Моделирање на анализа

Анализа на побарувања

- Анализа на побарувања
  - Ги специфицира оперативните карактеристики на софтверот
  - Упатува за интерфејс на софтверот со други системски елементи
  - Воспоставува ограничувања што софтверот мора да ги исполни
- Анализата на барањата му овозможува на софтверскиот инженер (наречен аналитичар или моденер во оваа улога): Да разработи основни барања утврдени во претходните задолжителни инженерски задачи Изградба на модели кои прикажуваат сценарија на корисници, функционални активности, класи на проблемот и нивните односи, однесување на системот и класата и протокот на податоци како што се трансформираат.
- Во текот на моделирањето на анализата, примарниот фокус на СЕ е на она што не е на тоа како.
- Моделот за анализа и спецификацијата на барањата им овозможуваат на инвеститорот и на клиентот средства за проценка на квалитетот откако ќе се изгради софтвер.

Принципи за моделирање на анализа

- Методите за анализа се поврзани со низа оперативни принципи:
  1. Информацискиот домен на проблемот мора да биде претставен и разбран.
  2. Функциите што се софтверски изведби мора да бидат дефинирани.
  3. Однесувањето на софтверот мора да биде претставено.
  4. Моделите што прикажуваат информации, функција и однесување мора да бидат поделени на начин што открива детали на слоевит начин.
  5. Задачата за анализа треба да се движи од основните информации кон деталите за имплементацијата.

1. Доменот на информации опфаќа дека податоците се влеваат во системот, надвор од системот и складираните податоци.
2. Функциите обезбедуваат директна корист за крајните корисници и исто така обезбедуваат внатрешна поддршка за оние карактеристики што се видливи од корисниците.
3. Однесување водено од неговата интеракција со надворешното опкружување.
   - На пр. Влез обезбеден од крајните корисници, контролни податоци обезбедени од надворешен систем или податоци за набудување.
4. Клучна стратегија на моделот за анализа, поделете го сложениот проблем во подпроблем сè додека секој под-проблем не е разбирлив. Овој концепт се нарекува поделба.
5. „Суштината“ на проблемот е опишана без да се земе предвид како ќе се примени решението.
   - На пр. Видео играта бара тој „играч“ за играч

   - Детали за имплементација (модел на дизајн) означуваат како суштината ќе се спроведе на пр. Командата на тастатурата може да биде внесена или да се користи џојстик.

Цели на моделот на анализа

- Три основни цели:
  1. Опишете што бара клиентот.
  2. Воспоставете основа за креирање на софтверски дизајн.
  3. Направете сет на барања што можат да се потврдат откако ќе се изгради софтверот.
- Нејзиниот мост го прави јазот помеѓу описот на ниво на систем кој ја опишува целокупната функционалност на системот и дизајнот на софтверот.

- Упатства:
  - Графиките треба да се користат секогаш кога е можно.
  - Разликувајте помеѓу логичките (суштински) и физичките (имплементација) размислувања.
  - Развијте начин за следење и проценка на корисничките интерфејси.

Analysis Rules of Thumb

- Моделот треба да се фокусира на барањата што се видливи во рамките на проблемот или деловниот домен. Нивото на апстракција треба да биде релативно високо. - Не влегувај во детали.
- Секој елемент од моделот за анализа треба да додаде на целокупното разбирање на барањата на софтверот и да обезбеди увид во доменот на информации, функцијата и однесувањето на системот.
- Одложување на инфраструктурата и другите нефункционални модели до дизајнирање.
- минимизирајте го спојувањето низ целиот систем. - Ако нивото на меѓусебна поврзаност е високо, треба да се направат напори за да се намали.
- Увери дека моделот на анализа им дава вредност на сите засегнати страни. - деловните заинтересирани страни треба да го потврдат барањето, Дизајнерите треба да го користат моделот како основа за дизајн.
- Чувајте го моделот колку што е можно поедноставно. - Нема потреба да користите дополнителен дијаграм и да користите нотации.

Елементи на модел на анализа

- Постојат два пристапа

1. Структурирана анализа:
   - Објектите со податоци се моделираат на начин што ги дефинираат нивните атрибути и врски.
   - Процеси што манипулираат со податочните објекти на начин што покажува како тие ги трансформираат податоците како што се движат објекти на податоци низ системите.
2. Ориентирана кон објект:
   - Се фокусира на дефиницијата на часовите и начинот на кој тие соработуваат едни со други.
   - UML е претежно ориентиран кон објекти.

Официјална употреба-случај

- Случај за користење:
- Итертација:
- Основен актер:
- Цел во контекст:
- Предуслови:
- Предизвикувач:
- Сценарио:
- Исклучоци:
- Приоритет:
- Кога е достапно:
- Фреквенција на употреба:
- Канал до актер:
- Средни глумци:
  - Закачи до секундарни актери:
- Отворени проблеми:

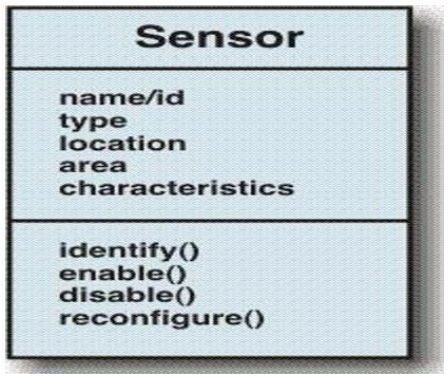
- Случај на користење: Пристапете до камера за надзор преку Интернет (ACS-DCV)
- Итерација: 2, последна модификација: 14 јануари од В. Раман.
- Основен актер: Сопственик на домови.
- Цел во контекст: Да гледате излез на камера поставена низ целата куќа од која било оддалечена локација преку Интернет.
- Предуслови: Системот мора да биде целосно конфигуриран; мора да се добијат соодветно корисничко име и лозинки.
- Предизвикувач: Сопственикот на домот одлучува да погледне во куќата додека е далеку.
- Сценарио: 1.… 2.…
- Исклучоци:
  1. ID или лозинки не се точни или не се препознаени / Валидација на ID и лозинки

2. Функција за надзор не е конфигурирана за овој систем - системот прикажува соодветна порака за грешка / Конфигурирајте ја функцијата за надзор.
3. Сопственикот на домот избира „Прикажи слики од сликички за сите камери“
4. План за подот не е достапен или не е конфигуриран - прикажете соодветна порака за грешка / Конфигурирајте го планот за подот.
5. Составена е состојба на алармот

- Приоритет: Умерен приоритет, што треба да се спроведе по основните функции.
- Кога е достапно: Трет прираст.
- Фреквенција на употреба: Умерена фреквенција.
- Канал до актер: Преку прелистувач базиран на компјутер и Интернет врска.
- Средни актери: Администратор на системот, фотоапарати.
- Канали до секундарни актери:
  1. Администратор на системот: Систем базиран на компјутер.
  2. Камери: безжична конекција.
- Отворени проблеми

1. Кои механизми ја штитат неовластената употреба на оваа можност од страна на вработените во SafeHome Products?
2. Дали безбедноста е доволна? Хакирањето во оваа одлика би претставувало голема инвазија на приватноста.
3. Дали одговорот на системот преку Интернет ќе биде прифатлив со оглед на широчината на опсегот потребен за прегледување на камерата?
4. Дали ќе развиеме можност да обезбедиме видео со поголема брзина на рамки на секунда кога се достапни врски со ширина на опсег?

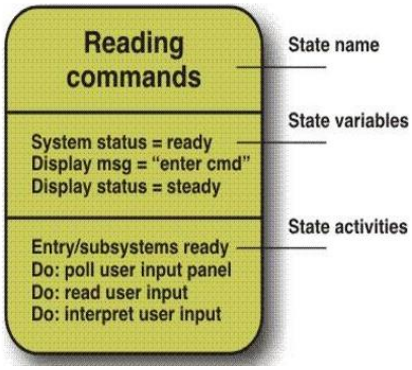
Класен дијаграм за сензор



- Елементи засновани на класа

- Различни системски објекти (добиени од сценарија) вклучувајќи ги нивните атрибути и функции (класа дијаграм)

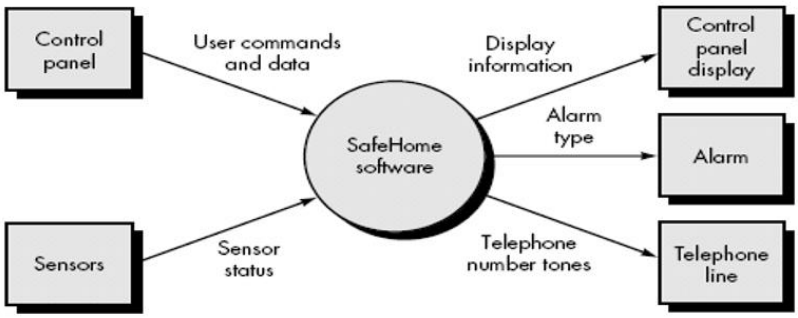
Однесувачки Елемент – Дијаграм на состојби



- Елементи на однесување

- Како системот се однесува како одговор на различни настани (државен дијаграм)

Елементи ориентирани кон проток



Елементи ориентирани кон проток

- Како се трансформираат информациите како да течат низ системот (дијаграм за проток на податоци)
- Системот прифаќа влез во различни форми; применува функции за да се трансформира; и произведува излез во различни форми.

Моделирање на податоци

- Моделот за анализа често започнува со моделирање на податоци.
- Моделот на податоци се состои од три меѓусебно поврзани делови:
  1. Податочен објект,
  2. атрибути што го опишуваат предметот на податоците и
  3. Односите што ги поврзуваат предметите со податоци едни на други

Податочен објект

- Објект со податоци е претстава на скоро сите сложени информации што мора да бидат разбрани од софтвер.
- композитни информации - број на различни својства или атрибути.
- Објект со податоци може да биде:
  - Надворешен ентитет (на пр., Што произведува или троши информации),
  - Нешто (на пр. Извештај или дисплеј),
  - Појава (на пр. Телефонски повик)
  - Настан (на пр. аларм),
  - Улога (на пр., продавач),
  - Организациска единица (на пр. сметководствен оддел),
  - Место (на пр. магацин),
  - Структура (на пр. датотека).

Податочни атрибути

- Дефинирајте ги својствата на податочниот објект.
- Атрибутите именуваат податочен предмет, ги опишуваат неговите карактеристики и во некои случаи, упатуваат на друг предмет.
- Покрај тоа, една или повеќе од атрибутите мора да бидат дефинирани како идентификатор (клучна вредност или единствена вредност).
  - Пр. Објект на податоци Автомобилот има број на идентификација како идентификатор.

Податочни односи

- Објектите со податоци се поврзани едни на други на различни начини.
- Размислете за два објекти со податоци
  - Книга
  - Книжарница
- Поставена е врска помеѓу книгата и книжарницата бидејќи двата предмети се поврзани

- За да се утврди врската помеѓу нив, мора да се разбере улогата на книгата и книжарницата.
- Може да дефинира збир на парови на предмети / врски што ги дефинираат релевантните односи.
- На пример:
  - Книжарница нарачува книги.
  - Книжарница прикажува книги.
  - Книжарница чува книги.
  - Книжарница продава книги.
  - Книжарница враќа книги.

Кардиналност и модалитет

- Дополнителен елемент на моделирање на податоци.
- Предмет X поврзан на предметот Y не дава доволно информации.
- Колку појави на објектот X се поврзани со колку појави на објектот Y наречен кардиналност.

Кардиналност

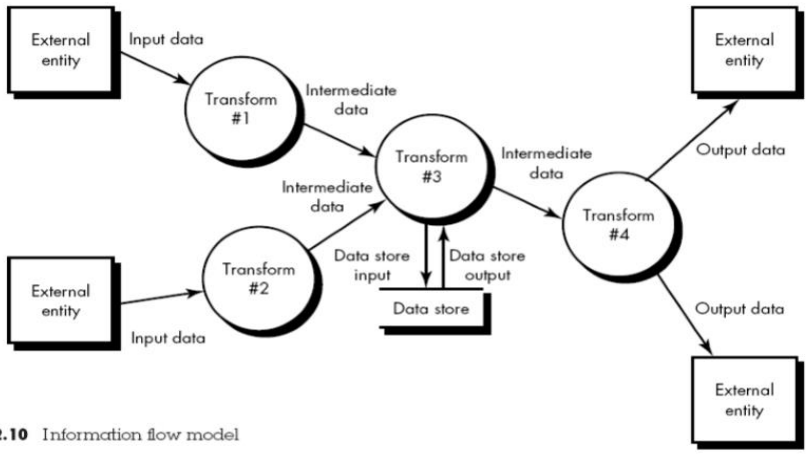
- Го претставува бројот на предмети што се појавуваат во дадена врска.
- Кардиналност е спецификација на бројот на појави на еден [предмет] што може да се поврзе со бројот на појави на друг.
- Кардиналноста обично се изразува како едноставно „една“ или „многу“.
- 1: 1 - Еден предмет може да се однесува само на еден друг предмет.
- 1: M - еден предмет може да се однесува на многу објекти.
- M: N - Некои бр. на појави на некој предмет може да се однесува на некој друг бр. на појави на друг предмет.

Модалитет

- Кардиналноста не дава индикација дали одреден предмет на податоци мора да учествува во врската.
- Модалитетот на врската е 0 ако нема експлицитна потреба да се појави врската или врската е по избор.
- Модалитетот е 1 ако настаните во врската се задолжителни.

Функционкциско моделирање и проток на информации

- Информациите се трансформираат како што течат низ компјутерски систем. Системот прифаќа влез во различни форми; применува хардвер, софтвер и човечки елементи за да ги трансформира; и произведува излез во различни форми
- Структурната анализа започна како техника за моделирање на проток на информации.
- Правоаголник се користи за да претставува надворешен ентитет (софтвер, хардвер, личност)
- Круг (понекогаш се нарекува меур) претставува процес или трансформација што се применува на податоци (или контрола) и го менува на некој начин.
- Стрелката претставува една или повеќе податочни елементи (податочни објекти) и треба да биде означена.
- Двојната линија претставува зачувана информација за складирање на податоци што ги користи софтверот.
- Првиот модел на проток на податоци (понекогаш наречен ниво 0 DFD или дијаграм на контекст) го претставува целиот систем.
- Дава детални детали со секое следно ниво.



12.10 Information flow model

Креирање на модел на проток на податоци

- Овозможува софтверски инженер да развива модели на домен на информации 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.
- Бидејќи рафинирањето на ДФД резултира во соодветно рафинирање на податоците додека се движи низ процесите што ја претставуваат апликацијата

Упатства за ДФД

- Опишете го системот како единечен меур на ниво 0. should Треба внимателно да се забележат основните влезни и излезни
- Прочистете се со изолирање на процесите на кандидатот и нивните поврзани, продавници за податоци и продавници со податоци
- Сите стрели и меури треба да бидат означени со значајни имиња. Continu Континуитетот на проток на информации мора да се одржува од ниво на ниво.
- Треба да се рафинира еден меур во исто време.

Модели на проток на податоци

- Ниво 0 DFD, исто така наречен фундаментален системски систем или контекст модел, го претставува целиот софтверски елемент како единечен меур со влезни и излезни податоци означени со влезни и појдовни стрели.
- Ниво 0 рафинирање на DFD во ниво 1 DFD со сите релевантни процеси на системот.
- Ниво 1 DFD, секој процес може да се рафинира во ниво 2 DFD.
- Рафинирањето на DFD продолжува сè додека секој меур не изврши едноставна функција.

Контролен модел на проток

- Апликацијата што содржи збирка класи зависи од настанот отколку од податоците, произведува контрола на информации отколку извештаи или прикази.
- Таквата апликација бара употреба на контролирано моделирање на проток, покрај моделирање на проток на податоци.

Упатство за проток на контрола

- Наведете ги сите процеси што ги изведува софтверот.
- Наведете ги сите услови за прекин.
- Наведете ги сите активности што ги извршува операторот или актерот.
- Наведете ги сите услови на податоци.
- Прегледајте ги сите „Контролни артикли“ што е можно за влезни / излезни контролни протоци.
- Опишете го однесувањето на системот со идентификување на неговите состојби; идентификувајте како се достигнува секоја држава; ги дефинираат транзициите помеѓу државите.
- Фокусирајте се на можен пропуст - многу честа грешка при одредување на контрола

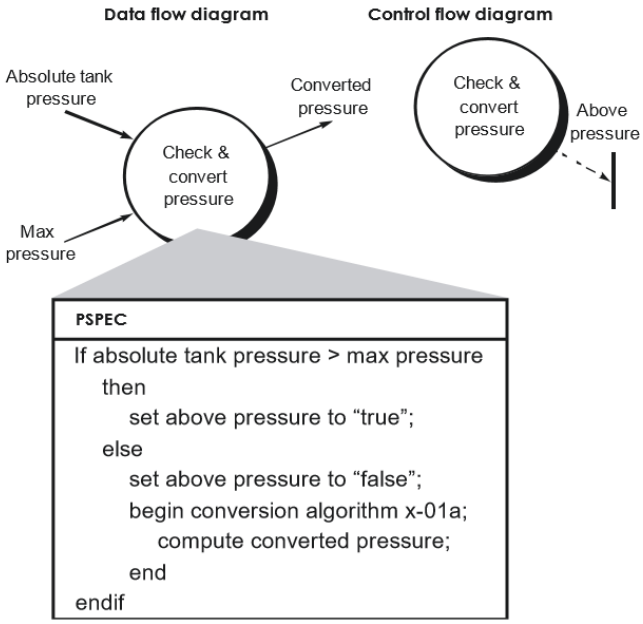
Контролна спецификација (CSPEC)

- CSPEC го претставува однесувањето на системот на два различни начина.
- Содржи државен дијаграм - секвенцијална спецификација на однесување.
- Исто така, содржи табела за активирање на програмата - комбинаторска спецификација на однесувањето.

- Со прегледување на државниот дијаграм, софтверски инженер може да утврди однесување на системот и може да открие дали има „дупки“ во одредено однесување.
- CSPEC го опишува однесувањето на системот, но не дава информации за внатрешното работење на процесите што активирале резултат.

Спецификација на процеси (PSPEC)

- Се користи за опишување на сите процеси на моделот на проток што се појавуваат на последното ниво на рафинирање.
- Вклучува наративен текст, јазик за дизајн на програма (PDL) опис на алгоритмот на процесот, математички равенки, табели, дијаграми или графикони.
- Со обезбедување на PSPEC да го придружува секој меур во моделот на проток, софтверскиот инженер создава „мини-спецификација“ што може да послужи како водич за дизајнирање на софтверската компонента што ќе го спроведе процесот.



Моделирање засновано на класи

- Идентификување на класи за анализа
- Одредување на атрибути
- Дефинирање на операциите
- Моделирање со CRC

1. Идентификување класи за анализа

- Идентификувајте ги класите со испитување на изјавата за проблемот или со извршување на „Граматичка парсија“ на случаи на употреба или обработка на наративи развиени за системот.
  Како се манифестираат класите за анализа?
- Надворешни субјекти (друг систем, луѓе, уреди) кои произведуваат или трошат информации
- Работи (извештаи, приказ, сигнали) кои се дел од информатичкиот домен за проблемот.
- Настани или настани - се случуваат во контекст на системските операции.
- Улоги (менаџер, инженер, продавач) играат од луѓе кои комуницираат со системот.
- Организациски единици (дивизија, група, тим) кои се релевантни за апликација,
- Места - да се воспостави контекст на проблемот и целокупната функција на системот.
- Структури (сензори, компјутери) кои дефинираат класа предмети или сродни класи на предмети.

Избор на критериуми - класи

- Задржани информации - Потенцијалната класа мора да се запомни за да може системот да функционира.
- Потребни услуги - Мора да има збир на препознатливи операции што можат да ја променат вредноста на неговите атрибути.
- повеќе атрибути - Класа со единечен атрибут, всушност, може да биде корисна за време на дизајнирањето, но веројатно подобро претставена како атрибути на друга класа.
- Заеднички атрибути - Овие атрибути се применуваат за сите случаи на часот

2. Одредување на атрибути
- Атрибути се збир на објекти со податоци што целосно ја дефинираат класата во контекст на проблемот.
- За да развијат атрибути за класата, може да го проучи случајот за употреба и да ги избере оние „работи“ што разумно „припаѓаат“ на часот.

3. Дефинирање на операциите
- Операциите го дефинираат однесувањето на некој предмет.
- Четири широки категории
    1. Операции што манипулираат со податоци на некој начин.
    2. Операции што вршат пресметка.
    3. Операции кои се распрашуваат за состојбата на објектот
    4. Операции кои следат предмет за појава на контролен настан.
- За да изведе збир на операции, аналитичарите изучуваат случај на употреба (или наратив) и изберете ги операциите што разумно припаѓаат.

4. Class-Responsibility-collaborator (CRC) моделирање
- CRC моделот е збирка на стандардни индекс картички што претставуваат класи. Картите се поделени на три дела.
    - Врвот на картичките напишете име на класата
    - Во телото, наведете ја одговорноста за часот лево.
    - Соработник од десно
- Едноставни средства за идентификување и организирање на часовите што се релевантни за системот или на производот.
- Користете вистински или виртуелни картички за индекс.

CRC - Инстантирање на класи

- Класи на три типа:
    1. Класи на субјекти (модел или деловни часови): - Претставуваат работи што треба да се чуваат во база на податоци и опстојуваат во текот на траењето на апликацијата.
    2. Гранична класа: - се користи за создавање интерфејс. Тој беше дизајниран со одговорност за управување со начинот на кој предметите на ентитетот се претставени пред корисниците.
    3. Класа на контролор: - управувате со „единица за работа“ од почеток до крај.
    - Создавање или ажурирање на предмети на ентитетот.
    - Претставување на гранични објекти бидејќи добиваат информации од предмети на ентитет.
    - Комплексна комуникација помеѓу множества предмети.
    - Верификација на податоците.

Модел на однесување

- Моделот на одмесување покажува како софтверот ќе реагира на надворешни настани.
- Да се создаде модел
  - Оценете ги сите случаи на употреба за да се разбере редоследот на интеракција во рамките на системот.
  - Идентификувајте ги настаните
  - Создадете низа за секој случај на употреба
  - Изградете државен дијаграм за системот.
  - Прегледајте го моделот на однесување за да ја проверите точноста или доследноста.

Идентификување на настани со случаи на употреба

- Користењето на случајот претставува низа активности што ги вклучуваат актерите и системот.
- Се појавува настан кога системот и актерот разменуваат информации.
- Настан не е информација што се разменила, туку е фактот дека информациите се разменети.
- За секој настан треба да се идентификува актер.
- Треба да се забележат информации што се разменуваат.
- Треба да се наведат какви било услови или ограничувања.

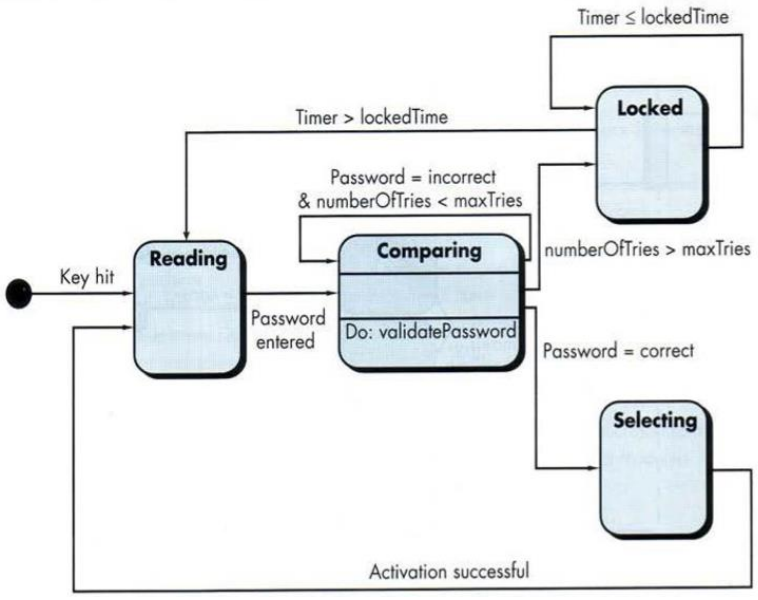
Репрезентација на состојби

- Во предвид треба да се земат две различни карактеристики.
  - Пасивна состојба
  - Активна состојба
- Пасивната состојба е едноставно тековниот статус на сите атрибути на објектот.
  Пр. Тековна позиција и ориентација на класата на играчи - атрибути.
- Активната состојба е моментална состојба на објектот бидејќи се подложува на постојана трансформација или обработка.
  Пр. Активна состојба на играчи од класа на играчи, повредени, заробени, изгубени и др.

  Треба да се случи настан за да се присили некој предмет да изврши премин од една активна состојба во друга.

Дијаграм на состојби за анализирачки класи

- UML - дијаграм што претставува активни состојби за секоја класа и настани што предизвикуваат промени помеѓу активната состојба.
- Едно дејство се случува истовремено со државната транзиција или како низа од истата и генерално вклучува една или повеќе операции на предметот



Дијаграм на секвенца

- Тоа покажува како настаните предизвикуваат транзиции од предмет до предмет.
- Откако настанот ќе се идентификува со испитување на случаи на употреба, моденерот создава дијаграм на секвенци.
- Претставување на тоа како настаните предизвикуваат проток од еден до друг предмет како функција на време.
- Нејзината сносилна верзија на дијаграмот за употреба, што претставува клучни класи и настаните што предизвикуваат однесување однесување од класа во класа.
- Предмет и настани на системот ќе помогнат во креирање на ефективен дизајн.

Механика на структурирана анализа

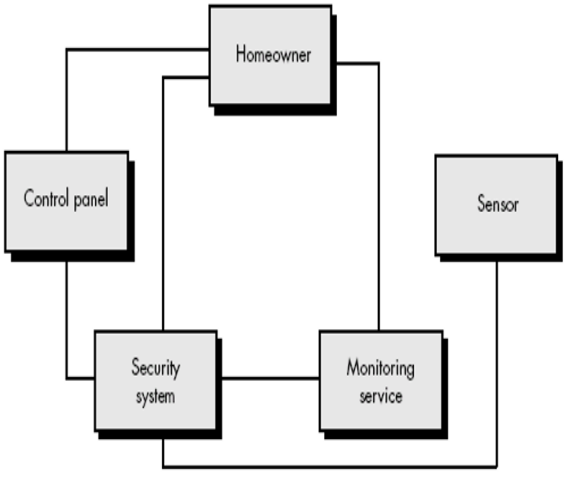
- Сето тоа за
  - Дијаграм за врски со ентитети (ERD)
  - Дијаграм за проток на дата (DFD)
  - Дијаграм за транзиција на состојба (STD)

ERD

Креирање на ERD дијаграм:

- Од клиентите се бара да ги наведат „нештата“ на кои се однесува апликацијата или деловниот процес
- Аналитичарот и дефинираната врска постојат од корисниците помеѓу предметот на податоци и другите предмети (доколку има)
- Каде и да постои врска, аналитичарот и клиент создава еден или повеќе парови на предмети / врски.
- За секој предмет / врска се истражува, кардиналност и модалитет.
- Чекорите 2 до 4 се продолжуваат повторливо, сè додека не се дефинираат сите предмети / врски.
- Дефинирани се атрибути на секој ентитет.
- Дијаграмот за односи со ентитетот е формализиран и разгледан.
- Чекорите од 1 до 7 се повторуваат додека не се заврши моделирањето на податоците

Безбеден систем за безбедност на домот:



Чекор 1: Идентификувани работи
- сопственик на домот
- контролен панел
- сензори
- систем за безбедност
- услуга за набудување

Чекор 2: еден или повеќе парови на предмети / врски се идентификуваат за секоја врска.

Чекор 3: безбедносен систем го следи сензорот
- безбедносниот систем овозможува / оневозможува сензор
- сензор за безбедносен систем
- сензор за програми за безбедност на системот

Чекор 4: Кардиналност и модалитет
- Кардиналноста меѓу безбедносниот систем и сензорот е една до многу.
- Модалитет на безбедносниот систем (задолжително) и сензор (задолжително)

Чекор 5: повторете 2 до 4 за сите објекти.

Чекор 6: Секој предмет се изучува за да се утврдат неговите атрибути.

На пример: сензор - тип на сензор, внатрешен идентификациски број, локација на зона и ниво на аларм.

Речник на податоци

- Речник на податоци е организиран список на сите елементи на податоци што се релевантни за системот, со прецизни, ригорозни дефиниции, така што и корисникот и системот аналитичар ќе имаат заедничко разбирање на влезовите, излезите, компонентите на продавниците и [дури] средните пресметки.
- Речникот може да се користи секогаш како дел од структурната алатка за анализа и дизајнирање на случајот.
- „Повеќето ги содржат следниве информации:
  - Име - примарно име на податоци или контролна точка, продавница за податоци или надворешен ентитет.
  - Alias - имиња што се користат за првиот запис.
  - Каде што се користи / како се користи - список на процеси што користат податоци или контролна ставка и како се користат (на пр. влез во процесот, излез од процесот, како продавница, како надворешен ентитет.
- Опис на содржината — нотација за претставување на содржина.
- Дополнителни информации — други информации за типови на податоци, претходно поставени вредности (ако се познати), ограничувања или ограничувања и слично.
- Откако името на објектот за податоци или контролниот елемент и неговите алијаси ќе бидат внесени во речникот, може да се зајакне конзистентност во именувањето. Тоа е, ако членот на тимот за анализа одлучи да именува нова изведена податочна точка xyz, но xyz веќе е во речникот, алатката CASE што го поддржува речникот објавува предупредување за да означи дупликат имиња.
- „Каде што се користат / како се користат“ информациите се запишуваат автоматски од моделите на проток. Кога се креира запишување во речникот, алатката CASE скенира DFD и CFD за да утврди кои процеси ги користат податоците или информациите за контрола и како се користат.
- За големи проекти, често е тешко да се утврди влијанието на промената. Многу софтверски инженер прашаа: "Каде се користи овој податочен предмет? Што друго ќе треба да се промени ако го измениме? Кое ќе биде целокупното влијание на промената?" Бидејќи речникот на податоци може да се третира како база на податоци,

Ознаката што се користи за развој на опис на содржината е наведена во следната табела:

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

Архитектонски дизајн

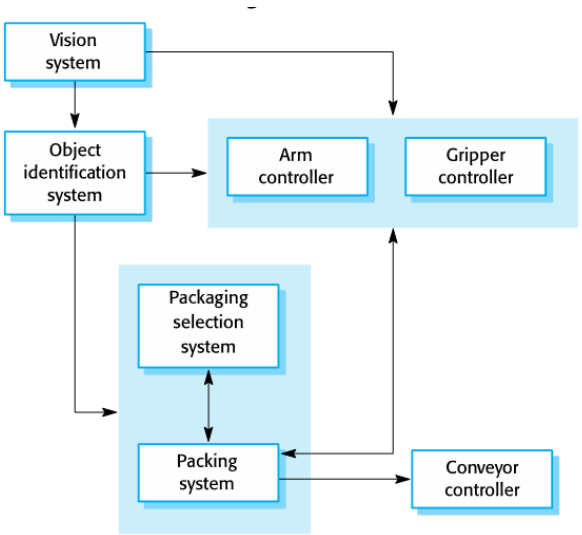
Архитектонски дизајн

- Архитектонскиот дизајн е заинтересиран да разбере како треба да се организира софтверски систем и да се дизајнира целокупната структура на тој систем.
- Архитектонскиот дизајн е критична врска помеѓу инженерството за дизајн и побарувачката, бидејќи ги идентификува главните структурни компоненти во системот и односите меѓу нив.
- Излезот на процесот на архитектонско дизајнирање е архитектонски модел кој опишува како системот е организиран како збир на компоненти за комуникација.

Агилност и архитектура

- Општо е прифатено дека раната фаза на агилни процеси е да се дизајнира целокупна архитектура на системи.
- Реновирањето на архитектурата на системот е обично скапо, бидејќи влијае на толку многу компоненти во системот

Архитектурата на систем за контрола на роботи за пакување



Архитектонска апстракција

- Архитектурата во мали се занимава со архитектурата на одделни програми. На ова ниво, ние сме загрижени за начинот на кој индивидуалната програма се распаѓа во компоненти.
- Архитектурата во голема мерка се занимава со архитектурата на комплексни системи на претпријатија кои вклучуваат други системи, програми и компоненти на програмата. Овие системи на претпријатието се дистрибуираат преку различни компјутери, кои можат да бидат во сопственост и раководење од различни компании.

Предности на експлицитна архитектура

- Комуникација на засегнатите страни - Архитектурата може да се користи како фокус на дискусија од страна на засегнатите страни во системот.
- Системска анализа - можна е анализа за тоа дали системот може да ги исполни своите нефункционални барања.
- Голема повторна употреба - Архитектурата може да биде повторна употребена во низа системи Може да се развијат архитектури со линија на производи.

Архитектонски репрезентации

- Едноставни, неформални блок дијаграми кои покажуваат субјекти и врски се најчесто користениот метод за документирање на софтверската архитектура.
- Но, овие се критикувани бидејќи немаат семантика, не ги покажуваат видовите врски помеѓу ентитетите, ниту видливите својства на субјектите во архитектурата.
- зависи од употребата на архитектонски модели. Барањата за семантика на моделот зависи од тоа како се користат моделите.

Дијаграми на кутии и линии

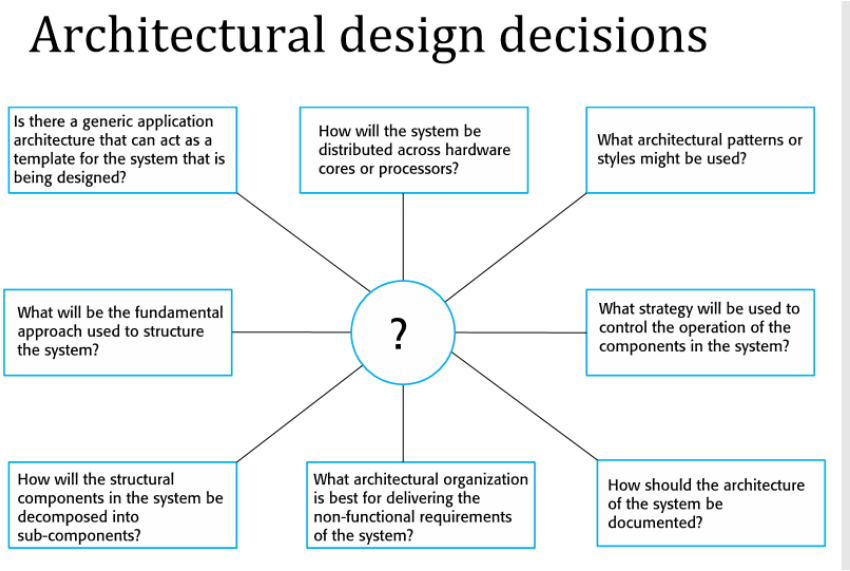
- Многу апстрактни - тие не ја покажуваат природата на односите со компонентите, ниту надворешно видливите својства на под-системите.
- Сепак, корисно за комуникација со засегнатите страни и за планирање на проекти.

Употреба на архитектонски модели

- Како начин за олеснување на дискусијата за дизајнот на системот
  - Високо ниво на архитектонски поглед на системот е корисно за комуникација со засегнатите страни во системот и за планирање на проекти, бидејќи не е преполно со детали. Засегнатите страни можат да се однесуваат на тоа и да разберат апстрактен поглед на системот. Потоа, тие можат да разговараат за системот како целина, без да бидат збунети од детали.
- Како начин за документирање на архитектурата што е дизајнирана
  - Целта е да се изработи целосен системски модел кој ги прикажува различните компоненти во системот, нивните интерфејси и нивните врски.

Одлуки за архитектонски дизајн

- Архитектонскиот дизајн е креативен процес, така што процесот се разликува во зависност од видот на системот што се развива.
- Сепак, голем број на вообичаени одлуки ги опфаќаат сите процеси на дизајнирање и овие одлуки влијаат на нефункционалните карактеристики на системот.



Повторна употреба на архитектурата

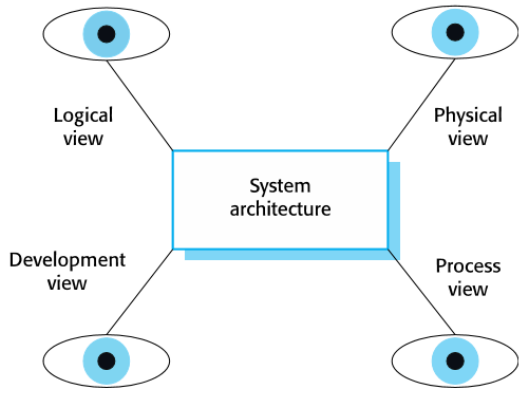
- Системите во ист домен често имаат слични архитектури кои ги рефлектираат доменските концепти.
- Линиите на производи за апликација се градат околу основната архитектура со варијанти кои задоволуваат одредени барања за клиент.
- Архитектурата на системот може да биде дизајнирана околу еден од повеќе архитектонски модели или „стилови“.
  - Овие ја доловуваат суштината на архитектурата и можат да бидат инстантирани на различни начини.

Архитектура и системски карактеристики

- Изведба - Локалисекритички операции и минималикомуникации. Користете големи, наместо компоненти со ситно зрно.
- Безбедност - Користете слоевит архитектура со критични средства во внатрешните слоеви.
- Безбедност - Локализирање на безбедносни критични карактеристики кај мал број на под-системи.
- Достапност - Вклучете непотребни компоненти и механизми за толеранција на дефект.
- Одржливост - Користете компоненти што можат да се заменуваат со ситно зрно.

Архитектонски погледи

- Кои погледи или перспективи се корисни при дизајнирање и документирање на архитектурата на системот?
- Кои нотации треба да се користат за опишување на архитектонски модели?
- Секој архитектонски модел покажува само еден поглед или перспектива на системот.
  - Може да покаже како системот се распаѓа во модули, како процесите на траење на времето комуницираат или различните начини на кои се дистрибуираат компонентите на системот низ мрежа. И за дизајнот и за документацијата, обично треба да презентирате повеќе погледи на архитектурата на софтверот.



4 + 1 преглед на модел на софтверска архитектура

- Логичен приказ, кој ги покажува клучните апстракции во системот како објекти или класи на објекти.
- Преглед на процес, кој покажува како, во време на извршување, системот е составен од интеракциски процеси.
- Разглед на развој, кој покажува како софтверот се распаѓа за развој.
- физички преглед, кој го прикажува системскиот хардвер и како софтверските компоненти се дистрибуираат низ процесорите во системот.
- Поврзани со случаи на употреба или сценарија (1)

Претставување на архитектонски погледи

- Некои луѓе тврдат дека Унифицираниот јазик за моделирање (UML) е соодветна нотација за опишување и документирање на архитектурите на системот
- Не се согласувам со ова бидејќи не мислам дека UML вклучува апстракции соодветни за опис на системот на високо ниво.
- Јазиците за опис на архитектурата (АДЛ) се развиени, но не се користат широко

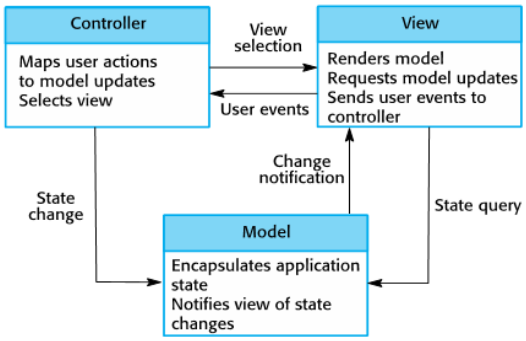
Архитектонски обрасци

- Моделите се средство за претставување, споделување и користење на знаења.
- Архитектонски образец е стилизиран опис на добра практика за дизајн, која е испробана и тестирана во различни средини.
- Моделите треба да содржат информации за тоа кога се и кога не се корисни.
- Моделите можат да бидат претставени со употреба на табеларни и графички описи.

The-Model-View-Controller(MVC)

- Опис - Одделува презентација и интеракција од податоците на системот. Системот е структуриран во три логички компоненти кои комуницираат едни со други. Компонентата Модел управува со податоците на системот и придружните операции на тие податоци. Компонентата Преглед дефинира и управува со тоа како се презентираат податоците на корисникот. Компонентата Контролер управува со интеракцијата со корисници (на пр., Притискање на копчињата, кликнување со глувчето и сл.) И ги пренесува овие интеракции на прегледот и на моделот.
- Пример - Слика 6.4 ја прикажува архитектурата на веб-базиран систем на апликации организиран во шема на MVC.
- Користено - се користи кога има повеќе начини за прегледување и интеракција со податоците. Исто така се користат кога идните барања за интеракција и презентација на податоците се непознати.
- Предности - Овозможува податоците да се менуваат независно од неговото претставување и обратно. Поддржува презентација на истите податоци на различни начини со промените направени во презентацијата, на сите нив.
- Недостатоци - може да вклучуваат дополнителни сложеност на код и код кога моделот на податоци и интеракциите се слични.

Огранизација на MVC

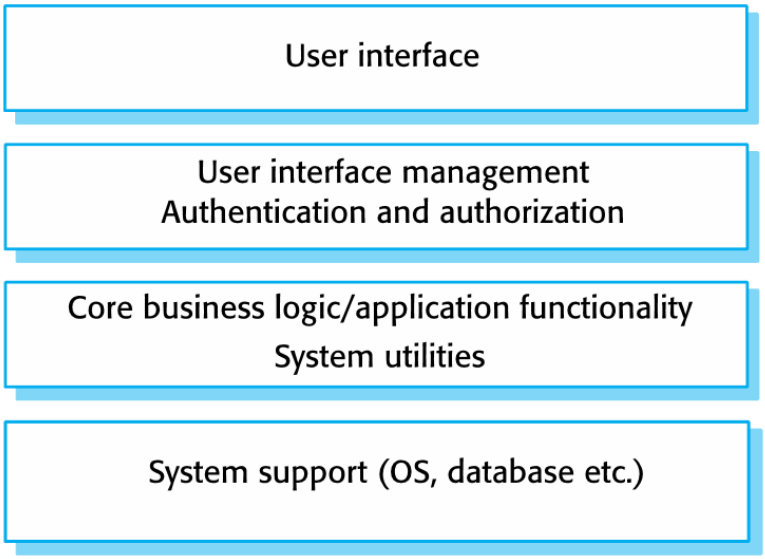


Слоевита архитектура

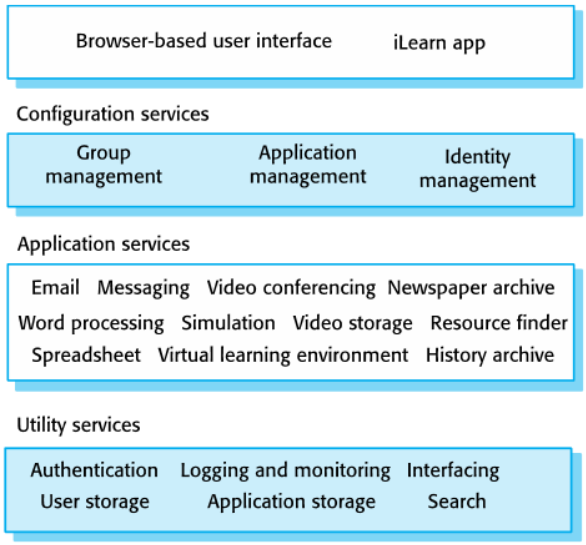
- Се користи за моделирање на меѓусебните системи на под-системи.
- Го организира системот во збир на слоеви (или апстрактни машини) од кои секоја обезбедува збир на услуги.
- Го поддржува постепениот развој на подсистемите во различни слоеви. Кога интерфејсот на слој се менува, само соседниот слој е засегнат.
- Сепак, честопати се вештачки за структурните системи на овој начин.
- Опис - Го организира системот во слоеви со поврзана функционалност поврзана со секој слој. Алајер обезбедува услуги до слојот над него, така што најниските нивоа слоеви претставуваат основни услуги што можат да бидат користени преку системот за употреба. Погледнете ја слика 6.6.
- Пример - Слоевит модел на систем за споделување документи за авторски права што се чуваат во различни библиотеки, asshowninFigure6.7.
- Користено - се користи при градење нови објекти над постојните системи; кога развојот се шири во неколку тимови со секоја екипа одговорност за слој на функционалност; кога постои задолжителна нивоа на сигурност.
- Предности - Овозможува замена на цели слоеви сè додека интерфејсот се одржува. Во секој слој може да се обезбедат излишни објекти (на пр., Автентикација) за да се зголеми зависноста на системот.

- Недостатоци - Во пракса, обезбедувањето чиста поделба помеѓу слоевите е често тешко и слојот на високо ниво можеби ќе треба директно да комуницира со слоевите на пониско ниво, отколку преку слојот веднаш под него. Перформансите можат да бидат проблем заради повеќе нивоа на толкување на барање за услуга бидејќи се обработува на секој слој.

Генерирачка слоевита архитектура



Архитектура на iLearn систем

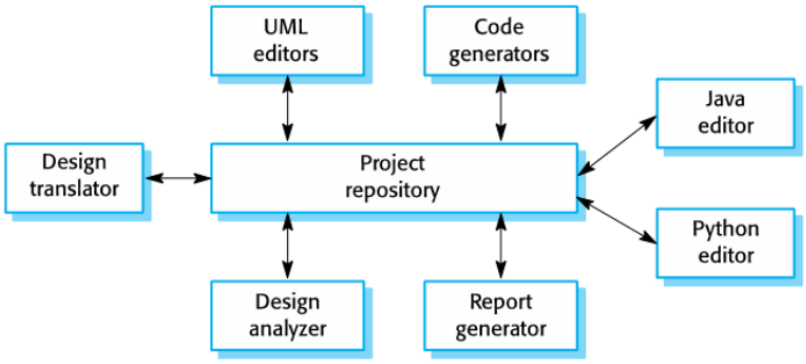


Архитектура на складиште

- Под-системите мора да разменуваат податоци. Ова може да се направи на два начина:
  - споделените податоци се чуваат во централна база на податоци или складиште и може да пристапат до сите потсистеми;
  - Секој под-систем одржува своја база на податоци и ги пренесува експлицитно податоците на други под-системи.
- Кога треба да се споделат големи количини на податоци, најчесто се користи моделот на складирање на складишта, што е ефикасен механизам за споделување податоци.
- Опис Сите податоци во системот се управуваат во централното складиште што е достапно за сите компоненти на системот. Компонентите не се меѓусебно меѓусебно интерактивни, само на размислување

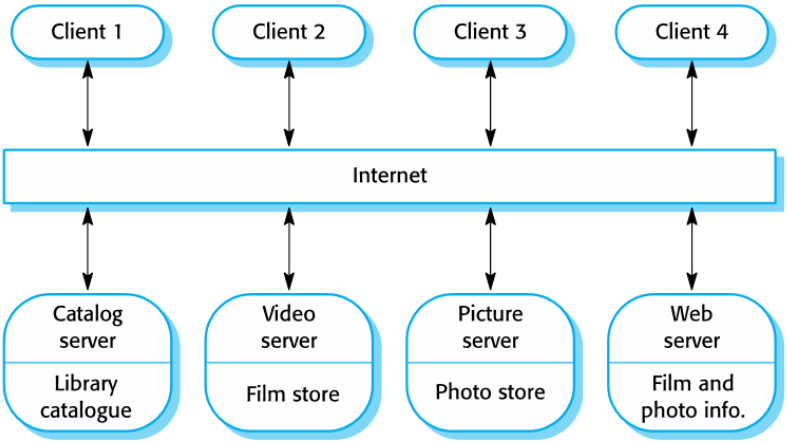
- Пример Слика 6.9 е пример за IDE каде компонентите користат складиште за информации за дизајн на системот. Секоја софтверска алатка генерира информации кои потоа се достапни за употреба од други алатки.
- Кога се користи - треба да ја користите оваа шема кога имате систем во кој се создаваат големи количини на информации што треба да се чуваат подолго време. Може да ги користите и во системи управувани со податоци кога вклучувањето на податоците во складиштето активира акција или алатка.
- Предности - Компонентите можат да бидат независни - не треба да знаат за постоењето на други компоненти. Промените направени од една компонента можат да бидат пропагирани на сите компоненти. Сите податоци можат да се управуваат доследно (на пр., Бекап направени во исто време) бидејќи сето тоа е на едно место.
- Недостатоци - Складиштето е единствена точка на неуспех, така што проблемите во складиштето влијаат на целиот систем. Може да биде неефикасност во организирањето на целата комуникација преку складиштето. Дистрибуирањето на складиштето на повеќе компјутери може да биде тешко.

Архитектура на складиште на IDE



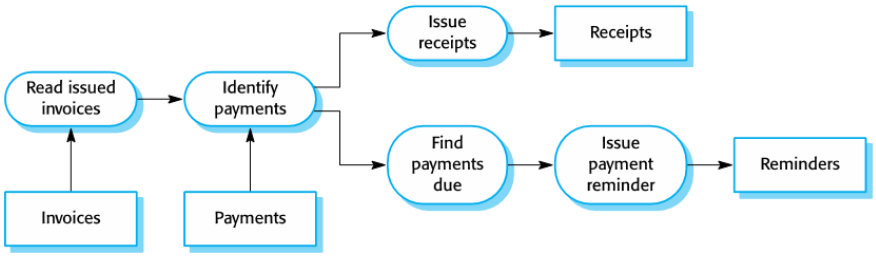
Клиент-сервер архитектура

- Дистрибуиран системски модел кој покажува како податоците и обработката се дистрибуираат низ низа компоненти. Ап Може да се спроведува на еден компјутер.
- Збир на самостојни сервери кои обезбедуваат специфични услуги, како што се печатење, управување со податоци, итн.
- Збир на клиенти кои ги повикуваат овие услуги.
- Мрежа што им овозможува на клиентите пристап до серверите.
- Опис - вВо архитектурата клиент-сервер, функционалноста на системот е организирана во услуги, при што секоја услуга се испорачува од посебен сервер. Клиенти се корисници на овие услуги и пристапуваат до серверите што ги користат.
- Пример Слика 6.11 е пример за филм и видео / ДВД библиотека организирана како сервер-систем на клиент-сервер.
- Кога се користи - Користејќи се кога податоците во заедничката база на податоци треба да се пристапи од голем број на локации. Бидејќи серверите можат да се реплицираат, може да се користат и со варијаблата на оптоварувањето на системот.
- Предности - Главната предност на овој модел е тоа што серверите можат да се дистрибуираат преку мрежа. Општата функционалност (на пр., Услуга за печатење) може да биде достапна за сите клиенти и нема потреба да се спроведува со сите услуги.
- Недостатоци - Секоја услуга е единствена точка на неуспех, толку подложна на негирање на напади на услуги или откажување на серверот. Перформансите може да бидат непредвидливи затоа што зависи од мрежата, како и од системот. Може да бидат проблеми со управувањето ако серверите се во сопственост на различни организации.



Архитектура на цевки и филтрирање

- Функционалните трансформации ги обработуваат нивните влезови за производство на излези.
- Може да биде наведен како модел на цевки и филтри (како во школка на UNIX).
- Варијантите на овој пристап се многу вообичаени. Кога трансформациите се последователни, ова е последователен модел кој се користи во системите за обработка на податоци.
- Не е баш погоден за интерактивни системи
- Опис - Обработката на податоците во системот е организирана така што секоја компонента за обработка (филтерот) е дискретна и извршува еден вид трансформација на податоците. Податоците течат (како во цевката) од една компонента до друга за процесирање.
- Пример - Слика 6.13 е пример за систем на цевки и филтрирање што се користи за обработка на производите.
- Кога се користи - Обично се користи во апликациите за обработка на податоци (и серија и трансакција-базирани) каде влезовите се обработуваат во одделни фази за да се генерираат излезни излези.
- Предности - Лесно да се разберат и поддржуваат повторна употреба на трансформацијата. Стилот на работа се совпаѓа со структурата на многу деловни процеси. Еволуцијата со додавање на трансформации е јасна. Може да се имплементира како систем за секвенцијално спроведување на курсот.
- Недостатоци - Форматот за трансфер на податоци треба да се договори помеѓу комуникацијата на трансформациите. Секоја трансформација мора да го анализира својот влез и да го распарчи својот излез во договорената форма. Ова го зголемува надземниот систем и може да значи дека е невозможно повторно да се употребат функционални трансформации, кои се користат со компатибилидирани структури.



Архитектури на апликации

- Системите за апликација се дизајнирани да одговорат на организацијата.
- Бидејќи деловните субјекти имаат многу заедничко, нивните апликативни системи, исто така, имаат тенденција да имаат заедничка архитектура што ги одразува барањата за апликација.
- Општа архитектура за апликации е архитектура за еден вид софтверски систем што може да се конфигурира и прилагодува за да создаде систем што одговара на специфични барања.

Употреба на апликациски архитектури

- Како појдовна точка за архитектонски дизајн.
- Како список за проверка на дизајнот.
- Како начин за организирање на работата на тимот за развој.
- Како средство за проценка на компонентите за повторна употреба.
- Како речник за зборување за типови апликации.

Примери на типови на апликации

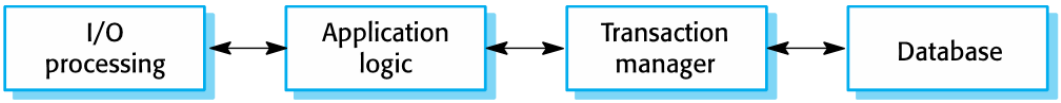
- Апликации за обработка на податоци
  - Апликации управувани со податоци кои обработуваат податоци во серии без експлицитна интервенција на корисникот за време на обработката.
- Апликации за обработка на трансакција
  - Централизирани апликации за податоци кои ги обработуваат барањата на корисниците и ги ажурираат информациите во системската база на податоци.
- Системи за обработка на настани
  - Апликации кога системските активности зависат од толкување на настаните од околината на системот.
- Системи за обработка на јазик
  - Апликации во кои намерите на корисниците се наведени на формален јазик, обработен и толкуван од системот.

Примери за типот на апликација

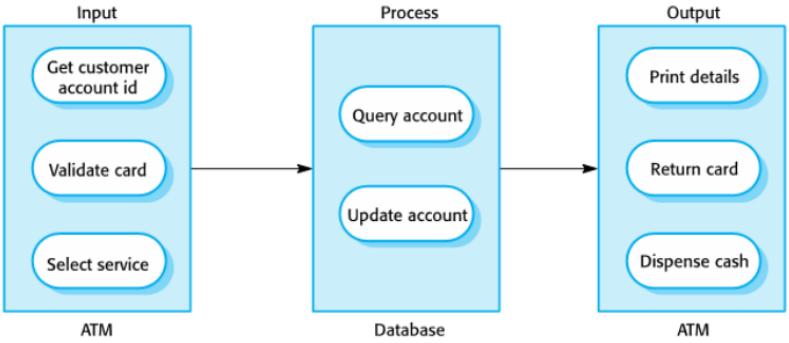
- Двете широко користени архитектури за генерички апликации се системи за обработка на трансакции и системи за обработка на јазик.
- Системи за обработка на трансакции
  - Системи за електронска трговија;
  - Системи за резервација.
- Системи за обработка на јазик
  - Компајлови;
  - Команда толкувачи

Системи за обработка на трансакции

- Процесирајте ги барањата на корисникот за информации од базата на податоци или барања за ажурирање на базата на податоци.
- Од гледна точка на корисникот, трансакцијата е:
  - Секоја кохерентна низа на операции што задоволува цел;
  - На пример - утврдете го времето на летови од Лондон до Париз.
- Корисниците прават асинхрони барања за услуга, кои потоа ги обработува менаџер на трансакција.



Софтверска архитектура на АТМ систем



Архитектура на информациски системи

- Информациските системи имаат генеричка архитектура која може да се организира со слоевита архитектура.
- Овие се системи засновани врз трансакција, бидејќи интеракцијата со овие системи генерално вклучува трансакции со бази на податоци.
- Слоевите вклучуваат:
  - Кориснички интерфејс
  - Кориснички комуникации
  - Враќање на информации
  - Системска база на податоци

Информациони системи базирани на веб.

- Системите за управување со информации и ресурси сега обично се мрежни системи, каде корисничките интерфејси се спроведуваат со помош на веб прелистувач.
- На пример, системите за е-трговија се системи за управување со ресурси базирани на Интернет, кои прифаќаат електронски нарачки за стоки или услуги и потоа организираат достава на овие добра или услуги на клиентот.
- Во системот за е-трговија, слојот специфичен за апликацијата вклучува дополнителна функционалност за поддршка на 'количката' во која корисниците можат да постават голем број на артикали во одделни трансакции, а потоа да плаќаат за сите заедно во една трансакција.

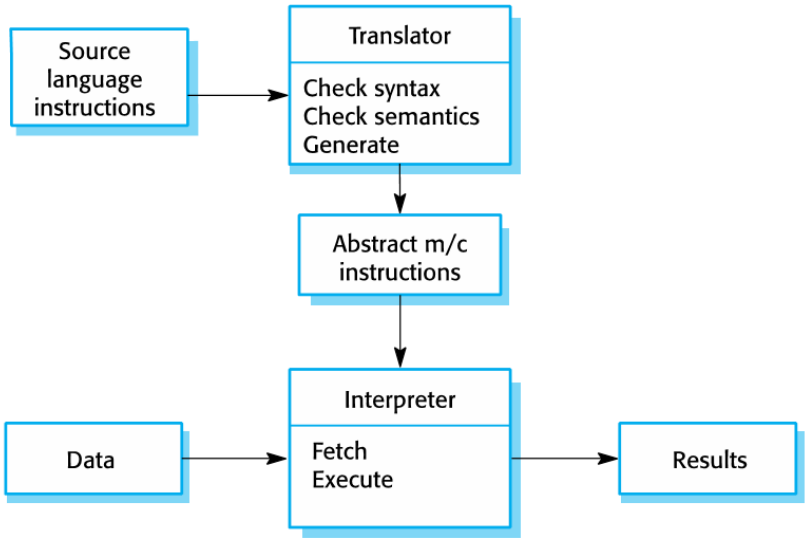
Имплементација на серверот

- Овие системи честопати се имплементираат како мулти-ниво на клиентски сервер / архитектури
  - Веб-серверот е одговорен за сите кориснички комуникации, при што корисничкиот интерфејс е имплементиран со помош на веб прелистувач;
  - Серверот за апликации е одговорен за спроведување на логика специфична за апликацијата, како и за чување информации и барања за пребарување;
  - Серверот за бази на податоци пренесува информации до и од базата на податоци и се справува со управувањето со трансакциите.

Системи за обработка на јазик

- Прифатете природен или вештачки јазик како влез и генерирајте друга застапеност на тој јазик.
- Може да вклучи толкувач да дејствува според упатствата на јазикот што се обработува.
- Се користи во ситуации кога најлесен начин да се реши проблем е да се опише алгоритм или да се опишат податоците за системот
  - Методи за описи на алатки за процесирање, правила за методи и сл. и генерирање алатки.

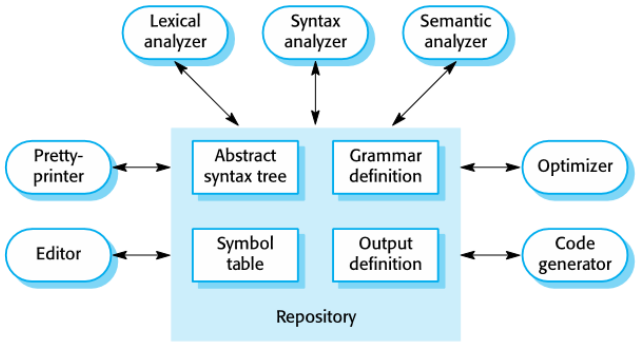
Архитектура на систем за обработка на јазик



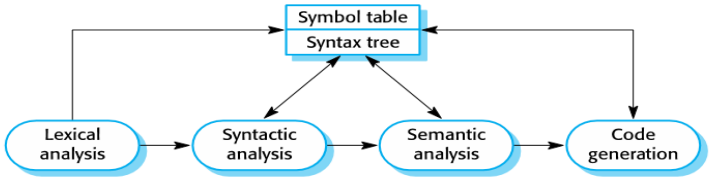
Компоненти на компајлерот

- Лексички анализатор, кој зема знаци за внесување јазик и ги претвора во внатрешна форма.
- Табела со симболи, која содржи информации за имињата на субјектите (варијабли, имиња на класи, имиња на предмети, итн.) Користени во текстот што се преведува.
- Синтаксички анализатор, која ја проверува синтаксата на јазикот што се преведува.
- Синтаксичко дрво, што е внатрешна структура што ја претставува програмата што се составува.
- Семантички анализатор што користи информации од синтаксата и табелата со симболи за да ја провери семантичката коректност на текстот за јазик за внесување.
- Генератор на кодови што го „шетаат“ синтаквото дрво и генерираат апстрактен код на машината

A repository architecture for a language processing system



A pipe and filter compiler architecture



Дизајн на ниво на компоненти

Што е компонента?

- Компонента: Модуларен градежен блок за компјутерски софтвер
- Дефиниција на компонента од OMG
  - „… модуларен, распоредлив и заменлив дел од асистемот што опфаќа имплементација и изложува збир на интерфејси.
- Компоненти
  1) Населете ја архитектурата на софтверот
  2) Играјте улога во постигнување на целите и барањата на системот што треба да се изгради
  3) Мора да комуницираат и да соработуваат со други компоненти и со субјекти (други системи, уреди, луѓе) кои постојат надвор од граници на софтверот

Компоненти - Гледна точка

- ОО преглед: компонента содржи збир на работни класи
- Конвенционален преглед: компонента содржи логика за обработка, внатрешни структури на податоци кои се потребни за спроведување на логиката на обработка и интерфејс што овозможува компонентата да се повикува и да се пренесат податоците до него
- Поглед поврзан со процесите: користете ги постојните компоненти на софтверот или моделите на дизајнирање, со избирање на компоненти или модели на дизајнирање од каталог

Компонента: ОО преглед

- компонентата содржи збир на часови за соработка
  - Секоја класа во рамките на компонентата е целосно разработена за да вклучува сите атрибути и операции што се релевантни за нејзино спроведување.
  - Сите интерфејси што им овозможуваат на часовите да комуницираат и да соработуваат со други класи на дизајни, исто така, мора да бидат посочени.
- За да дефинираме компонента, започнуваме со моделот на барања и елаборирани класи за анализа (за компоненти кои се однесуваат на доменот на проблемот) и класи на инфраструктура (за компоненти кои обезбедуваат услуги за поддршка за проблемот домен).

- Пример во PrintShop

  - Размислете за создавање софтвер за софистицирана продавница за печатење.
  - Софтвер за печатење PrintShop
    Целта е да се соберат барањата на клиентот во предниот дел, да се чине печатење и потоа да се предаде работата на автоматски производствен погон.
  - За време на инженерството на барања, беше изведена класа за анализа наречена PrintJob со атрибути и операции.

  - Изработка на дизајнерска компонента

  - PrintJob има две интерфејси, computeJob, што овозможува способност за чистење на работното место и иницијатива Job, која ја минува работата заедно во производниот објект.

  - Дизајн на ниво на компоненти започнува во овој момент

- Деталите за компонентата PrintJob мора да бидат образложени за да се обезбедат доволно информации за водење на имплементацијата.

- Оригиналната класа на анализа е разработена за да ги ослободи сите атрибути и операции потребни за спроведување на класата како компонента PrintJob.

- Елаборираната дизајн класа PrintJob содржи подетални информации за атрибути, како и проширен опис на операциите потребни за спроведување на компонентата

- Интерфејсите computeJob и initiateJob значат комуникација соработка со други компоненти
  - Операцијата computePageCost () (дел од компјутерскиот интерфејс computeJob) може да соработува со компонентата PricingTable што содржи информации за цени за работа.
  - Операцијата checkPriority () (дел од почетниот интерфејс за активирање) може да соработува со компонентата JobQueue за да се утврдат видовите и приоритетите на работните места што во моментот чекаат за производство.

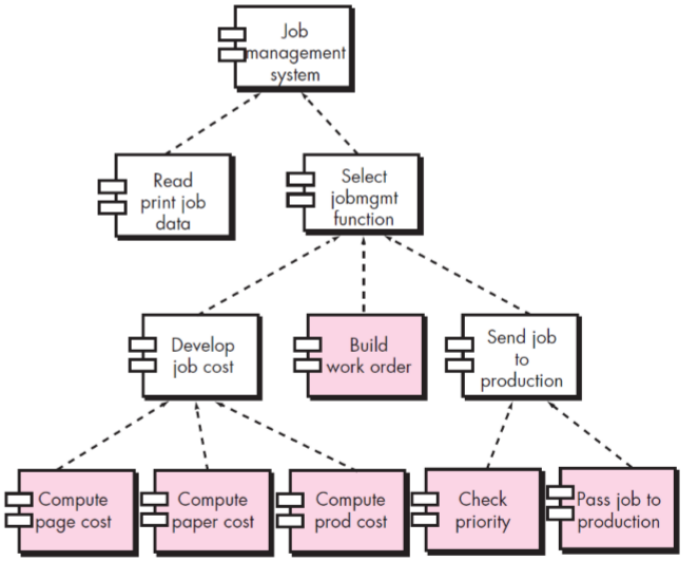
Компонента: Преглед на ОО

- Оваа елаборатска активност се применува на секоја компонента дефинирана како дел од архитектонскиот дизајн.
- Откако ќе се заврши, понатамошна елаборација се применува на секој атрибут, операција и интерфејс.
- Мора да бидат наведени структурите на податоци соодветни за секој атрибут.
- Дизајнирани се алгоритмички детали потребни за спроведување на логиката на обработка поврзана со секоја операција.
- Конечно, дизајнирани се механизмите потребни за спроведување на интерфејсот
- За софтвер ориентиран кон објект, ова може да опфати опис на сите пораки што се потребни за да се изврши комуникација помеѓу предметите во системот.

Компонента: Традиционален преглед

- компонентата е функционален елемент на програмата што вклучува…
  1) логика на обработка,
  2) внатрешни структури на податоци што се потребни за спроведување на логиката на обработка, и
  3) интерфејс што овозможува компонентата да се повикува и податоците да се пренесат на истата.
- Традиционална компонента, исто така наречена модул, живее во рамките на софтверската архитектура и служи на една од трите важни улоги:
  1) контролна компонента која ја координира поканата на сите други компоненти на доменот на проблемот,
  2) компонента на доменот на проблемот што имплементира целосна или делумна функција што ја бара клиентот, или
  3) инфраструктурна компонента која е одговорна за функции кои ја поддржуваат процедурата потребна во доменот на проблемот.
- Како компоненти ориентирани кон предмети, традиционалните компоненти на софтверот се изведени од моделот за анализа.
- Во овој случај, сепак, компонентен елемент за елаборирање на моделот за анализа служи како основа за деривацијата
- Секоја компонента претставена во хиерархијата на компонентите се мапира во модерна хиерархија
- Контролните компоненти (модули) престојуваат близу горниот дел од хиерархијата (програмска архитектура)
- Компонентите на доменот на проблемот имаат тенденција да живеат кон дното на хиерархијата.

Традиционален поглед – пример во PrintShop



- Секое поле претставува софтверска компонента.
- Забележете дека засенчените кутии се еднакви во функција на операциите дефинирани за класата PrintJob.
- Во овој случај, сепак, секоја операција е претставена како посебен модул што се повикува.
- Други модули се користат за контрола на обработката и затоа се контролни компоненти.

Традиционален поглед - Дизајн на ниво на компоненти

- За време на дизајнот на ниво на компонентите, секој модул е обработен
- Интерфејсот на модулот е дефиниран експлицитно. Тоа е, претставено е секој податок или контролен предмет што тече низ интерфејсот.
- Дефинирани се структурите на податоци што се користат внатрешно во модулот.
- Алгоритмот што му дозволува на модулот да ја оствари својата наменета функција е дизајниран со помош на пристапот за чекор дополнување
- Однесувањето на модулот понекогаш се претставува со помош на државен дијаграм

Традиционален поглед - Дизајн на ниво на компоненти – пример во PrintShop

- Размислете за модулот ComputePageCost
  - Целта на овој модул е да се пресмета перспективата на трошоците за печатење заснована врз спецификациите дадени од корисникот.
- Потребни се податоци за извршување на оваа функција:
  - број на страници во документот, вкупен број на документи што треба да се произведат, печатење на едно или двострано ниво, барања за боја и барања за големина.
- Овие податоци се доставуваат до ComputePageCost преку интерфејсот на модулот.
- ComputePageCost ги користи овие податоци за да утврди цена на страница што се заснова на големината и сложеноста на работата
- Цената на страницата е обратно пропорционална со големината на работата и е директно пропорционална со сложеноста на работата.
- Модулот ComputePageCost пристапува до податоците со повикување на модулот getJobData, кој овозможува да се пренесат сите релевантни податоци до компонентата, и интерфејс за база на податоци, пристапCostsDB, што му овозможува на модулот пристап до базата на податоци што ги содржи сите трошоци за печатење.
- Како што продолжува дизајнот, ComputePageCost модулот е обработен за да обезбеди детали за алгоритмот и детали за интерфејсот
- Деталите на алгоритмот можат да бидат претставени со помош на текстот псевдокод, прикажано на сликата или со UML дијаграм за активност.
- Интерфејсите се претставени како збирка на предмети или предмети за влезни и излезни податоци.

- Изработката на дизајнот продолжува сè додека не се дадат доволно детали за да се води конструкцијата на компонентата.

Компонента: Поглед поврзан со процесите

- Објектно-ориентираните и традиционалните погледи на дизајнот на ниво на компоненти претпоставуваат дека компонентата е дизајнирана од нула.
- Тоа е, ние треба да создадеме нова компонента заснована на спецификациите добиени од моделот на барања.
- Во текот на изминатите две децении, заедницата за софтверско инженерство ја нагласи потребата за градење системи што ги користат постојните софтверски компоненти или моделите на дизајнирање.
- Во суштина, каталог на докажан дизајн или компоненти на ниво на код е достапен за нас, како што се одвива со дизајнерската работа.
- Избираме компоненти или модели на дизајнирање од каталогот и ги користиме за да ја населиме архитектурата.
- Бидејќи овие компоненти се создадени со повторна употреба, целосен опис на нивниот интерфејс, функциите (итеите) што ги извршуваат и комуникацијата и соработката што тие ги бараат се достапни за нас.

Дизајнирање на класа – базирани компоненти

- Дизајн на ниво на компоненти се базира на информации развиени како дел од моделот на барања и претставен како дел од архитектонскиот модел
- За пристапот кон софтверското инженерство на ОО, дизајнот на ниво на компоненти се фокусира на изработка на специфични класи на домен на проблемот и дефинирање и рафинирање на инфраструктурата класи содржани во моделот на барања.
- Деталниот опис на атрибутите, операциите и интерфејсите што ги користат овие класи се деталите за дизајнот што се бараат како претходник на градежната активност.

Основни принципи на дизајнирање

- Принцип на отворено затворено (OCP). „Модулот [компонента] треба да биде отворен за проширување, но затворена форматификација.
- Принцип на замена на Лисков (ЛСП). „Подкласите треба да бидат заменливи за нивните базени за чаши.
- Принцип на инверзија во зависност (ДИП). „Зависи од апстракциите. Не зависи од бетони “.
- Принцип за поделба на интерфејсот (ISP). „Многу интерфејси специфични за клиенти се подобри од една општа намена.

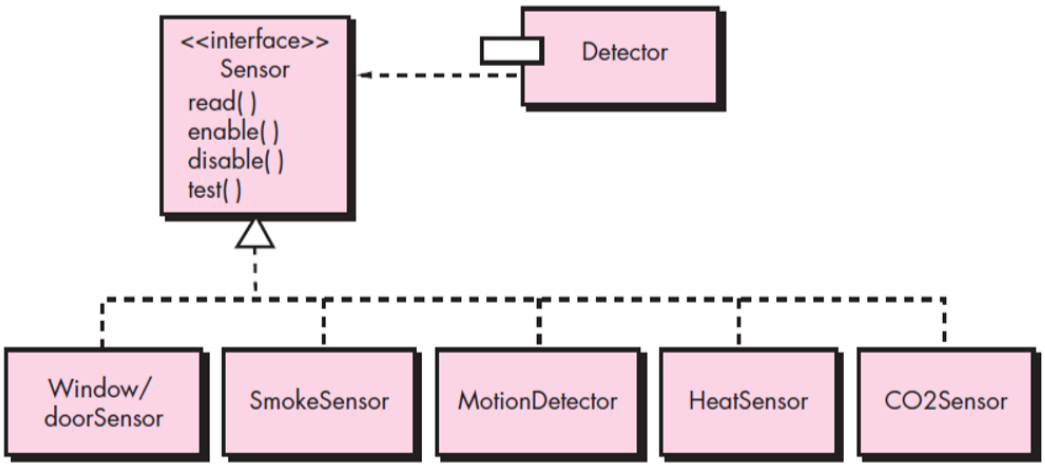
Во многу случаи, индивидуалните компоненти или класи се организираат во под-системи или пакети

- Принцип на еквивалентност на повторна употреба на ослободување (РЕП). „Гранулот на повторна употреба е гранул за ослободување“.
- Заедничкиот принцип на затворање (ССР). „Класите што се менуваат заедно припаѓаат заедно“.
- Заедничкиот принцип за повторна употреба (ЦРП). „Класите што не се користат повторно не треба да се групираат заедно.“

Отворен затворен принцип (OCP)

- Ние треба да ја специфицираме компонентата на начин што ќе овозможи да се прошири (во рамките на функционалниот домен на кој се обраќа) без потреба да се направат внатрешни (код или ниво на логика) модификација на самата компонента.
- За да го постигнеме ова, ние создаваме апстракции што служат како тампон помеѓу функционалноста што веројатно ќе се прошири и самата дизајн класа.

Пример за OCP



- Интерфејсот на сензорите претставува постојан преглед на сензорите до компонентата на детектор. Ако се додаде нов вид сензор, не е потребна промена за класата Детектор (компонента).

Принцип на замена на Лисков

- компонентата што користи основна класа треба да продолжи да функционира правилно ако наместо тоа се предава класа добиена од основната класа.
- LSP бара секоја класа добиена од основна класа да го исполни секој имплицитен договор помеѓу основната класа и компонентите што го користат.
- „договор“ е
  - предуслов што мора да биде вистина пред компонентата да користи основна класа и
  - пост-услов што треба да биде вистина откако компонентата користи основна класа.

Принципот на инверзија на зависноста (DIP)

- Како и OCP, апстракциите се место каде што дизајнот може да се продолжи без голема компликација.
- Колку повеќе компонента зависи од другите бетонски компоненти (наместо од апстракции како што е интерфејс), толку е потешко да се прошири.

Принцип за поделба на интерфејсот (ISP)

- Постојат многу случаи во кои повеќе компоненти на клиентот ги користат операциите обезбедени од класата на серверот.
- интернет провајдер сугерира дека треба да создадеме специјализиран интерфејс за да им служиме на секоја главна категорија клиенти.
- Само оние операции што се релевантни за одредена категорија клиенти треба да бидат наведени во интерфејсот за тој клиент.
- Ако повеќе клиенти бараат исти операции, треба да бидат наведени во секоја од специјализираните интерфејси.

ISP – пример во SafeHome Project

- За безбедносните функции, FloorPlan се користи само за време на конфигурациските активности и ги користи операциите placeDevice (), showDevice (), groupDevice () и removeDevice () за да ги постави, покаже, групира и отстранува сензорите од подот.
- Функцијата за набудување на SafeHome ги користи четирите операции наведени за безбедност, но исто така бара посебни операции за управување со фотоапарати: showFOV () и showDeviceID ().

- ISP сугерира дека компонентите на клиентот од двете функции „SafeHome“ имаат специјализирани интефејси дефинирани само за нив.
- Интерфејсот за безбедност: опфатени се само местата за операциии: placeDevice(), showDevice(), groupDevice(), and removeDevice()
- Интерфејс за набљудување: вметнете ги операциите placeDevice (), showDevice (), groupDevice () и removeDevice (), заедно со showFOV () и showDeviceID ().

Принцип на ослободување на еквивалентност на повторна употреба (РЕП).

- Кога класи или компоненти се дизајнирани за повторна употреба, постои имплицитен договор што се воспоставува помеѓу развивачот на субјектот што може да се употреби за повторна употреба и луѓето што ќе го користат.
- Наместо да се обраќате на секоја класа поединечно, честопати се препорачува да ги групирате повторните часови во пакетите што можат да се управуваат и контролираат бидејќи се развиваат нови верзии.

Принцип на заедничко заклучување (ССР).

- Класите треба да се пакуваат кохезивно
- Кога часовите се пакуваат како дел од дизајнот, треба да се осврнат на истата функционална или област на однесување.
- Кога некоја карактеристика за таа област мора да се промени, веројатно е дека само оние класи во рамките на пакетот ќе бараат модификација.

Принцип на заедничка повторна употреба (CRP).

- Кога една или повеќе класи во рамките на пакетот се менуваат, бројот на ослободување на пакетот се менува
- Сите други класи или пакети што се потпираат на променетиот пакет, сега мора да се ажурираат до најновото издание на пакетот и да бидат тестирани за да се осигури дека новото издание работи без инциденти.
- Само класи што се користат повторно треба да бидат вклучени во пакетот

Упатства за дизајн на ниво на компоненти

- Компоненти ❑
  - Конвенциите за именување треба да се воспостават за компонентите што се наведени како дел од архитектонскиот модел, а потоа да се рафинираат и елаборираат како дел од моделот на ниво на компоненти
- Интерфејси
  - Интерфејсите обезбедуваат важни информации за комуникација и соработка (како и за да ни помогнат да постигнеме ОРС)
- Зависности и наследство
  - добра идеја е да се моделираат зависностите од лево кон десно и наследството од дното (изведените класи) до врвот (основни класи).

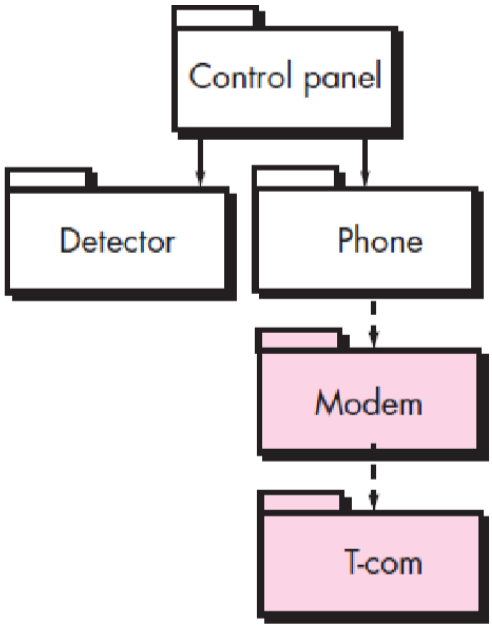
Кохезија

- Конвенционален поглед:
  - „едноумието“ на компонентата
- ОО преглед:
  - кохезијата подразбира дека компонента или класа опфаќа само атрибути и операции кои се тесно поврзани едни со други и со самата класа или компонента

- Ниво на кохезија
  - Функционално: Ова ниво на кохезија е изложено пред се со операции, ова ниво на кохезија се јавува кога компонентата врши насочена пресметка и потоа враќа резултат.
  - Слој: Изложена од пакети, компоненти и класи, овој вид кохезија се јавува кога повисок слој пристапува до услугите на долниот слој, но долните слоеви не пристапуваат до повисоки слоеви.
  - Комуникациски: Сите операции што пристапуваат до истите податоци се дефинирани во една класа. Во принцип, ваквите класи се фокусираат исклучиво на односните податоци, пристапувајќи до нив и ги чуваат.

Ниво – слој кохезија ПРИМЕР - SafeHome

- Барањето за безбедносна функција за да оствари појдовен телефонски повик ако се почувствува алармот.
- засенчените пакувања содржат компоненти на инфраструктурата.
- Пристапот е од пакетот на контролниот панел надолу.



- Класите и компонентите што покажуваат функционална, слој и комуникациска кохезија се релативно лесни за спроведување, тестирање и одржување.
- Треба да се обидеме да ги постигнеме овие нивоа на кохезија секогаш кога е можно.
- Меѓутоа, важно е да се напомене дека прашањата за прагматичен дизајн и спроведување понекогаш нè принудуваат да се определиме за пониско ниво на кохезија

Спојка

- Конвенционален поглед:
  - Степенот до кој компонентата е поврзана со другите компоненти и со надворешниот свет
- ОО поглед:
  - квалитативна мерка за степенот до кој се поврзани часовите едни на други
- Важен цел во дизајнот на ниво на компонента е да се одржи спојувањето што е можно пониско

Спојки на класи:категории

- Спојување на содржина: Се појавува кога една компонента „заштитено ги модифицира податоците што се внатрешни на друга компонента“
  - Ова го крши криењето на информациите - основен концепт за дизајн.
- Контролна спојка: Се појавува кога операцијата А () повикува операција
  - В () и го пренесува контролното знаме на В.
  - Контролното знаме потоа го „насочува“ логичкиот проток во В.

- Проблемот со оваа форма на спојување е тоа што не е поврзана промена во Б може да резултира во неопходност да се смени значењето на контролното знаме што поминува

- Екстерно спојување: Се појавува кога компонента комуницира или соработува со компонентите на инфраструктурата
  - (на пр., функции на оперативниот систем, можност за база на податоци, телекомуникациски функции). Иако овој вид спојка е неопходен, тој треба да биде ограничен на помал број компоненти или класи во рамките на асистемот

Дизајн на ниво на компоненти-I

- Чекор 1. Идентификувајте ги сите класи на дизајнирање што одговараат на доменот на проблемот.
- Чекор 2. Идентификувајте ги сите класи на дизајнирање кои одговараат на доменот на инфраструктурата.
- Чекор 3. Да ги разработиме сите класи на дизајнирање кои не се стекнуваат како компоненти што можат да се користат за еднократно
  - Чекор 3а. Наведете детали за пораки кога соработуваат часови или компоненти.
  - Чекор 3б. Идентификувајте соодветни интерфејси за секоја компонента.

Развој на база на компоненти

- Во контекст на софтверското инженерство, повторна употреба е идеја и стара и нова.
- Програмерите ги користеа идеите, апстракциите и процесите уште од најраните денови на пресметување, но раниот пристап за повторна употреба беше ад хок
- Денес, сложените, висококвалитетни компјутерски системи мора да се градат во многу кратки временски периоди и да бараат поорганизиран пристап кон повторна употреба.
- Софтверското инженерство базирано на компоненти (CBSE) е процес што го нагласува дизајнирањето и изградбата на компјутерски системи базирани на употреба на „компоненти“ на софтвер за еднократно користење.
- Но, се појавуваат голем број прашања.
  - Дали е можно да се конструираат комплексни системи со нивно составување од каталог на компоненти што можат да се користат за еднократно?
  - Дали ова може да се постигне во рамките на трошоците и времето ефикасно работење?
  - Може ли да се воспостават соодветни стимулации за да ги охрабрат инженерите на софтвер да употребуваат повторна употреба отколку да реинвентни?
  - Дали раководството е подготвено да претрпи дополнителен трошок поврзан со создавање на компоненти што можат да се користат за еднократно?
  - ……. ➔ Одговор е ДА

Развој на база на компоненти: Доменско инженерство

- Целта на инженерството на домени е да идентификува, конструира, каталогизира и дистрибуира збир на софтверски компоненти кои имаат применливост на постојниот и идниот софтвер во одреден домен на апликација
- Да се воспостават механизми што овозможуваат софтверските инженери да споделуваат овие компоненти - за да ги користите - за време на работата на нови и постојни системи.
- Доменското инженерство вклучува три главни активности, анализи, градежништво и дисеминација.

Доменско инженерство

- Целокупниот пристап кон анализа на доменот често се карактеризира во контекст на објектно-ориентираното инженерство.
- Чекор 1. Дефинирајте го доменот што треба да се истражува.
- Чекор 2. Категоризирајте ги предметите извлечени од доменот.
- Чекор 3. Соберете репрезентативен примерок на апликации во доменот.

- Чекор 4. Анализирајте ја секоја апликација во примерокот и дефинирајте ги часовите за анализа.
- Чекор 5. Изгответе модел на барања за часовите.

Квалификација на компонентата, адаптација и состав

- Доменското инженерство обезбедува библиотека на еднократни компоненти што се потребни за софтверско инженерство базирано на компоненти.
- Некои од овие компоненти што можат да се употребат се користат, други можат да бидат извлечени од постојните апликации, а други може да се стекнат од трети страни.
- За жал, постоењето на еднократно компоненти не гарантира дека овие компоненти можат лесно или ефикасно да се интегрираат во архитектурата избрана за нова апликација.

Квалификација на компонентите

- Квалификацијата на компонентата осигурува дека кандидатската компонента ќе ја изврши потребната функција, правилно ќе се „вклопи“ во архитектонскиот стил наведен за системот и ќе ги прикаже карактеристиките за квалитет (на пр. Перформанси, сигурност, употребливост) што се потребни за апликацијата.
- Описот на интерфејс обезбедува корисни информации за работењето и користењето на софтверската компонента, но не ги обезбедува сите потребни информации за да се утврди дали предложената компонента, всушност, може да биде повторно користена ефективно во нова апликација.

Квалификација на компонентите: Фактори што треба да се земат во предвид

- Интерфејс за програмирање на апликации (API).
- Алатки за развој и интеграција потребни од компонентата.
- Барања за време на извршување, вклучувајќи употреба на ресурси (на пр. Меморија или складирање), тајминг или брзина и мрежен протокол.
- Барања за услуги, вклучително и интерфејси на оперативниот систем и поддршка од други компоненти.
- Безбедносни карактеристики, вклучително и контроли за пристап и протокол за автентикација.
- Вградени претпоставки за дизајн, вклучувајќи употреба на специфични нумерички или ненумерички алгоритми.
- Справување со исклучоци.

Адаптација на компонентата

- Во идеален амбиент, инженерството на домени создава библиотека со компоненти кои можат лесно да се интегрираат во архитектурата на апликации.
- Во реалноста, дури и откако компонентата е квалификувана за употреба во рамките на архитектурата за апликации, може да се појават конфликти во една или повеќе од само наведените области.
- За да се избегнат овие конфликти, понекогаш се користи техника за адаптација наречена обвивка за компоненти.
  - Завиткување со бела кутија
  - Завикување со сиво кутија
  - Завиткување со црна кутија
- Завиткување со бела кутија се применува кога софтверски тим има целосен пристап до внатрешниот дизајн и шифрата за компонентата (честопати не е случај, освен ако не се користат компонентите COTS со отворен извор),
  - завитката со бела кутија ги разгледува деталите за внатрешната обработка на компонента и прави измени на ниво на код за да се отстрани секој конфликт.
- Завитувањето со сиво кутија се применува кога библиотеката со компоненти обезбедува јазик за проширување на компонентата или API што овозможува да се отстранат или маскираат конфликтите.

- Завиткување со црни кутии бара воведување пред и после обработка на интерфејсот на компонентата за да се отстранат или маскираат конфликтите.

Состав на компонентата.

- Задачата на составот на компонентата е составува квалификувани, адаптирани и инженерски компоненти за да ја насели архитектурата основана за апликација.
- За да се постигне ова, мора да се воспостави инфраструктура за да се врзат компонентите во оперативен систем.
- Инфраструктурата (обично библиотека со специјализирани компоненти) обезбедува модел за координација на компонентите и специфичните услуги што овозможуваат компонентите да се координираат едни со други и да извршуваат заеднички задачи.

Стандарди за компонентен софтвер

- ОМГ / КОРБА. Групацијата за управување со предмети објави архитектура за брокер за обичен предмет (OMG / CORBA).
- Мајкрософт COM и .NET. Мајкрософт има развиено модел на компонентен предмет (COM) кој обезбедува спецификација за употреба на компоненти произведени од различни добавувачи во рамките на една апликација што работи под оперативниот систем Виндоус.
- Компоненти на Java JavaBeans. Системот за компоненти JavaBeans е преносна, платформно зависна CBSE инфраструктура развиена со употреба на програмскиот јазик Java.

Архитектонски неусогласеност

- Еден од предизвиците со кои се соочува распространетата повторна употреба е архитектонскиот неусогласеност
- Дизајнерот на еднократно компоненти често прави имплицитни претпоставки за околината во која е поврзана компонентата
- Овие претпоставки честопати се фокусираат на моделот за контрола на компонентите, природата на приклучните компоненти (интерфејси) , самата архитектонска инфраструктура и природата на градежниот процес
- Доколку овие претпоставки не се точни, се јавува архитектонска неусогласеност
- Сите дизајнерски концепти придонесуваат за создавање на софтверски компоненти кои можат да се употребат и да се спречи архитектонско несогласување
  - Апстракција, криење, функционална независност, рафинирање и структурно програмирање, заедно со објективно ориентирани методи, тестирање, гарантирање на квалитетот на софтверот (SQA) и коректност методи за верификација),
- Рано откривање на архитектонски неусогласеност може да се случи ако претпоставките на засегнатите страни се експлицитно документирани
- Употребата на модел-процес управуван од ризик ја нагласува дефиницијата на раните архитектонски прототипи и укажува на области на неусогласеност
- Поправката на архитектонски неусогласеност е често многу тешка без да се користат механизми како омоти или адаптери
- Понекогаш, неопходно е целосно да се редизајнирате интерфејс на компонентата или самата компонента за да се отстранат проблемите за спојување

Анализа и дизајн за повторна употреба

- Елементите на моделот на барања се споредуваат со описите на еднократните компоненти во процес што понекогаш се нарекува „појавување на спецификации“

- Ако совпаѓањето на спецификациите укажува на постоечка компонента што одговара на потребите на тековната апликација, можете да ја извадите компонентата од библиотека за повторна употреба (складиште) и користете ја во дизајнирање на нов систем.
- Ако компонентите не можат да се најдат (т.е., нема совпаѓање), се креира нова компонента.
- Токму во овој момент - кога ќе започнете да создавате нова компонента - треба да се земе предвид дизајнот за повторна употреба (DFR).
- DFR бара да примените концепти и принципи на цврст дизајн на софтвер. Но, карактеристиките на доменот на апликацијата исто така мора да бидат земени предвид.
- Основи за дизајн за повторна употреба:
  - Стандардни податоци. Ако доменот на апликации има стандардни структури на податоци во глобални компоненти, компонентата треба да биде дизајнирана да ги користи овие стандардни структури на податоци
  - Треба да се усвојат стандардни протоколи за интерфејс во рамките на доменот на апликацијата,
  - Архитектонски стил кој е соодветен за доменот може да послужи како образец за архитектонскиот дизајн на новости

Дизајн и имплементација

- Дизајн и имплементација на софтверот е фаза во процесот на софтверско инженерство во кое се развива извршен софтверски систем.
- Активностите за дизајнирање и спроведување на софтверот се непроменливи меѓусебно.
  - Дизајнот на софтверот е креативна активност во која ги идентификувате компонентите на софтверот и нивните односи, врз основа на барањата на клиентот. ¨
  - Имплементација е процес на реализирање на дизајнот како програма.

Изградба или купување

- Во широк спектар на домени, сега е можно да се купат системи надвор од полиците (COTS) кои можат да бидат прилагодени и прилагодени на барањата на корисниците.
  - На пример, ако сакате да спроведете систем на медицински записи, можете да купите пакет што веќе се користи во болниците. Може да биде поевтино и побрзо да се користи овој пристап наместо да се развива систем на конвенционален јазик за програмирање.
- Кога развивате апликација на овој начин, процесот на дизајнирање се грижи за тоа како да ги користите конфигурациските карактеристики на тој систем за да ги испорачате барањата на системот.

Објектно-ориентиран процес на дизајнирање

- Структурните процеси ориентирани кон објектите вклучуваат развој на голем број на различни модели на системот.
- Тие бараат многу напор за развој и одржување на овие модели и, за малите системи, ова можеби не е рентабилно.
- Сепак, за големите системи развиени од различни групи моделите за дизајн се важен комуникациски механизам.

Фази на процеси

- Постојат различни различни процеси ориентирани кон предмети, кои зависат од организацијата што го користи процесот.
- Заеднички активности во овие процеси вклучуваат:
  - Да се утврди контекстот и начините на употреба на системот;
  - Исцртување на системската архитектура;
  - Идентификувајте ги главните објекти на системот;

- Развој на модели за дизајн;
- Специфицирајте интерфејси за објекти.

➢ Процес илустриран овде користејќи дизајн за временска станица во пустината.

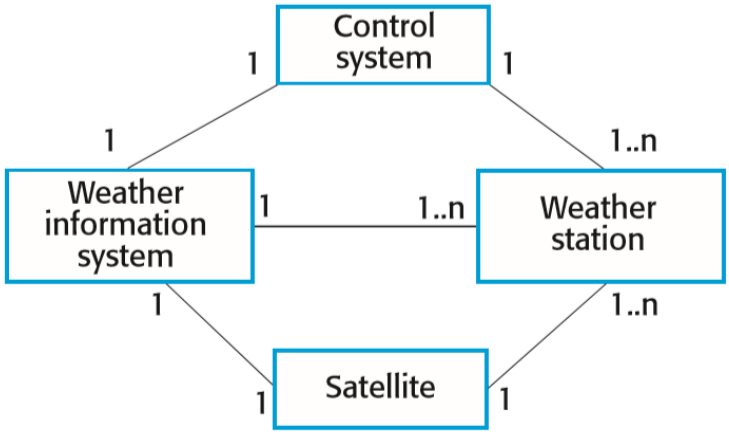
Контекст на системот и интеракции

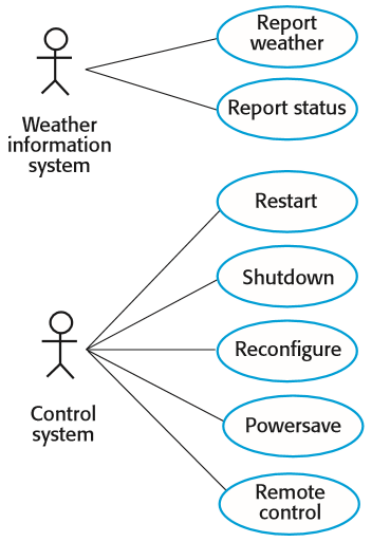
➢ Разбирањето на односите помеѓу софтверот што е дизајниран и неговото надворешно опкружување е неопходно за да се одлучи како да се обезбеди потребната функционалност на системот и како да се структурира системот за да комуницира со неговата околина.
➢ Разбирањето на контекстот исто така ви овозможува да ги поставите границите на системот. Поставувањето на системските граници ви помага да одлучите какви карактеристики се имплементираат во системот што е дизајниран и кои карактеристики се во другите придружни системи.

Контекст и модели на интеракција

➢ Системски контекст модел е структурен модел кој ги демонстрира другите системи во околината на системот што се развива.
➢ Модел на интеракција е динамичен модел кој покажува како системот комуницира со неговата околина како што се користи.

Контекст на системот за временската станица

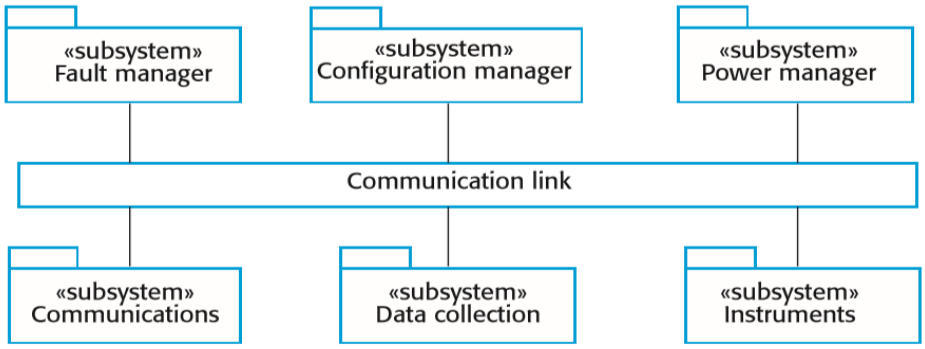




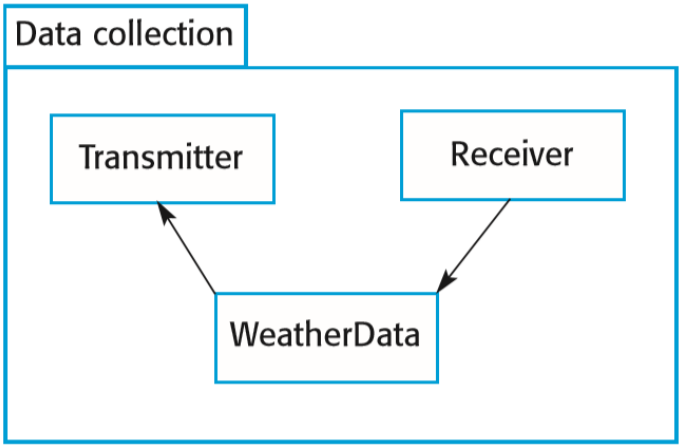
Архитектонски дизајн

➢ Откако ќе се разберат интеракциите помеѓу системот и неговата околина, овие информации ги користите за дизајнирање на архитектурата на системот.
➢ Вие ги идентификувате главните компоненти што го сочинуваат системот и нивните интеракции, а потоа можете да ги организирате компонентите користејќи архитектонска шема, како што е слоевит или клиент-сервер модел.
➢ Метеоролошката станица е составена од независни под-системи кои комуницираат со емитување пораки за заедничка инфраструктура.

Архитектура на високо ниво на станицата за време



Архитектура на систем за собирање на податоци



Идентификација на класата на објекти

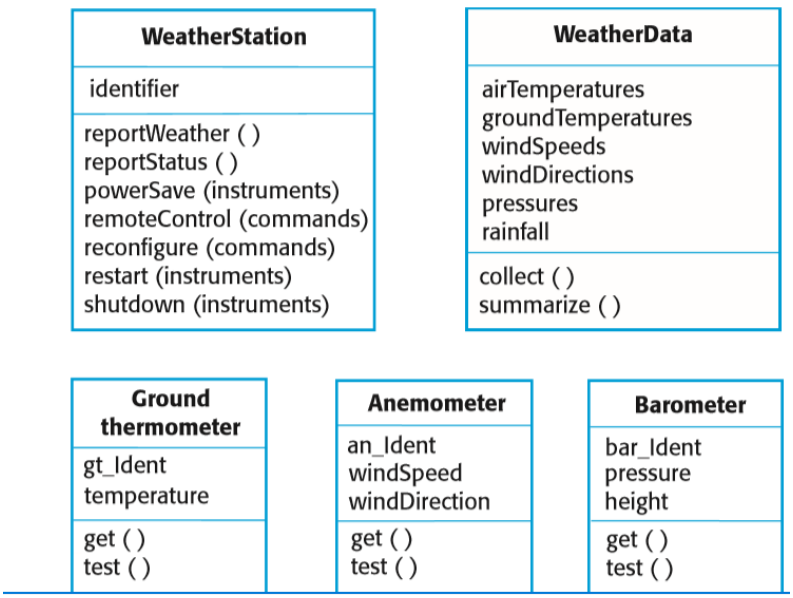
- Идентификувањето на класите на објекти е честопати тежок дел од дизајнот ориентиран кон објектите.
- Не постои „магична формула“ за идентификација на предмети. Се потпира на вештина, искуство и знаење на доменот на дизајнерите на системот.
- Идентификација на објектот е итеративен процес. Вие веројатно нема да го добиете тоа прв пат.

Пристапи кон идентификација

- Користете граматички пристап заснован на описот на системот за природен јазик.
- Базирајте ја идентификацијата за материјални работи во доменот на апликацијата.
- Користете пристап во однесувањето и идентификувајте ги предметите засновани врз она што учествува во какво однесување.
- Користете анализа базирана на сценарио. Идентификувани се предметите, атрибутите и методите во секое сценарио.

Класи на објекти на временската станица

- Идентификација на објекти од класа во системот за метеоролошка станица може да се заснова врз материјалниот хардвер и податоците во системот:
  - Земјишен термометар, анемометар, барометар - Предмет на домен објекти што се „хардверски“ објекти поврзани со инструментите во системот. .
  - Метеоролошка станица - Основен интерфејс на метеоролошката станица до неговата околина. Затоа ги одразува интеракциите утврдени во моделот на употреба случај.
  - Податоци за временските услови - ги опфаќа сумираните податоци од инструментите.



Дизајн модели

- Дизајн модели ги покажуваат предметите и класи класи и односите помеѓу овие субјекти.
- Постојат два вида на модерен дизајн:
  - Структурните модели ја опишуваат статичката структура на системот во однос на класи на објекти и врски.
  - Динамичките модели ја опишуваат динамичката интеракција помеѓу предметите.

Примери на модели за дизајнирање

- Модели на потсистеми кои покажуваат логички групирање на предмети во кохерентни подсистеми.
- Модели на секвенца кои ја покажуваат низата на интеракции на објектите.
- Модели на државни машини кои покажуваат како индивидуалните предмети ја менуваат својата состојба како одговор на настаните.
- Другите модели вклучуваат модели за случај на употреба, модели за агрегација, модели за генерализација итн.

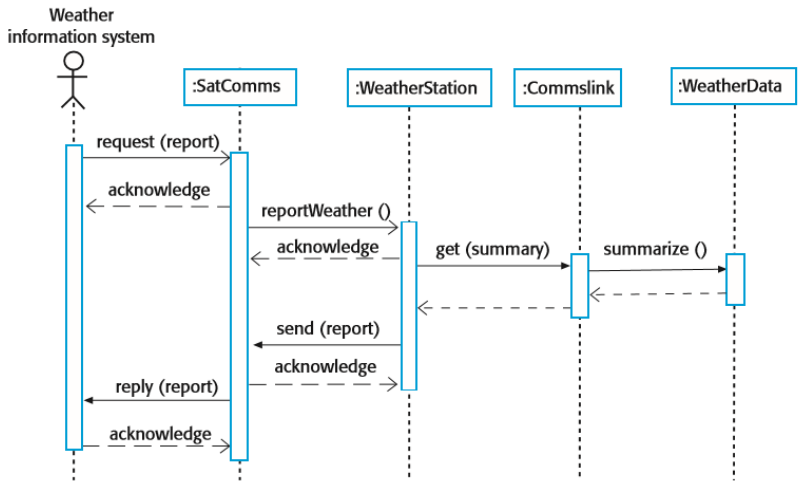
Под-системски модели

- Покажува како дизајнот е организиран во логички поврзани групи на предмети.
- Во UML, овие се прикажани со употреба на пакети со конструктивна структура. Ова е логичен модел. Вистинската организација на предметите во системот може да биде различна.

Секвентни модели

- Секвентните модели ја покажуваат низата на интеракција на објектите што се одвиваат
  - Објектите се распоредени хоризонтално преку горниот дел;
  - Времето е претставено вертикално, така што моделите се читаат одгоре до дното;
  - Интеракциите се претставени со означени стрели, Различни стилови на стрела претставуваат различни видови на интеракција;
  - Тенок правоаголник во животен век на објектот претставува време кога предметот е контролен предмет во системот.

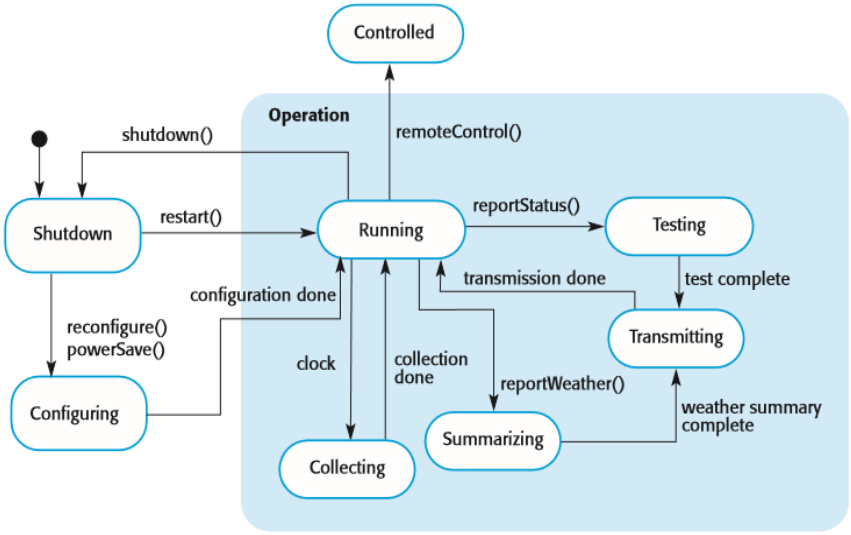
Секвентен дијаграм го опипува прибирањето на податоци



Дијаграми на состојба

- Дијаграмите за состојба се користат за да се покаже како предметите реагираат на различни барања за услуги и државните транзиции предизвикани од овие барања.
- Дијаграмите за состојба се корисни модели на високо ниво на систем или однесување на време на објектот.
- Обично не ви е потребен дијаграм за состојба за сите објекти во системот. Многу од предметите во системот се релативно едноставни и моделот за состојба додава непотребни детали во дизајнот.

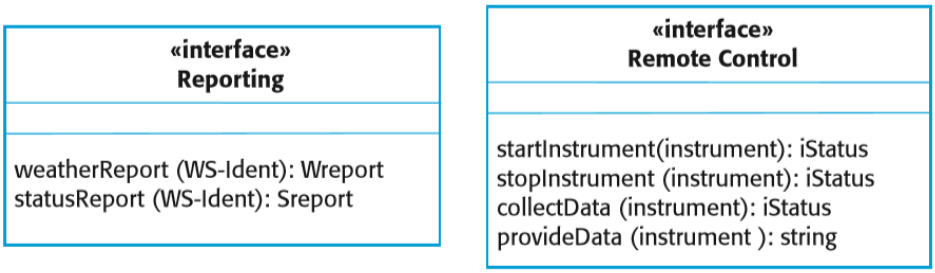
Дијаграм на состојба на временска станица



Спецификација на интерфејс

- Интерфејсите за објекти треба да бидат наведени така што предметите и другите компоненти можат да бидат дизајнирани паралелно.
- Дизајнерите треба да избегнуваат дизајнирање на застапеноста на интерфејсот, но треба да го кријат тоа во самиот предмет.
- Предметите можат да имаат неколку интерфејси што се гледишта за дадените методи.
- UML користи дијаграми за класи за спецификација на интерфејс, но може да се користи и Java.

Интерфејси на временска станица



Шеми на дизајнирање.

- Дизајн-шема е начин за повторна употреба на апстрактното знаење за проблем и за негово решавање.
- Модел е опис на проблемот и суштината на неговото решавање.
- Треба да биде доволно апстрактно за да се користи повторно во различни поставувања.
- Описите на моделите обично користат карактеристики ориентирани кон предмети, како што се наследство и полиморфизам.

Шеми

- Шеми и јазици на обрасци се начини да се опишат најдобрите практики, добри дизајни и да се доживее искуство на начин на кој е можно за другите да го користат ова искуство.

Елементи на шемите

- Име - идентификатор на значајна шема.
- Опис на проблемот.
- Опис на решението - Не конкретен дизајн, туку образец за идејно решение што може да се изврши инстант порака на различни начини.
- Последици - Резултатите и промените на примената на шемата.

Моделот на Обсервер

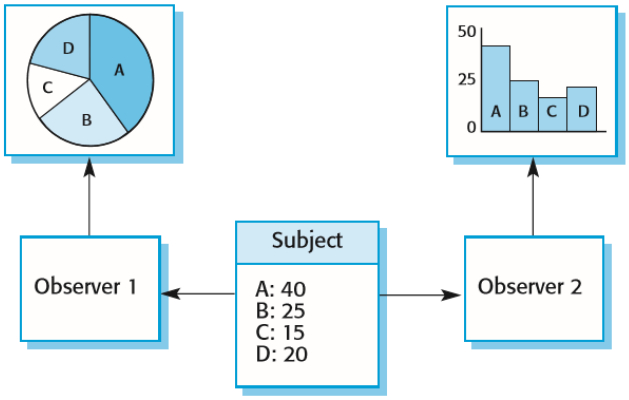
- Име - Набудувач.
- Опис - Одделува приказ на состојбата на предметот од самиот предмет.
- Опис на проблемот - Се користи кога се потребни повеќекратни прикажувања на состојба.
- Опис на решението - Погледнете го слајдот со описот UML.
- Последици - Оптимизациите за подобрување на перформансите на екранот се непрактични.

- Опис - Одделува приказ на состојбата на некој предмет од самиот предмет и дозволува да се обезбедат алтернативни дисплеи. Кога состојбата на предметот се менува, сите дисплеи автоматски се известуваат и ажурираат за да ја одразуваат промената.

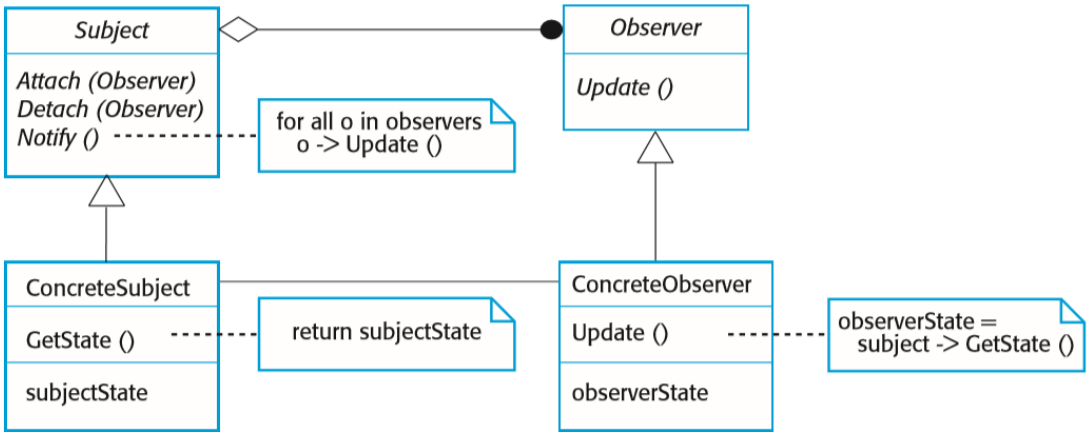
- Опис на проблемот - Во многу ситуации, треба да обезбедите повеќекратни прикази на информации за состојбата, како што е графички приказ и табеларен приказ. Не може да се знаат сите овие кога ќе се наведат информациите. Сите алтернативни презентации треба да поддржуваат интеракција и, кога ќе се смени состојбата, сите дисплеи мора да се ажурираат. Овој образец може да се користи во сите ситуации кога се потребни повеќе од еден формат на дисплеј за државни информации и кога не е неопходно за предметот што ги одржува државните информации да знае за специфичните формати на приказ што се користат.

- Опис на решението - Ова вклучува два апстрактни предмети, Субјект и Обсервер, и два бетонски предмети, БетонскиСубјект и БетонскиОбјект, кои ги наследуваат атрибутите на сродните апстрактни предмети. Апстрактните предмети вклучуваат општи операции кои се применливи во сите ситуации. Состојбата што треба да се прикаже се одржува во „ConcreteSubject“, кој наследува операции од Предмет дозволувајќи му додавање и отстранување на набудувачите (секој набудувач одговара на дисплеј) и да издава известување кога државата се сменила.
  Concrete Observer одржува копија од состојбата на ConcreteSubject и го спроведува интерфејсот Ажурирање () на Обсервер кој им овозможува на овие копии да се чуваат во чекор. ConcreteObserver автоматски ја прикажува состојбата и ги рефлектира промените секогаш кога државата ќе се ажурира.
- Последици - Предмет само го знае апстрактниот Набегудувач и не знае детали за конкретната класа. Затоа, постои минимална спојка помеѓу овие објекти. Поради овој недостаток на знаење, оптимизациите кои ги подобруваат перформансите на дисплејот се непрактични. Промените на оваа тема може да предизвикаат генерирање на сложени ажурирања на набудувачите, од кои некои не се потребни.

Повеќекратни дисплеи користејќи Обсервер шема



UML модел за Обсервер шема



Проблеми со дизајнот

- За да користите обрасци во вашиот дизајн, треба да препознаете дека секој проблем со дизајнот со кој се соочува може да има поврзана шема што може да се примени.
  - Кажете неколку предмети дека состојбата на некој друг предмет е променета (Шаблон на набудување).

- Наместете ги интерфејсите на голем број сродни предмети што честопати се развиваат постепено (шема на фасадата).
- Обезбедете стандарден начин за пристап до елементите во колекцијата, без оглед на тоа како се применува оваа колекција (Iteratorpattern).
- Овозможете ја можноста за проширување на функционалноста на постојната класа во времетраење (шема на украсување).

Проблеми имплементација

- Фокусот тука не е на програмирање, иако тоа е очигледно важно, но за други прашања поврзани со имплементацијата кои честопати не се опфатени во програмските текстови:
  - Повторна употреба - Повеќето современ софтвер е дизајниран со употреба на постојни компоненти или системи. Кога развивате софтвер, треба да искористите што е можно повеќе од постојниот код.
  - Управување со конфигурацијата - За време на процесот на развој, треба да водите сметка за многу различни верзии на секоја софтверска компонента во систем за управување со конфигурација.
  - Развој на таргетираниот на домаќинот - Производство на софтвер обично не се извршува на ист компјутер како околина за развој на софтвер. Наместо тоа, вие го развивате на еден компјутер (систем домаќин) и го извршувате на посебен компјутер (цел систем).

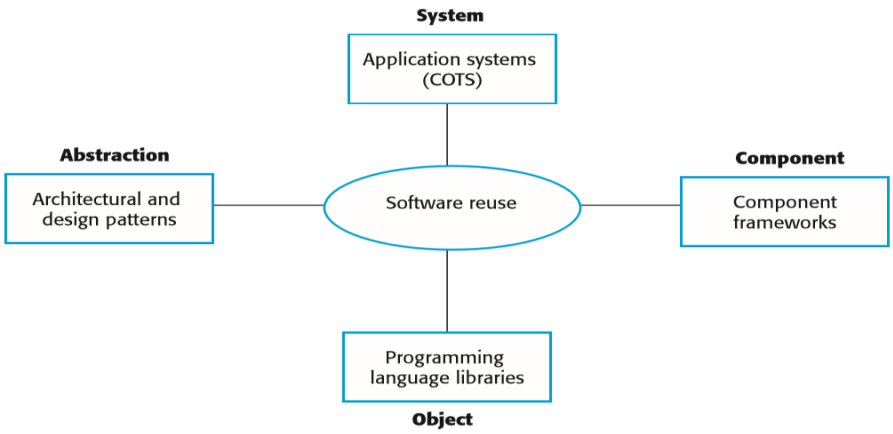
Повторна употреба

- Од 1960-тите до 1990-тите, повеќето нови софтвер беа развиени од нула, со пишување на целиот код на високо ниво на програмски јазик.
  - Единствената значајна повторна употреба или софтвер беше повторна употреба на функциите и предметите во библиотеките за програмирање јазик.
- Трошоците и притисокот во распоредот значат дека ваквиот пристап стана сè поневидлив, особено за комерцијалните и Интернет-системите.
- Се појави пристап за развој базиран околу повторна употреба на постојниот софтвер и сега генерално се користи за деловен и научен софтвер.

Нивоа на повторна употреба

- Ниво на апстракција - На ова ниво, не го употребувате директно софтверот, туку користете знаење за успешни апстракции при дизајнирање на вашиот софтвер.
- Ниво на објектот - На ова ниво, вие директно користите повторно предмети од библиотека, наместо сами да го напишете кодот.
- Ниво на компонентата - Компонентите се збирки предмети и класи на предмети што ги користите повторно во апликациските системи.
- Системско ниво - На ова ниво, можете повторно да го користите целиот систем на апликација.

Повторна употреба на софтвер



Трошоци за повторна употреба

- Трошоците за потрошеното време во потрага по софтвер за повторна употреба и проценка дали ги задоволува или не вашите потреби.
- Каде што е применливо, трошоците за купување на употреблив софтвер. За големи системи надвор од полиците, овие трошоци можат да бидат многу високи.
- Трошоците за прилагодување и конфигурирање на компонентите или системите за еднократно користење, за да ги одразуваат барањата на системот што го развивате.
- Трошоците за интегрирање на еднократните софтверски елементи едни со други (ако користите софтвер од различни извори) и со новиот код што сте го развиле.

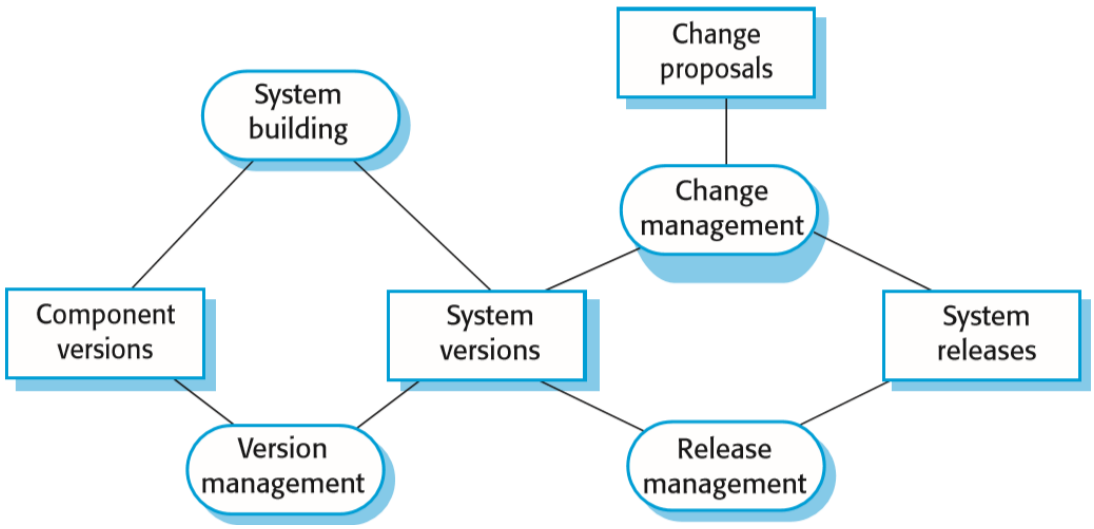
Управување со конфигурацијата

- Управувањето со конфигурацијата е името дадено на генералниот процес на управување со софтверски систем што се менува.
- Целта на управувањето со конфигурацијата е да го поддржи процесот на системска интеграција, така што сите развивачи можат да пристапат до кодексот на проектот и документите на контролиран начин, да дознаат какви се направени промените и да ги состави и да ги поврзува компонентите за да создадат систем.

Активности за управување со конфигурација

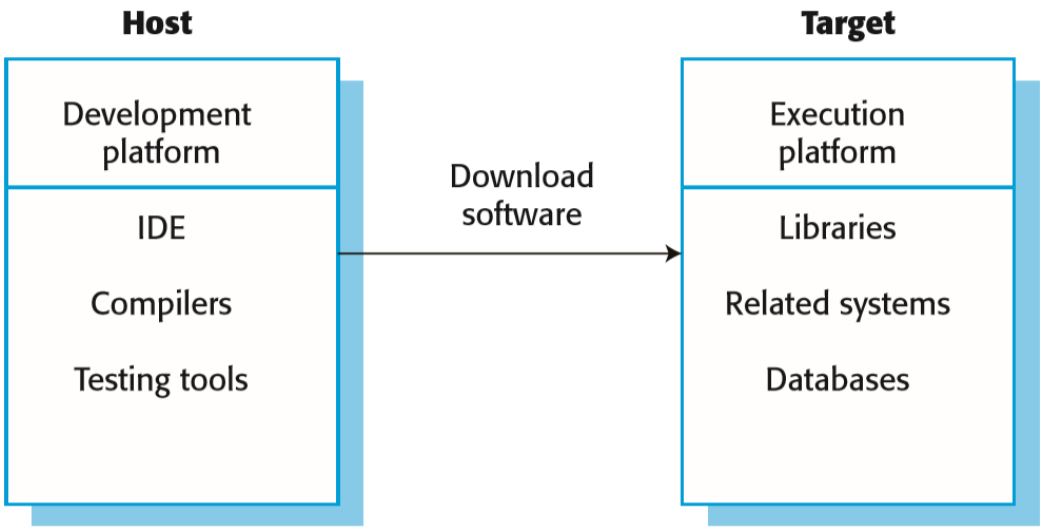
- Управување со верзијата, каде што е дадена поддршка за да се следат различните верзии на софтверските компоненти. Системите за управување со верзиите вклучуваат можности за координирање на развојот од страна на неколку програмери.
- Системска интеграција, каде што е дадена поддршка за да им помогне на развивачите да дефинираат кои верзии на компонентите се користат за создавање на секоја верзија на системот. Овој опис потоа се користи за да се изгради систем автоматски со составување и поврзување на потребните компоненти.
- Следење на проблеми, каде што е обезбедена поддршка за да им се овозможи на корисниците да пријавуваат грешки и други проблеми и да им овозможат на сите развивачи да видат кој работи на овие проблеми и кога се решени.

Интеракција – управување со конфигурацијата



Развој на таргетираните домаќини

- Повеќето софтвер се развиваат на еден компјутер (домаќин), но работи на посебна машина (целта).
- Општо, можеме да зборуваме за развојна платформа и платформа за извршување.
  - Платформата е повеќе од само хардвер.
  - Вклучува инсталиран оперативен систем, плус друг софтвер за поддршка, како што е систем за управување со базата на податоци или, за развојни платформи, интерактивно опкружување за развој.
- Развојната платформа обично има различен инсталиран софтвер од платформата за извршување, овие платформи можат да имаат различни архитектури.



Алатки за развојна платформа

- Интегриран систем за уредување насочен кон компајлерот и синтаксата, кој ви овозможува да креирате, уредувате и составувате код.
- Систем за дебагирање на јазик.
- Алатки за графичко уредување, како што се алатки за уредување на UML модели.
- Алатките за тестирање, како што е Junitthat, можат автоматски да извршат сет на тестови на нова верзија на програмата.
- Алатки за поддршка на проекти кои ви помагаат да го организирате кодот за различни развојни проекти.

Integrated development environments (IDEs) – интегрирана развојна околина

- Алатките за развој на софтвер честопати се групирани за да создадат интегрирано развојно опкружување (IDE).
- IDE е збир на софтверски алатки кои поддржуваат различни аспекти на развој на софтвер, во рамките на некоја заедничка рамка и корисничкиот интерфејс.
- IDEs се создадени за поддршка на развојот на специфичен јазик за програмирање, како што е Java. Јазикот ИРО може да се развива специјално или може да претставува иницијатива на IDE од општа намена, со специфични алатки за поддршка на јазици.

Фактори на распоредување на компонентата / системот

- Ако компонентата е наменета за специфична хардверска архитектура или се потпира на некој друг софтверски систем, тој очигледно мора да биде распореден на платформа која ја обезбедува потребната хардверска и софтверска поддршка.
- Системите за висока достапност може да бараат компонентите да бидат распоредени на повеќе од една платформа. Ова значи дека, во случај на дефект на платформата, достапна е алтернативна имплементација на компонентата.

- Ако има високо ниво на комуникациски сообраќај помеѓу компонентите, обично има смисла да се распоредат на иста платформа или на платформи кои се физички блиски едни до други. Ова го намалува одложувањето помеѓу времето кога една компонента ја испраќа пораката и ја добива од друга.

Развој на отворен извор

- Развој на отворен извор е пристап кон развој на софтвер во кој се објавува изворниот код на софтверскиот систем и се поканети доброволци да учествуваат во процесот на развој.
- Неговите корени се во Фондацијата за слободен софтвер (www.fsf.org) , кој се залага дека изворниот код не треба да биде комерцијален, туку треба секогаш да биде достапен за корисниците да ги испитаат и модифицираат како што сакаат.
- Софтверот со отворен извор ја прошири оваа идеја со користење на Интернет за да регрутира многу поголема популација на волонтери на развивачи. Многу од нив се исто така корисници на кодот.

Системи со отворен извор

- Најпознатиот производ со отворен извор е, се разбира, оперативниот систем Linux, кој широко се користи како серверски систем и, сè повеќе, како работна околина.
- Други важни производи со отворен извор се Java, веб-серверот Apache и системот за управување со базата на податоци mySQL.

Проблеми со отворен извор.

- Дали производот што се развива треба да користи компоненти на отворен извор?
- Дали треба да се користи пристап со отворен извор за развој на софтверот?

Бизнис со отворен извор

- Се повеќе и повеќе производи на производи користат пристап со отворен извор за развој.
- Нивниот деловен модел не зависи од продажба на софтверски производ, туку на продажба на поддршка за тој производ.
- Тие веруваат дека вклучувањето заедница со отворен код ќе овозможи развој на софтвер поевтино, побрзо и ќе создаде заедница на корисници за софтверот.

Лиценцирање на отворен извор

- Основен принцип на развој на отворен код е дека изворниот код треба да биде слободно достапен, тоа не значи дека секој може да го стори како што сакаат со тој код.
  - Правно, инвеститорот на кодот (или компанија или индивидуа) сè уште го поседува кодот. Тие можат да стават ограничувања за тоа како се користат со вклучување на законски обврзувачки услови во лиценца за софтвер со отворен извор.
  - Некои развивачи на софтвер со отворен извор веруваат дека ако компонента со отворен извор се користи за развој на нов систем, тогаш тој систем треба да биде и со отворен извор.
  - Други се подготвени да дозволат нивниот код да се користи без ова ограничување. Развиените системи може да бидат комерцијални и да се продаваат како системи со затворен извор.

Модели на лиценци

- GNU Општа јавна лиценца (GPL). Ова е т.н. "реципрочна" лиценца што значи дека ако користите софтвер со отворен извор кој е лиценциран под GPL лиценцата, тогаш мора да го направите тој софтвер со отворен извор.
- GNU Помалата општа јавна лиценца (LGPL) е варијанта на GPL лиценцата каде што можете да напишете компоненти што водат до код со отворен извор, без да го објавувате изворот на овие компоненти.

- Лиценца за стандардна дистрибуција на Беркли (BSD). Ова е нерециprочна лиценца, што значи дека не сте должни повторно да објавувате какви било промени или измени направени за код со отворен код. Може да го вклучите кодот во комерцијални системи што се продаваат.

Управување со лиценца

- Воспоставете систем за одржување информации за компонентите со отворен извор што се преземаат и се користат.
- Бидете свесни за различните видови на лиценци и разберете како компонентата е лиценцирана пред да се користи.
- Бидете свесни за развојните патеки за компонентите.
- Едуцирајте ги луѓето за слободен софтвер.
- Да се воспостават системи за ревизија.
- Учествувајте во заедницата со отворен извор.

Тестирање на стратегии

- Тестирањето е процес на вежбање програма со специфична намера да пронајде грешки пред испорака до крајниот корисник. Тестирањето е наменето да покаже дека програмата го прави она што е наменето да го направи.
- Поради човечката неспособност да работи и да комуницира совршено (без грешки), софтверот мора да се провери
- Софтверското тестирање е функција за контрола на квалитет која има една примарна цел - да пронајде грешки. Задача за обезбедување на квалитет на софтверот е да обезбеди дека тестирањето е правилно планирано и ефикасно спроведено така што има најголема веројатност за постигнување на својата примарна цел.
- Тестирањето честопати вклучува повеќе напори во проектот отколку која било друга акција софтверско инженерство

Стратегиски пристап

- За да извршите ефективно тестирање, треба да спроведете ефективни технички прегледи. Со тоа, многу грешки ќе бидат елиминирани пред да се започне тестирањето.
- Тестирањето започнува на ниво на компонентите и работи „нанадвор“ кон интеграција на целиот компјутерски базиран систем.
- Различни техники за тестирање се соодветни за различни пристапи за софтверско инженерство и во различни периоди на време.
- Тестирањето го спроведува развивачот на софтверот и (за големи проекти) независна тест група.
- Тестирањето и дебагирање се различни активности, но дебагирање мора да се смести во секоја стратегија за тестирање.

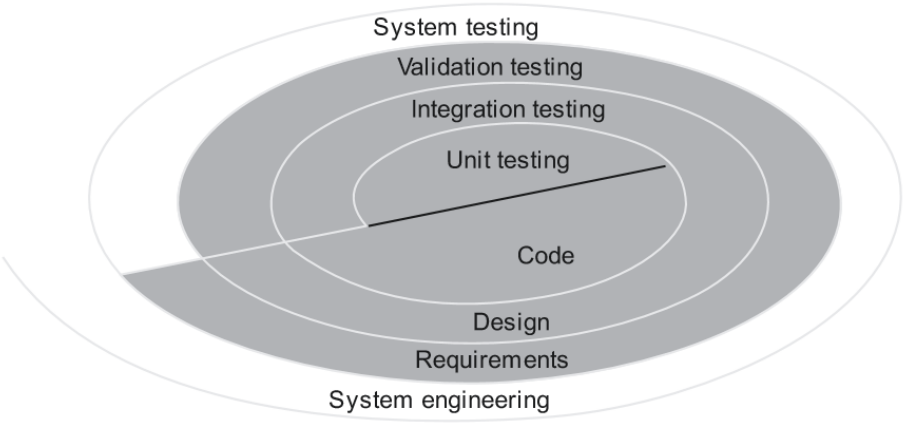
В и В

- Верификацијата се однесува на збир на задачи што гарантираат дека софтверот правилно спроведува специфична функција.
- Валидацијата се однесува на различен пакет задачи што гарантираат дека софтверот што е изграден е следлив на барањата на клиентот.

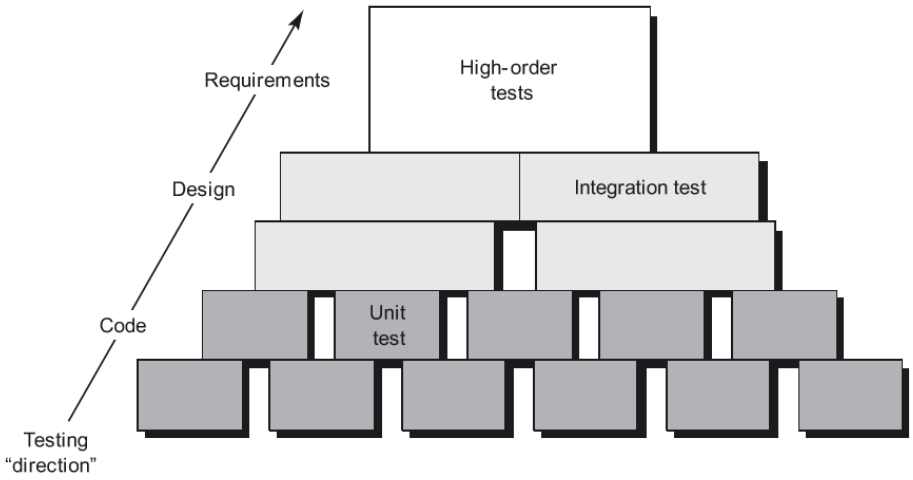
Кој го тестира софтверот

- Девелопер - Го разбира системот, но, ќе тестира „нежно“ и е управуван од „испораката”
- Независен тестер - Мора да научи за системот, но, ќе се обиде да го сруши и, управувано од квалитет

Стратегија на тестирање



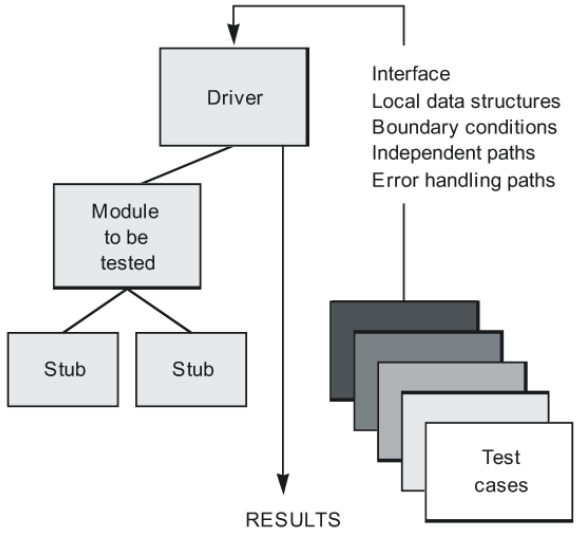
Чекори во тестирањето на софтвер



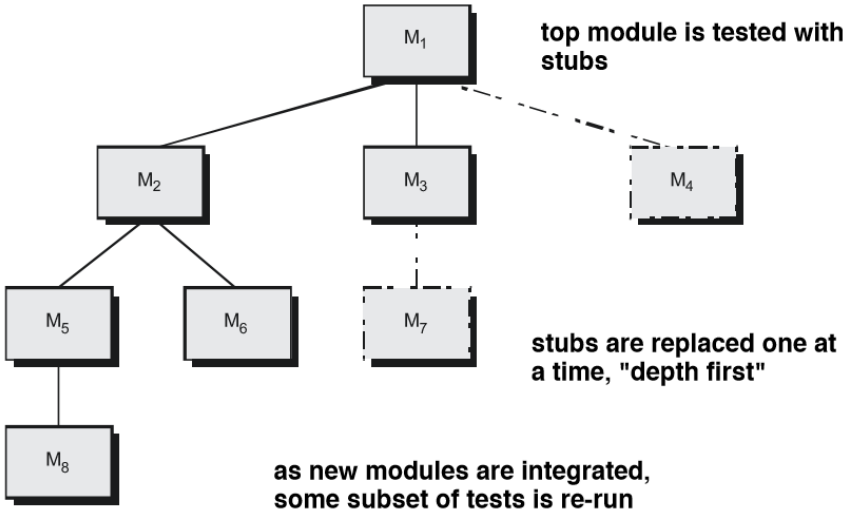
Стратегиски проблеми

- Наведете ги барањата на производот на мерливи количини пред да започнете со тестирање.
- Експлицитно цели на државно тестирање.
- Разберете ги корисниците на софтверот и развијте профил за секоја категорија на корисници.
- Развијте план за тестирање кој го потенцира „тестирањето на брз циклус“.
- Изградете „стабилен“ софтвер што е дизајниран да се тестира.
- Користете ефикасни технички прегледи како филтер пред тестирање.
- Спроведете технички прегледи за да ја процените стратегијата за тестирање и самите случаи за тестирање.
- Развијте пристап за континуирано подобрување за процесот на тестирање.

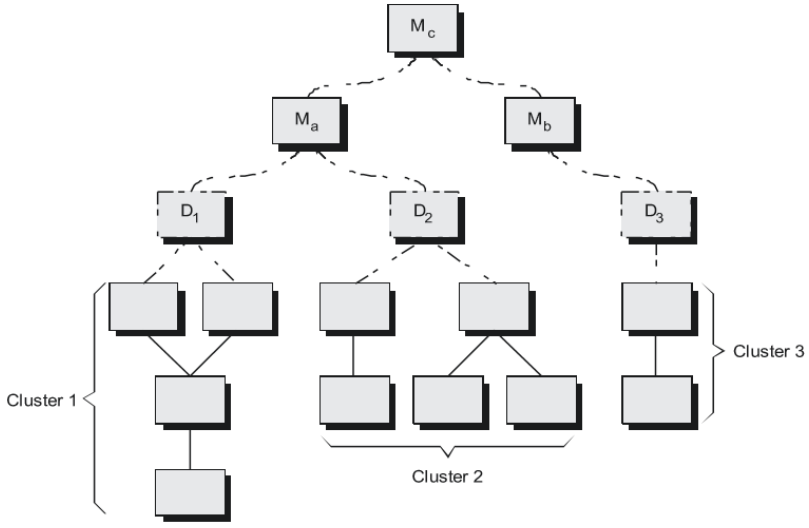
Околина за единица за тестирање



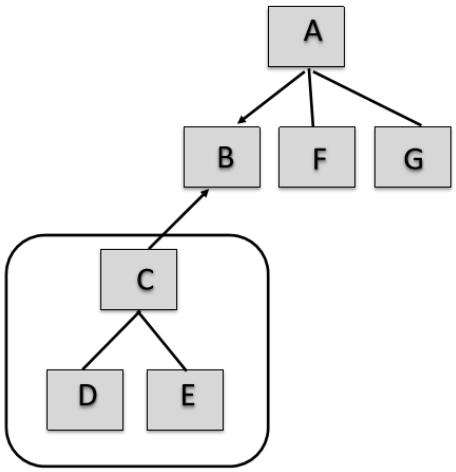
Top Down Integration



Bottom-up Integration



Sandwich Testing



Тестирање на регресија

- Регресивно тестирање го изврши повторното извршување на некои подгледи на тестови што се веќе спроведени за да се обезбеди дека промените не пропагирале несакани несакани ефекти
- Кога и да е корегиран софтверот, некој аспект на конфигурацијата на софтверот (програмата, неговата документација или податоците што поддржете го) се менува.
- Регулационото тестирање помага да се осигури дека промените (поради тестирање или од други причини) не воведуваат ненамерно однесување или дополнителни грешки.
- Регулационото тестирање може да се спроведе рачно, со повторна извршување на подмножество на сите тест случаи или со користење на автоматски алатки за снимање / репродукција.

Тестирање на чад.

- Заеднички пристап за создавање „секојдневно градење“ за софтверски производи
- Чекори за тестирање на чад:
  - Софтверските компоненти што се преведени во код се интегрирани во „градење“.
    - А изградбата ги вклучува сите датотеки со податоци, библиотеки, модули за еднократно користење и инженерните компоненти кои се потребни за спроведување на една или повеќе функции на производи.
  - Серија тестови се дизајнирани да изложат грешки што ќе го спречат правилното извршување на својата функција.
    - Намерата треба да биде да се откријат грешките во „покажете го стоперот“ кои имаат најголема веројатност за фрлање на софтверскиот проект зад предвиденото.
  - Изградбата е интегрирана со други гради и целиот производ (во сегашна форма) се тестира чад секој ден.
    - Пристапот за интеграција може да биде од горе или долу нагоре.

Придобивки за тестирање на чад

- Тестирање на чад обезбедува голем број придобивки кога се применува на сложени, временски критични проекти за софтверско инженерство:
  - Ризикот за интеграција е минимизиран.
  - Квалитетот на крајниот производ е подобрен.
  - Дијагностицирање и корекција на грешки се поедноставени.
  - Напредокот е полесен за проценка.

Документација за тест за интеграција

- Севкупниот план за интеграција на софтверот и описот на специфичните тестови е документиран во Спецификација за тестови
- Овој работен производ вклучува тест план и тест постапка и станува дел од конфигурацијата на софтверот.
- Критериуми за тест фази:
  - Интегритет на интерфејс
  - Функционална важност
  - Содржина на информации
  - Преформанси

Објектно – ориентирано тестирање

- Започнува со проценка на исправноста и конзистентноста на моделите за анализа и дизајн
- Промени во стратегијата за тестирање
- Дизајн на тест случаи се базира на конвенционални методи, но опфаќа и посебни карактеристики

ОО тестирачка стратегија

- класното тестирање е еквивалентно на единечното тестирање
  - се тестираат операциите во рамките на часот
  - состојбата на однесувањето на часот се испитува
- интеграција се применуваат три различни стратегии
  - тестирање врз основа на написи — интегрира множество на класи потребни за да одговорат на еден влез или настан
  - тестирање врз основа на употреба — го интегрира множеството класи потребни за да одговорат на еден случај за употреба
  - „тестирање на блок“ - интегрира сет на класи потребни за да се демонстрира една соработка

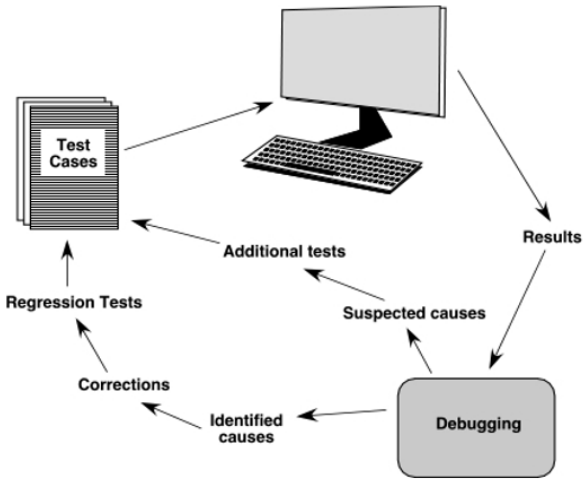
Тестирање на валидација

- Започнува со кулминација на тестирањето за интеграција
- Валидацијата успева кога софтверот функционира на начин што може разумно да се очекува од клиентот
  - да се фокусира на кориснички видливи активности и да може да се препознае корисник од системот (Барања на софтвер за извршување)
- Alpha and BetaTesting

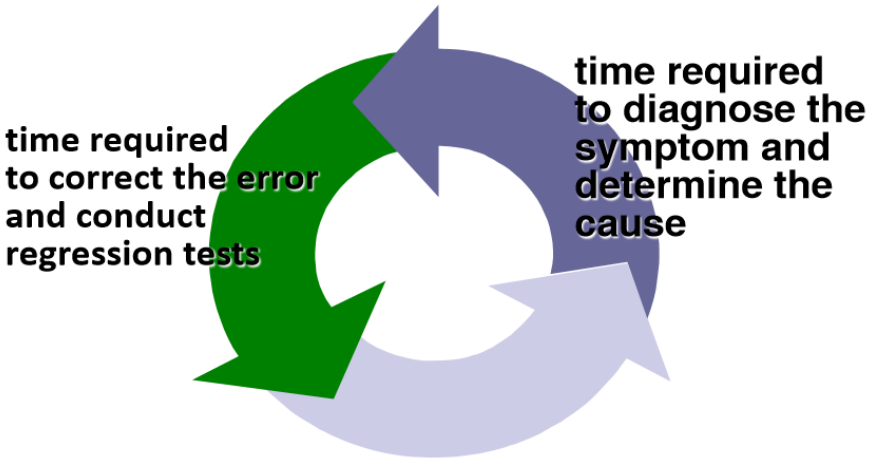
Системско тестирање

- Системско тестирање - Фокусот е на системска интеграција
- Тестирањето за обновување - го принудува софтверот да не успее на најразлични начини и потврдува дека обновувањето е правилно извршено
- Безбедносно тестирање - потврдува дека механизмите за заштита вградени во систем, всушност, ќе штитат тоа од неправилно навлегување
- Стресното тестирање - извршува систем на начин што бара ресурси во абнормална количина, фреквенција или волумен
- Тестирање на перформанси - го тестира извршувањето на перформансите на софтверот во контекст на интегриран систем
- Тестирање на распоредување

Процес на дебагирање



Debugging Effort



Симптоми и причини

- симптомот може да исчезне кога е утврден уште еден проблем,
- може да биде резултат на комбинација на не-грешки
- Може да биде тешко точно да се репродуцираат влезните услови,
- симптомот може да биде предизвикан од човечка грешка што не се проследи лесно.
- симптомот може да биде резултат на проблеми со тајмингот, отколку со проблеми со обработка.

Поправање на грешката

- Дали причината за грешката се репродуцира во друг дел од програмата? Во многу ситуации, дефект на програмата е предизвикан од погрешна шема на логика што може да се репродуцира на друго место.
- Која „следна грешка“ може да ја воведе со фиксот што ќе го направам? Пред да се изврши корекцијата, треба да се процени изворниот код (или, подобро, дизајнот) за да се процени спојувањето на логиката и структурите на податоците.
- Што можевме да сториме за да ја спречиме оваа грешка на прво место? Ова прашање е првиот чекор кон воспоставување пристап за статистичко обезбедување на квалитет на софтвер. Ако го поправите процесот, како и производот, грешката ќе се отстрани од тековната програма и може да биде елиминирана од сите идни програми.

Техники на тестирање на софтвер

Тестирање

- Софтверот мора да биде тестиран за да открие (и корегира) што е можно повеќе грешки пред испорака до вашиот клиент
- Цел: да се дизајнираат серија тест случаи кои имаат голема веројатност да најдат грешки
- Како: техники за тестирање на софтвер - системски упатства за дизајнирање на тестови што
  - ја извршуваат внатрешната логика и интерфејси на секоја софтверска компонента
  - ги вежбаат влезните и излезните домени на програмата за да откриваат грешки во функцијата на програмата, однесувањето и перформансите.

Кој го прави тоа?

- Софтверски инженер - во текот на раните фази на тестирање, ги изведува сите тестови
- Програмер - код
- Специјалисти - со тоа што процесот на тестирање напредува, специјалистите за тестирање може да бидат вклучени
- Клиент - секое време на програмата е извршено, клиентот го тестира

Зошто е важно?

- Прегледите и другите активности на SQA можат да направат и да откриат грешки, но тие не се доволни
- Со цел да се најде најголем можен број грешки, тестовите мора да се спроведуваат систематски, а случаите на тестирање мора да бидат дизајнирани со користење на дисциплинирани техники

Кои се чекорите?

- Софтверот се тестира од две различни перспективи:
  - „бела кутија“ техники за дизајнирање на тест-случај - се практикува логика на надворешна програма
  - „црна кутија“ техники за дизајнирање тест-случај - вежби за надградба се остваруваат
- Намерата е да се најде максималниот број на грешки со минимална количина на напор и време.
- При тестирање, гледната точка треба да се смени - треба да се обидеме напорно да го „срушиме“ софтверот

Цели

- Тестирање е процес на извршување програма со намера да се најде грешка.
- Добар тест случај е оној што има голема веројатност да пронајде грешка што сè уште не е откриена.
- Успешен тест е оној што открива грешка што се уште е откриена.
- Целта е да се дизајнираат тестови што систематски откриваат различни класи на грешки и да го сторат тоа со минимално време и напор.

Секундарна корист

- Тестирањето покажува дека функциите на софтверот се чини дека работат според спецификациите.
- Податоците собрани како тестирање се спроведуваат даваат добар показател за сигурноста на софтверот и индикација за квалитетот на софтверот како целина.
- Тестирањето не може да покаже отсуство на грешки и дефекти; може да покаже само дека грешките и дефектите на софтверот се присутни

Принципи на тестирање

1. Сите тестови треба да се следат според барањата на клиентите.
2. Тестовите треба да бидат планирани многу пред да започне тестирањето.
3. Принципот Парето важи за тестирање на софтвер.
4. Тестирањето треба да започне „во мали“ и да напредува кон тестирање „воопшто“.
5. Исцрпно тестирање не е можно.

Тестабилност

- Тестабилност = колку лесно може да се тестира компјутерска програма
- Треба да дизајнирате и имплементирате компјутерски систем или производ со „тестабилност“ во умот

Оперативност

- „Колку подобро работи, толку поефикасно ќе може да се тестира“
- Системот е дизајниран и имплементиран со квалитетно предвид
- Релативно неколку грешки ќе го блокираат извршувањето на тестовите

Набудување

- „Она што го гледате е она што го тестирате“
- Внесите обезбедени како дел од тестирањето, произведуваат различни резултати.
- Состојбите на системот и променливите се видливи или извршени со зададено движење.
- Неправилниот излез лесно се идентификува.
- Внатрешните грешки автоматски се откриваат и пријавуваат.
- Изворниот код е достапен

Контролирање

- „Колку подобро можеме да го контролираме софтверот, толку повеќе тестирањето може да се автоматизира и оптимизира“
- Сите можни излези можат да се генерираат преку одредена комбинација на влез, а форматите I / O се конзистентни и структурирани
- Сите шифри се извршни преку одредена комбинација на внес .
- Тестовите можат погодно да се наведат, автоматизираат и репродуцираат.

Разградување

- „Со контролирање на обемот на тестирање, можеме побрзо да ги изолираме проблемите и да извршиме попаметно повторување на тестирањето“
- Софтверскиот систем е изграден од независни модули што можат да се тестираат независно.

Едноставност

- „Колку е помалку да се тестира, толку побрзо можеме да го тестираме“
- Функционална едноставност - сетот на функција е минимум неопходен за да се исполнат барањата
- Структурна едноставност - архитектурата е модулализирана за да се ограничи размножувањето на грешките
- Кодна едноставност - кодирање усвоен е стандард за леснотија на инспекција и одржување

Стабилност

- "Колку помалку промени, толку помалку прекинувања при тестирањето."
- Промените во софтверот се ретки, контролирани кога се случуваат и не ги валидираат постојните тестови
- Софтверот добро се обновува после неуспеси

Разбирливост

- „Колку повеќе информации имаме, толку попаметни ќе ги тестираме“
- Архитектонскиот дизајн и зависностите помеѓу внатрешните, надворешните и споделените компоненти се добро разбрани,
- Техничката документација е веднаш достапна, добро организирана, специфична и детална и точна.
- Промените во дизајнот се соопштуваат на тестерите.

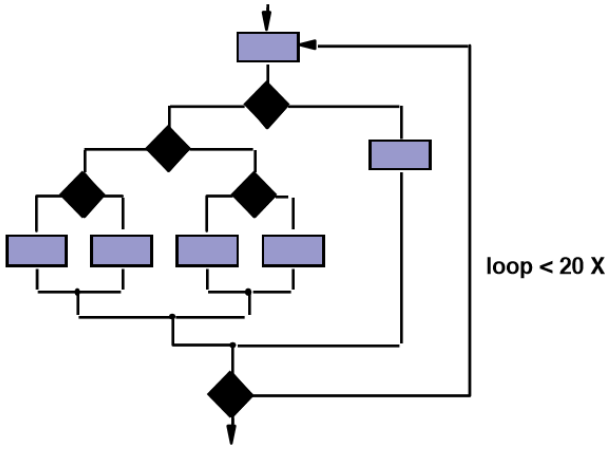
Што е „добар“ тест?

- Добар тест има голема веројатност да се најде грешка
- Добар тест не е вишок. n Добар тест треба да биде „најдобар од раса“
- Добар тест не треба да биде ниту премногу едноставен, ниту премногу сложен

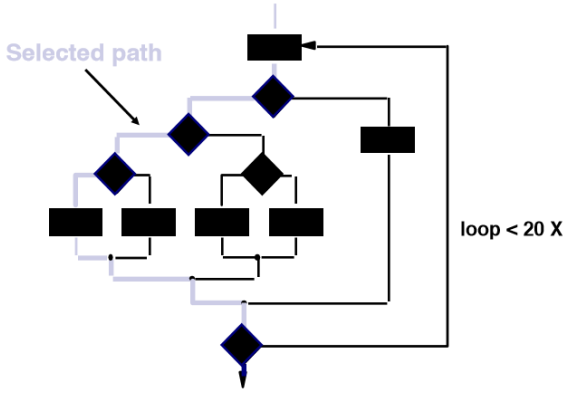
Внатрешни и надворешни погледи

- Секој инженерски производ (и повеќето други работи) може да се тестира на еден од два начина:
  - Со знаење на наведената функција што производот е дизајниран да ја изврши, може да се спроведат тестови што покажуваат дека секоја функција е целосно оперативна додека е на исто време во потрага по грешки во секоја функција;
  - Сознавајќи ги внатрешните работи на производот, може да се спроведат тестови за да се обезбеди „сите решетки на решетката“, односно внатрешните работи да се изведуваат според спецификациите и сите внатрешни компоненти се соодветно остварени.

Исцрпно тестирање



Селективно тестирање



Зошто да се покријат?

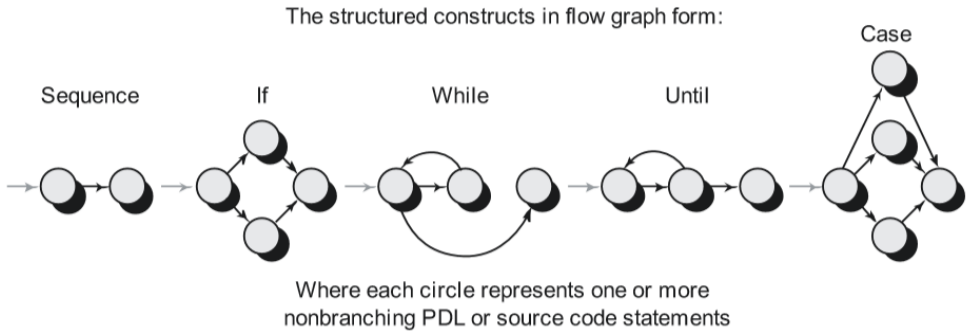
- логичките грешки и неточните претпоставки се обратно пропорционални со веројатноста за извршување на патеката,
- честопати веруваме дека патеката веројатно нема да биде извршена; всушност, реалноста е често контра интуитивна
- типографска грешка е случајна; веројатно е дека неиспитаните патеки ќе содржат некои

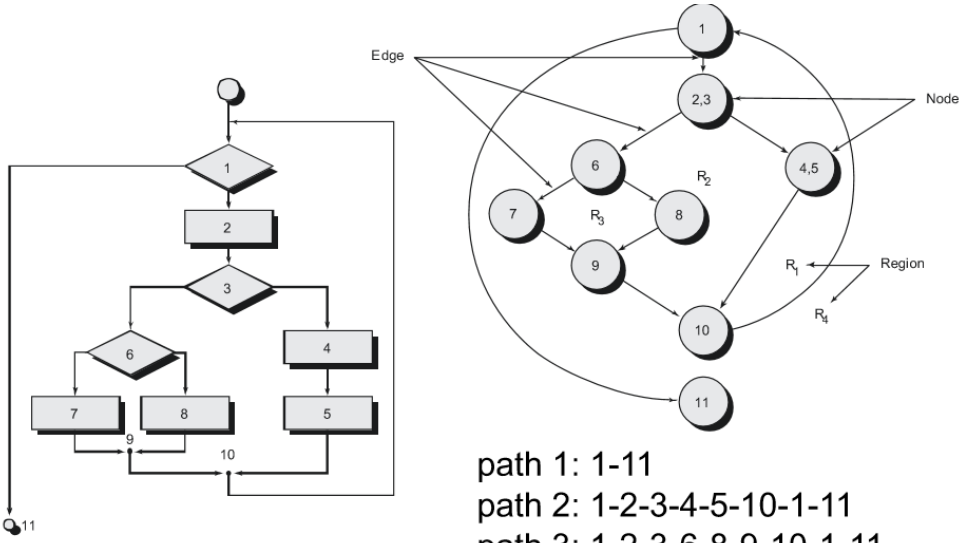
Основно тестирање на патеката.

- Тестирањето на патеката е техника за тестирање во бела кутија (МекКабе, 1976) - му дозволува на дизајнерот на терминали да донесе логична мерка за сложеност на процедуралниот дизајн и да ја користи оваа мерка како упатство за дефинирање основен пакет на патеки за извршување. Случаите за тестирање добиени за вежбање на поставената основа се гарантирани за извршување на секоја изјава во програмата барем едно време за време на тестирањето.

График на проток

- Графикот на протокот го прикажува логичкиот проток на контрола
- Секоја структурирана конструкција има симбол на графиконот на проток

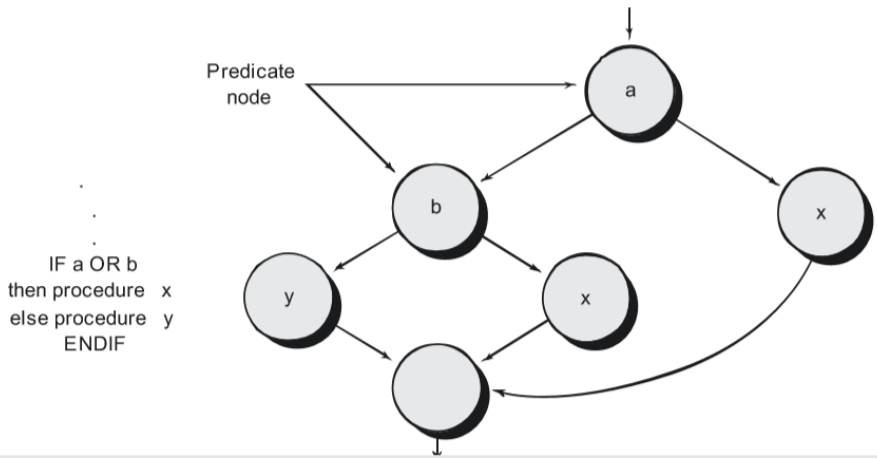




1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

path 1: 1-11
path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11

- Се јавува сложена состојба - еден или повеќе оператори (логички ИЛИ, И, НАНД, НОР) се присутни во условна изјава; PDL (јазик за дизајн на програми) сегментот се претвора во графиконот на проток прикажан
- IF a OR b е создаден посебен јазол за секој од условите a и b
- предикативен јазол - секој јазол што содржи состојба и се карактеризира со две или повеќе рабови што произлегуваат од тоа

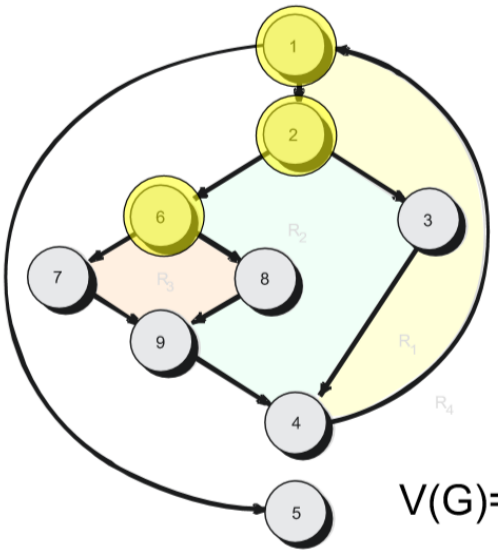


Цикломатска комплексност

- Цикломатска сложеност - е софтверска метрика која обезбедува квантитативна мерка за логичката сложеност на програмата
- Кога се користи во контекст на методот за тестирање на основната патека, вредноста пресметана за цикломатска сложеност
  - го дефинира бројот на независни патеки во основен сет на програма и
  - ви обезбедува горна граница за бројот на тестови што мора да бидат дизајнирани и извршени за да се осигурате дека сите изјави се извршени барем еднаш.

Пресметка на цикломатска сложеност

- Цикоматска сложеност се пресметува на еден од трите начини:
  1. Бројот на региони на графиконот на проток одговара на цикломатската сложеност
  2. Цикломатска сложеност V (G) за график на проток G дефиниран како: V (G) = E – N + 2 каде што е бројот на рабовите на графиконот на протокот и, N е бројот на јазли на графиконот на проток.
  3. V (G) = P + 1 каде што P е бројот на предикативни јазли содржани во графиконот на проток G



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

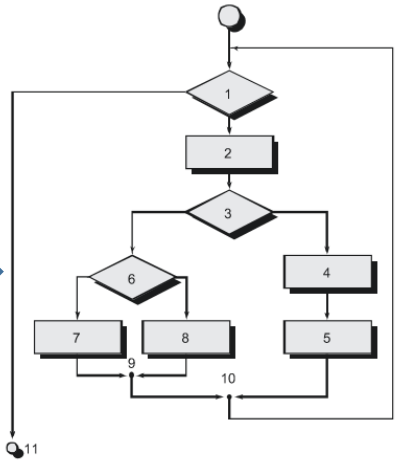
Number of regions = 4

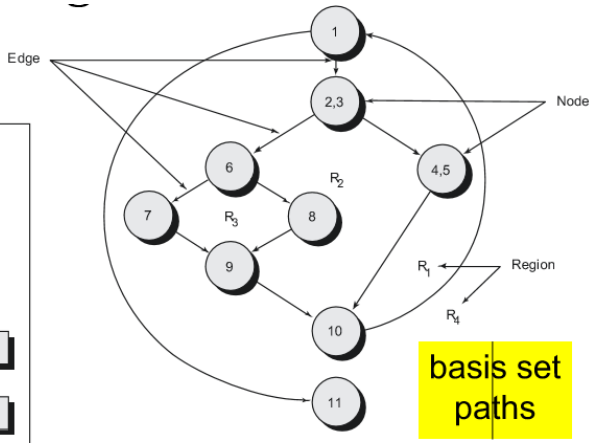
V(G)= 11 edges – 9 nodes + 2 = 4

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

- Голем број на студии во индустријата покажаа дека колку е повисок V (G), толку е поголема веројатноста за грешки

Основно Тестирање на патеката





basis set paths

path 1: 1-11
path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11

Случаи за испитување на изведување

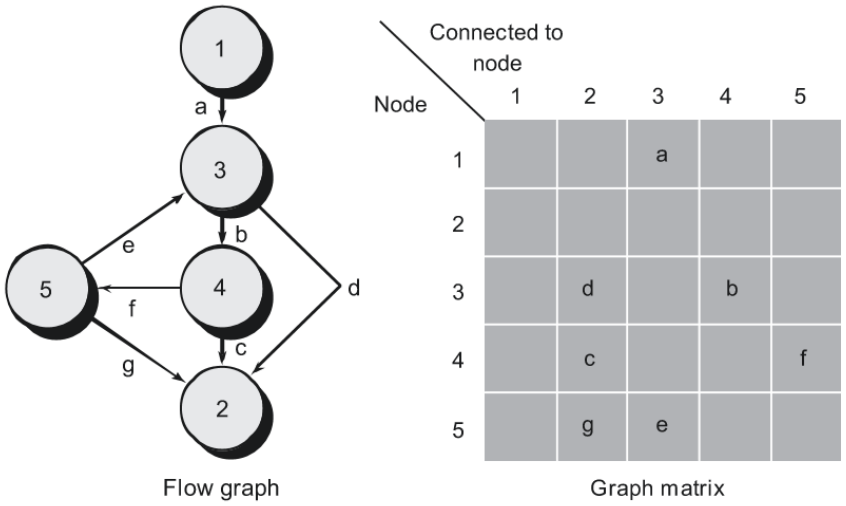
Резимирање:

- Користејќи го дизајнот или кодот како основа, нацртајте соодветен графикон на проток.
- Одредете ја цикломатската сложеност на графиконот на проток на резултат.
- Определи основен сет на независни патеки.
- Подгответе случаи на тест што ќе принудат извршување на секоја патека во поставената основа

Тука има голем пример, не за копирање туку за слушање. Лол ☺

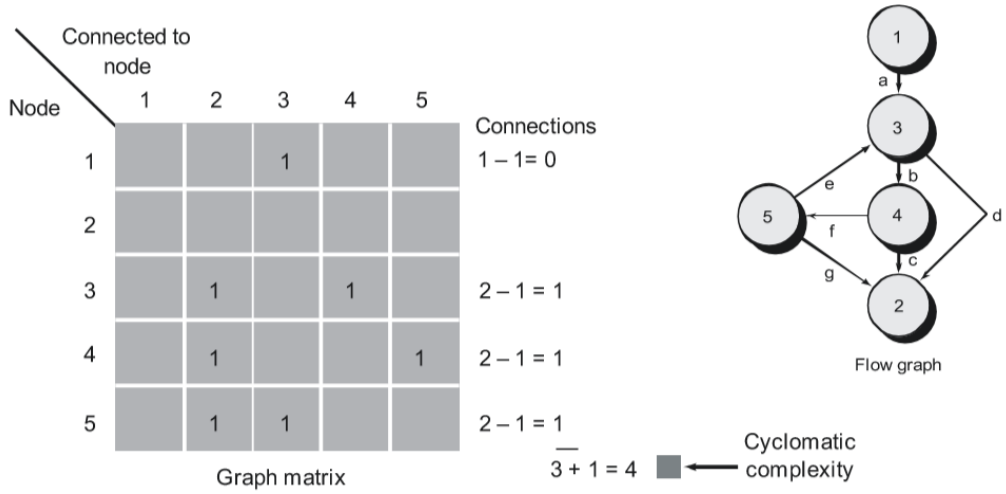
Графички матрици

- Графичка матрица е квадратна матрица чија големина (т.е. број на редови и колони) е еднаква на бројот на јазли на графикон на проток
- Секој ред и колона одговара на идентификуван јазол, а записите од матрицата одговараат на врски (раб) помеѓу јазли.
- Со додавање на тежина на врската кон секоја влез во матрицата, матрицата на графиконите може да стане моќна алатка за оценување на контролната структура на програмата за време на тестирањето.



Матрица за поврзување

- 1 - врска постои помеѓу i и j јазли,
- 0 - врска не постои.
- Цикломатска сложеност - пресметана врз основа на матрицата за поврзување



Тестирање на контролната структура

- Условно тестирање - метод на дизајнирање на тест случаи, кој ги практикува логичките услови содржани во програмскиот модул.
- Тестирање на проток на податоци ги избира патеките за тест на програмата според локацијата на дефинициите и употребата на променливите во програмата

Условно тестирање

- Метод на дизајнирање на тест-случај што ги практикува логичките услови содржани во програмскиот модул - Булова променлива или релационо изразување, што му претходи еден оператор НЕ (¬): E1<рационален - оператор>E2 каде E1 и E2 се аритметички изрази, а рационалниот оператор е еден од следните<, ≤, =, ≠,> или ≥
- Соединение е составено од два или повеќе едноставни состојби, булеоператори (И, ИЛИ, НЕ) и загради
- Ако условот не е точен, тогаш барем една компонента на состојбата е неточна
  - Грешка кај Булеан оператор
  - Грешка кај Булеан променлива
  - Грепка кај заградите
  - Грешка кај рационален оператор
  - Грешка кај аритметички израз

Различни стратегии

- Тестирање на гранки - ја анализира секоја гранка присутна во модулот на програмата барем еднаш за да ги открие сите грешки присутни во филијалата.
- Тестирање на домен - тестира релациони изрази присутни во програмата.
  - потребни се 3-4 тестови за релационият израз E1 <релационо-оператор> E2
  - 3 за да се покријат случаите: E1 is>, <and = до E2
  - За булеански изрази со n променливи -2^n
- BRO - ги тестира гранките присутни во модулот на програмата користејќи ограничувања на состојбата

Тестирање на проток на податоци

- Методот за тестирање на протокот на податоци избира патеки за тестирање на програмата според локацијата на дефинициите и употребата на променливите во програмата.
  - Да претпоставиме дека на секоја изјава во програмата му е доделен единствен број на изјави и дека секоја функција не ги менува неговите параметри или глобалните променливи. За изјава со Sas нејзиниот број на изјави
    - DEF(S) = {X| изјава S содржи дефиниција од X}
    - USE(S) = {X| изјава S содржи користење од X}
- Една едноставна стратегија за тестирање на протокот на податоци е да се бара секој ланец DU да биде покриен барем еднаш.

Тестирање на јамка: едноставни јамки.

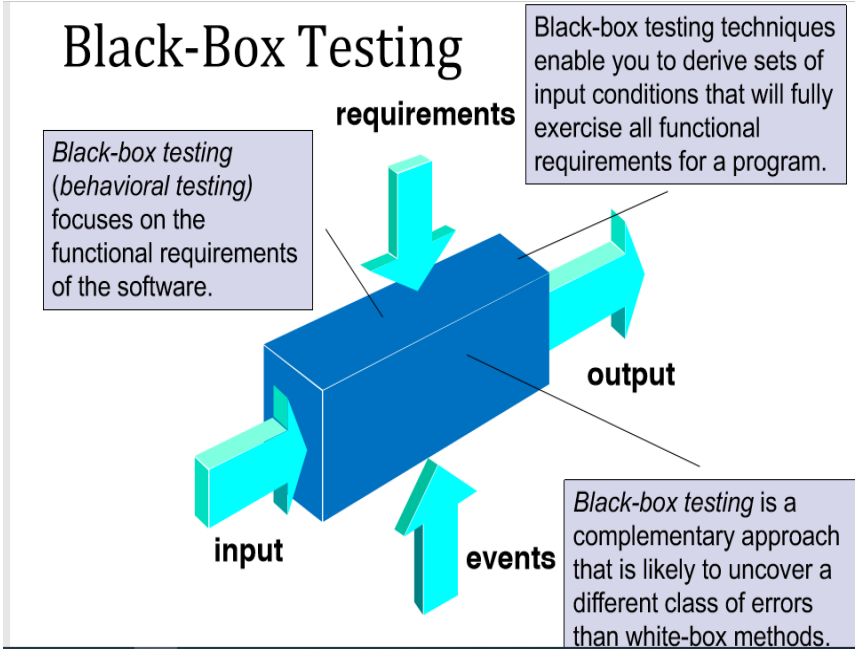
- Минимални услови — едноставни јамки
  1. прескокнете ја јамката во целост
  2. само една поминува низ јамката
  3. две поминува низ јамката
  4. m поминува низ јамката m <n 5. (n-1), n и (n + 1) поминува низ јамката
     каде n е максималниот број на дозволени додавања

Тестирање на јамка: Вгнездени јамки

- вгнездени јамки
  - Започнете со најоддалечената јамка. Поставете ги сите надворешни јамки на нивните минимални вредности на параметарот за итерација.
  - Тестирајте ги мин + 1, типичен, макс-1 и максимум за најоддалечената јамка, додека ги држите надворешните јамки на нивните минимални вредности.
  - Испуштете една јамка и поставете ја како во чекор 2, држејќи ги сите други јамки во типични вредности. Продолжете со овој чекор се додека не се јамкаат други песни по типични вредности. Продолжете со овој чекор сè додека не се тестира најоддалечената јамка.

Тестирање на јамка: Здружени и неструктурирани јамки.

- Здружени јамки
  Ако петелките се независни едни од други,
  тогаш третирајте ја секоја како едноставна јамка
  инаку место третирајте како вгнездени јамки
  endif
- Неструктурирани јамки
  - Кога е можно, тие треба да бидат редизајнирани за да ја рефлектираат употребата на структуираните програмирани конструкции.



- Тестирање на црни кутии. Како се тестира функционалната валидност?
- Како се тестираат однесувањето и перформансите на системот?
- Кои класи на влез ќе направат добри тест случаи?
- Дали системот е особено чувствителен на одредени влезни вредности?
- Како се изолираат границите на класата на податоци?
- Кои стапки на податоци и волуменот на податоците можат да го толерираат системот?
- Каков ефект ќе имаат специфичните комбинации на податоци врз работењето на системот?

Тестирање на црна кутија

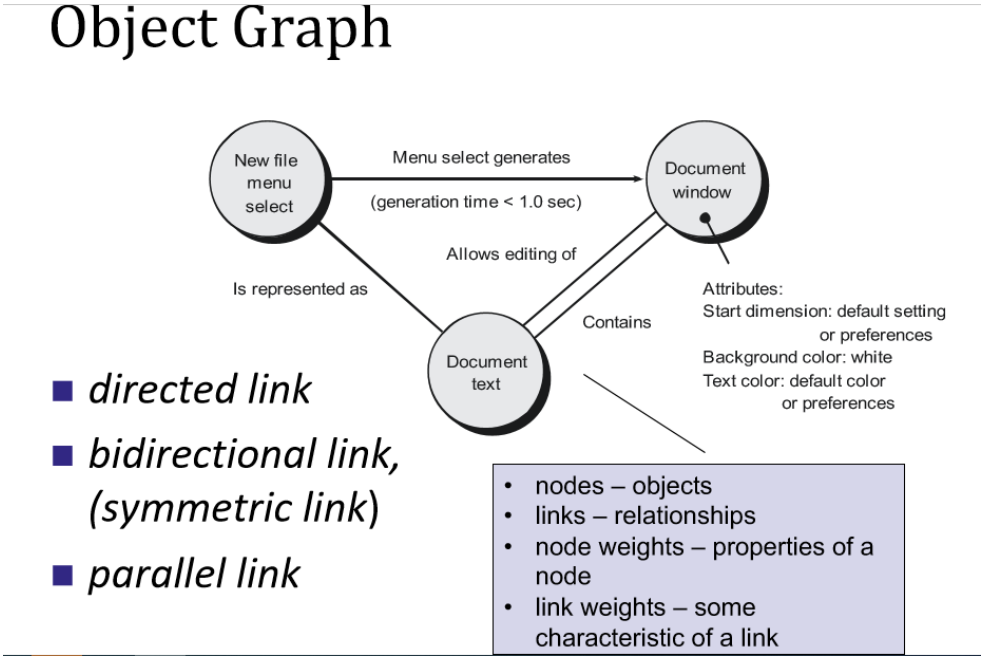
- Категории на грешки. Тестирањето на „Black Box“ се обидува да пронајде грешки во следниве категории:
  1. неточни или исчезнати функции,
  2. грешки во интерфејсот,
  3. грешки во структурите на податоци или надворешен пристап до базата на податоци,
  4. грешки во однесувањето или работењето, и
  5. грешки во иницијализацијата и прекинувањето.

Техники на црната кутија

- Со примена на техники на црни кутии, добивате сет на тест случаи што ги исполнуваат следниве критериуми:
  - Тест случаи што го намалуваат, со пребројување на броевите поголема од една“, бројот на дополнителни тест случаи што мора да бидат дизајнирани да постигнат разумно тестирање и други
  - тест случаи што ќе ви кажат нешто во врска со присуството или отсуството на часови на грешки, отколку за грешка поврзана само со специфичниот тест што е на располагање.

Методи за тестирање засновани врз графикони

- Првиот чекор во тестирањето на црната кутија е да се разберат предметите што се моделираат во софтвер и односите што ги поврзуваат овие објекти.
- Следниот чекор е да се дефинираат серија тестови што потврдуваат „сите предмети имаат очекувана врска еден со друг“
- Со тестирање на софтвер започнува со создавање графикон на важни објекти и нивни односи и
- Измислување серија тестови што ќе го опфатат графиконот така што секој предмет и врска се остварува и се откриваат грешките.



Еквивалентност на партиционирање

- Метод за тестирање во црна кутија што го дели влезниот домен на програмата во класи на податоци од кои може да се изведат тест случаи.
- Идеален случај за тест едногласно открива класа на грешки, инаку може да бара многу случаи на тестирање пред да се забележи општата грешка.
- Дизајн на тест-случај, базиран на проценка на класи на еквивалентност за влезна состојба.
- Класа на еквивалентност претставува збир на валидни или невалидни состојби за влезни услови (специфична нумеричка вредност, опсег на вредности, збир на сродни вредности или состојба на Булеан).

Класи на еквивалентност

- Класите за еквивалентност можат да бидат дефинирани според следниве упатства:
  1. Доколку влезниот услов одредува опсег, дефинирани се една валидна и две класи на невалидна еквивалентност.

2. Доколку влезниот услов бара одредена вредност, дефинирани се една валидна и две класи на невалидна еквивалентност.
3. Ако влезниот услов одредува член на множеството, дефинирани се една валидна и една класа на невалидна еквивалентност.
4. Ако влезниот услов е Булоан, дефинирани се една валидна и една невалидна класа.

Анализа на гранична вредност

- Поголем број грешки се појавуваат на границите на влезниот домен отколку во „центарот“.
- Анализата на граничната вредност е техника за дизајн-тест-случај што ја надополнува поделбата на еквивалентноста.
- Наместо да одберете кој било елемент од класата за еквивалентност, BVA води до избор на тест случаи на „рабовите“ на часот.

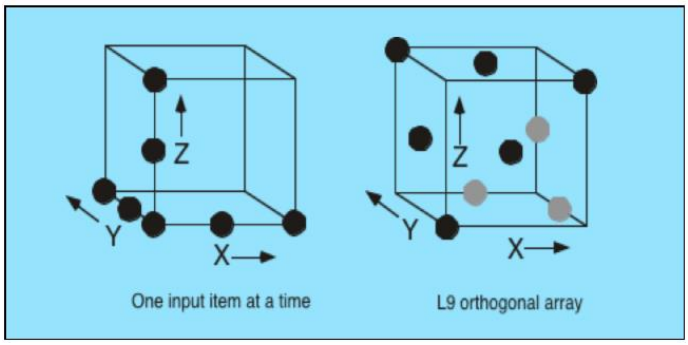
- Насоките за BVA се слични во многу аспекти со оние предвидени за поделба на еквивалентност:
  1. Ако влезната состојба наведете опсег ограничен со вредностите a и b, случајот со тест може да биде дизајниран со вредности a и b и веднаш над и веднаш под нивото б.
  2. Доколку влезната состојба наведете голем број на вредности, треба да се развијат тест случаи што ги користат минималните и максималните броеви. Исто така, се тестираат вредности веднаш над минимум и максимум.
  3. Примени упатства 1 и 2 за излезни услови.
  4. Доколку структурите на податоци за внатрешна програма имаат пропишано граници (на пр. Низа со дефиниран лимит од 100 записи), сигурно ќе дизајнирате случај за тестирање за да ја искористите структурата на податоците на нејзината граница.

Тест за споредување

- Се користи само во ситуации во кои веродостојноста на софтверот е апсолутно критична (на пр., Хуманирани системи)
  - Одделни тимови за софтверско инженерство развиваат независни верзии на апликација користејќи ги истите спецификации
  - Секоја верзија може да се тестира со исти тест податоци за да се обезбеди дека сите обезбедуваат идентичен излез
  - Потоа сите верзии се извршуваат паралелно со споредбата на резултатите во реално време за да се обезбеди конзистентност

Тестирање на ортогонална низа (OAT)

- Се користат кога бројот на влезните параметри е мал и вредностите што може да ги земе секој од параметрите се јасно ограничени



"Една влезна ставка во исто време" наспроти OAT

- "Една влезна ставка во еден момент" се приближува - секоја ставка за влез во еден момент може да се разликува во низа по секоја влезна оска.
  - релативно ограничено покриеност на влезниот домен
- Ортогонална низа за тестирање - рамнотежа на сопственост - дисперзирана рамномерно во целиот домен на тестот

L9 ортогонална низа

- Откријте ги и изолирајте ги сите грешки во единечен режим.
- Откријте ги сите грешки во двоен режим.
- Мултимодни грешки.

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Тестирање засновано врз модели

- Анализирајте постоечки модел на однесување за софтверот или создадете таков.
  - Потсетете се дека однесувањето моделира како софтвер ќе реагира на надворешни настани или стимули.
- Чекори за тестирање засновани врз модели:
  1. Анализирајте постоечки модел на однесување за софтверот или создадете еден.
  2. Пресечете го моделот на однесување и наведете ги влезовите што ќе го принудат софтверот да направи премин од држава во држава.
     - Влезовите ќе активираат настани што ќе предизвикаат транзиција.
  3. Прегледајте го моделот на однесување и забележете ги очекуваните излези бидејќи софтверот го прави преминот од држава во држава.
  4. Изврши случаи на тестирање.
  5. Споредете ги реалните и очекуваните резултати и преземете корективни активности по потреба.

Тестирање за специјализирани опкружувања, архитектури и апликации

- Тестирање на GUIs
- Тестирање на клиент-сервер архитектури
- Тестирање на документација и помошни средства
- Тестирање за системи во реално време
  - тестирање на задачи.
  - Тестирање на однесувањето.
  - Тестирање на интертаскинг работи
  - Системско тестирање.

Еволуција на софтверот

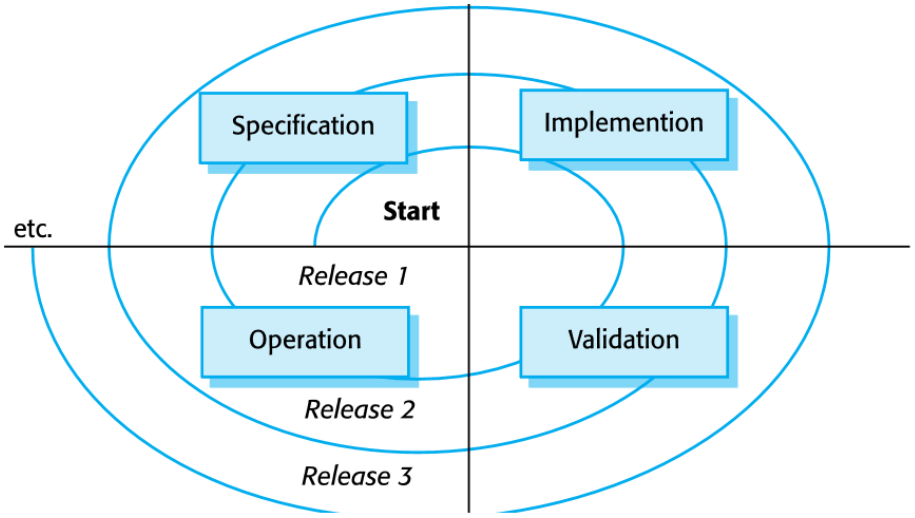
Промена на софтвер

- Промената на софтверот е неизбежна
  - Се појавуваат нови барања кога се користи софтверот;
  - Промените во деловното опкружување;
  - Грешките мора да се санираат;
  - Во системот се додаваат нови компјутери и опрема;
  - Перформансите или сигурноста на системот може да треба да се подобрат.
- Клучен проблем за сите организации е спроведување и управување со промените во постојните софтверски системи.

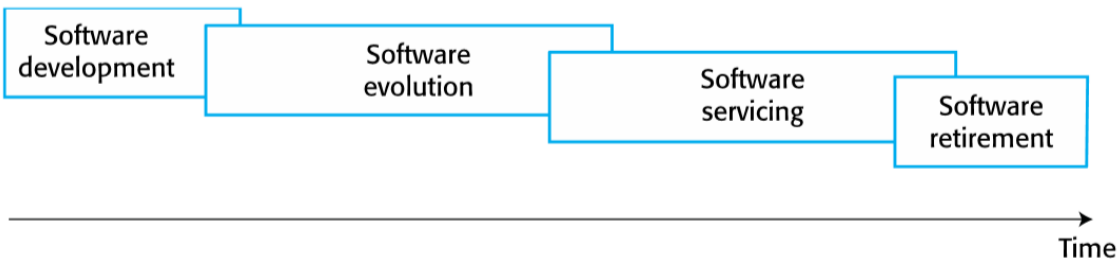
Важноста на еволуцијата

- Организациите имаат огромни инвестиции во нивните софтверски системи - тие се клучни деловни средства.
- За да се одржи вредноста на овие средства во бизнисот, тие мора да бидат променети и ажурирани.
- Поголемиот дел од буџетот на софтверот во големите компании е посветен на промена и развој на постојниот софтвер наместо развој на нов софтвер.

Спирален модел на развој и еволуција



Еволуција и сервисирање



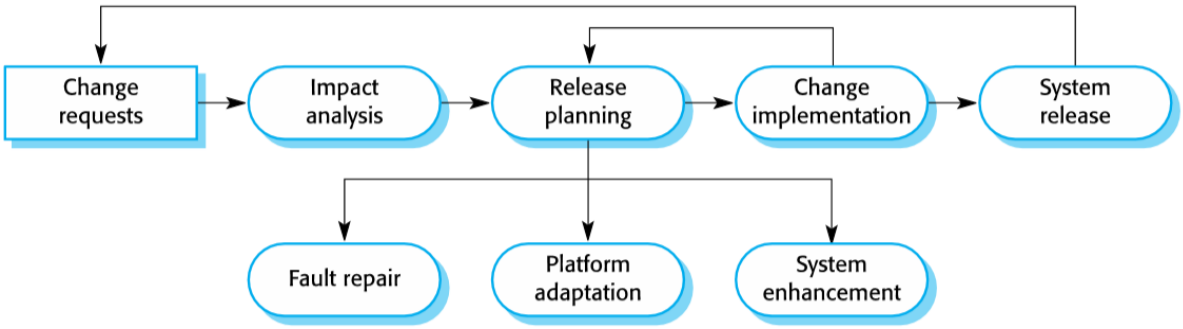
- Еволуција - Фазата во животниот циклус на софтверскиот систем каде што е во оперативна употреба и се развива со оглед на тоа што во системот се предлагаат и имплементираат нови барања.

- Сервисирање - Во оваа фаза, софтверот останува корисен, но единствените направени промени се оние кои се потребни за да се одржи функционален, т.е. поправки на грешки и промени за да ги одрази промените во околината на софтверот. Не е додадена нова функционалност.
- Исклучок - Софтверот сè уште може да се користи, но не се направени дополнителни промени во него.

Процеси на еволуција

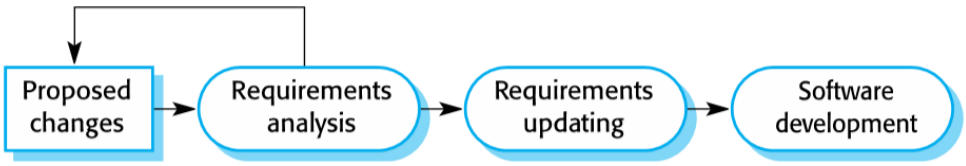
- Процесите за развој на софтвер зависат од
  - Видот на софтверот што се одржува; Used Користените развојни процеси;
  - Вештини и искуство на вклучените луѓе.
- Предлозите за промена се драјвер за развој на системот.
  - Треба да бидат поврзани со компонентите кои се погодени од промената, со што ќе се овозможи проценка на трошоците и влијанието на промената.
- Промената идентификација и еволуцијата продолжува во текот на целиот животен век на системот.

Процес на развој на софтвер



Имплементација на промена

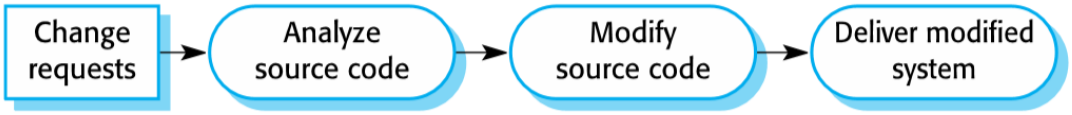
- Итерација на процесот на развој, каде што ревизиите на системот се дизајнирани, имплементирани и тестирани.
- Критична разлика е во тоа што првата фаза на спроведување на промените може да вклучува разбирање на програмата, особено ако оригиналните развивачи на системот не се одговорни за спроведувањето на промената.
- За време на фазата на разбирање на програмата, треба да разберете како е структурирана програмата, како таа обезбедува функционалност и како предложената промена може да влијае на програмата.



Барања за итна промена

- Можеби треба да се спроведат итни промени без да поминат низ сите фази на процесот на инженерство на софтвер
  - Ако треба да се поправи сериозната грешка на системот за да се овозможи нормално функционирање;
  - Ако промените во околината на системот (на пр. Надградба на оперативниот систем) имаат неочекувани ефекти;
  - Ако има деловни промени што бараат многу брз одговор (на пр. Објавување на конкурентскиот производ).

Процес на итна поправка



Агилни методи и еволуција

- Агилните методи се засноваат на развојниот развој, така што преминот од развој во развој е непречен.
  - Еволуцијата е едноставно продолжение на процесот на развој, засновано на чести изданија на системот.
- Автоматското тестирање на регресија е особено вредно кога се прават промени во системот.
- Промените може да бидат изразени како дополнителни кориснички приказни.

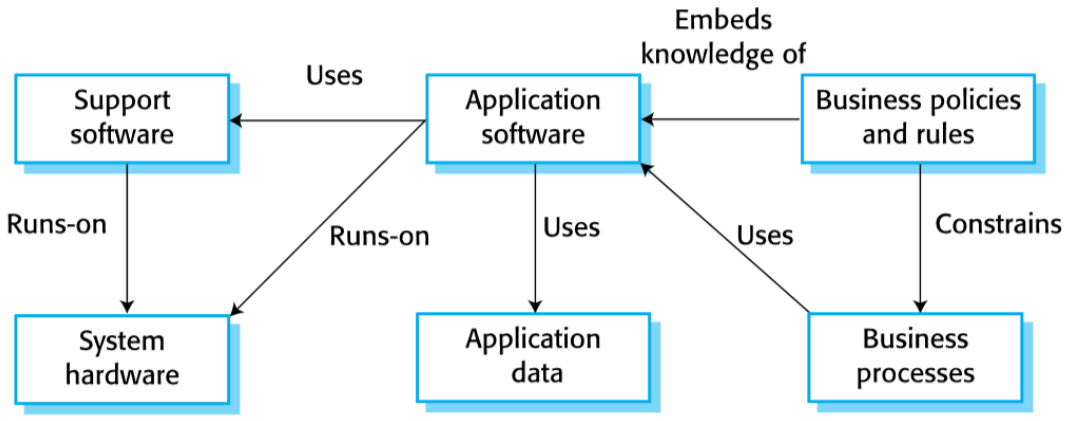
Проблеми со предавање

- Каде што тимот за развој користел агилен пристап, но тимот за еволуција не е запознаен со агилни методи и претпочита пристап заснован на план.
  - Тимот за еволуција може да очекува детална документација за поддршка на еволуцијата и тоа не се произведува во агилни процеси.
- Каде што се користиплан заснован за развој за развој, но тимот за еволуција претпочита да користи агилни методи.
  - Тимот за еволуција можеби треба да започне од нула развој на автоматски тестови и кодот во системот може да не е преработен и поедноставен како што се очекува во агилниот развој.

Наследни системи

- Наследните системи се постари системи што се потпираат на јазиците и технологијата кои веќе не се користат за развој на нови системи.
- Наследниот софтвер може да зависи од постариот хардвер, како што се компјутери со голема сума и може да има поврзани процеси и процедури за наследство.
- Наследните системи не се само софтверски системи, туку се и поширок социо-технички системи кои вклучуваат хардвер, софтвер, библиотеки и други придружни софтверски и деловни процеси.

Елементи на наследен систем



Компоненти на наследен систем

- Системски хардвер Системите за наследство може да се напишани за хардвер кој веќе не е достапен.
- Софтвер за поддршка Наследството систем може да се потпира на голем број на софтвер за поддршка, кој може да биде застарен или неподдржан.

- Софтвер за апликација Системот за апликации што обезбедува деловни услуги обично се состои од голем број на апликативни програми.
- Податоци за апликација Овие се податоци што се обработени од системот на апликации. Може да бидат неконзистентни, дуплирани или чувани во различни бази на податоци.
- Деловни процеси Овие се процеси што се користат во бизнисот за да се постигне одредена деловна цел.
- Деловните процеси можат да бидат дизајнирани околу систем на наследство и ограничен од функционалноста што ја обезбедува.
- Деловни политики и правила Овие се дефиниции за тоа како треба да се спроведува бизнисот и ограничувања на деловната активност. Употребата на системот за примена на наследство може да биде вметната во овие правила и правила

Замена на наследниот систем

- Замената на системот за наследство е ризична и скапа, така што деловните активности продолжуваат да ги користат овие системи
- Замената на системот е ризична од повеќе причини
  - Недостаток на целосна спецификација на системот
  - Тесна интеграција на системот и деловните процеси
  - Некументирани деловни правила вградени во системот на наследство
  - Но развој на софтвер може да биде доцна и / или над буџет

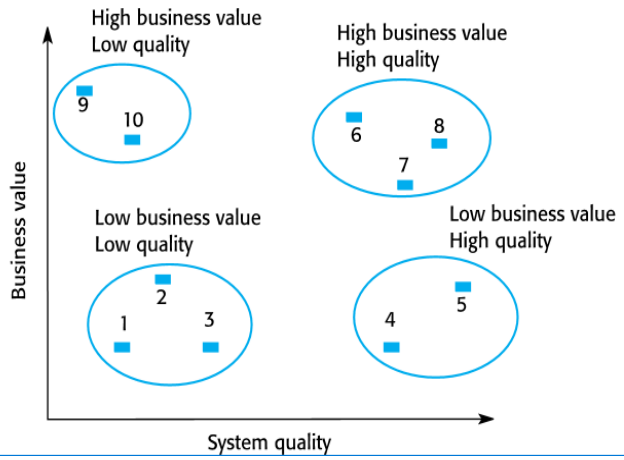
Промена на наследен систем

- Легитимните системи се скапи за промена од повеќе причини:
  - Не постојан стил на програмирање
  - Употреба на застарени јазици за програмирање со неколку луѓе што се достапни со овие јазични вештини
  - Идеална системска документација
  - Системирање на структурата на системот
  - Оптимизациите на програмите може да ги отежнат разбирливо
  - Грешки во податоците, дуплирање и недоследност

Управување со наследниот систем

- Организациите што се потпираат на наследни системи мора да изберат стратегија за развој на овие системи
  - Отпакувајте го системот целосно и изменете ги деловните процеси така што веќе не се бара;
  - Продолжете со одржување на системот;
  - трансформација на системот со реинженеринг за подобрување на неговата одржливост;
  - Заменете го системот со нов систем.
- Избраната стратегија треба да зависи од квалитетот на системот и неговата деловна вредност

Figure 9.13 An example of a legacy system assessment



Категории на наследни системи

- Низок квалитет, ниска деловна вредност
  - Овие системи треба да бидат уништени.
- Ниско-квалитетна, висока деловна вредност
  - Овие придонесуваат за важен деловен придонес, но се скапи за одржување. Треба да се преработат или заменат доколку е достапен соодветен систем.
- Висококвалитетна, ниска деловна вредност
  - Заменете со COTS (комерцијална-полица), ставете го целосно или одржете го.
- Висок квалитет, висока деловна вредност
  - Продолжете со работа со користење на нормално одржување на системот.

Проценка на деловната вредност

- Проценката треба да има предвид различни гледишта
  - крајни корисници на системот;
  - Бизнис клиенти;
  - Линиски менаџери;
  - ИТ менаџери;
  - Постари раководители.
- Интервјуирајте различни заинтересирани страни и соберете резултати.

Проблеми во проценката на деловната вредност

- Употреба на системот - Ако системите се користат само повремено или мал број луѓе, тие може да имаат мала деловна вредност.
- Деловните процеси што се поддржани - Системот може да има мала деловна вредност ако принуди употреба на неефикасни деловни процеси.
- Сигурност на системот - Ако системот не е зависен и проблемите директно влијаат врз деловните клиенти, системот има ниска деловна вредност.
- Излези на системот - Ако бизнисот зависи од излезни системи, тогаш системот има висока деловна вредност.

Оценување на квалитетот на системот

- Процена на деловните процеси - Како добро деловните процеси ги поддржуваат тековните цели на бизнисот?
- Оцена на животната средина - Колку е ефективно опкружувањето на системот и колку е скапо да се одржува?
- Проценка на апликацијата - Кој е квалитетот на системот за апликативен софтвер?

Проценка на деловниот процес

- Користете пристап ориентиран кон гледиште и побарајте одговори од засегнатите страни во системот
  - Дали има дефиниран модел на процеси и дали се следи?
  - Дали различни делови на организацијата користат различни процеси за иста функција?
  - Како е прилагодено процесот?
  - Кои се односите со другите деловни процеси и дали се неопходни?
  - Дали процесот ефикасно е поддржан од софтверот за наследни апликации?
- Пример - системот за нарачки за патувања може да има мала деловна вредност заради широко распространетата употреба на нарачки засновани на веб.

Фактори кои се користат во проценката на животната средина

- Стабилност на снабдувачот - Дали снабдувачот сè уште постои? Дали снабдувачот е финансиски стабилен и веројатно ќе продолжи да постои? Ако добавувачот повеќе не е во деловна активност, дали некој друг ги одржува системите?
- Стапка на неуспех - Дали хардверот има голема стапка на пријавени неуспеси? Дали рестартираниот систем за паѓање и сила на системот за поддршка?
- Возраст - Колку години е хардверот и софтверот? Колку е постар хардверот и софтверот за поддршка, толку ќе биде застарен. Сè уште може да функционира правилно, но може да има значителни економски и деловни придобивки за преминување кон посовремен систем.
- Изведба - Дали перформансите на системот се соодветни? Дали проблемите со перформансите имаат значително влијание врз корисниците на системот?
- Барања за поддршка - Која локална поддршка е потребна од хардверот и софтверот? Ако има големи трошоци поврзани со оваа поддршка, можеби вреди да се размисли за замена на системот.
- Трошоци за одржување - Кои се трошоците за лиценците за одржување и хардверскиот софтвер? Постариот хардвер може да има поголеми трошоци за одржување од современите системи. Софтверот за поддршка може да има високи годишни трошоци за лиценцирање.
- Интероперабилност - Дали има проблеми со поврзување на системот со други системи? Може, на пример, компајлерите да се користат со тековните верзии на оперативниот систем? Дали е потребна емулација на хардвер?

Фактори кои се користат при проценката на апликацијата

- Разбирливост - Колку е тешко да се разбере изворниот код на тековниот систем? Колку се комплексни контролните структури што се користат? Дали варијаблите имаат значајни имиња што ја отсликуваат нивната функција?
- Документација - Која системска документација е достапна? Дали документацијата е комплетна, конзистентна и тековна?
- Податоци - Дали постои експлицитен модел на податоци за системот? До кој степен податоците се дуплираат преку датотеки? Дали податоците се користат од системот ажурирани и конзистентни?
- Изведба - Дали ефикасноста на апликацијата е соодветна? Дали проблемите со перформансите имаат значително влијание врз корисниците на системот?
- Програмски јазик - Дали се современи компајлери достапни за програмскиот јазик што се користат за развој на системот? Дали се уште се користи програмскиот јазик за развој на нов систем?
- Управување со конфигурацијата - Дали сите верзии на сите делови на системот управуваат со систем за управување со конфигурација? Дали постои експлицитен опис на верзиите на компонентите што се користат во тековниот систем?
- Податоци за тестот - Дали постојат податоци за тестот за системот? Дали има евиденција за тестовите за регресија кога се додадени нови функции во системот?
- Вештини за персонал - Дали има луѓе кои имаат вештини за одржување на апликацијата? Дали има достапни луѓе кои имаат искуство со системот?

Мерење на системот

- Може да соберете квантитативни податоци за да направите проценка на квалитетот на системот за апликација
  - Бројот на барања за промена на системот; Колку е поголема оваа акумулирана вредност, толку е понизок квалитетот на системот.
  - Бројот на различни кориснички интерфејси што ги користи системот; Колку повеќе интерфејси, толку е поголема веројатноста дека ќе има недоследности и технолошки вишок во овие интерфејси.

- Обемот на податоците што ги користи системот. Како што се зголемува обемот на податоци (број на датотеки, големината на базата на податоци, итн.) Обработени од системот, исто така прават недоследности и грешки во тие податоци.
- Чистењето на старите податоци е многу скап и одзема многу време

Одржување на софтвер

- Измена на програма откако ќе биде пуштена во употреба.
- Терминот најмногу се користи за промена на сопствен софтвер. Се вели дека генеричките софтверски производи се развиваат за да создадат нови верзии.
- Одржувањето вообичаено не вклучува големи промени во архитектурата на системот.
- Промените се спроведуваат со модифицирање на постојните компоненти и додавање на нови компоненти во системот.

Вид на одражувања (ISO/IEC 14764:2006)

- Корективно одржување: Реактивна модификација на софтверски производ извршен по породувањето за да се поправат откриените проблеми;
- Адаптивно одржување: модификација на софтверски производ извршен по породувањето за да се задржи софтверски производ употреблив во променета или променлива околина;
- Совршено одржување: Измена на софтверски производ по испорака за подобрување на перформансите или одржливоста;
- Превентивно одржување: Измена на софтверски производ по породувањето за откривање и корекција на латентни грешки во софтверот, пред да станат ефективни грешки.

Дистрибуција на напор за одржување

- Корективно (приближно 21%)
  - 12,4% дебагирање во итни случаи
  - 9,3% рутинско дебагирање
- Адаптивно (приближно 25%)
  - 17,3% адаптација на околината на податоците
  - 6,2% промени во хардвер или оперативен систем
- Совршен (приближно 50%)
  - 41.8% додатоци за корисници
  - 5,5% подобрување на документација
  - 3,4% други
- Превентивни (приближно 4%)
  - 4,0% подобрување на ефикасноста на кодот

Видови на одржување

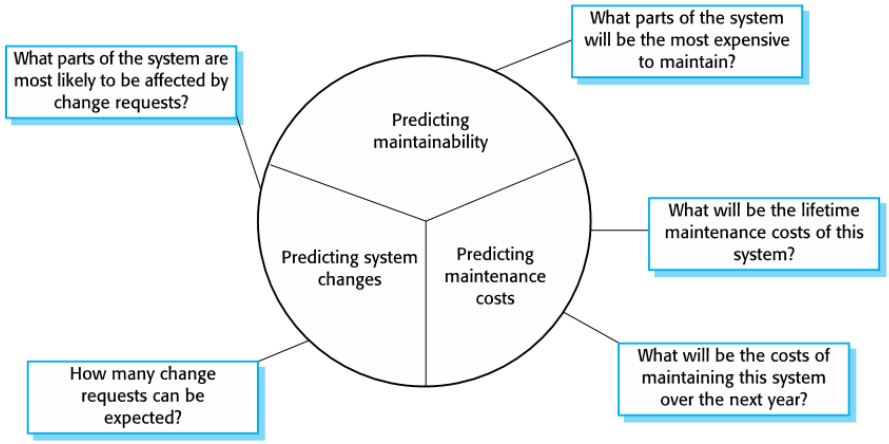
- Поправки на грешки - 24%
  - Промена на систем за отстранување на грешки / слабости и точни недостатоци на начинот на кој ги исполнува неговите барања.
- Прилагодување на животната средина – 19%
  - Одржување за прилагодување на софтверот во различна оперативна околина
  - Промена на систем така што работи во различно опкружување (компјутер, оперативен систем и сл.) Од првичното спроведување.
- Додавање и модификација на функционалност – 58%
  - Изменување на системот за задоволување на новите барања.

Трошоци за одржување

- Обично е поголема од трошоците за развој (2 * до 100 * во зависност од апликацијата).
- Погодени од технички и од нетехнички фактори.
- Се зголемува како што се одржува софтверот. Одржувањето ја корумпира структурата на софтверот и го отежнува понатамошното одржување.
- Софтверот што стареење може да има големи трошоци за поддршка (пр. Стари јазици, компајлери и сл.)
- Обично е поскапо да додавате нови функции на системот за време на одржувањето, отколку да ги додадете истите карактеристики за време на развојот
  - Новиот тим треба да ги разбере програмите што се одржуваат
  - Одвојување на одржување и развој значи дека нема поттик за развојниот тим да напишете одржлив софтвер
  - Работата за одржување на програмата е непопуларна
  - Персоналот за одржување е често неискусен и има ограничено знаење за домен.
  - Како што стареат програмите, нивната структура се деградира и тие стануваат потешки за промена

Предвидување за одржување

- Предвидувањата за одржување се занимаваат со проценка кои делови од системот можат да предизвикаат проблеми и имаат големи трошоци за одржување
  - Прифаќањето на промената зависи од одржливоста на компонентите погодени од промената;
  - Спроведувањето на промените го деградира системот и ја намалува неговата одржливост;
  - Трошоците за одржување зависат од бројот на промени и трошоците за промена зависат од одржливоста.



Предвидување на промените

- Предвидувањето на бројот на промени бара разбирање на односите помеѓу системот и неговата околина.
- Цврсто спојните системи бараат промени секогаш кога се менува околината.
- Фактори кои влијаат на оваа врска се:
  - Број и сложеност на системските интерфејси;
  - Број на барања за својствено непостојан систем;
  - Деловните процеси каде се користи системот.

Методи на сложеност

- Предвидувања за одржливост може да се направат со проценка на комплексноста на компонентите на системот.
- Студиите покажаа дека повеќето напори за одржување се трошат на релативно мал број на компоненти на системот.
- Комплексноста зависи од
  - Комплексноста на контролните структури;
  - сложеност на структурите на податоци;
  - Предмет, метод (постапка) и големина на модулот.

Методи на процеси

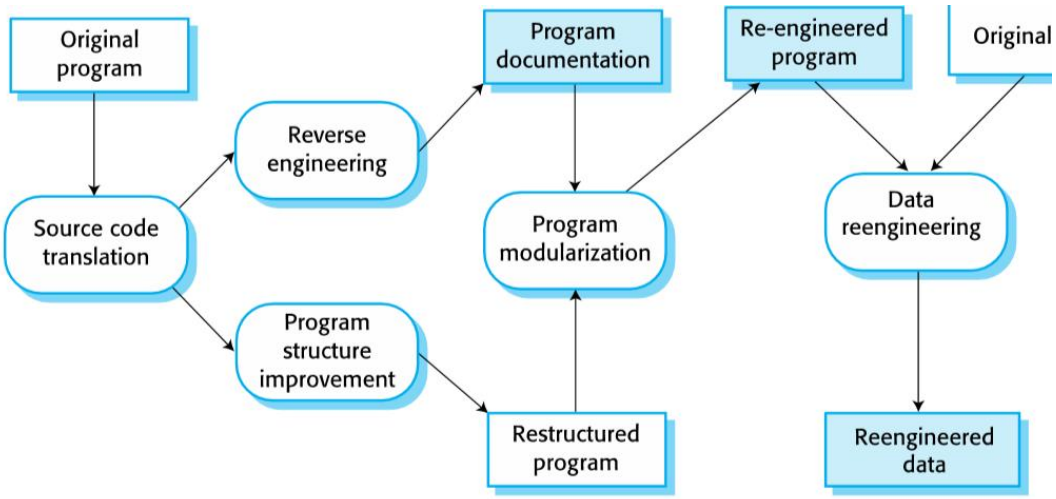
- Методите на процеси може да се користат за проценка на одржливоста
  - Број на барања за корективно одржување;
  - Потребно време за анализирање на влијанието;
  - Потребно време за спроведување на барање за промена;
  - Број на барања за исклучителна промена.
- Ако некој или сите овие се зголемуваат, ова може да укаже на пад на одржливоста.

Реинженеринг на софтвер

- Преструктуирање или препишување на дел или на сите наследни системи без промена на неговата функционалност.
- Се применува кога некои, но не сите под-системи на поголем систем, бараат често одржување.
- Реинженеринг вклучува додавање напор за да се олесни одржувањето. Системот може да се реструктуира и повторно да се документира.

Предности на реинженеринг

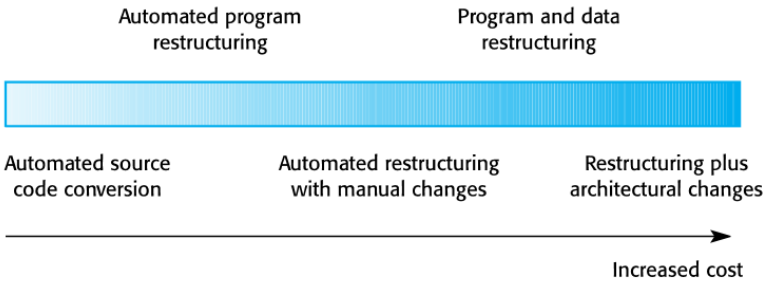
- Намален ризик
  - Постои висок ризик во развојот на новиот софтвер. Може да има проблеми со развојот, проблеми со вработените и проблеми со спецификации.
- Намалена цена
  - Цената на реинженерингот е често значително помала од трошоците за развој на нов софтвер.



Активности на процесот на реинженерирање

- Превод на изворен код - Претворете го кодот на нов јазик.
- Обратно инженерство - Анализирајте ја програмата за да ја разберете;
- Подобрување на структурата на програмата - Автоматска реконструкција за разбирливост;
- Модуларизација на програмата - Реорганизирање на структурата на програмата;
- Реинженерирање на податоците - Чистење и реструктуирање на податоците на системот.

Пристапи кај реинженерството



Фактори на трошоците за реинженерирање.

- Квалитетот на софтверот што треба да се реинженерира.
- Алатката за поддршка е достапна за реинженерирање.
- Степенот на конверзија на податоците што е потребно.
- достапност на стручен персонал за реинженеринг.
  - Ова може да биде проблем со старите системи засновани врз технологија која веќе не е широко користена.

Реновирање - Refactoring

- Реновирање е процес на правење подобрувања во програмата за да се забави деградацијата преку промени.
- Може да размислите за преобличување како„ превентивно одржување “што ги намалува проблемите со идните промени.
- Реновирање вклучува измена на програма за подобрување на нејзината структура, намалување на неговата сложеност или полесно разбирање.
- Кога одбивате програма, не треба да додавате функционалност, туку да се концентрирате на подобрување на програмата.

Реновирање и реинженерирање

- Реинженерингот се одвива откако системот се одржува некое време и трошоците за одржување се зголемуваат. Користете автоматски алатки за обработка и реинженерирање на наследен систем за да создадете нов систем што е пооддржлив.
- Реновирање е континуиран процес на подобрување во текот на процесот на развој и еволуција. Наменет е да се избегне деградацијата на структурата и кодот што ги зголемува трошоците и тешкотиите во одржувањето на системот.

„Лоши мириси“ во програмскиот код

- Дупликат код - Истиот или многу сличен код може да биде вклучен на различни места во програмата. Ова може да се отстрани и спроведе како единствен метод или функција што се нарекува како што се бара.
- Долги методи - Ако методот е предолг, тој треба да се редизајнира како голем број пократки методи.

- Изјави за прекинувач - Овие често вклучуваат дуплирање, кога прекинувачот зависи од видот на вредноста. Изјавите за прекинувачот може да бидат расфрлани околу програмата. На јазично-ориентирани јазици, честопати можете да користите полиморфизам за да постигнете иста работа.
- Збир на податоци - Групи на податоци се јавуваат кога истата група на податоци (полиња во класи, параметри во методите) повторно се појавуваат на неколку места во програмата. Овие често може да се заменат со предмет што ги опфаќа сите податоци.
- Шпекулативна општост - Ова се случува кога програмерите вклучуваат генералност во програмата, доколку се бара во иднина. Ова често може едноставно да се отстрани.