Софтуерна архитектура и разработка на софтуер

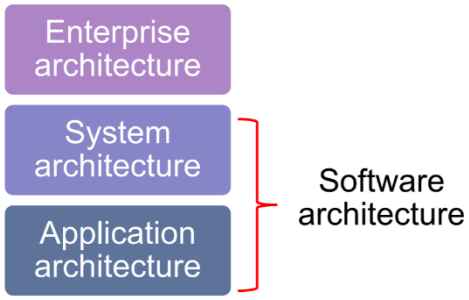
I. Софтуерна архитектура

1. Какво е софтуерна архитектура?

- Обикновено СА се създава като първа стъпка по време на проектирането, като целта е да се гарантира наличието на дадени качества в системата.
- Детайли като алгоритми, представяне на данни, реализация, и т.н. не са предмет на СА.
- Предмет на СА е поведението и връзките между различни елементи, разглеждани като "черни кутии".
- ***Архитектура на дадена софтуерна система е съвкупност от структури, показващи различните софтуерни елементи на системата, външно видимите им свойства и връзките между тях.***
- **The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.**

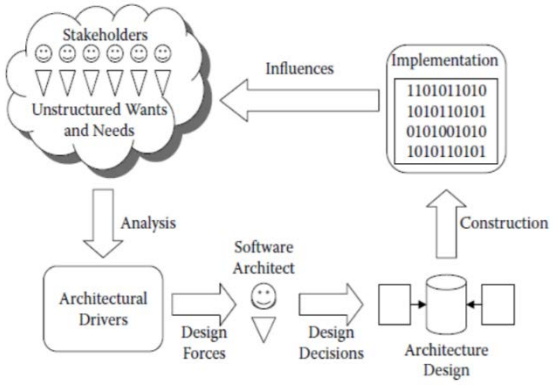
2. По-общо понятие за архитектура

- **Организационна архитектура (Enterprise architecture)** – основните процеси, технологичните и бизнес-стратегии в дадена организация.
- **Системна архитектура (System architecture)** – организацията на програмите и инфраструктурата, върху която те се изпълняват.
- **Архитектура на приложението (Application architecture)** – организация на приложение, подсистема или компонент.



3. От какво се определя СА?

- Широко разпространено е схващането, че СА зависи само от изискванията.
- Истината е, че се намесват и много други **фактори на обкръжението** (**environment**), а именно:
  a. **технически, бизнес и социални влияния**
  b. **опит, знания и умения на архитекта**
  c. **съвременните технологии**
- От друга страна, самото създаване на СА повлиява върху обкръжението, т.е. **процесът е цикличен**.



4. Stakeholder

- ***Stakeholder (заинтересовано лице) – това са всички, които имат отношение към създаването на софтуерната система*** – напр. собствениците, управителите, специалистите по продажби, ръководителя на проекта, разработчиците, екипа по поддръжка, различни прослойки от страна на клиента, крайните потребители и т.н.
- **Интересите на различните ЗЛ най-често си противоречат.**
- Архитектът е в неблагоприятна позиция – какъвто и ход да предприеме, все някой от списъка със ЗЛ ще е недоволен.
- Ролята му е да балансира между различните ЗЛ за бъдат конкретните интереси отразени в спецификацията на изискванията!

5. **Как стоят нещата всъщност:**
   - В много редки случаи изискванията, породени от бизнес целите, както и различните влияния, са напълно разбрани, обяснени и документирани.
   - Това води до конфликти между различните ЗЛ, които трябва да се разрешават.
   - За целта архитекта трябва да разбере същността, източниците и приоритетите на различните ограничения и трябва да управлява нуждите и очакванията на ЗЛ.
   - Крайната цел е ЗЛ да бъдат притиснати да приближат позициите си така, че да се намери пресечна точка между противоречивите на пръв поглед изисквания.
6. **Софтуерен архитект**
   - **Качества**
     a. отлично познаване на технологиите
     b. отлично аналитично и абстрактно мислене
     c. комуникативност, дипломатичност и умение за убеждаване и въобще за водене на преговори
   - **Дейности**
     a. **Вземане на бизнес решения за създаване на системите.**
        - Каква е целевата функция?
        - Колко ще струва?
        - За колко време?
        - В каква среда ще работи (интерфейси с други системи)?
        - Някакви ограничения?

     b. **Разбиране на изискванията.**
        - Изискванията (функционални и нефункционални) обуславят СА.
        - Те трябва да се дефинират по възможно най-недвусмислен начин.

     c. **Създаване или избор на архитектура.**
        - Същественото тук е, че успешен проект и разработка могат да се изградят само ако е налице идейна цялост, а идейна цялост може да се постигне само посредством последователен и подреден мисловен процес от страна на специализирани (малко на брой) архитекти.

     d. **Документиране на СА.**
        - Втората част от същинската работа на архитекта.
        - И най-добрата архитектура е безполезна, ако тя не бъде по подходящ начин представена на всички ЗЛ.
        - Формата, под която следва да бъде представена СА, зависи от конкретните ЗЛ.

     e. **Анализ и оценка на СА.**
        - Както при всеки процес на проектиране, и при създаване на СА най-вероятно има няколко варианта, които следва да се оценят и анализират, и да се избере най-добрият.
        - Архитектурите подлежат на оценка както по отношение на изпълняване на изискванията, така и по отношение на финансови параметри.

     f. **Създаване на системата.**
        - Ролята на архитекта по време на създаването (implementation) на системата е основно да следи дали се спазват предписанията на СА.
        - Това, че има прекрасна, добре документирана и преразказана архитектура е добре, но ако хората, които правят системата не я следват, ефектът е нулев.

     g. **Следене за наличие на съответствие между системата и СА.**
        - След като системата бъда разработена и премине във фаза на поддръжка, архитектът трябва да следи за съответствието между СА и системата – по време на поддръжката се налагат промени. Тяхната реализация следва да е съгласно принципите на архитектурата.

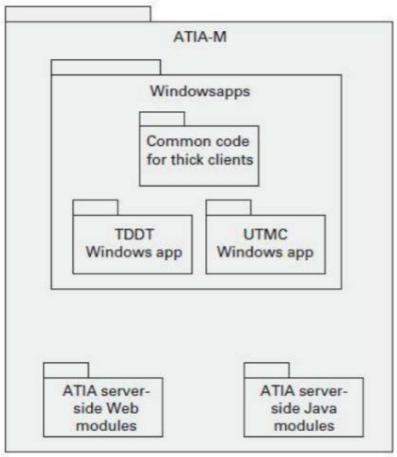
- От своя страна, СА също трябва да се адаптира към промените.

7. Архитектурни структури

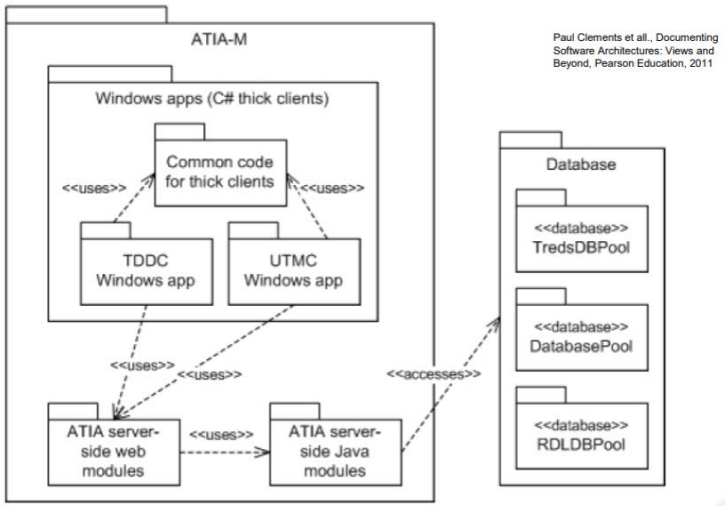
- ***Структура – съвкупност от софтуерни елементи, външно видимите им свойства и връзките между тях.***
- ***Изглед (view) – конкретно документирано представяне на дадена структура.***
- Двете понятия в голяма степен са взаимозаменяеми.
- Архитектурните структури се делят най-общо казано на 3 групи:

Модулни структури

- ***Елементите в модулните структури са модули – единици работа за изпълнение. Модулите предлагат поглед, ориентиран към реализацията на системата, без значение какво става по време на изпълнението.***
- Коя функционалност в кой модул се реализира?
- Кои други модули може да използва (и използва) дадения модул?
- Как са свързани модулите по отношение на специализация и генерализация (наследяване)?
- **Декомпозиция на модулите**
  a. **Декомпозиция на модулите – връзките между модулите са от вида "X е под-модул на Y".**
  b. Това се прави рекурсивно до момента, в който елементите станат достатъчно прости, че да могат да бъдат разбрани лесно.
  c. Декомпозицията на модулите обуславя в голяма степен възможността за лесна промяна, като обособява логически свързани функционалности на едно място.
  d. Много често служи и като основа на разпределението на работата между екипите на изпълнителя.

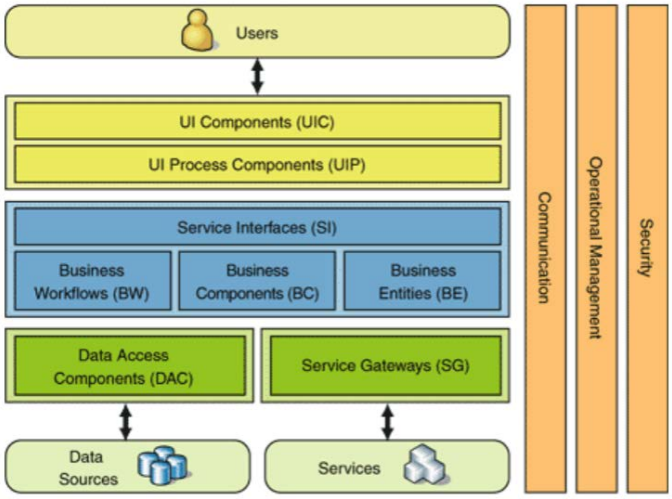


- **Употреба на модулите**
  a. **Употреба на модулите – връзките между модулите са от вида "X използва Y".**
  b. Ако има нужда от по-детайлно описание, може връзките да са насочени към конкретен интерфейс или ресурс на модула.
  c. Структурата за употребата на модули обуславя възможността за лесно добавяне на нова функционалност, обособяване на (в голяма степен) самостоятелни подмножества от функционалност, както и позволява последователната разработка, много важна и мощна техника за работа.



- **Структура на слоевете**

a. Като частен случай на структурата на употребата на модули е структурата на слоевете – когато върху употребата са наложени стриктни правила се обособяват слоеве.
b. **Модулите от слой номер N могат да се възползват само от услугите на модулите от слой N-1.**
c. Слоевете често са реализирани като виртуални машини или обособени подсистеми, които скриват детайлите относно работата си от следващия слой.

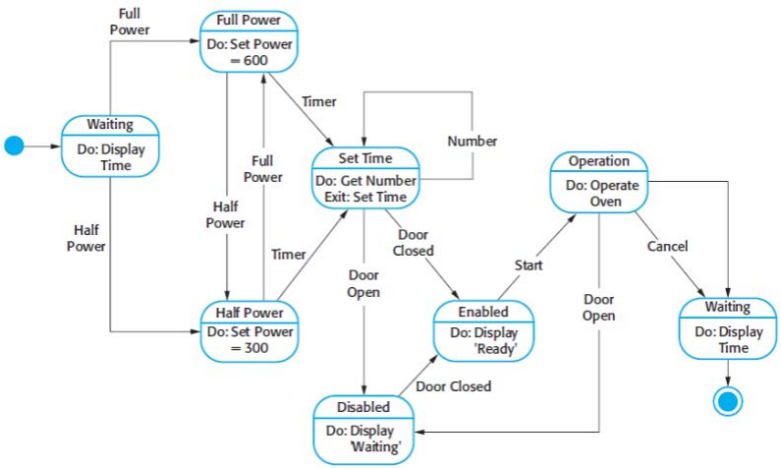


d. Не е прието (и е признак на лошо възпитание) слоеве да се прескачат.
e. Структурата позволява без особени сътресения да бъде подменен цял един слой (напр. да се смени СУБД).

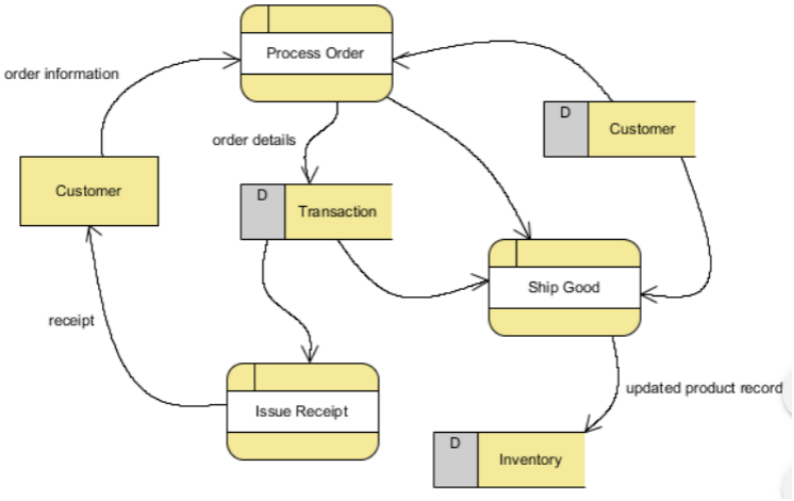
- **Йерархия на класовете**
  a. В терминологията на ООП, модулите се наричат “класове”, а в настоящата структура връзките между класовете са от вида “класът X наследява класа Y” и “обекта X е инстанция на клас Y”.
  b. Тази структура обосновава наследяването – защо подобни поведения или въобще функционалности са обособени в супер-класове или пък защо са дефинирани под-класове за обслужване на параметризирани различия.

Структури на процесите

- ***Елементите са компоненти, които се проявяват по време на изпълнението (т.е. основните изчислителни процеси) и средствата за комуникация между процесите.***
- ***Елементите са процеси (или нишки), изпълнявани в системата (компоненти) и комуникационни, синхронизационни или блокиращи операции между тях (конектори). Връзките между тях (attachments) показват как компонентите и конекторите се отнасят помежду си.***
- Структурата е полезна, тъй като има отношение по въпросите на бързодействието по време на изпълнението и високата надеждност.
- Кои са основните изчислителни процеси и как те си взаимодействат?
- Кои са основните споделени ресурси?
- Как се развиват данните в системата?
- Кои части от системата могат да работят паралелно?
- Как се променя структурата на системата докато тя работи?

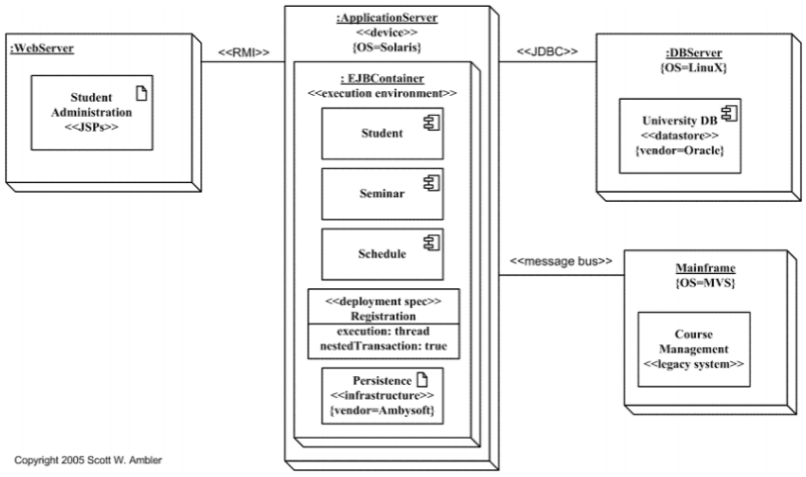


Структура на потока на данните



Структури на разположението

- ***Структурите на разположението показват връзката между софтуерните елементи и елементите на околната среда, в която се намира системата по време на разработката или по време на изпълнението.***
- На кой процесор се изпълнява всеки от елементите?
- В кои файлове се записва сорс кода на елементите по време на разработката?
- Какво е разпределението на софтуерните елементи по екипи, които създават системата?
- **Структура на внедряването**
  a. Показва как софтуера се разполага върху хардуера и комуникационното оборудване.
  b. Елементите са процеси, хардуерни устройства и комуникационни канали.
  c. Връзките са напр. “внедрен върху” или “мигрира върху”.
  d. Представлява интерес при разпределени системи и позволява да се разберат особеностите относно бързодействието, интегритета на данните, надеждността, сигурността и т.н.



- **Файлова структура**
  a. Показва кой модул къде се помещава, по време на различните фази на реализация.
  b. Структурата е критична за управлението на дейностите по разработка и за създаването и поддържането на обкръжение за build-ове.
- **Разпределение на работата**
  a. Показва кой модул от кой екип се реализира.
  b. Елементите са модули и екипи, а връзките са кой модул от кой екип се разработва.
  c. Под “кой екип” не се има предвид конкретен списък от хора, а по-скоро виртуална група хора с подходящ опит, знания и умения.
  d. Архитектът трябва да знае какви хора са необходими за изработка на модулите и да участва във вземането на управленчески решения.

e. Структурата помага и за това дадени общи функционалности да бъдат обособени и разработени от един екип, вместо всеки сам да си ги прави.

Кои структури да използваме?

- **Logical** – елементите са ключови абстракции, а връзките – взаимодействие между тях. Може да се класифицира като обектно-ориентирана модулна структура.
- **Process** – адресира паралелното изпълнение и разпределението на процесите. Типична СКК.
- **Development** – типична структура на разпределението на работата.
- **Physical** – кой процес на кой хардуер се изпълнява, типична структура на внедряването.

8. 4 + 1 модел на софтуерната архитектура

- **Логически изглед** – показва основните абстракции в системата, като обекти, класове и компоненти.
- **Изглед на процесите** – показва системата като съвкупност от взаимодействащи си процеси по време на изпълнение.
- **Изглед на кода** – показва как отделните елементи на системата се разполагат във файлове код.
- **Физически изглед** – показва как софтуерните компоненти са разпределени между хардуерните възли в системата.
- **+1** – съответните сценарии на употреба.

II. Софтуерна архитектура и изисквания към системата

1. Архитектурни драйвери

- За проектиране на архитектурата е достатъчно да се започне с няколко най-важни изисквания.
- **Най-важните изисквания ги наричаме архитектурни драйвери.**

Съществуват две големи групи изисквания към софтуерните системи – функционални и качествени изисквания.

- **Функционалните изисквания** определят **какво** трябва да прави софтуерната система.
- **Качествените изисквания** определят **как** софтуерната система да работи. На практика качествата поставят ограничения върху начина, по който системата ще се изпълнява.

2. Видове качествени характеристики

- **Технологични качества** – надеждност, изменяемост, производителност, сигурност, изпитаемост, използваемост и др.
- **Бизнес качества** – време за пускане на продукта на пазара и др.
- **Архитектурни качества** – присъщи на самата архитектура като напр. идейна цялост (влияят косвено върху всички останали качества).

3. Функционалност и качества

- **СА определя структурите на системата в контекста на нефункционалните изисквания**. Например, дадена система може да бъде разделена на модули така, че няколко екипа да могат да работят паралелно по тях, което намалява времето за пускане на продукта на пазара. Това на практика е нефункционално (бизнес) изискване.
- **Бизнес целите определят качествата, които трябва да бъдат вградени в архитектурата на системата**. Тези качества поставят изисквания отвъд функционалните (описание на основните възможности на системата и услугите, които тя предоставя).
- **Въпреки, че функционалността и качествата са тясно свързани, функционалността често е единственото, което се взема под внимание по време на проектирането.**
- **Като следствие, много системи се преправят не защото им липсва функционалност, а защото е трудно да се поддържат, трудно е да се смени платформата, не са скалируеми, прекалено са бавни, или пък са несигурни.**
- **СА е тази стъпка в процеса на създаването на системата, в която за пръв път се разглеждат качествените изисквания и в зависимост от тях се създават съответните структури, на които се приписва функционалност**.
- Решенията, които прави архитекта по време на създаването на СА, са определящи за постигането на необходимите нива на съответните качества.
- Ако функционалността беше единственото изискване, то системата може да се реализира и като един монолитен блок, без нуждата от описание на вътрешната структура.
- **Декомпозицията на модули прави системата разбираема и осигурява постигането на много други цели. В този смисъл, декомпозицията на модули е в голяма степен независима от функционалността.**

4. Архитектура и качества

- За да притежава дадена система изискваните качествени характеристики, те трябва да се имат предвид както по време на проектирането, така и по време на разработката и внедряването.
- Постигането на задоволителни резултати е въпрос на правилното разбиране както на общата картина (СА), така и на детайлите (разработката).
- Постигането на качествата е въпрос както на архитектурни, така и на не-архитектурни решения.

Примери

- **Използваемост** – не-архитектурните аспекти включват това, интерфейсът да е ясен и лесен за употреба, например разположение на елементите по екрана, избор на ясен шрифт и подходяща цветова схема. Архитектурните решения включват предоставянето на възможности за cancel, undo, reuse на предишно въведени данни и т.н.
- **Изменяемост** – определя се в голяма степен от декомпозицията на модулите (арх. решение), но дори и най-добрата архитектура няма да помогне за изменяемостта, ако програмистите не се придържат към нея.
- **Производителност** – зависи от комуникацията между компонентите, разпределението на функционалността между компонентите, разпределението на споделени ресурси (архитектурни аспекти), изборът на алгоритми и тяхната реализация (не-архитектурни аспекти).

5. Отношения между качествата

- Качествата не могат да се разгледат в изолация – постигането на дадено качество обикновено оказва влияние (положително или отрицателно) върху постигането на другите качества.

Примери

- **Сигурност vs. отказоустойчивост** – въпреки, че обикновено ги поставят в една и съща графа, те си противоречат – сигурността изисква наличието на single point of failure, т.е. системата да може да се компроментира от едно единствено място. Обратно, отказоустойчивостта предполага репликация.
- **Преносимост vs. производителност** – основната техника за постигането на преносимост е изолирането на зависимостта от ОС и хардуер в специализирани модули, което внася допълнителна тежест по време на изпълнението и намалява производителността.

6. Качество

- Качеството е субективно възприятие – различните ЗЛ могат да не одобрят даден дизайн, тъй като тяхната идея за качество се различава от идеята за качество на архитекта.
- Изискванията за качество трябва да се формализират от архитекта посредством т.н. "**сценарии за качество**", за да бъдат те поставени на обективна основа.
- Сценариите демонстрират какво е качество в рамките на създаваната система, като дават на архитекта и на ЗЛ еднозначна основа за оценка на дизайна.

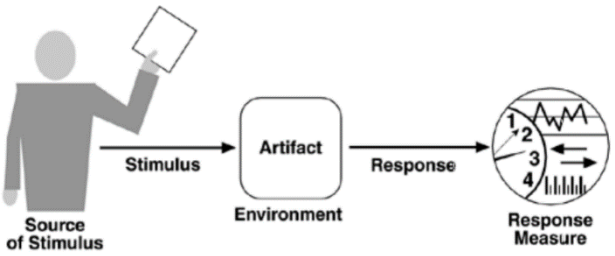
7. Технологични качества

- Безмислено е да се говори за дадена система, че е "изменяема" – всяка система е изменяема по отношение на дадена промяна и неизменяема по отношение на друга промяна. Ситуацията с другите качества е подобна.
- Често се водят спорове към кое качество принадлежи даден аспект на системата. Дали отказ в системата е аспект на наличността, на сигурността или на надеждността?
- За всяко качество си има собствен речник. Специалистите по производителност говорят за "събития", тези по сигурност – за "атаки", тези по надеждност – за откази, и т.н. Всички тези термини всъщност могат да обозначават едно и също събитие.
- За избягването на тези проблеми е необходимо да се борави с т.нар. **сценарии за качество**.

8. Сценарии за качество

- **Сценарият за качество е специфично изискване към поведението на системата в дадена ситуация, в светлината на дадено качество.**
- Сценариите за качество играят същата роля за дефиниране на нефункционалните изисквания, каквато роля играят сценариите за употреба (usecases) за дефиниция на функционалните изисквания.
- **Всеки сценарий описва някаква случка и се характеризира с 6 компонента:**
  a. **Въздействие** – състояние или събитие, което подлежи на обработка.

b. **Източник** – обект (човек, система или нещо друго), който генерира въздействието.
c. **Обект** – системата, или конкретна нейна част, върху която се случва въздействието.
d. **Контекст** – условията, при които се намира обекта по време на обработка на въздействието.
e. **Резултат** – действията, предприети от обекта при случването на въздействието.



f. **Количествени параметри** – резултатът трябва да подлежи на някакви количествени измервания, така че да позволи проверката дали сценарият се изпълнява съгласно Изискванията.

Примери

- **Сценарий за надеждност**

По време на експлоатация на системата, външен източник изпраща на процеса X, съобщение за препълване на опашката с потребителски заявки. X трябва да информира оператора за получаването на съобщението и да продължи работа без прекъсване.

„По време на експлоатация на системата,… (**Контекст**) …външен източник… (**Източник**) …изпраща на процеса X…. (**Обект**) ...съобщение (**Въздействие**) за препълване на опашката с потребителски заявки. Процесът трябва да информира оператора за получаването на съобщението и да продължи работа… (**Резултат**) …без прекъсване" (**Количествени параметри**).

- **Сценарий за изменяемост**

Преди пускането на системата в експлоатация, клиентът желае промяна на фоновия цвят на екрана. За целта се променя изходния код. Това не трябва да предизвиква никакви странични ефекти в поведението на системата, като времето за извършване на промяната, вкл. тестването й трябва да отнеме по-малко от 3 часа.

„Преди пускането на системата в експлоатация… (**Контекст**)…клиентът… (**Източник**)…желае промяна на фоновия цвят на екрана…. (**Въздействие**). За целта се променя изходния код…. (**Обект**). Това не трябва да предизвиква никакви странични ефекти в поведението на системата. (**Резултат**). Времето за извършване на промяната, вкл. тестването й трябва да отнеме по-малко от 3 часа" (**Количествени параметри**).

9. Изправност (dependability)

- **надеждност (reliability) + готовност или наличност(availability) + безопасност (safety) + сигурност (security) + отказоустойчивост (fault-tolerance) + възможност за промяна (modifiability)**
- **Изправността (dependability) се занимава с отказите (failures), наричани още сривове в системата и свързаните с това последствия.**
- **Отказ в системата настъпва, когато тя престане да предоставя услугите си съгласно спецификацията.**
- Отказът е наблюдаем от потребителите (хора или други системи).
- Интерес представляват обстоятелствата, свързани с откриването на отказ, колко често може да се случва отказ, за колко време системата може да преустанови работата си, кога отказите са безопасни, как могат да бъдат избегнати и какви уведомления се генерират, когато се получи отказ.

- **Трябва да се прави разлика между отказ и дефект. Дефектът (fault) може да стане срив ако не бъде поправен или замаскиран. Сривът е наблюдаемо събитие, докато дефектът не е. Когато дефектът стане наблюдаем, той се превръща в срив.**
- **Докато системата е отказала, важна характеристика става времето, за което тя ще бъде поправена**.
- Това време може да трае от няколко милисекунди до няколко дни.
- Наличността на системата се дефинира като вероятността тя да бъде в изправност, когато има нужда от нея. Представя се чрез формулата -> Δt f е средното време между отказите, а Δtc е средното време за отстраняване на повредата.

$$\alpha = \frac{\Delta t_f}{\Delta t_f + \Delta t_c}$$

- **Сценарии за изправност**

Компо-нент	**Възможни стойности**
Източник	Вътрешен за системата, външен за системата
Въздействие	Срив: липса на отговор, счупване, ненавременно събитие, неправилни отговори
Обект	Хардуерни устройства, комуникационни канали, постоянна памет, процеси
Контекст	Нормална работа, временна намалена работоспособност, временно решение за отстраняване на срив
Резултат	Системата трябва да регистрира събитието и да: - го запише; - извести когото е необходимо; - да забрани входа от източника; - да остане недостъпна; - да продължи да работи в нормален или специален режим;
Количес-твени параметри	Процент надеждност (α) Интервали (или продължителност), в които системата трябва да е работоспособна Време за отстраняване на срива Интервали (или продължителност), в които системата може да е в състояние на намалена работоспособност

10. Изменяемост (modifiability)

- **Изменяемостта на системата е свързана със себестойността на промените.**
- **Сценарии за изменяемост**

Компонент на сценария	**Стойности**
Източник	Разработчици, системни администратори, крайни потребители
Въздействие	Искат да се добави/премахне/промени/ функционалност, качествено свойство, капацитет и др.
Обект	Потребителския интерфейс, платформата, обкръжението, конкретен модул, системата и т.н.
Контекст	По време на разработката, по време на компилацията, по време на билд-а, по време на конфигурацията, по време на изпълнението
Резултат	Намира местата, които подлежат на промяна; прави промените, без това да се отразява на останалата функционалност; проверява промените; внедрява промените
Количествени параметри	Стойност, в смисъл на брой променени елементи, усилие, пари и доколко промяната се отразява на останалата функционалност и свойства

11. Производителност (performance)

- **Различни събития (прекъсвания, съобщения, заявки от потребителя, или пък самото преминаване на времето) се случват и системата трябва да реагира на тях.**
- **Предмет на производителността е времето, за което системата реагира на възникващите събития.**

- Сценарият за производителност започва със заявка за извършване на някаква услуга. Обслужването на заявката е свързано с консумацията на някакви ресурси. В същия момент системата може да обслужва паралелно и други заявки.
- Схемата, по която се случват събитията, може да бъде характеризирана като периодична или стохастична.
- Например, периодично събитие може да се случва всеки 10 ms. Най-често периодичните събития се срещат в real-time системите.
- Стохастичните събития пристигат в съответствие с някакво вероятностно разпределение.
- Събитията могат да пристигат и спорадично, т.е. не се поддават нито на периодично, нито на стохастично описание.
- **Сценарий за производителност**

Компонент на сценария	**Стойности**
Източник	Множество източници, потребителски заявки, други системи, вътрешни за системата и т.н.
Въздействие	Случва се периодично, стохастично или спорадично събитие
Обект	Системата или конкретен неин процес
Контекст	Нормален режим; режим на претоварване
Резултат	Обработва събитието; променя качеството на обслужване
Количествени параметри	Латентност; времева граница; пропускливост; отклонение; брой необработени заявки и т.н.

Примери

- **Уеб финансова система получава заявки от потребителите, вероятно десетки или стотици хиляди на ден.**

За уеб-базираната финансова система е важно колко транзакции могат да бъдат обработени за 1 мин.

- **Система, за управление на двигател на автомобил получава събития просто с течение на времето и контролира както запалването, така и смесването на гориво и въздух за постигането на максимална мощност.**

За системата за управление на двигателя е важно максималното допустимо отклонение от идеалната точка за задействане.

12. Сигурност (security)

- **Сигурността е мярка за способността на системата да устоява на опити за неразрешена употреба, без това да пречи на легитимните потребители.**
- Опитът да се компроментира сигурността се нарича “атака” (или “заплаха”) и може да приеме множество различни форми.
- За сигурните системи са характерни невъзможността за отричане (non-repudiation), поверителността (confidentiality), интегритет, осигуреността (assurance), наличността (availability), проверяемостта (auditing).
  a. **Невъзможност за отричане** – системата не позволява на страните в транзакцията отричането ѝ.
  b. **Поверителност** – данните и функционалността са защитени от неправомерен достъп.
  c. **Интегритет** – данните и услугите се предоставят във вида, в който това е предвидено.
  d. **Осигуреност** – участниците в транзакцията са тези, за които се представят.
  e. **Наличност** – системата е достъпна за легитимна употреба.
  f. **Проверяемост** – системата проследява събитията, които се случват в нея.

- **Сценарий за сигурност**

Компонент на сценария	**Стойности**
Източник	Човек или система, който/която е: - идентифициран; неправилно идентифициран; неидентифициран; - външен; вътрешен; оторизиран или не-оторизиран; - с ограничен достъп; с неограничен достъп
Въздействие	Опитва се да види данни; да промени/изтрие данни; да използва системни услуги; да предотврати/намали достъпа до системни услуги
Обект	Системни услуги и данни
Контекст	Online; Offline; свързана; несвързана; с/без защитна стена
Резултат	Допуска потребителя; скрива идентичността му; блокира достъпа до данни/услуги; дава достъп до данни/услуги; дава/отнема права за достъп до данни/услуги; записва опитите за промяна/изтриване; записва данните в криптиран вид; различава необяснимо високи нива на активност; информира заинтересовани лица; ограничава ползваемостта
Количествени параметри	Време/усилия/ресурси, необходими за заобикаляне на сигурността с вероятност за успех; вероятност за разкриване на атаката; вероятност за разкриване на самоличността на извършителите; процент на работоспособност (напр. при DoS); възможности за възстановяване; обхват на пораженията

13. Изпитаемост (testability)

- **Изпитаемостта е лекотата, с която софтуерът може да бъде накаран да покаже дефектите си посредством извършване на различни тестове.**
- Около 40% от себестойността на грамотно изработените системи е свързана с тестването.
- **По конкретно, изпитаемостта е вероятността по време на следващото пускане на тестовете да бъде открит поне един дефект на системата (ако има такива).**
- На практика пресмятането на тази вероятност не е възможно и количествените параметри на сценариите обикновено включват други метрики.
- За да бъде една система правилно изпитаема, трябва да е възможно да се контролира вътрешното състояние и входовете на всеки един от компоненти ѝ и да се наблюдават изходите му.
- Тестването се прави от различни програмисти, тестери, потребители, по различно време. Тестват се порции от кода, от дизайна, цялата система и т.н.
- Количествените параметри са свързани с това, колко са ефективни тестовете и колко време отнемат за да се достигне предварително ниво покритие.
- **Сценарий за изпитаемост**

14. Използаемост (usability)

Компонент на сценария	**Стойности**
Източник	Разработчик; интегратор; проверител; проверител на клиента; потребител
Въздействие	Завършена е първоначална версия на анализа; архитектурата; проекта; класа; подсистемата; завършена е интеграцията на подсистемите; системата е цялостно завършена
Обект	Части от анализа, проекта, парчета код, цялата система
Контекст	По време на проектирането; разработката; компилацията; внедряването
Резултат	Дава достъп до вътрешните променливи и състоянието, изчислените стойности, създава тестова установка
Количествени параметри	Процент на изпълнените операции, вероятност за регистриране на дефектите, време за изпълнение на тестовете, дължина на най-дългата верига от зависимости в тестовете, време за подготовка на тестовата установка

- **Използваемостта е свързана с това, колко лесно потребителя успява да свърши дадена задача и каква подкрепа му оказва системата в това му начинание.**
  a. **Обучение** – ако потребителя не е запознат с даден аспект на системата, какво може тя да направи за да улесни процеса на обучение?

b. **Ефикасност** – какво може да направи системата за да работи потребителя по ефикасно?
c. **Устойчивост** – какво може да направи системата, така че да намали последствията от потребителски грешки?
d. **Адаптивност** – как може потребителя (или системата) да се адаптира, така че да улесни работата?
e. **Увереност** – с какво помага системата за това, потребителя да е сигурен, че е извършил правилно дадено действие?

- **Сценарий за използваемост**

Компонент на сценария	**Стойности**
Източник	Крайния потребител
Въздействие	Иска да научи нова функционалност; да използва системата ефикасно; да намали ефекта от грешки; да адаптира системата; да се чувства уверен
Обект	Системата или някаква нейна част
Контекст	По време на изпълнението/конфигурирането
Резултат	- за научаване на нова функционалност: контекстно ориентирана система за помощ; стандартизиран интерфейс, познат на потребителя; - за постигане на ефикасност: агрегация на данни и функционалност, повторна употреба на вече въведени данни и команди, ефикасна навигация в рамките на екраните и между тях, постоянство в интерфейса, разширено търсене, възможност за няколко едновременни действия; - за намаляване на ефекта от грешки: отмени; откажи; възстановяване от грешки; разпознаване и отстраняване на потребителски грешки; възстановяване на забравена парола; проверка на системните ресурси; - за постигане на увереност: показва моментното състояние; прогрес при дълги операции; визуална обратна връзка; работи в ритъма на потребителя
Количествени параметри	Време за извършване на задачите; брой грешки; брой на решените проблеми; потребителска удовлетвореност; научаване от страна на потребителя; отношението на успешните операции към неуспешните; изгубено време/данни

15. Бизнес качества

- В допълнение към технологичните качества, обикновено системата се оформя от набор от нефункционални бизнес изисквания.
- Те обикновено са свързани със себестойността, времето за изработка, пазара и пазарните условия.
- По същият начин, по който се формализират технологичните качества, се формализират и бизнес изискванията, за да няма двусмислие при тяхната комуникация.
- **Време за пускане на продукта на пазара (Time to market, TTM)** – ако има натиск от страна на конкуренцията, или пък има тесен прозорец за пускане на продукта, времето за разработка става важно. Обикновено TTM се намалява чрез използването на COTS продукти или преизползването на компоненти от предишни разработки (което пък е свързано тясно с декомпозицията на системата и структурата на употреба).
- **Себестойност и печалба** – всеки проект има бюджет, който не бива да се надвишава. Различните архитектури влекат след себе си различна себестойност. Например, ако архитектурата разчита на непозната технология, то най-вероятно тя ще излезе по-скъпа. Или ако пък е високо изменяема, тя най-вероятно ще струва повече (но пък ще се спести от поддръжката).
- **Предвидено време за живот на системата** – ако се предвижда системата да се експлоатира за дълъг интервал от време, следва тя да бъде проектирана така, че да е високо изменяема, скалируема и преносима. Това от своя страна би увеличило TTM, така че отново се налага компромис и за да бъде взето правилното решение, е необходимо да се формализират и приоритизират изискванията.

III. Архитектурни стилове

1. Логическия изглед има 4 нива на анстракция

- Компоненти и конектори
- Техните интерфейси
- Архитектурни конфигурации – конкретна топология за вътресвързани компоненти и конектори.
- Архитектурни стилове – шаблони за успешни и практически доказани архитектурни конфигурации.
  a. Определят набор от системи от гледна точка на шаблон на структурна организация.
  b. Стиловете определят граматиката на компоненти и конектори, които могат да бъдат използвани в инстанции на този стил.

2. Компонент

- ***Изпълнима част, която има определена функционалност, достъпвана чрез добре дефинирани интерфейси (входен и изходен).***

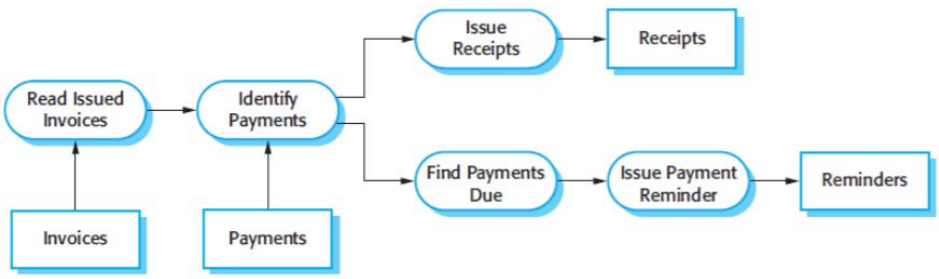
3. Конектор

- Същност от първи клас, която показва комуникационния механизъм (протокола между компонентите).
- Конекторите също имат интерфейси, понякога наричани роли.
- И компонентите и конекторите могат да бъдат преизползвани.

4. Архитектурни стилове

Pipe and Filter

- **Всеки компоненти (филтър) предава информация в последователен ред на следващия компонент.**
- **Конекторите (pipes) между филтрите показват предавателния механизъм на данните.**
- Филтрите представляват изчислителните части в системата.
  a. Четат данни от входния интерфейс, изпълняват ги и изпращат данните на изходния интерфейс.
  b. Не знаят нищо за съседите си.
- **Пайповете са длъжни да предадат изхода от единия филтър на входа на другия.**



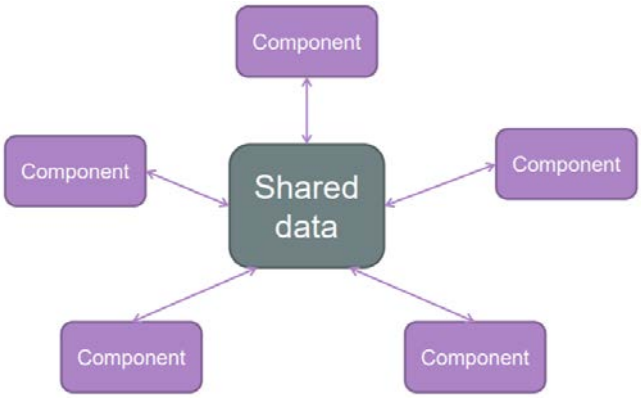
- **Вариации от гледна точка на комуникационен протокол**
  a. **Pipeline/stream** – изпълнението на данните може да започне веднага след като първият байт е получен от филтъра.
  b. **Batch-sequential style** – изисква цялата информация да бъде изпратена преди филтъра да започне да ги обработва.
- **Предимства**
  a. Интуитивен и лесен за разбиране.
  b. Филтрите са самостоятелни и могат да бъдат ретирани като “black boxes”, което води до гъвкавост от гледна точна на поддръжка и преизползване.
  c. Лесни да имплементират конкурентност.

d. Лесно приложими в структури с много бизнес процеси – когато изпълнението, изисквано от приложение, може да бъде декомпозирано в множество дискретни, независими стъпки.

- **Недостатъци**
  a. Заради последователните стъпки на изпълнение е трудно да се имплементират интерактивни приложения.
  b. Слаб performance – всеки филтър трябва да parse/unparse data, трудно е да се споделят глобални данни.
  c. Сложност (complexity) – в разпределена среда филтрите се изпълняват на различни сървъри.
  d. Надеждност (reliability) – използват инфраструктура, която осигурява прехода на данни между филтри в pipeline да не бъде загубен.
  e. Idempotency (excuse me what the fuck!?) – намиране и премахване на дублиращи се съобщения.
  f. Контекст и състояние (contex and state) – всеки филтър трябва да бъде доставен с достатъчен контекс, чрез който да работи. Това може да изисква значително количество информация за състоянието.

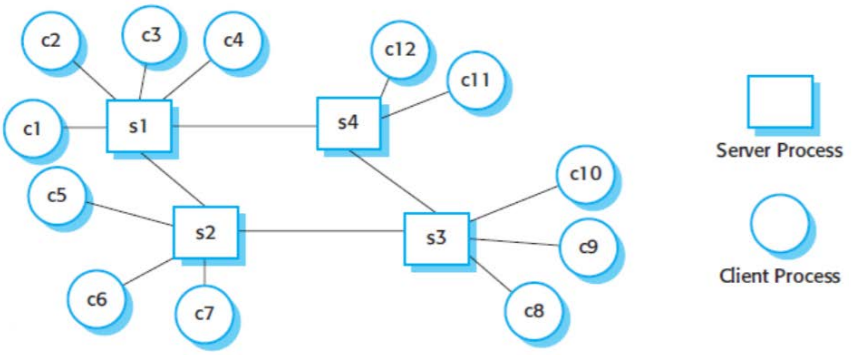
Shared data style

- **Активно използвана в системи, където компонентите трябва да предават големи количества данни.**
- **Споделените данни могат да се разглеждат като конектори между компонентите.**
- **Вариации**
  a. **Blackboard** – когато някакви данни са изпратени до конектора на споделената памет, всички компоненти трябва да бъдат информирани за това. С други думи, споделените данни са активен агент (!?!).
  b. **Repository** – споделените данни са пасивни. Никакви уведомления не се изпращат до компонентите.



- **Предимства**
  a. Scalability – нови компоненти може да се добавят и въпреки това системата да може да се мултиплицира.
  b. Concurrency – всички компоненти могат да работят паралелно.
  c. Високо ефективна, когато големи количества от данни се обменят.
  d. Централизирано управление на данни – по-добри условия за сигурност, backup и т.н.
- **Недостатъци**
  a. Трудно приложими в разпределена среда.
  b. Споделените данни трябва да поддържат uniform data model.
  c. Промени в модела могат да доведат до ненужни разходи.
  d. Тясна връзка между balckboard и източника на знания.
  e. Проблем при твърде много клиенти.

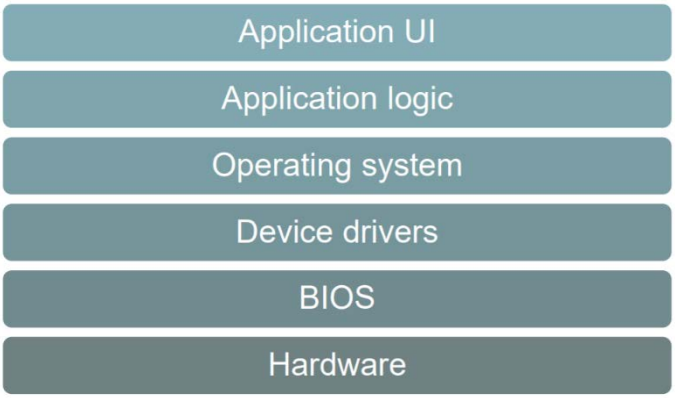
Client-server style



- **Системата е дизайната като множество от сървъри, които предлагат услуги на множество клиенти, които ги използват.**
- Не е задължителни сървърите да знаят за клиентите.
- Класическа имплементация – **thin client**
  a. **Клиентът имплементира UI функционалност.**
  b. **Сървъра имплементира функционалности по управление на данните и приложението.**
- **Fat клиентите могат да имплементират някои от функционалностите на приложението.**
- **Three tier client-server model** – по-добра производителност и сигурност.
- **Предимства**
  a. централизация на данните
  b. сигурност
  c. лесна имплементация
  d. backup and recovery
- **Недостатъци**
  a. Пренатоварване на сървъра при много клиенти.

Layered style

- **Представя системата като организация от йерархично подредени слоеве.**
- Класическа имплементация
  a. **Всеки слой предоставя услуги чрез интерфейс, но само за слоя, който е директно над него и използва услугите на слоя директно под него.**
  b. **По този начин един слой представлява сървър за горния слой и клиент за долния.**
  c. Интерфейсът може да бъде подобен на APIs.



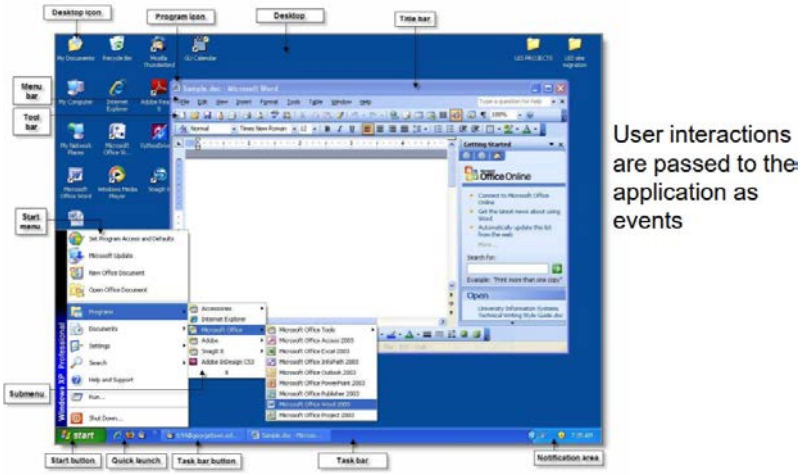
- Най-широко разпространеният стил във всички видове софтуерни системи.
- **Предимства**
  a. Вътрешната структура на слоя е скрита, ако интерфейсът е поддържан.
  b. Абстракция – минимизира сложността.
  c. По-добра свързаност – всеки слой поддържа подобни задачи.
- **Надостатъци**
  a. За много системи е трудно да разграничат отделните слоеве и това води до увеличаване на усилията за дизайн.
  b. Стриктните правила за комуникация компроментират производителността.
  c. Понякога вертикални слоеве могат да бъдат имплементирани.

Object-oriented style

- **Обектите представляват изчислителни единици.**
- **Обектите са отговорни за тяхната вътрешна цялостност.**
- **Вътрешното представяне е скрито от други обекти.**
- Конекторите са съобщения и/или извикване на методи.
- **Предимства**
  a. Енкапсулация на данните и логиката на програмата.
  b. Композиция на системата в множества от взаимодействащи си агенти.
- **Недостатъци**
  a. Обектите могат да знаят същностите на други обекти, за да си взаимодействат с тях.
  b. Странични ефекти в извикването на методи на обекта.

Implicit invocation style (publish-subscribe, event-based, message passing)

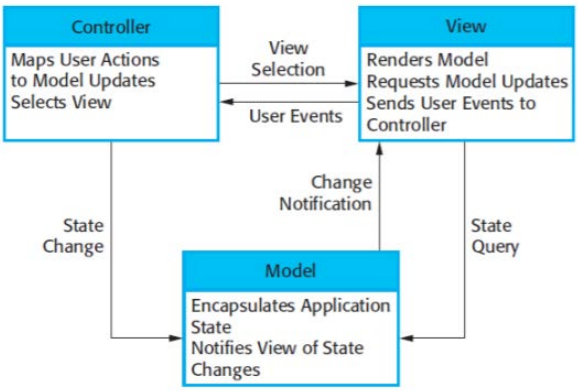
- **Компонентите в системата си взаимодействат чрез събития.**
- Събитията могат да съдържат не само контролни съобщения, а също и данни.
- Компонентите на този стил се изпълняват конкурентно и си комуникират чрез получаване или излъчване на събитие.
- Конекторите са event bus.



- **Предимства**
  a. Разграничаване – компонентите са лесни за замяна или преизползване.
  b. Голяма ефективност за разпределените системи – събитията са независими и могат да се обменят през мрежата.
  c. Сигурност – събитията са лесно проследими и logged.
- **Недостатъци**
  a. Неясна структура на системата – поредица от изпълнения на компоненти е трудна за контрол, трудно дебъгване.
  b. Не е сигурно дали съществува компонент, който да реагира на дадено събитие.
  c. Големи количества данни са трудни да се пренасят от събития.
  d. Проблем с надеждността – неизправност на event bus-а ще доведе до срив на цялата система.

Model-view controller style (MVC)

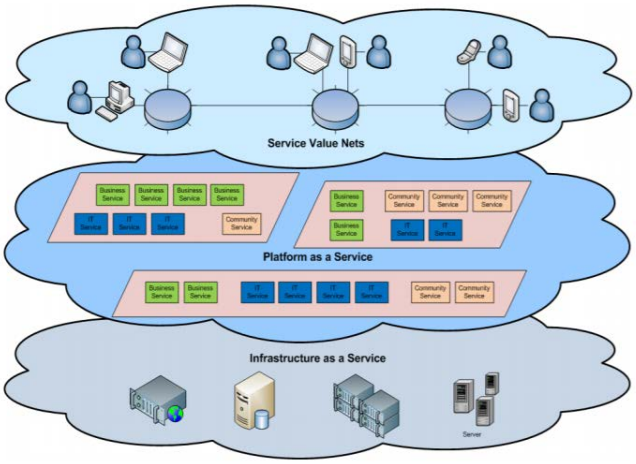
- **Предава независимост между данните, тяхното представяне и потребителя.**
- MVC представя знание. Управлява поведението на данните в домейна на приложението, изпраща информация за нейното състояние (на изгледа) и отвръща на инструкции за смяна на състоянието (обикновено от контролера).



- Изгледа има задача да управлява представянето на информация на потребителя.
- Контролера управлява взаимодействието с потребителя (mouse clicks, key pressed) и информира модела или изгледа да вземат нужните действия.
- **Предимства**
  a. Лесно се поддържа и имплементират бъдещи подобрения.
  b. Ясно разграничение между логиката на представяне и бизнес логиката.
  c. Лесна поддържка на нови типове потребители.
  d. Изгледът е отделен и в повечето системи претърпява много промени.
- **Недостатъци**
  a. Дори ако моделът на данните е прост, този стил може да предостави сложност и да изисква ненужен код.
  b. Не е подходящ за малки приложения.
  c. Проблем с производителността, когато се налагат чести ъпдейти в модела.

5. Архитектурни стилове в облака (никой не разбира облака)

- **Какво е cloud computing?**
  a. Главен термин, използван, за да описва нов клас на network based изчисление, което се намира в нета.
     - Група от интегриран и мрежово свързан хардуер, софтуер и интернет инсфрастуктура (платформа).
     - Използването на интернет за комуникация и транспорт осигурява хардуерні, софтуерни и мрежови услуги за клиента.



  b. Платформите прикриват сложността и детайлите на прилежащата инфраструктура от потребители и приложения, като осигуряват много прост графичен интерфейс или API.
  c. Cloud computing се използва за реферирane към интернет базираната разработка и услуги.
  d. **Характеристики, определящи cloud data, applications services and infrastructure:**
     - **Отдалечено съхранение** – услугите или данните са съхранявани на отдалечена инфраструктура.
     - Услугите или данните са достъпни отвсякъде.
     - Плащаш само за това, което искаш.
- **SaaS (Software as a service)**
  a. **Методология за доставка на софтуер, която предоставя лицензиран достъп от много „наематели“ до софтуер и неговата функционалност, отдалечно като web-базирана услуга.**
  b. Крайните потребители са консуматори на облака и не управляват прилежащата инфраструктура на облака.

- **PaaS (platform as a service)**
  a. **Осигурява всички съоръжения, нужни за поддръжка на целия жизнен цикъл на разработване и доставяне на web приложения и услуги изцяло в интернет.**

b. Обикновено приложенията трябва да бъдат разработвани на специфична платформа.
c. Cloud консуматорите са разработчици или ситемни администратори.
d. Платформата осигурява услуги като бази от данни, scalability, среди за разработка.
e. Консуматорите внедряват приложения, използвайки способностите, осигурени от доставчиците на cloud.
f. Консуматорите не управляват cloud инфраструктурата, но контролират внедряването на приложения.
g. Cloud доставчиците предоставят интернет базирана платформа на разработчиците, които искат да създават услуги, но не искат да изграждат собствен cloud.

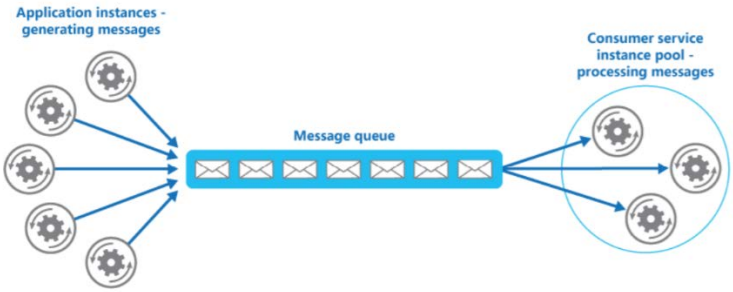
- **IaaS (infrastructure as a service)**
  a. **Представлява доставка на изисквана техническа инфраструктура като scalable услуга.**
  b. Cloud консуматорите са разработчици или системи администратори.
  c. Обработка, съхранение, мрежи и други ресурси, където консуматорите внедряват и изпълняват софтуера.
  d. Консуматорите имат контрол над операционните системи, съхранението и внедрените приложения.
  e. Потребителите на cloud-а наемат място за съхранение, изпълнение и поддържка от cloud доставчиците.

Circuit breaker style (прекъсвач)

- **Услугата се проваля в разпределена среда.**
- **Класическо решение е да се имплементира изчакване за други услуги, които използват circuit breaker.**
- Това може да доведе до ненужно използване на ресурси в cloud средата (например, стотици потребители, чакащи провалена услуга и всеки от тях да чака за определен timeout).
- Предотвратява приложенията да изпълняват операция, която е слабо вероятно да успее.
- Държи се като proxy за операции, които могат да се провалят.
- Показва броят на последните сривове, които са се появили и използва тази информация да реши дали тази операция да продължи или да хвърли грешка.
- **Помага да се повиши надеждностаа на системата.**
- Едно или много приложения, пращащи заявки, не биха чакали ненужно за timeout, когато услугата е счупена, няма интернет или услугата е заета и не може да отговори.

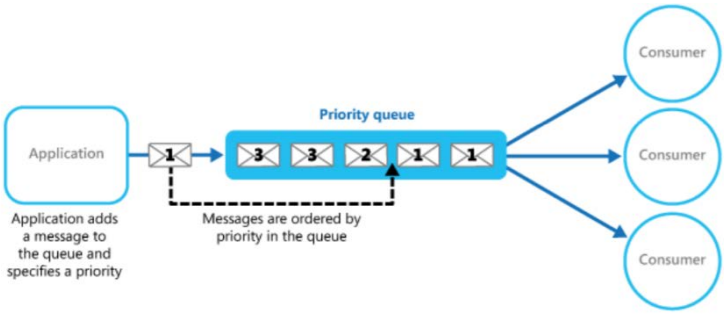
Queue

- В cloud-а може да има услуги, които са затрупани от множество конкурентни заявки от други услуги. В този случай те може да се пренатоварят. Трябва да се намери решение за големи пренатоварвания, които могат да накарат услугата да се провали или да й забавят изпълнението.



- **Типове**
  a. **Standart queue**
    - Опашката се държи като буфер, който записва съобщенията, докато не се достъпят от услугите.
    - Услугата достъпва съобщенията от опашката и ги изпълнява.
    - Минимизира риска на наличността чрез много на брой конкурентни заявки.

b. **Priority queue**
   - Имплементира политика за сортиране на входните заявки в зависимост от техния приоритет.
   - Приоритетът на заявките може да бъде зададен от изпращащото приложение или от самата опашка.



c. **Fixed length queue**
   - Може да проектираме опашката така, че да изпраща изключения, когато определен брой съобщения е достигнат.
   - В този случай задачата, която изпраща заявката, ще знае, че заявката няма да бъде изпълнена и може да предприеме адекватни действия без да чака.

Cache

- **Използва се, за да оптимизира повтарящ се достъп до информация.**
- **Четене**
  a. Търси елемент в кеша.
  b. Ако го няма, взема го от data store-а.
  c. Запазва копие в кеша.
- **Писане**
  a. Прави промени в data store-а.
  b. Анулира елемента в кеша.

Sharding style

- **Целта е да се повиши scalability, когато се използват големи количества данни.**
- **Идеята е да се раздели data store-а в множество хоризонтални подразделения (парчета).**
- Решението за това как да се раздели информацията между парчетата е важно.
- Едно парче обикновено съдържа части, които попадат в определен интервал, определен от един или няколко атрибута.
- Атрибутите формират т.нар **shard key**.
- Физически организира информация – когато приложение съхранява или достъпва данни, sharding логиката ориентира приложението към подходящото парче.
- Проблеми, свързани с имплементацията на sharding логиката:
  a. Могат да бъдат имплементирани като част от кода за достъп до данните в приложнието.
  b. Могат да бъдат имплементирани от системата за съхранение на данните, ако тя поддържа sharding (не мога вече с тази дума).
- Стратегии за имплементация
  a. Lookup strategy
  b. Range strategy
  c. Hash strategy
- **Предимства**
  a. По-добро управление на данните – абстракция на физическото местоположение на данните.
  b. Повишава производителността за съхранение на данните.
- **Недостатъци**
  a. Парчетата може да съдържат небалансирано количество данни.
  b. Понякога е много трудно да се проектира shard key, който да отговаря на изискванията на всяка възможна заявка.

IV. Техники (тактики) за постигане на качество на софтуера

- ***Тактиката е архитектурно решение, чрез което се контролира резултата на даден сценарий за качество. Наборът от конкретни тактики се нарича архитектурна стратегия.***

1. Тактики за изправност (Dependability)

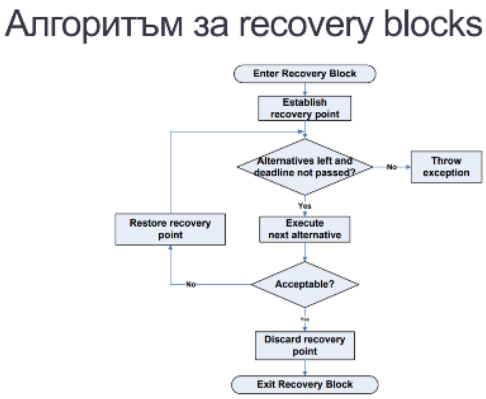
A. Тактики за изправност – откриване на откази

- **Exo (Ping/echo)** – компонент А пуска сигнал до компонент Б и очаква да получи отговор в рамките на определен интервал от време. Ако отговорът не се получи навреме, се предполага, че в компонент Б (или в комуникационния канал до там) е настъпила повреда и се задейства процедурата за отстраняване на повредата. Използва се например от група компоненти, които солидарно отговарят за една и съща задача. Също така от клиенти, които проверяват дали даден сървърен обект и комуникационния канал до него работят съгласно очакванията за производителност. Вместо един детектор да ping-ва всички процеси, може да се организира йерархия от детектори – детекторите от най-ниско ниво ping-ват процесите, с които работят заедно върху един процесор, докато детекторите от по-високо ниво ping-ват детекторите от по-ниско ниво и т.н. – така се спестява мрежови трафик.
- **Heartbeat, Keepalive** – даден компонент периодично излъчва сигнал, който друг компонент очаква. Ако сигналът не се получи, се предполага, че в компонент А е настъпила повреда и се задейства процедура за отстраняване на повредата. Сигналът може да носи и полезни данни – например, банкоматът може да изпраща журнала от последната транзакция на даден сървър. Сигналът не само действа като heartbeat, но и служи за логване на извършените транзакции.
- **Изключения (Exceptions)** – обработват се изключения, които се генерират, когато се стигне до определено състояние на отказ. Обикновено процедурата за обработка на изключения се намира в процеса, който генерира самото изключение.

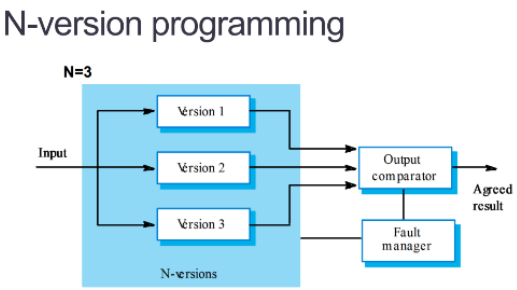
B. Тактики за изправност – отстраняване на откази

- **Активен излишък (Active redundancy, hot restart)** – важните компоненти в системата са дублирани (вкл. многократно). Дублираните компоненти се поддържат в едно и също състояние (чрез подходяща синхронизация, ако се налага това). Използва се резултатът само от единния от компонентите (т.н. активен). Обикновено се използва в клиент/сървър конфигурация, като например СУБД, където се налага бърз отговор дори при срив. Освен излишък в изчислителните звена се практикува и излишък в комуникациите, в поддържащият хардуер и т.н. Downtime-ът обикновено се свежда до няколко милисекунди, тъй като резервният компонент е готов за действие и единственото, което трябва да се направи е той да се направи активен.
- **Пасивен излишък (Passive redundancy, warm restart)** – един от компонентите (основният) реагира на събитията и информира останалите (резервните) за промяната на състоянието. При откриване на отказ, преди да се направи превключването на активния компонент, системата трябва да се увери, че новият активен компонент е в достатъчно осъвременено състояние. Обикновено се практикува периодично форсиране на превключването с цел повишаване на надеждността. Обикновено downtime-ът е от няколко секунди до няколко часа. Синхронизацията се реализира от активния компонент.
- **Резерва (Spare)** – поддръжка на резервни изчислителни мощности, които трябва да се инициализират и пуснат в действие при отказ на някой от компонентите. За целта може да е необходима постоянна памет, в която се записва състоянието на системата и която може да се използва от резервната система за възстановяване на състоянието. Обикновено се използва за хардуерни компоненти и работни станции. Downtime-ът обикновено е от няколко минути до няколко часа.

- **Основни предизвикателства:**
  - Синхронизация на състоянието на отделните дублирани модули.
  - Данните трябва да са консистентни във всеки един момент.
  - Има ли отстраняване на откази при копиране на един и същи код?
- **Разнородност** – отказите в софтуера обикновено се предизвикват от грешки при проектирането. Мултиплицирането на грешка в проектирането чрез репликация не е добра идея. Просто увеличаване на броя идентични копия на програмата (подобно на техниките в хардуера) не е решение. Трябва да се въведе разнородност в копията на програмата.
  a. **Разнородност в проектирането**
     - различни програмни езици
     - различни компилатори
     - различни алгоритми
     - ограничен/липса на контакт между отделните екипи
     - модул за избор на резултат от изпълнението на отделните копия
     - **Recovery Blocks** - разработват се няколко алтернативни модула на програмата. Извършват се тестове за одобрение, за да се определи дали получения резултат е приемлив.



     - **N-version programming (Voting)** - на различни процесори работят еквивалентни процеси, като всички те получават един и същ вход и генерират един и същ резултат, който се изпраща на т.н. voter (output comparator), който решава крайния резултат от изчислението. Ако някои от процесите произведе изход, който се различава значително от останалите, voter-ът решава да го изключи от обработката. Алгоритъмът за изключване на процес от обработката може да бъде различен и изменян, например отхвърляне чрез мнозинство, предпочитан резултат и т.н.



  b. **Разнородност по време –** предполага възникването на определени събития, които касаят работата на програмата, по различно време.
     - Методи за реализация на разнородност по време – чрез стартиране на изпълнението в различни моменти от време, чрез подаване на данни, които се използват или четат в различни моменти от времето.
     - **Извеждане от употреба (Removal from service)** – премахва се даден компонент от системата, за да се избегнат очаквани сривове. Типичен пример – периодичен reboot на сървърите, за да не се получават memory leaks и така да се стигне до срив. Извеждането от употреба може да става автоматично и ръчно, като и в двата случая това следва да е предвидено в системата на ниво архитектура.
     - **Следене на процесите (Process Monitoring)** – посредством специален процес се следят основните процеси в системата. Ако даден процес откаже, мониторинг процеса може да го премахне, преинициализира, да създаде нов екземпляр и т.н.

C. Тактики за изправност – при повторно въвеждане в употреба

- **Паралелна работа (shadow mode)** – преди да се въведе в употреба компонент, който е бил повреден, известно време се оставя той да работи в паралел в системата, за да се уверим, че се държи коректно, точно както работещите компоненти.
- **Ре-синхронизация на състоянието (State resynchronization)** – тактиките за пасивен и активен излишък изискват състоянието на компонентите, които се въвеждат повторно в употреба да бъде ре-синхронизирано със състоянието на останалите работещи компоненти. Начинът, по който ще се извърши ресинхронизацията зависи от downtime-а, който може системата да си позволи, размера на данните за ре-синхронизация и т.н.
- **Контролни точки и rollback (Checkpoint/rollback)** – контролната точка е запис на консистентно състояние, създаван периодично или в резултат на определени събития. Понякога системата се разваля по необичаен начин и изпада в не-консистентно състояние. В тези случаи, системата се възстановява (rollback) в последното консистентно състояние (съгласно последната контролна точка) и журнала на транзакциите, които са се случили след това.

2. Тактики за производителност

- Целта на тактиките за производителност е да се постигне реакция от страна на системата на зададено събитие в рамките на определени времеви изисквания. За да реагира системата е нужно време, защото ресурсите, заети в обработката го консумират, работата на системата е блокирана поради съревнование за ресурсите, не-наличието на такива, или поради изчакване на друго изчисление.

A. Тактики за производителност – намаляване на изискванията

- **Увеличаване на производителността на изчисленията** – подобряване на алгоритмите, замяна на един вид ресурси с друг (напр. кеширане) и др.
- **Намаляване на режийните (overhead)** – не-извършване на всякакви изчисления, които не са свързани конкретно с конкретното събитие (което веднага изключва употребата на посредници).
- **Промяна на периода** – при периодични събития, колкото по-рядко идват, толкова по-малки са изискванията към ресурсите.
- **Промяна на тактовата честота** – ако върху периода, през който идват събитията нямаме контрол, тогава можем да пропускаме някои от тях (естествено, с цената на загубата им).
- **Ограничаване на времето за изпълнение** – например при итеративни алгоритми.
- **Опашка с краен размер** – заявките, които не могат да се обработят веднага, се поставят в опашка. Когато се освободи ресурс, се обработва следващата заявка. Когато се напълни опашката, заявките се отказват.

B. Тактики за производителност – управление на ресурсите:

- **Паралелна обработка** – ако заявките могат да се обработват паралелно, това може да доведе до оптимизация на времето, което системата прекарва в състояние на изчакване.
- **Излишък на данни/процеси** – cache, loadbalancing, клиентите в c/s и т.н.
- **Включване на допълнителни ресурси** – повече (и по-бързи) процесори, памет, диск, мрежа и т.н.

C. Тактики за производителност – арбитраж на ресурсите

- Когато има недостиг на ресурси (т.е. спор за тях), трябва да има институция, която да решава (т.е. да извършва арбитраж) кое събитие да се обработи с предимство. Това се нарича scheduling.
- В scheduling-а се включват два основни аспекта – как се приоритизират събитията и как се предава управлението на избраното високо-приоритетно събитие. Някои от основните scheduling алгоритми са:
  - **FIFO** – всички заявки са равноправни и те се обработват подред;.

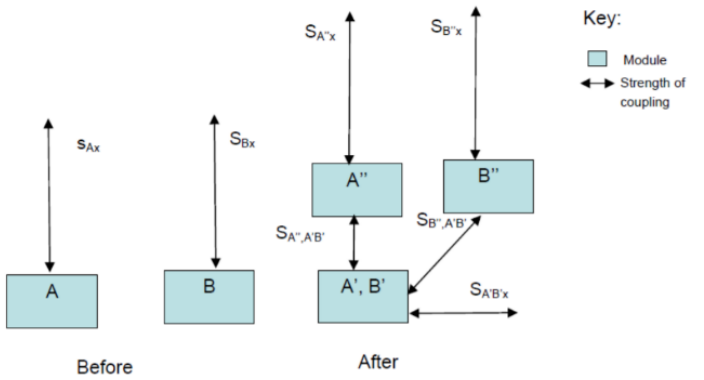
- **Фиксиран приоритет** – на различните заявки се присвоява различен фиксиран приоритет. Пристигащите заявки се обработват по реда на техния приоритет.

3. Тактики за изменяемост (modifiability)

- Тактиките за постигане на изменяемост също се разделят на няколко групи, в зависимост от техните цели.
  - **Локализиране на промените** – целта е да се намали броят на модулите, които са директно засегнати от дадена промяна.
  - **Предотвратяване на ефекта на вълната** – целта е модификациите, необходими за постигането на дадена промяна, да бъдат ограничени само до директно засегнатите модули .
  - **Отлагане на свързването** – целта е да се контролира времето за внедряване и себестойността на промяната.

A. Тактики за изменяемост – локализиране на промените

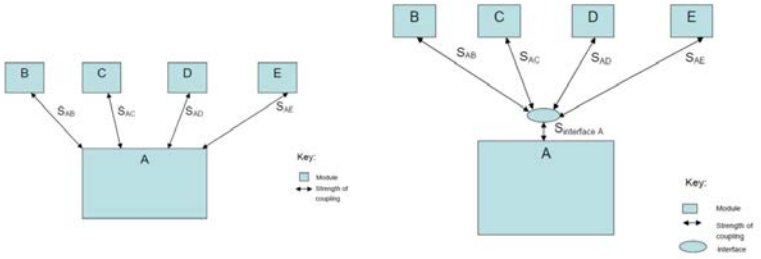
- Въпреки, че няма пряка връзка между броя на модулите, които биват засегнати от дадена промяна и себестойността на извършване на промените, е ясно, че ако промените се ограничат във възможно най-малък брой модули, цената ще намалее.
- Целта на тази група тактики е отговорностите и задачите да бъдат така разпределени между модулите, че обхватът на очакваните промени да бъде ограничен.
- **Поддръжка на семантична свързаност** – семантичната свързаност (semantic coherence) се отнася до отношенията между отговорностите в рамките на даден модул. Целта е задачите да се разпределят така, че тяхното изпълнение и реализация да не зависят прекалено много от други модули. Постигането на тази цел става като се обединят в рамките на един и същ модул функционалности, които са семантично свързани, при това разглеждани в контекста на очакваните промени. Пример за подход в тази насока е и използването на общи услуги (напр. чрез използването на стандартизирани application frameworks или middleware).



- **Очакване на промените** – прави се списък на най-вероятните промени (това е трудната част). След което, за всяка промяна се задава въпроса “Помага ли така направената декомпозиция да бъдат локализирани необходимите модификации за постигане на промяната?”. Друг въпрос, свързан с първия е “Случва ли се така, че фундаментално различни промени да засягат един и същ модул?”. За разлика от поддръжката на семантична свързаност, където се очаква промените да са семантично свързани, тук се набляга на конкретните най-вероятни промени и ефектите от тях. На практика двете тактики се използват заедно, тъй като списъкът с най-вероятни промени никога не е пълен. Доброто обмисляне и съставянето на модули на принципа на семантичната свързаност в много от случаите допълва така направения списък.
- **Ограничаване на възможните опции** – промените, могат да варират в голяма степен и следователно да засягат много модули. Ограничаването на възможните опции е вариант за намаляване на този ефект. Например, вариационна точка в дадена фамилия архитектури (продуктова линия – product line) може да бъде конкретното CPU. Ограничаването на смяната на процесори до тези от една и съща фамилия е възможна тактика за ограничаване на опциите.

B. Тактики за изменяемост – продотвратяване на ефекта на вълната

- Ефект на вълната има тогава, когато се налагат модификации в модули, които не са директно засегнати от дадена промяна. Например, ако модул А се модифицира, за да се реализира някаква промяна и се налага модификацията на модул Б само защото модул А е променен. В този смисъл Б зависи от А.
- **Скриване на информация** – декомпозиция на отговорността на даден елемент (система или конкретен модул) и възлагането ѝ на по-малки елементи, като при това част от информацията остава публична и част от нея се скрива. Публичната функционалност и данни са достъпни посредством специално дефинирани за целта интерфейси. Това е най-старата и изпитана техника за ограничаване на промените и е пряко свързана с "очакване на промените", тъй като именно списъка с очакваните промени е водещ при съставянето на декомпозицията, така че промените да бъдат сведени в рамките на отделни модули.



- **Ограничаване на комуникацията** чрез ограничаването на модулите, с които даден модул обменя информация (доколкото това е възможно). Т.е. – ограничават се модулите, които консумират данни, създадени от модул А и се ограничават модулите, които създават информация, която се използва от модул А.
- **Поддръжка на съществуващите интерфейси** – ако Б зависи от името и сигнатурата на даден интерфейс от А, то съответният синтаксис трябва да се поддържа непроменен. За целта се прилагат следните техники – добавяне на нов интерфейс – вместо да се сменя интерфейс се добавя нов, добавя се адаптер – А се променя и същевременно се добавя адаптер, чрез който се експлоатира стария синтаксис, създава се stub – ако се налага А да се премахне, на негово място се оставя stub – процедура със същия синтаксис, която обаче не прави нищо (NOOP).
- **Използване на посредник** – Ако Б зависи по някакъв начин от А (освен семантично), е възможно между А и Б да бъде поставен „посредник", които премахва тази зависимост. Посредник – wrapper, mediator, façade, adaptor, proxy и т.н.
- **Name server** - When the client (A) does not know about the location of the server (B), then the wrapper is called "name server".
- **Object request broker** - very popular for distributed objectoriented systems about a decade ago. Combines many features of different wrapper implementations. Manages the communication between objects. Has information about all object in the system and their interfaces. The so-called stub defines the interface of the object being called, while the skeleton defines the connection with the caller. An Interface Definition Language (IDL) is used to define the stub and the skeleton.

C. Тактики за изменяемост – отлагане на свързването

- Двете категории тактики, които разгледахме до тук служат за намаляване на броя на модулите, които подлежат на модификации при нуждата от промяна. Само че сценариите за изменяемост включват и елементи, свързани с възможността промени да се правят от не-програмисти, което няма нищо общо с брой модули и т.н. За да бъдат възможни тези сценарии се налага да се инвестира в допълнителна инфраструктура, която да позволява именно тази възможност – т.н. отлагане на свързването. Различни "решения" могат да бъдат "свързани" в изпълняваната система по различно време. Когато свързването става

по време на програмирането, промяната следва да бъде изкомуникирана с разработчика, след което тя да бъде реализирана, да се тества и внедри, и всичко това отнема време. От друга страна, ако свързването става по време на зареждането и/или изпълнението, това може да се направи от потребителя/администратора, без намесата на разработчика. Необходимата за целта инфраструктура (за извършване на промяната и след това нейното тестване и внедряване) трябва да е вградена в самата система.

- Съществуват много тактики за отлагане на свързването, по-важните от които са:
  - Включване/изкл./замяна на компоненти, както по време на изпълнението (plug-and-play), така и по време на зареждането (component replacement).
  - Конфигурационни файлове, в които се задават стойностите на различни параметри.
  - Дефиниране и придържане към протоколи, които позволяват промяна на компоненти по време на изпълнение.

4. Тактики за сигурност (Security)

A. Тактики за сигурност – устояване на атаки

- **Автентикация на потребителите** – проверка за това, дали потребителя е този, за който се представя (пароли, сертификати, биометрика и т.н.)
- **Оторизация на потребителите** – проверка за това дали потребителя има достъп до определени ресурси.
- **Конфиденциалност на данните** – посредством криптиране (на комуникационните канали и на постоянната памет).
- **Интегритет** – включване на различни механизми на излишък – чек-суми, хеш-алгоритми и т.н.
- **Ограничаване на експозицията**, т.е. на местата, чрез които можем да бъдем атакувани.
- **Ограничаване на достъпа** – firewall, DMZ, и др., вкл. и средства за ограничаване на физическия достъп.

B. Тактики за сигурност – възстановяване след атака

- Тактиките за възстановяване след атака са свързани с възстановяване на състоянието и с идентификация на извършителя.
- Тактиките за възстановяване на системата донякъде се препокриват с тактиките за Изправност, тъй като извършена атака може да се разгледа като друг срив в работата на системата. Трябва да се внимава обаче в детайли като пароли, списъци за достъп, потребителски профили и т.н. Тактиката за идентифициране на атакуващия е поддръжка на audit trail – копие на всяка транзакция, извършена върху данните, заедно с информация, която идентифицира извършителя по недвусмислен начин. Audit Trail-а може да се използва да се проследят действията на извършителя, с цел осигуряване на невъзможност за отричане (nonrepudiation), а също и за възстановяване от атаката.
- Важно е да се отбележи, че audit trail-овете също са обект на атака, така че при проектирането на системата трябва да се вземат мерки достъп до тях да се осигурява само при определени условия.

5. Тактики за изпитаемост (Testability)

- Целта на тактиките за изпитаемост е да подпомогнат тестването, когато част от софтуерната разработка е приключила.
- За фазата преди приключването на разработката за това обикновено се прилагат тактики като code review.
- За тестване на работеща система обикновено се използва софтуер, който чрез изпълнението на специализирани скриптове подава входни данни на системата и анализира резултата от нейната работа .
- Целта на тактиките за изпитаемост е да подпомагат този процес.

- **Запис и възпроизвеждане** – прихващане на информацията, която преминава през даден интерфейс. Може да се използва както за генериране на входни данни, така и за запис на изходно състояние с цел последващо сравнение.
- **Разделяне на интерфейса от реализацията** – позволява замяна на реализацията за тестови цели.
- **Специализиран интерфейс за тестване** – предоставяне на интерфейс за специализиран тестващ софтуер, който е различен от нормалния. Това дава възможност различни мета-данни да бъдат предоставени на тестващия софтуер, които да го управляват и т.н.
- Тестващият софтуер може да тества и тестовия интерфейс, чрез сравнение с резултатите, постигнати чрез използване на нормалния интерфейс.
- **Вградени модули за мониторинг** – самата система поддържа информация за състоянието, натовареността, производителността, сигурността и т.н. и я предоставя на определен интерфейс. Интерфейсът може да е постоянен или временен, включен посредством техника за инструментиране.

6. Тактики за използваемост (Usability)

- Използваемостта се занимава с това, колко лесно даден потребител успява да свърши дадена задача и доколко системата подпомага това му действие. Различаваме два вида тактики за използваемост, всяка от тях насочена към различна категория "потребители".
  - **Тактики за използваемост по време на изпълнението (runtime)** – насочени към крайния потребител.
  - **Тактики за използваемост, свързани с UI** – насочени към разработчика на интерфейса. Този вид тактики са силно свързани с тактиките за изменяемост. Разделяне на интерфейса от реализацията, например при MVC.

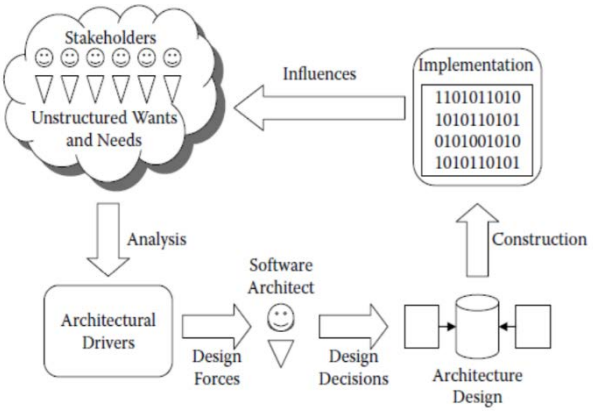
A. Тактики за използваемост – по време на изпълнението

- По време на работа на системата, използваемостта обикновено е свързана с предоставянето на достатъчно информация, обратна връзка и възможност за изпълнение на конкретни команди (cancel, undo, aggregate, multiple views и т.н.) .
- В областта на Human-Computer Interaction (HCI) обикновено се говори за "потребителска инициатива", "системна инициатива" или "смесена инициатива".
- Например, когато се отказва дадена команда, потребителя извършва "cancel" (потребителска инициатива) и системата отговаря. По време на извършването на това действие обаче, системата предоставя индикатор за прогреса (системна инициатива).
- Когато става въпрос за потребителска инициатива, архитектът проектира реакцията на системата както при всички останали функционалности, т.е. трябва да се опише поведението на системата в зависимост от получените входни събития. Напр., при cancel:
  - Системата трябва да е подготвена за cancel (т.е. да има процес, който следи за такава команда, който да не се блокира от действието на процеса, който се отказва).
  - Процесът, който се отказва трябва да се преустанови.
  - Ресурсите, които са били заети да се освободят.
  - Евентуално компонентите, които взаимодействат с отказания процес да се уведомят, за да предприемат съответните действия.
- Когато става въпрос за системна инициатива, системата трябва да разчита на някаква информация (модел) относно потребителя, задачата или моментното си състояние.
- Различните модели изискват различни входни данни за постигането на инициативата. Тактиките за постигането на използваемост по време на системната инициатива са свързани с избора на конкретен модел.
- **Моделиране на задачата** – системата знае какво прави потребителя и може да му помогне в зависимост от създадения модел (напр. AutoCorrection);

- **Моделиране на потребителя** – моделът съдържа информация за това, както потребителя знае за системата, как работи с нея, какво очаква и т.н. Това позволява на системата да бъде пригодена към поведението на потребителя.
- **Моделиране на системата** – моделът на системата включва очакваното поведение, така че тя да предоставя адекватна обратна връзка. (напр. предвиждане на времето за обработка, за да бъде показан progress bar).

V. Процес за проектиране на софтуерна архитектура

1. **Как започва проектирането на архитектурата?**
   - Проектирането започва при наличието на изисквания. От друга страна, не се изисква наличието на много изисквания, за да започне проектирането.
   - Архитектурата се оформя от няколко (десетина) основополагащи функционални, качествени и бизнес изисквания – т.н. архитектурни драйвери.



2. **Как се избират драйверите?**
   - Идентифицират се тези цели на системата, които са с най-висок приоритет.
   - Тези цели се превръщат в сценарии за употреба или за постигане на качествено свойство.
   - От тях се пробират не повече от 10 – тези, които имат най-голямо влияние върху архитектурата.
   - След като драйверите бъдат избрани, започва проектирането. Започва и задаването на въпроси, което може да доведе до промяна в изискванията, което пък може да доведе до промяна в драйверите и т.н.
3. **ADD – Attribute Driven Design**
   - ADD е подход за проектиране, в който основна роля играят качествените свойства (атрибути).
   - Това е рекурсивен процес на дефиниране на архитектурата, като на всяка стъпка се използват тактики и архитектурни модели за постигане на желаните качествени свойства.
   - В следствие на приложението на ADD се получават първите няколко нива на модулната декомпозиция и на други структури, в зависимост от случая.
   - Това е първото проявление на архитектурата и като такова е на достатъчно високо ниво, без излишни детайли.
   - **Стъпки на ADD**
     - Избира се модул, който ще се декомпозира. Изискванията към модула трябва да са известни.
     - Модулът се детайлизира – избират се архитектурните драйвери (най-важните изисквания за този етап).
     - Избира се архитектурен модел, който удовлетворява драйверите. Модела се избира или създава въз основа на тактиките за постигане на избраните свойства. Идентифицират се типовете под-модули, необходими за постигането на тактиките.
     - Създават се под-модули от идентифицираните типове и им се приписва функционалност съгласно сценариите за употреба. Създават се всички необходими структури.
     - Дефинират се интерфейсите към и от под-модулите.
     - Проверяват се и се детайлизират изискванията, като същевременно се поставят ограничения върху под-модулите. На тази стъпка се проверява дали всичко съществено е налично и се подготвят под-модулите за по-нататъшна декомпозиция.
     - Това се повтарят за всички модули, които се нуждаят от понататъшна декомпозиция
   - **Входни данни на ADD – набор от изисквания**
     - Функционални изисквания, под формата на сценарии за употреба (use-cases).
     - Функционални ограничения (constraints).
     - Качествени свойства, под формата на специфични сценарии за проявление.
4. **Стъпки на ADD**

A. **Избира се модул за декомпозиция** – първоначално това е цялата система, която се разлага на подсистеми. Подсистемите се разлагат на модули. Модулите на под-модули и т.н.

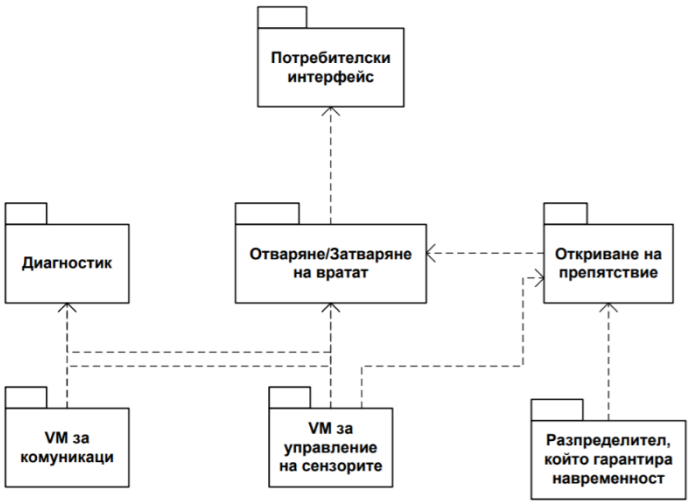
B. **Избират се архитектурните драйвери -** драйверите се избират измежду изискванията с най-висок приоритет. Изборът на драйвери оформя декомпозицията на модула на под-модули. Въпреки, че има и други изисквания към модула, те се считат за по-маловажни и ще бъдат реализирани в контекста на най-важните

C. **Избира се архитектурния модел -** за постигането на всяко изискване има тактика. Конфигурацията от избрани тактики определя архитектурния модел. Тактиките обикновено имат странични ефекти (+, -) върху другите свойства, затова се налага да се разгледа цялостната картина, т.е. архитектурния модел.

D. **Създаване на подмодулите** - за всеки от идентифицираните типове се създават по един или няколко под-модула. (заедно с E и F)

E. **Декомпозиция**

F. **Приписване на функционалност** - всеки сценарий на употреба, който е приложим към модула, който декомпозираме се прилага, като при това се описва кои под-модули за коя част от функционалността отговарят. В резултат се получават сценарии, приложими за подмодулите. Възможно е в следствие на това да се премахнат или създадат нови под-модули. По време на този процес изпъкват нуждите от обмяна на информация между под-модулите, които също трябва да бъдат документирани.



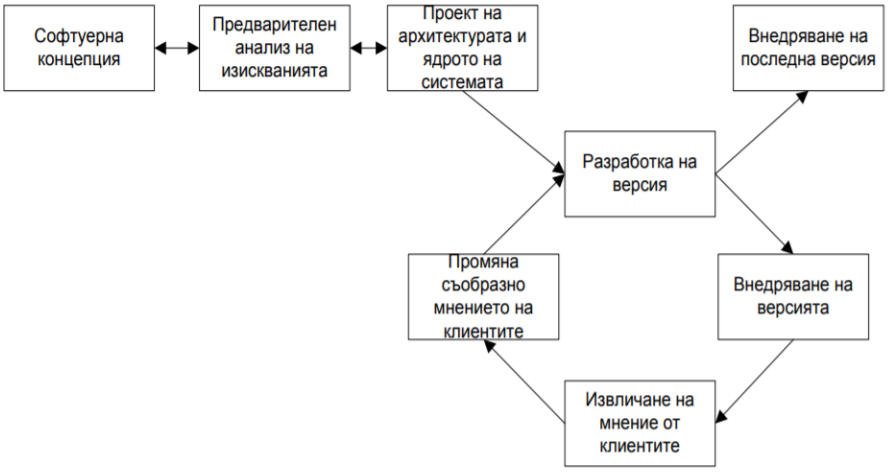
G. **Създаване на други структури**
   a. **Структура на декомпозицията** - структура на декомпозицията – структурата, която директно се получава от метода. Получават се модули, на които се приписват функционалности и отговорности. Освен това структурата показва основните потоци на обмен на информацията.
   b. **Структура на внедряването -** наличието на няколко хардуерни устройства диктува необходимостта от структура на внедряването. Логическите връзки се декомпозират и техните части се разполагат върху различните процесори, а между тях се оформят комуникационни канали. Помага да се изясни дали са няколко инстанции на даден модул, дали е необходим специализиран хардуер. Създаването на структурата се базира на архитектурните драйвери и избраните тактики за постигането на изискванията.

H. **Дефинират се интерфейсите на под-модулите -** под интерфейс се разбира съвкупността от услуги и свойства, които модула предлага/изисква. Документират се всички свойства и услуги от всички структури.

I. **Проверява се декомпозицията -** проверява се как функционалните изисквания се изпълняват от подмодулите, дали се изпълняват ограниченията, за да се изпълнят ограниченията може да се сложи отделен под-модул или да се вмени отговорността за това на един или повече под-модули. Как се покриват сценариите за качествата? Напълно, с ограничения, без влияние или противоречат на декомпозицията?

J. **Рекурсивен ADD -** След стъпка 2 имаме списък от под-модули, за които имаме списък с отговорности, сценарии за употреба, интерфейси, сценарии за качества, списък с ограничения. Това е достатъчно, за да послужи за вход на рекурсивно извикване на ADD за някои от модулите, които се нуждаят от детайлизация.

5. **Цикличен процес на създаване на архитектурата**



6. **Формиране на екипи**
   - След като се идентифицират първите няколко нива на декомпозицията, могат да се формират екипи, които да работят по съответните модули.
   - Структурата на екипите обикновено отговаря на структурата на декомпозицията.
7. **Създаване на скелетна система**
   - Когато архитектурата е готова донякъде и има сформирани екипи може да се започне работа по системата. По време на разработката обикновено се използват стъбове, за да могат модулите да се разработват и тестват поотделно.
   - Но с кои модули да започне създаването на скелетна система?
8. **Последователност на реализацията**
   - Първо се реализират компонентите, свързани с изпълнението на и взаимодействието между архитектурните компоненти (middleware).
   - След това се реализират някои прости функционалности.
   - Последователността по нататък може да се диктува от:
     - Намаляване на риска – първо най-проблематичните.
     - В зависимост от наличния персонал и квалификацията му.
     - Бързото създаване на нещо, което се продава.
   - След това на база структурата на употребата се определят следващите функционалности.

VI. Документиране на архитектурата

1. Зависимост на съдържанието от предназначението

- Документацията на архитектурата зависи то това, кой ще я ползва.
- Трябва да е достатъчно абстрактна, така че да бъде разбрана от нови служители; от друга страна трябва да е достатъчно детайлна, че да послужи за основа на проектирането.
- За някои документацията е предписание; за други – описание. Всичко това означава, че един прост документ няма да стигне за да задоволи изискванията на всички заинтересовани (stakeholders).
- Най-често се създава набор от документи с обогатено съдържание, което указва къде каква информация се съдържа.
- Основен принцип – по време на документирането архитектът трябва да може да се постави на мястото на четящия.

2. Перспективи (views)

- Съгласно дефиницията на СА тя е съвкупност от структури от софтуерни елементи, техните видими свойства и взаимовръзките между тях.
- Перспектива (изглед, view) – формално понятие – описание на дадена структура на архитектурата.
- Нито една от тези структури (перспективи) поотделно не е архитектура. Всички те заедно изграждат и изразяват архитектурата на сградата.

3. Основен принцип на документирането на СА

- Концепцията за структури и перспективи позволява да се дефинира основния принцип на документирането на СА: “Документирането на СА е въпрос на документиране на всички съставляващи я структури поотделно и последващо добавяне на документация, която се отнася за няколко структури”.

4. Три [лесни?] стъпки за избор на подходящите перспективи

- Създава се таблица от вида “кой от какво се интересува”, подобна на тази от следващата страница, но различна за всяка конкретна ситуация.
- Комбинират се перспективите – тъй като неминуемо ще се получи прекалено голям набор перспективи, се намира подходящ начин те да се комбинират (напр. декомпозиция+слоеве, декомпозиция+разработка, процеси+внедряване и т.н.).
- Задава се приоритет на създаването на всяка перспектива. Това зависи от множеството заинтересовани лица и други съображения.

5. Съдържание на документацията

- Няма изграден индустриален стандарт за съдържанието на документацията на дадена структура. От съществено значение е да има някакъв стандарт, т.е. да има последователност при създаването на документацията. За описание на структурите, тук е описано 7- елементно съдържание, което се е доказало в практиката, но принципно може да се избере друга методика, стига да се спазва навсякъде:
  - **A. Първично представяне**
  - **B. Описание на елементите и връзките**
  - **C. Описание на обкръжението**
  - **D. Описание на възможните вариации**
  - **E. Архитектурна обосновка**
  - **F. Терминологичен речник**
  - **G. Допълнителна информация**

Първично представяне

- Показва елементите (и връзките между тях) от които се изгражда документираната структура.
- Трябва да съдържа информация от първостепенна важност за структурата. Второстепенни детайли (напр. обработка на грешки) може да се пропускат.
- Обикновено първичното представяне е графично, с легенда или препратка към описанието на използваната символика.
- Случва се да се използва и чисто текстово или таблично описание.
- Най-често, за първичното представяне се използва UML.

Описание на елементите и връзките

- Съдържа детайлно описание на елементите и връзките, показани в първичното представяне, а също така и други второстепенни детайли.
- Първичното представяне представлява в голяма степен схематично описание. Тук следва да се опише смисъла и ролята на всеки един елемент и връзка.
- Други важни аспекти на описанието са поведението и интерфейса на елементите.
- Интерфейс – това е мястото, където два независими софтуерни елемента се срещат и си взаимодействат.
- Описанието на интерфейсите разкрива информация за начините за употребата на ресурсите, предоставени от дадения софтуерен елемент.
- Недостатъчното описание води до проблеми при създаването на взаимодействието.
- Прекалено детайлно описани води до трудности при промяната.

Описание на обкръжението

- Съдържа информация за това как елементите от документираната структура си взаимодействат с обкръжението – други системи, интерфейси, протоколи и т.н.

Описание на възможните вариации

- Изброяване на всички позволени варианти или детайлно описание на условията, на които трябва да отговаря избрания вариант.
- Кога трябва да се направи окончателния избор (по време на проектирането, разработката, внедряването или работата на системата).

Архитектурна обсновка

- Обяснява на заинтересованите лица защо структурата е проектирана по начина, по който е описана. Целта е да се дадат убедителни аргументи, че архитектурния проект е издържан.
- Обосновка на взетите относно настоящата структура решения, наличните алтернативи и защо те са отхвърлени.
- Аналитични резултати, които потвърждават предприетите решения.
- Предположения, направени по време на проектирането.

Терминологичен речник

- Кратко описание на използваната стандартна и нововъведена терминология.

Допълнителна информация

- Често тук се включва и административна информация – автор, история на промените и т.н. Във всеки случай, раздела за допълнителна информация трябва да започва с описание на съдържанието си.

6. Допълнителна документация

- Освен документиране на самите структури се налага създаването на документация, приложима към няколко структури или към целия пакет. Допълнителната документация има три аспекта:
  - КАК е организирана документацията
  - КАКВА е цялостната архитектура
  - ЗАЩО архитектурата е проектирана така

A. КАК е организирана документацията?

- Този раздел съдържа два основни елемента - каталог на структурите и шаблон за описание на структурите.
- **Каталогът на структурите** помага на четящия по-лесно да открие интересуващата го информация. За всяка структура се описват името и типа на структурата, описание на типовете елементи, връзки и характеристики, за какво служи структурата, административна информация – автор, версия, къде се помещава самият документ и т.н.
- Каталогът на структурите описва самата документация, а не архитектурата!
- Шаблонът за описание на структурите е документ със заглавията на разделите и кратко описание на това, какво се съдържа в тях.

B. КАКВА е цялостната архитектура

- Вторият раздел от допълнителната информация съдържа описание на архитектурата като цяло, под формата на: Кратко описание на системата; Карта на съответствието между структурите; Списък на софтуерните елементи; Разширен терминологичен речник.
- Краткото описание на системата е къс разказ за това каква е необходимостта от системата, нейната функционалност, потребители, конюнктурни обстоятелства и т.н.
- Тъй като различните описания на структури се отнасят за една и съща система, с увереност може да се твърди, че между тях има много общи неща.
- Картата на съответствието между структурите помага за ориентацията.
- Съответствието може да е от всякакъв характер – 1-1, 1-N, N-1, N-N
- Потенциалните съответствия са много, описват се само тези, които биха били полезни за осмисляне на архитектурата.
- Списъка на софтуерните елементи представлява индекс на всички описани в документацията СЕ и препратки към срещанията им.
- Разширения терминологичен речник съдържа кратко описание на използваната стандартна и нововъведена терминология.

VII. Анализ на архитектурата

ATAM (Architecture Tradeoff Analysis Method)

1. **Въведение в ATAM**
   - ATAM (Architecture Tradeoff Analysis Method) – метод за анализ, базиран на компромисите**.**
   - Разкрива до каква степен архитектурата удовлетворява индивидуалните качествени изисквания.
   - И по-важното – как архитектурните решения си взаимодействат и съответно какви компромиси се правят за това.
   - Анализът е сложен процес – обикновено големите системи имат сложна архитектура, многобройни бизнес цели и голям брой заинтересовани лица, а времето за анализ е твърде ограничено.
2. **Участници в ATAM**
   - **Оценяващ екип** – от 3 до 5 души, външни за проекта, чиято архитектура се оценява. Всеки от оценяващия екип изпълнява една или няколко роли.
   - **Ръководен екип** – група, от която зависят решенията по проекта. Обикновено включва ръководителя на проекта, представител на клиента, архитекта и лицето, наредило анализа.
   - **Останалите заинтересовани** – разработчици, тестери, поддръжка, потребители, разработчици на трети системи, интегратори и т.н.
3. **Роли в оценяващия екип**
   - **Ръководител на екипа** – формира екипа, ръководи цялостния процес, установява необходимите контакти, координира действията с клиента и т.н.
   - **Ръководител на анализа** – физически ръководи анализа, подпомага извличането на сценарии, администрира процеса на избор и приоритизация на сценариите и т.н.
   - **Автор на сценарии** – формулира сценариите, използвайки съгласувана терминология! Изключително важна роля, може да спре дискусията докато не се постигне съгласие от всички по окончателната формулировка.
   - **Стенограф** – записва дискусиите колкото е възможно подетайлно, вкл. документира окончателните формулировки.
   - **Хронометрист** – подпомага ръководителя на анализа с данни за това, колко отнема дискусията на всеки сценарии и може да прекратява дискусиите при прекалено задълбаване.
   - **Наблюдател** – тихо наблюдава как върви процеса и предлага на ръководителя възможни подобрения, както за настоящия, така и за бъдещи анализи; в задълженията му влиза и да подготви отчет за проведения анализ и евентуално да предложи подобрения на процеса за анализ на архтектурата.
   - **Специалист по процеса** – знае ATAM наизуст и подпомага ръководителя да определя следващите стъпки.
   - **Разпитващ** – задава необходимите въпроси на всички участници.
4. **Резултати от ATAM**
   - Сбито представяне на архитектурата - обикновено архитектурата се отъждествява с обемисти списъци от обекти, интерфейси и сигнатури. ATAM води до създаването на сбито представяне, което да бъде демонстрирано в рамките на 1 час.
   - Изясняване на бизнес целите – често се случва разработчици да не са и чували за бизнес целите преди да се направи анализа.
   - Непосредствено се получават сценарии за по-важните изисквания към качеството (следствие от бизнес целите).
   - Разглежда се съответствието между архитектурни решения и качествени изисквания.

- Идентифицират се решенията, към които качествените изисквания са чувствителни и които представляват компромиси.
- Идентифицират се множествата от рискове и безопасни решения (nonrisks).
- **Риск** – решение, което може да доведе до нежелани последствия.
- Идентифицират се рискови тематики – проблемни области от архитектурата, където рисковете са систематични и чието присъствие неминуемо носи заплаха за проекта**.**
- И други вторични резултати, вкл. и нематериални такива (напр. допълнителна документация, подобряване на комуникацията в екипа, по-добро разбиране на проекта от всички участници).

5. Четири фази на ATAM

-
- Фаза 0 е подготвителна – ръководствата на екипа за оценка и на ръководния екип се събират и уточняват всички подробности по анализа, напр. каква експертиза е необходима, логистиката, заинтересовани лица, план за изпълнение, налична документация и изисквания към представянията на ръководителя на проекта и на архитекта.
- Фази 1 и 2 са същинските фази на оценка – екипът по оценка вече е изучил наличната документация, има идея за системата, общата насоченост на архитектурата и т.н. Организират се срещи с ръководния екип (на първа фаза) и с останалите заинтересовани лица (на втора фаза) и се осъществява същинския анализ.
- Фаза 3 е заключителна – подготвя се и се доставя окончателния доклад за състоянието на архитектурата. Друга важна дейност е самооценка на извършената работа и изводи за бъдещи проекти. Наблюдателя на процеса съставя съответния доклад за работата на екипа. След време (5-6 месеца) ръководителя на екипа се допитва до клиента с цел установяване на положителен (или отрицателен) резултат.

6. Стъпки в ATAM

A. Стъпка #1 – представяне на метода. Ръководителя на екипа по оценка представя метода на всички участници във фазата, задават се въпроси, дават се отговори и се установява контекста и набора от дейности по време на процеса.

B. Стъпка #2 – дискусия относно бизнес целите на системата. Ръководителя на проекта или представител на клиента представя системата от гледна точка на бизнеса:

- Най-важните функции на системата.
- Всякакви приложими технически, управленски, икономически и политически ограничения.
- Бизнес целите и бизнес обкръжението и как те се отнасят към проекта.
- Главните заинтересовани лица.
- Архитектурните драйвери, т.е. най-важните качествени характеристики които оформят архитектурата.

C. Стъпка #3 – представяне на архитектурата. Архитекта, в рамките на 1 час (20-на слайда) представя архитектурата:

- Основни изисквания, измерваеми величини свързани с тях, стандартни подходи за постигането им (2-3 слайда).
- Важна архитектурна информация (4-8 слайда).
- Контекстна диаграма – системата и нейното обкръжение (хора и системи, с които си взаимодейства).

- Декомпозиция на модулите – разпределението на функционалността по модули, обекти, процедури, функции и връзки между тях.
- Диаграма на процесите – процеси, нишки, синхронизация между тях, потоци от данни и събития, които ги свързват.
- Схема на внедряването – процесори, устройства, сървъри, датчици, комуникационни устройства, разположението и връзките на софтуерните елементи с тях.
- Архитектурни похвати, схеми и тактики, както и качествените характеристики, които те преследват (3-9 слайда).
- COTS (component-off-the-shelf) и тяхната интеграция (0-2 слайда).
- Не повече от 3 от най-важните сценарии за употреба, вкл. ресурсите (1-3 слайда).
- Не повече от 3 от най-важните сценарии за промяна, вкл. въздействието (1-3 слайда).
- Архитектурни рискове относно изпълнението на изискванията (1-3 слайда).
- Кратък терминологичен речник (0-1 слайда).

D. Стъпка #4 – идентифициране на архитектурния подход. Архитектът изброява използваните архитектурни стилове и схеми и обосновава тяхната употреба.

E. Стъпка #5 – създаване на дърво на качествените атрибути.

- Коренът на дървото е описание на качествените характеристики на системата– т.нар. Utility (практичност).
- Практичността се разделя на няколко основни качествени характеристики, като конкретните наименования не са толкова важни (напр. “бързодействие”).
- Всяка от качествените характеристики се разделя на конкретни изисквания (напр. “системата трябва да поддържа поне 20 fps”).
- Сценариите за качество са листата на дърветата. Сценариите тук са опростени – имат само по 3 елемента (стимул – събитие, кой го генерира и какво бива стимулирано; състояние (какво се прави в момента); и резултат (измерваем, какво се случва в резултат на стимула).
- Обикновено дървото съдържа няколко десетки сценария, които трябва да се приоритизират, както по важност, така и по трудност.

F. Стъпка #6 – анализ на архитектурния подход.

- Сценариите с най-висок приоритет от предишната стъпка се анализират един по един. Архитектът обяснява как архитектурата поддържа сценария. Екипът документира рискове, нерискове, чувствителни сценарии и компромиси. Зад всичко това стои дискусия с архитекта, който трябва да защити позицията си. В следствие на дискусията анализът може да се задълбочи, като целта е екипа да бъде убеден в извлечените относно сценария детайли. А може и да не се задълбочи, ако архитектът все още няма виждане по някои от въпросите. На края на стъпка 6 екипът по оценка би следвало да има информация относно:
- Най-важните аспекти на цялостната архитектура.
- Обосновката на основните архитектурни решения.
- Списък с рисковете, не-рисковете и компромисите.

G. Пауза - Фаза 1 приключва със стъпка #6

- Следва пауза от 2-3 седмици. През това време екипът по оценка обобщава наученото, неформално комуникирайки с архитекта (примерно по телефон, e-mail и т.н.), анализират се допълнителни второстепенни сценарии, изчистват се детайли, въпроси и т.н. Ръководителя на екипа по анализ запознава останалите заинтересовани лица със резултатите до тук и заедно с тях започва Фаза 2.

H. Стъпка #7 – брейнсторм и приоритизиране на сценариите. На тази стъпка останалите заинтересовани дискутират кои според тях са най-важните сценарии, като най-добре това се прави на т.н. брейнсторм сесия – събират се всички вкупом и си обменят мнения. На всеки се дава право да гласува за 30% от всички сценарии и тези с най-много гласове печелят. Ако избраните сценарии съвпадат с тези от т. 5, значи има добро разбирателство между архитекта и заинтересованите лица. Ако има големи разлики – това само по себе си е риск за проекта.

I. **Стъпка #8** – **повтаря се стъпка #6, но за сценариите от стъпка #7**. Където е възможно се използва досега натрупаната информация.

J. **Стъпка #9** – **прави се обобщение на всичко научено дотук**
   - Документираните архитектурни подходи.
   - Сценариите и техния приоритет.
   - Дървото със сценариите от т. 5.
   - Рисковете и не-рисковете.
   - Чувствителните точки и компромисите.
   - Рисковете се обобщават в теми.
   - За всяка тема екипът идентифицира бизнес целите, които биха могли да бъдат засегнати. Така на пръв поглед чисто технически второстепенен проблем може да се окаже критичен за постигането на бизнес целите и съответния мениджър да бъде известен за това

9. ATAM - обобщение - Какво не е ATAM:

- НЕ Е оценка на изискванията – от него не става ясно дали ВСИЧКИ изисквания ЩЕ бъдат покрити. Това което става ясно е дали настоящата архитектура ще поддържа най-важните от тях.
- НЕ Е оценка на кода – обикновено се случва в ранните стадии на жизнения цикъл, преди още да се говори за код.
- НЕ включва същинско тестване на системата – по същите причини.
- НЕ Е прецизен инструмент – дава възможност за идентифициране на рискове, като се анализират компромисните и чувствителните точки. Всичко това зависи от знанията и уменията на оценителите, както и от представянето на самия архитект.
- НЕ дава стойността ($) на рисковете. Това е предмет на друг вид анализ – CBAM (Cost Benefit Analysis Method).

CBAM

1. Мястото на CBAM

- В ATAM става въпрос за анализ на компромиси, но всъщност най-важните компромиси се правят по икономически съображения.
- ATAM не отговаря на въпроса как дадената организация може да увеличи печалбите и да намали риска от даден софтуерен проект.
- В миналото се е обръщало повече внимание на разходите (при това на преките такива), докато напоследък има тенденция да се отделя голямо внимание на ползите от дадена система.
- Имайки предвид, че ресурсите необходими за построяването на СА са изчерпаеми, добре би било да има рационален процес, съгласно който да изберем един или друг архитектурен подход. Всички решения са свързани с определени разходи, допринасят с нещо (т.е. има полза от тях) и съдържат определен риск. Нужен е икономически модел, който да взема предвид тези фактори.
- На практика, CBAM се базира на ATAM и дава оценка на технико-икономическите аспекти на архитектурните решения.
- Целта на архитекта е да увеличи разликата между ползите, извлечени от софтуерната система и разходите направени за производството й.
- CBAM започва там, където ATAM свършва и всъщност разчита на резултите от него.
- Всяка софтуерна архитектура както техническа така и икономическа страна.
- Икономическата страна на архитектурата е свързана както с цената на системата, така и с икономическия ефект от различните качествени произведен в следствие на различните качествени характеристики.
- Връзките между бизнес цели, архитектурни решения и реализирани качествени характеристики се дава от ATAM.
- CBAM се базира на разкритията относно тези връзки за да изгради представа за стойностите и ползите на всяко от решенията и за тяхната разлика – ROI (Return of Investment).

2. **Основни идеи**
   - Методът се базира на оценка на полезността на взетите решения и оценка на тяхната цена.
   - За целта за всяко от качествата се разглеждат множество сценарии, като на всеки възможен техен резултат се присъжда степен на полезност (от страна на заинтересованите лица).
   - Разглеждат се възможните архитектурни стратегии, които водят до различни вариации относно резултатите от сценариите.
   - Всяка стратегия си има цена и влияе върху едно или няколко качества, променяйки резултата от изпълненията им.
   - Отчетената полезност на всеки един от резултатите се сумира и така се формира окончателната ROI за съответния вариант на архитектурна стратегия.
3. **Вариации на сценариите**
   - В CBAM (подобно на ATAM), качествените характеристики на системата се описват чрез сценарии (с 3 елемента – стимул, обкръжение, резултат).
   - За разлика от ATAM тук се разглеждат няколко сценария за една и съща характеристика, като на всеки резултат се присвоява степен на полезност.
4. **Криви на полезността (utility)**
   - Извличането на точната форма на кривата е дълъг и продължителен процес, но пък и това не е толкова необходимо.
   - Практиката показва, че 4 точки за сценарии са горедолу достатъчни.
   - Най-добрия случай – (best-case scenario) – резултат, чието повишаване не води до повишаване на полезността. Полезност = 100.
   - Най-лошия случай – (worst-case scenario) – разрешения санитарен минимум за системата – всеки по-лош резултат не е допустим за клиента. Полезност = 0.
   - Настоящ резултат – сегашното състояние на нещата.
   - Желан резултат – процент (може и 100%) от най-добрия.
   - Може да има и 5та, специфична за системата точка
5. **Приоритизиране на сценариите за качество**
   - Различните сценарии имат различна значимост за различните заинтересовани лица.
   - За постигане на правилни резултати е редно да се постигне консенсус относно относителната тежест на сценариите.
   - Това става на две стъпки:
     - Заинтересованите гласуват за подредбата на сценариите по важност (базирани на очаквания резултат).
     - На така подредените сценарии се дава тежест, най-важния според общото мнение е оценен с 1, а останалите – с усреднена дробна стойност.
6. **Архитектурни стратегии**
   - Целта на архитекта е да се определи архитектурните стратегии, които да доведат нивото на резултата за всяка от качествените характеристики от “текущото” на “желаното”.
   - Част от CBAM се занимава и с това. За всяка стратегия можем да определим:
     - Най-вероятен резултат
     - Страничните ефекти на архитектурната стратегия върху другите качества
     - Оценка на стойността на съответната стратегия
7. **Стъпки в CBAM**

A. **Стъпка #1 – сортиране на сценариите** – събират се всички сценарии от ATAM, дава се възможност на заинтересованите лица да допринесат с още сценарии. Всички събрани сценарии се приоритизират в зависимост от това как помагат за изпълняване на поставените бизнес цели. За целите на по-нататъшния анализ се може да се изберат част от най-високо оценените сценарии.
B. **Стъпка #2 – рафиниране на сценариите** – фокусира се вниманието върху резултата от сценариите, определят се най-добрия, най-лошия, текущия и желания резултати.
C. **Стъпка #3 – приоритизиране на сценариите** – на всеки заинтересован се дават 100 гласа, които той да разпредели между сценариите, като се базира на желания резултат. Резултатите се обобщават и се избират част от найвисоко оценените от всички сценарии. На най-високо оцененият сценарии се дава тежест 1, другите се оценяват от заинтересованите с дробен коефициент
D. **Стъпка #4 – присвояване на полезност** – на всеки от сценариите, останали след стъпка 3 се присвоява полезност за всяко от четирите нива на резултата.
E. **Стъпка #5 – разработка на архитектурните стратегии** – за всеки от сценариите се изработва архитектурна стратегия и се преценява очаквания резултат върху всеки от засегнатите сценарии.
F. **Стъпка #6 – оценка на полезността на очаквания от стратегиите резултат** – за всеки засегнат сценарии чрез използване на интерполация се оценява полезността на очаквания резултат.
G. **Стъпка #7 – оценка на общата полезност на стратегията** – извършва се сумиране на полезността за всеки засегнат от стратегията сценарии.
H. **Стъпка #8 – избор на стратегиите на база ROI** – определя се цената на стратегията, съответно ROI. Стратегиите се подреждат съгласно коефициента на възвръщаемост и в зависимост от цената и др. ограничения (напр. във времето). Избират се онези най-полезни стратегии, които се побират в бюджетните и времевите ограничения.
I. **Стъпка #9 – потвърждаване на резултатите** – по интуиция. За всяка от избраните стратегии се проверява дали съвпада с бизнес целите. Ако не съвпада – най-вероятно има нещо пропуснато по време на анализа. Това става изцяло на база предишен опит на анализатора.

8. Обобщение – CBAM

- Методът е итеративен и често се базира на неформална комуникация с заинтересованите лица, които на базата на консенсус вземат решения за полезността и цената на дадените стратегии.
- Естествено, това отнема много време и е необходимо гъвкавото му приложение в зависимост от обстановката.
- Въпреки това, при големи проекти икономическият ефект от приложението на структуриран метод за анализ е винаги положителен в сравнение с ad hoc анализа и вземането на решения по метода “гледане в точка”.