18. Софтуерна архитектура. Проектиране и документиране на софтуерни архитектури. (САРС)

Анотация: Изложението по въпроса трябва да включва следните по-съществени елементи:
1. Дефиниция на софтуерна архитектура. Структури и изгледи (structures and views) на архитектурата.
2. Изисквания към качеството (нефункционални изисквания) на системата.
3. Проектиране на софтуерна архитектура. Процес за проектиране. Избор на подходящи структури. Последователност на създаване на архитектурата.
4. Тактики (архитектурни решения) за постигане на желаните качествени показатели.
5. Архитектурни стилове.
6. Документиране на софтуерна архитектура. Предназначение на документацията. Основен принцип на документиране. Съдържание на документацията. Структура на документацията.

1. Софтуерна архитектура

1.1. Дефиниция.
Архитектурата на дадена софтуерна система е съвкупност от структури, показващи различните софтуерни елементи на системата, външно видимите им свойства и връзките между тях. Софтуерната архитектура е абстракция, която разглежда елементите като черни кутии (т.е. не се интересува от алгоритми и прочие), като се фокусира над взаимодействията между тях.

Съгласно дефиницията става ясно, че системите имат повече от една структура. Нито една от тях самостоятелно не представлява архитектурата на системата.

1.2. Структури и изгледи на архитектурата

Структура – съвкупност от софтуерни елементи, техните външно видими свойства и връзките между тях.
Изглед – конкретно документирано представяне на дадена структура. (Двете понятия в голяма степен са взаимозаменяеми.)

Структурите се делят на няколко типа:

1.2.1. 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.
Ще разгледаме няколко типа модулни структури.

- **Декомпозиция на модулите.** При тази структура връзките между модулите са от вида „**X е подмодул на У**“ и модулите биват рекурсивно разбивани на по-прости единици, докато станат лесни за разбиране. Декомпозицията на модулуте обуславя в голяма степен възможността за лесна промяна, като обособява логически свързани функционалности на едно място.
- **Употреба на модулите**. При този вид структура връзките между модулите са от вида „**X използва У**“. 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 последователна разработка.
- **Структура на слоевете**. Частен случай на „употреба на модулите“. Модулите са разделени на слоеве, като модулите от слой N може да ползват услугите само на модули от слой N-1. Слоевете скриват детайлите относно работата си от следващия слой, позволявайки лесна смяна на даден слой.
- **Йерархия на класовете**. В терминологията на ООП, модуите се наричат „класове“, а в настоящата структура, връзките между класовете са от вида „**класът X наследява класа У**“ и „**обектът X е инстанция на класа У**“. Тази структура обосновава наследяването – защо подобни функционалности са обособени в супер-класове или пък защо са дефинирани под-класове за обслужване на параметризирани различия.

1.2.2. Структури на 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.

1.2.3. 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.
Ще разгледаме няколко типа структури на разпределението.

- **Структура на внедряването**. Показва как софтуера се разполага върху хардуера и комуникационното оборудване. Елементите са процеси, хардуерни устройства и комуникационни канали. Връзките са например „внедрен върху“ или „мигрира върху“ – показвайки върху кое устройство е разположен даден софтуерен елемент. Тази структура може да се използва за поглед върху производителността, сигурността и други метрики на дадена система.
- **Файлова структура**. Показва кой модул къде се помещава във файловата структура по време на различните фази на реализация. Структурата е критична за управлението на дейностите по разработка и за създаването и поддържането на обкръжение за build-ове.
- **Разпределение на работата**. Показва кой модул от кой екип се реализира. Елементите са модули и екипи. Екипите често не са списък от хора, а по-скоро виртуална група хора с подходящ опит, знания и умения.

2. Изисквания към качеството (нефункционални изисквания) на системата

Качествените изисквания определят как софтуерната система да работи. Качеството е субективно понятие – различните заинтересовани лица (ЗЛ) имат различни възприятия за него. **Бизнес целите определят качествата**, които трябва да бъдат вградени в архитектурата на системата.

Качествата се разделят на следните три основни групи:

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

Качествата поставят изискванията отвъд функционалните изисквания (описание на основните възможности на системата и услугите които тя предоставя). Често функционалността е единственото нещо, което се взима под внимание. Като следствие много системи се преправят защото имат операционни проблеми. Софтуерната архитектура се изгражда на база качествените изисквания, като от тях се създават съответните структури, на които се вменява функционалност. качествени характеристики трябва да се имат предвид както по време на проектирането, така и по време на разработката и внедряването.

Понеже ЗЛ, в това число и архитектът, имат различна представа за качествата, се налага начин на формализация. Това се постига чрез сценарии за качество.

2.1. Сценарии за качество

Сценарият за качество е специфично изискване към поведението на системата в дадена ситуация, в светлината на дадено качество. Те 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 **ВИОКРК**:

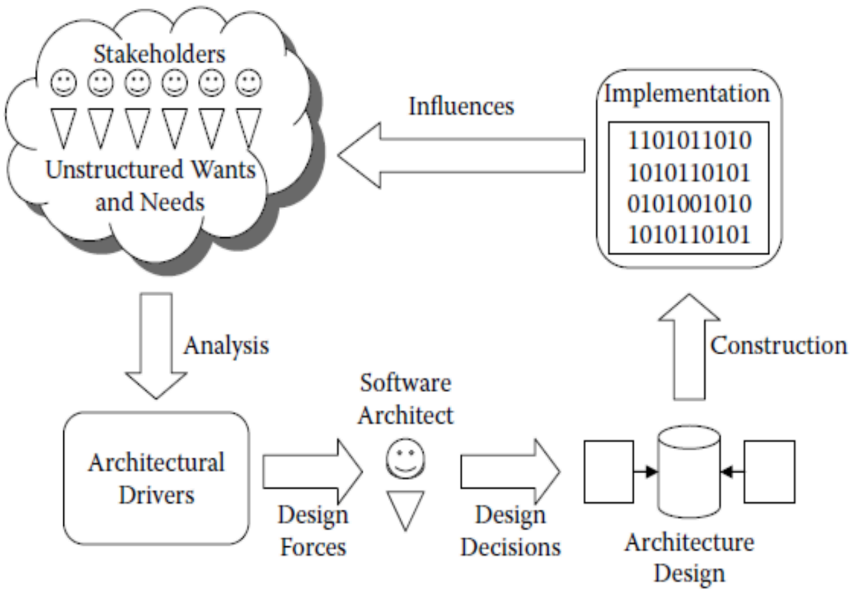
- **Въздействие**– състояние/събитие, което подлежи на обработка;
- **Източник** – обект (човек, система или нещо друго), който генерира въздействието;

- **Обект** – системата или конкретна нейна част, върху която се случва въздействието;
- **Контекст** – условията, при които се намира обекта по време на обработката на въздействието;
- **Резултат** – действията, предприети от обекта при случването на въздействието;
- **Количествени параметри** – резултатът трябва да подлежи на някакви количествени измервания, така че да позволи проверката дали сценарият се изпълнява съгласно изискванията.

Пример: По време на работа на системата (контекст), потребителят (източник) прави годишна справка (въздействие) и системата (обект) обработва заявката и връща pdf файл (резултат) за по-малко от 15 секудни. (количествени параметри)

3. Проектиране на софтуерната архитектура

3.1. Процес за проектиране



Процесът на проектиране представлява циклично изпълнение на последователност от стъпки, в следствие на което се изгражда дадена софтуерна архитектура.

В началото се избират целите (изискванията) с най-висок приоритет, които се превръщат в use-case или качествени сценарии. Това са така наречените **driver**-и на архитектурата, като се избират не повече от 10.

След като driver-ите бъдат избрани се пристъпва към проектиране. По време на проектиране могат да се появят други проблеми за разрешаване, които да доведат до промяна на driver-ите и така да се зацикли до брзкрайност.

3.2. Избор на подходящи структури

Структурата, която се получава директно от този подход, е декомпозиция на модулите. В зависимост от driver-ите, наличните ресурси и приоритетите на заинтересованите лица се избират и други подходящи структури за документиране.

3.3. Последователност на създаване на архитектурата

Ще разгледаме **Attribute-Driven Design (ADD)** като подход за проектиране. Той е рекурсивен процес на дефиниране на архитектурата, като на всяко ниво задълбаване основна роля играят качествените изисквания. Крайният резултат от **ADD** е първоначалната декомпозиция на модулите.

ADD приема следните входни данни:
- функционални изисквания, като сценарии за употреба (use-cases);
- функционални ограничения (constraints);
- качествени сценарии.

Той включва следните стъпки:
1. Избира се модул за декомпозиция.
2. (а) Избират се архитектурни driver-и, които се отнасят за модула;
   (б) Избира се архитектурния модел на модула, чрез избор и конфигурация на тактитки за осъществяване на driver-ите;
   (в) Създават се подмодулите спрямо избрания модел;
   (г) Приписва се функционалност на подмодулите;
   (д) Създават се други структури за подмодулите (например на процесите);
   (е) Дефиниция на интерфейсите на подмодулите;
   (ж) Проверява се декомпозицията, дали отговаря на изискванията.
3. Задълбаваме рекурсивно ако има нужда от още детайлизация на някои подмодули.

4. Тактики за постигане на желаните качествени показатели

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

4.1. Тактики за изправност

4.1.1. Откриване и предпазване от откази

- Ping/Echo, Healthcheck – компонент А пуска сигнал до компонент Б и очаква отговор в рамките на определен интервал от време;
- Heartbeat, Keepalive – компонент периодично изпраща сигнал, който друг компонент очаква;
- Изключения – обработват се изключения, които се генерират, когато се стигне до определено състояние на отказ.

4.1.2. Отстраняване на откази

- Активен излишък – активно дублиране и поддържане в едно състояние на важните компоненти. Downtime за милисекунди;
- Пасивен излипък – активният компонент реагира на събитията и синхронизира резервните. Downtime за секунда до часове;
- Резерва (Spare) – поддръжка на резервни изчислителни мощности. Downtime за минута до часове;
- Извеждане от употреба – спиране на компонент за избягване на сривове. Например за предпазване от mem-leaks;
- Следене на процесите (Process monitoring) – процес следи други процеси, които ако откажат се преинициализират.

4.1.3. Повторно въвеждане в употреба

- Паралелна работа (shadow mode) – компонент, който е бил счупен, се оставя да поработи за известно време в паралел с другите компоненти, преди да бъде въведен обратно;
- Ре-синхронизация на състоянието;
- Контролни точки и rollback – при счупване, системата се възобновява (rollback) до последно запомненото консистентно състояние (checkpoint).

4.2. Тактики за производителност

4.2.1. Намаляване на изискванията

- Увеличаване на производителността на изчисленията;
- Премахване на overhead от ненужни изчисления;
- Промяна на периода на настъпване на периодични събития;
- Промяна на тактовата честота (rate limiting) – пропускаме само част от събитията когато не можем да променим периода на идването им;

- Опашка с краен размер – аналог на leacky bucket алгоритъма.

4.2.2. Управление на ресурсите
- Паралелна обработка на заявките;
- Излишък на данни/процеси – cache, load-balancing и други;
- Включване на допълнителни ресурси – повече/по-мощен хардуер.

4.2.3. 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.

4.3. Тактики за изменяемост
Тактиките за изменяемост са:
- Локализиране на промените (намалява броя на директно засегнатите модули);
- Предотвратяване ефекта на вълната (например open-closed SOLID принципа);
- Отлагане на свързването при внедряване.

4.4. Тактики за сигурност
Тактиките за сигурност са:
- Устояване на атаките (doorlock);
- Откриване на атаките (alarm);
- Възстановяване след атака (застраховка).

4.5. Тактиките за изпитаемост (Testability)
Testability тактиките са:
- Запис и възпроизвеждане на информацията, която минава през даден интерфейс;
- Разделяне на интерфейса от реализацията, за да може да се mock-ва реализацията;
- Специализиран интерфейс за тестване (например mock server);
- Вградени модули за мониторинг (например излъчване на метрики към интерфейс).

5. Архитектурни стилове
Архитектурният стил дефинира семейство от системи чрез използването на шаблон за структурна организация. Стиловете обуславят речника на компонентите и конекторите, използвани в тях, както и ограниченията в начина на употребата им, например топологията и семантиката на изпълнението им.

5.1. Pipe-and-filter

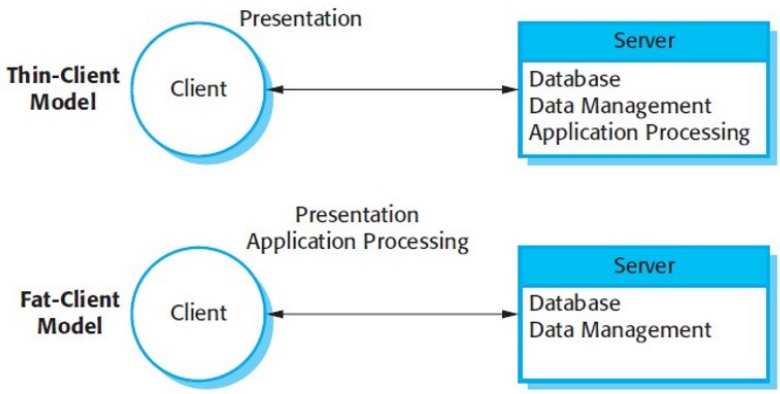
Модулите се делят на **филтри**, **канали**, **източник на данни и консуматор на данни**. каналите са потоци от данни изпълняващи ролята на конектори между филтрите. Филтрите са модули, трансформиращи входните си данни, които могат да работят паралелно и независимо едни от други, като всеки филтър започва да предава изходните данни към следващия филтър преди да е свършил с обработката на всичките данни. Този начин за обработка на данни позволява паралелизъм (конвейеризация).

Филтрите нямат никаква информация за съседите си, което прави архитектурата **гъвкава и лесна за разбиране**. От друга страна, **производителността страда заради parse-ването на данни** от всеки филтър.

5.2. Layered
Тук компонентите са вертикално разделени на слоеве, като всеки слой може да предлага услуги на слоя над себе си и да използва услугите на слоя под себе си. Може да разгледаме този стил като разширение на клиент сървър архитектурата – всеки слой е клиент за слоя под себе си и сървър за слоя над себе си.

Предимствата включват **абстракция и лесна изменяемост**, докато недостатъците включват **компромиси с производителността** поради стриктните правила на комуникация, както и **по-сложен дизайн процес**.

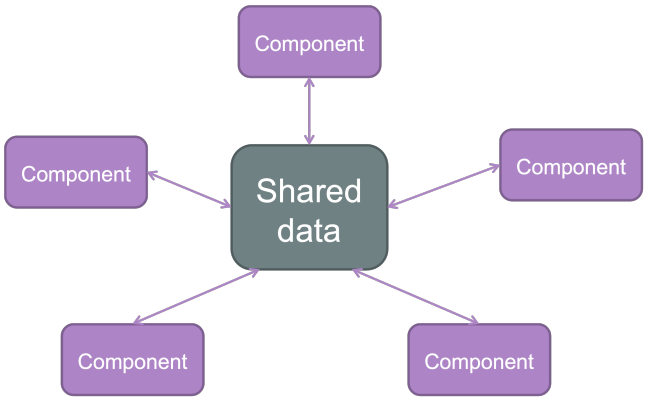
5.3. Client-server



Системата е построена като набор от сървъри, предлагащи услуги, и клиенти, които използват тези услуги. Клиентите могат да бъдат „тънки“ или „дебели“ – дебелите имплементират част от функционалността, а тънките – единствено потребителския интерфейс.

Предимствата на този стил са **централизираните данни и сигурността**, а недостатъците – **риск от претоварване**, както и **нужда от резерви в случай на server failure**.

5.4. Repository/Blackboard



Repository и Blackboard са две вариации на архитектурния стил „споделени данни“. Тук данните могат да се разглеждат като конектор между компонентите (агентите). При repository вариацията, когато се изпратят данни към общия конектор, всички агенти биват уведомени. При blackboard вариацията инициативата е на модула с данните, а агентите са абонати за събития (event listeners).

Предимствата на този стил включват **скалируемост** – компоненти се добавят лесно, и висока ефективност при размяна на голямо количество данни. От друга страна, споделената памет представлява **bottleneck** при голямо количество компоненти и заявки, също така е **трудна за имплементация** при разпределени системи.

5.5. Model-View-Controller

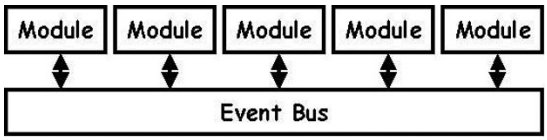
Състои се от:

- **модел** – изпраща данни към изгледа, приема заявки за промяна на данните и управлява операциите с тях;
- **изглед** – управлява презентацията на данните пред потребителите;

- **контролер** – отговаря за взаимодействието с потребителя – натискане на клавиш, кликване и т.н., като изпраща заявки към модела и изгледа за необходимите действия.

Този стил прави системата **гъвкава и разширяема**, но добавя **много сложност към архитектурата** както и може да доведе до **компромиси с производителността**.

5.6. Implicit invocation/Message passing



При този стил компонентите си комуникират чрез т.нар. „събития“. Тези събития може да съдържат както контролни съобщения, така и данни. Компонентите при такава архитектура работят паралелно, а конектора между тях е шина, по която се предават събитията.

Предимствата включват **сигурност**, както и **гъвкавост** (**loose coupling на компонентите)**. Недостатъците са **компромис с надеждността (ако шината умре)**, както и **несигурност** (ами ако няма компонент, който да отговори на дадено събитие?).

6. Документиране на софтуерна архитектура

6.1. Предназначение на документацията

Документирането на СА е въпрос на документиране на всички съставящи я структури потделно. Документацията е от важност с оглед на това в бъдеще да може системата да бъде лесно поддържана, безпроблемно изменяна и ясно разказвана на клиенти и потребители. Трябва да е достатъчно абстрактна, така че да бъде разбрана от нови служители, но и да е достатъчно детайлна, че да послужи за основа на проектирането.
За различните лица са предназначени различни документи. Най-често се създава набор от документи с обогатено съдържание, което указва къде каква информация се съдържа.

6.2. Основен принцип на документацията

Документацията се състои от описание на различните структури и когато тя се пише се мисли най-вече кой ще я чете. Техноописецът трябва да се поставя на мястото на четящия. В документацията **различните структури може да се групират по различни начини** в зависимост от това, за **кого конкретно е предназначена съответната документация**.

6.3. Съдържание на документацията

Няма изграден индустриален стандарт за съдържанието на документацията на дадена структура. Тук разглеждаме 7-елементно съдържание, доказано в практиката.

1. **Първично представяне** – обикновено графично, често UML, само с първостепенна информация;
2. **Описание на елементите и връзките** – детайлно описание на елементите и връзките + второстепенни детайли. Следва да се опише смисъла и ролята на всеки един елемент и връзка, както и поведенията и интерфейсите на елементите;
3. **Описание на обкръжението** – информация за това как елементите от документираната структура си взаимодействат с обкръжението – други системи, интерфейси, протоколи и т.н.;
4. **Описание на възможните вариации** – често архитектурата е изградена така, че да позволява варианти за някои от детайлите, като конкретния вариант може да бъде избран на по-късен етап. Обикновено се дават като ограничителни условия и изисквания. Трябва да бъдат описани вариантите и условията;
5. **Архитектурна обосновка** – обяснява на заинтересованите лица защо структурата е проектирана по начина по който е описана. Обоснование на взетите решения, алтернативи, резултати, предположения;
6. **Терминологичен речник** – кратко описание на използваната стандартна и нововъведена терминология;
7. **Допълнителна информация** – всичко останало, административна информация, описание на съдържанието си.

6.4. Избор на структури, които да бъдат документирани

Различните перспективи преследват различни цели и имат различно предназначение. Кои структури ще бъдат документирани зависи от това, кой ще чете документацията.
Алгоритъмът за избор на структури е:

1. **Създава се таблица** с разпределение на приоритетите на различните ЗЛ по различни изгледи;
2. **Комбинират се перспективите** за да се намали броя им (например декомпозиция + слоеве, процеси + внедряване и други);
3. **Задават се приоритети** на различните перспективи.