cost factors
-Квалитетот на софтверот кој треба да се реинженира
-Алатките за поддршка кои се достапни за реинженирање
-Обемот на конверзија на податоците кои се побарани/
-Достапност на стручен персонал за реинженирање

28. Реструктурирање (refactoring) – процес на подобрување на програма за да се забави нејзината деградација преку промените.
-Превентивно одржување, што ги намалува проблемите со идните промени.
-Вклучува модификација на програма за да се подобри нејзината структура, намалување на сложеноста и полесно разбирање
-Кога се реструктурира една програма, не се додава функционалност, туку се концнентрираме на подобрување на програмата.

29. Реструктурирање и реинженирање (refactoring and reengineering)
-Реинженирањето се одвива откако системот се одржува некое време и трошоците за одржување се зголемуваат. Се користат автоматизирани алатки за да се обработат и реинженираат наследните системи за да се креира нов систем кој е поодржлив.
-Реструктурирањето е континуиран процес на подобрување за време на процесот на развој и еволуција. Наменет е да се избегне деградација на структурата и кодот што ги зголемува трошоците и тешкотиите при одржување на системот.

30. ‘Bad smells’ во програмски код

1. Двоен код – ист или многу сличен код да биде вклучен на различни места во програмата. Ова може да се отстрани и да се спроведе како единствен метод или функција.

2. Долги методи – ако некој метод е предолг, треба да се редизајнира на неколку пократки методи.

3. Switch (case) statements – често вклучуваат дуплирање, каде прекинувачот зависи од видот на вредноста. Изјавите за прекинувачот може да се расфрлани низ програмата. Кај ОО јазици, често може да се користи полиморфизам за да се постигне истата работа.

4. Data clumping – се појавуваат кога иста група на податочни елементи се појвуваат на неколку места во програмата. Овие често може да се аменат со објект кој ги опфаќа сите податоци.

5. Speculative generality – кога програмерите вклучуваат општост во програма, во случај да се бара во иднина. Често може едноставно да се отстрани.