Павел Даниелов Иванчев, № 471219001, 76група

**Реферат на тема “Софтуерна архитектура”**

Софтуерната архитектура е множеството от структури, необходими за описанието на дадена система. Тя се състои от софтуерни елементи (компоненти), отношенията между тях и свойствата им. Софтуерната архитектура представлява “ранни” или “важни” решения в проектирането на софтуерна система. Това напрактита не е съвсем вярно, тъй като при гъвкавите методологии на разработка, важни решения могат да бъдат взети и на по-късен етап. От друга страна понякога е трудно да се определи кои от взетите решения са важни и кои не. Всяка структура съставлява софтуерни елементи, връзки между тях и свойствата на двете – елементи и връзки. Архитектурата на софтуера е метафора, аналогично е на архитектура на сграда. Тези структури са необходими за аргументиране на софтуерните системи. Чрез софтуерната архитектура се вземат структурни решения, които са скъпи за промяна след реализация. За това се препоръчва да са добре обмислени преди реализирането им. Софтуерната архитектура НЕ Е неразбираема, просто кутийки, без никаква информация за тях, без контекст, без бизнес цел и без заинтересовани от нея. Софтуерната архитектура е множество от структури, разглеждани в различен контекст. Тя е абстракция, необходима за разбиране на системата. Софтуерната архитектура е област с нарастващо значение за практикуващите и служителите в правителството, индустрията и академичните среди. Тъй като софтуерната архитектура все още е относително незряла както от изследване, така и от практическа перспектива има малко консенсус относно терминологията и представянето. Всички въпроси и всички неясноти в областта на софтуерната архитектура тепърва трябва да бъдат разглеждани. Тези структурни проблеми са свързани с дизайна, софтуерната архитектура в крайна сметка е форма на софтуерния дизайн, който се появява най-рано при създаването на системата, но по-абстрактно ниво от алгоритми и структури от данни. Според това, което е останало смятан за основен документ за софтуерната архитектура, Мери Шоу и Дейвид Гарлан предполагат, че тези "Структурните въпроси включват брутна организация и глобална структура за контрол; протоколи за комуникация, синхронизация и достъп до данни; възлагане на функционалност на дизайнерски елементи; физическо разпределение; композиция на дизайнерски елементи; мащабиране и производителност; избор между алтернативи на дизайна **"[1].**Всяко от различните определения на софтуерната архитектура подчертава някои от тях структурни въпроси и съответни начини за тяхното описание. Всяка от тези позиции обикновено може да се проследи до идея за това, което поддръжникът иска да направи. Важно е да разберем, че макар да изглежда объркващо да имаме много тълкувания, тези различни интерпретации не се изключват взаимно, нито представляват афундаментален конфликт за това какво е софтуерна архитектура. В моят реферат съм обърнал внимание на тези различни тълкувания или възгледи. На този етап обаче е важно да се знае, че заедно представляват спектър в изследванията на софтуерната архитектурна общност за акцента, който трябва да се постави върху архитектурата - нейна съставни части, цялото образувание, начина, по който се държи веднъж построен, или изграждането на него. Взети заедно, те формират консенсус по отношение на софтуерната архитектура и осигуряват по-голяма картина. Една структура е архитектурна, ако описва системата и нейните свойства. Изводът е, че софтуерната архитектура е свързана със структурните свойства на системата. Структурните свойства могат да бъдат изразени чрез компоненти, взаимовръзка с принципи и насоки за тяхното използване. Точните структурни свойства към и начините за тяхното представяне варират в зависимост от това какво е структурният интерес за потребителя на архитектурата. Изследването на софтуерната архитектура е до голяма степен изследване на софтуерната структура, която започна през 1968 г., когато Edsger Dijkstra посочи, че си струва да бъдеш загриже как софтуерът е разделен и структуриран, за разлика от простото програмиране, където трябва да се получи правилен резултат [2]. Dijkstra пише за операционна система и първо изложи идеята за слоеста структура, в която бяха програмите групирани в слоеве, а програмите в един слой могат да комуникират само с програма в съседни слоеве. Дийкстра посочи елегантната концептуална експозиция идва от такава организация, с произтичащите печалби в развитието и лекота на поддръжка. Цялата работа в областта на софтуерната архитектура може да се разглежда като развиваща се към парадигма на разработването на софтуер, базиран на принципите на архитектурата, и за абсолютно същите причини, дадени от Dijkstra и Parnas: Структурата е важна и правилното изграждане на структурата носи ползи. В съчетание с това важно академично разбиране за структурата на програмите и системите дойде дълга поредица от практически опит в работата със системи в няколко силно населени домейни, като компилатори. През 70-те и 80-те години, дизайнът на компилатора е еволюирал от поредица от отделни усилия, всяка от които новаторска и безпрецедентно, в едно със стандартни, кодифицирани парчета и взаимодействия. Днес за това как да се изгради компилатор изобилства от информация и домейнът е узрял доточка, от която днес никой не би помислил за миг да изгради компилатор от нищото, без повторно използване на опита на стотиците предишни примери. Повторното използване на структурните решения и компонентите за система също позволява повторно използване неговата структура на разбивка на работата, оценки, организация на екипа, планове за тестове, интеграция на планове, документация и много други трудоемки активи. Сега съществуват много други домейни, които чрез практика и повторение и споделяне сред многото ранни членове на семейството, сега проявяват обща структура, интернет-стратегии за свързване, разпределяне на функционалност към компоненти, взаимодействие между компонентилица и цялостна обосноваваща обосновка. Работа в софтуерна архитектура може да се разглежда като опит за кодифициране на структурната общност между членовете на семейство програми, така че решенията за проектиране на високо ниво, присъщи на всеки членна семейство програми не е необходимо да се изобретяват отново, да се валидират и описват. Изследването на софтуерната архитектура се е развило чрез спазване на принципите на проектиране и действия, които дизайнерите предприемат, когато работят върху реални системи. Това е опит да бъдат абстрахирани общите черти, присъщи на дизайна на системата, и като такива, тя трябва да отчете за широк спектър от дейности, концепции, методи, подходи и резултати. Тази разлика се отдалечава от по горният подход, който определя софтуерната архитектура и след това картографира съответстващите текущи дейности на термина. Това, което виждаме да се случва е, че хората наблюдават много дейности на дизайнерите и се опитват да ги приспособат чрез по-широк термин софтуерна архитектура. Тъй като това проучване продължава, сближаване на определението не се е случило. Въпреки че притежава дълги корени, областта на софтуерната архитектура е наистина съвсем нова, както се съди по неотдавнашното вълнение от книги, конференции и литература за него. Хората често правят аналогии с други употреби на думата архитектура, за която имат известна интуиция. Те обикновено свързват архитектурата с физическата структура(сграда, улици, хардуер) и физическа подредба. Архитект на сграда има перспектива на архитектурата, която се движи от необходимостта да се проектира сграда, която като обектът отговаря на нуждите и изискванията, включително достъпност, естетика, светлина, поддържаемост и др. [3]. Софтуерният архитект има перспектива, която е водена от необходимостта от проектиране на система, която да отговаря на нужди като паралелност, преносимост, еволюбилност, използваемост, сигурност и др. Използва се, за да разбере, че перспективата е важна и структурата и може да има различни значения в зависимост от мотивацията за изследване на структурата. От какво да черпим тази дискусия е, че точното определение на софтуерната архитектура не е почти толкова важно като концепцията и какво ни позволява разследването. Софтуерната архитектура обикновено се отнася до някаква комбинация от структурни възгледи на системата, с всеки изглед легитимна абстракция на системата по отношение на определени критериитерии, която улеснява определен тип планиране или анализ. Тази относително просто концепцията е обкръжена от голямо разнообразие от заинтересовани страни и участници в разработване на изделия. Архитектурата се превърна в концепция, която представлява много неща на много хора. Архитектурата представя публичната част на системата, чрез нейните интерфейси. Интерфейсите скриват конкретната си реализация и предоставят контакт за взаимодействие с други елементи. В този смисъл, архитектурата може да има множество реализации и за това е абстракция. Всяка система има архитектура – документирана или не. Архитектурата описва поведението на елементите по време на изпълнение. Не всички архитектури са добри по отношение на бизнес целите. Структурата (structure) е самото множество от елементи, така както те съществуват в софтуера или хардуера. Изгледът (view) е представяне на кохерентно множество от архитектурни елементи, така както са записани или четени от заинтересованите в системата. Той представя множеството от елементи и отношенията между тях. Архитектурните структури са:

модулните структури;

структурите компонент-конектор;

структурите на разпределението

Архитектурата е важна за нас поради няколко неща:

Взаимна комуникация - Софтуерната архитектура представлява общо високо ниво абстракция на системата, която повечето, ако не всички, заинтересовани страни в системата могат. Използване като основа за създаване на взаимно разбирателство, формиране на консенсус и комуникиращи помежду си.

Ранни дизайнерски решения. Софтуерната архитектура представлява въплъщение на най-ранният набор от дизайнерски решения за система и тези ранни обвързвания носят тегло далеч непропорционално на индивидуалното им тегло по отношение на останалата разработка на системата, нейното обслужване при внедряване и нейната поддръжка живот.

Софтуерната архитектура олицетворява сравнително малък, интелектуално разбираем модел за това как е структурирана системата и как нейните компоненти работят заедно. Този модел може да се прехвърля през системи, по-специално, той може да бъде приложен към други системи, показващи подобни изисквания и може да насърчи повторно използване в голям мащаб.

Качеството на една система се определя от фактори, различни от функционалността й. Качеството на системата главно се определя от способностите и поведението на системата. Атрибути като мащабируемост, изменяемост, сигурност и т.н., не са заложени в реализацията на функционалните изисквания. Атрибутът на качеството е измеримо или подлежащо на тестване свойство на системата, което определя доколко системата задоволява нуждите на заинтересованите лица. Атрибутите за качество на софтуера са показатели, които описват предвиденото поведение на системата в средата, за която е предназначена. Атрибутите за качество осигуряват средства за измерване на възможностите на даден продукт. Софтуерната архитектура влияе дълбоко върху повечето качества по един или друг начин, а атрибутите за качество на софтуера влияят на архитектурата. Има различни атрибути за оценка на качеството. Пример за такъв е RESPONSIVENESS. Този атрибут се дефинира като бързина на системата да реагира на потребителската заявка. Дългите закъснения могат да бъдат основна причина за неудовлетвореност на потребителя. Той може да сметне, че системата е повредена или че поставената от него команда е игнорирана. Този атрибут не се припокрива с ефективността. Разпределеността е друг важен атрибут дефинира като способността да се управлява група от системи като едно цяло и да се извършва управление от всяко място в мрежата. Трябва да се определи дали даден компонент ще се изпълнява в основния процес или в самостоятелен процес същата машина или на отдалечена машина (сървър). Възможността за сервизно обслужване Отнася се до способността за наблюдение на системите по време на тяхната работа, идентифициране на проблеми, извършване на анализ на основни причини за тези проблеми. Основната цел е откриване и решаване на проблем, т.е. възстановяване на системата във функциониращо състояние. Този атрибут се отнася до способността за техническа поддръжка, идентифициране и отстраняване на грешки без системите да се извеждат от работостобност. Проблеми могат да бъдат да бъдат докладвани от клиенти или представители на техническо обслужване. Всяка система трябва да притежава модул, поддържащ журнал за грешки (Error log). При необходимост този модул се конфигурира да генерира съобщения за грешки с различна степен на детайлност. Модулът може автоматично да известява отговорните за поддръжката на система лица за възникването на проблем чрез е-mail или друг комуникационен канал. Тя е важен фактор, влияещ върху удовлетвореността на потребители при работата със системата. Високата сложност на софтуера е основната причина за проблемите с надеждността на софтуера. Надежността на системата може да се провери чрез различни видове тестове – load тестове (за натоварване), тестове за отказоусточивост и др. Показателите за надеждност, които могат да бъдат използвани за количествена оценка на надеждността на софтуерния продукт, са следните: Средно време до отказ/грешка (Mean Time to Failure - MTTF)- описва се като интервал от време между двата последователни отказа. Средно време за ремонт (Mean Time to Repair - MTTR) – при възникване на грешка, е необходимо известно време за нейното отстраняване. Проверка на софтуера е степента, в която софтуерният артефакт (т.е. софтуерна система, софтуерен модул, изисквания или документ за проектиране) поддържа тестване в даден тестов контекст. Мащабируемостта е способността на системата да се справи с увеличаването на натоварването без да намалява производителността си. Необходимост от мащабируемост може да възникне при: Увеличаване на броя потребители на системата и увеличаване на обема на обработваните данни. Има два вида мащабируемост Хоризонтална (Scale Out) – представлява добавяне/внедряване на допълнителни модули от същия тип, изпълнявани най-често на отделни сървъри (Load Balancing). Вертикална (Scale up) – добавяне на ресурси към вече съществуващ изчислителен възел (сървър) – добавяне на процесор, памет, пропускателна способност на мрежата. Мащабируемост на база данни. Функционалните изисквания определят какво ще прави системата. Всяка система извършва работата, за която е предназначена. Функционалните изисквания се постигат чрез разпределяне на отговорностите в различните модули на системата. Изискванията на атрибутите за качество са квалифициране на функционалните изисквания в определен контекст, например:

• Колко бързо системата отговаря на заявка?

• Как реагира на грешка

• Колко е гъвкава системата при изменение на функционално изискване?

Изискванията на атрибутите за качество се постигат чрез проектни решения и структури в системата, чрез поведението и комуникацията между елементите. Ограничението е проектно решение, което вече е направено. Над ограничението надграждаме. Ограничението може да бъде избрана архитектура, избран език за програмиране, функционален стек, рамка и т.н. Ограниченията са задължителни. Ограниченият трябва да се съчетават с проектните решения, за да бъдат постигнати. Защо един проект притежава висока сигурност, а друг висока производителност? Постигането на тези качества е въпрос на фундаментални архитектурни решения – тактиките. Тактиката е архитектурно решение, чрез което се контролира резултата на даден сценарии за качество. Наборът от конкретни тактики се нарича архитектурна стратегия. Тактиките се описват чрез: Разпределение на отговорности-идентифицират се основните, водещи отговорности (системни функции, инфраструктура ...) Разпределение им става по модули, процеси, и т.н. Стратегии, които се използват за разпределение на отговорностите са функционална декомпозиция, моделиране на процеси и т.н. Модел на координация Моделът на координация е механизъм за интеграция. Трябва да се определят моделите, които се нуждаят от координация помежду си, а също така свойствата и мерките на координацията – време, точност, честота. Моделът на координация включва канали и механизми на взаимодействие. Модел на данните Моделът на данните се състои от водещите структури от данни и връзките между тях. Важен елемент от модела на данните е начинът на съхранение в база от данни, във файлова система и т.н. Управление на ресурсите При управлението на ресурсите трябва да се определи кои ресурси и кои системни елементи ще се управляват; как тези ресурси ще си взаимодействат и какво въздействие ще имат върху процесора, изпълнението и т.н. Връзките между архитектурните елементи могат да са между моделите, между елементите по време на изпълнение. Такава връзка е разпределението на елементите по процесорите по време на изпълнение или пък на артикулите в модела на базата от данни. Избор на технология При избор на технологии трябва да се види кои технологии са налични. Изборът на технология включва определянето на среда за разработка и среда за тестване. Трябва да се отчитат и страничните ефекти от избора на технология. Необходимо е да се следят зависимостите между технологиите и версиите им

Архитектурата е средство за комуникация със заинтересованите страни. Всеки участник в софтуерна система - клиент, потребител, мениджър на проекти, програмист, тестер и т.н. се занимава с различни аспекти на системата, за която архитектурата е важен фактор. Те могат да се занимават със същите аспекти, но от различни предходни перспективи. Например потребителят е загрижен, че системата отговаря на своите изисквания за наличност и надеждност. Клиентът е загрижен, че архитектът може да се изпълни по график и по бюджет. Мениджърът е притеснен (в допълнение цената и графика), че архитектурата ще позволи на екипите да работят до голяма степен независимо, взаимодействайки по дисциплиниран и контролиран начин. Разработчикът разказва за стратегии за постигане на всички тези цели. Архитектурата предоставя общо език, на който конкурентните опасения могат да бъдат изразени, договорени и разрешени на ниво, което е интелектуално управляемо дори за големи сложни системи. Без такъв език е трудно да се разберат достатъчно големи системи, за да се оправи информирани ранни решения, които до голяма степен влияят на тяхното качество и полезност.

Архитектурата олицетворява най-ранния набор от дизайнерски решения. За система. Тези ранните решения са най-трудни за получаване на правилните, най-трудните за промяна и имат най-широкообхватни последствия надолу по веригата, някои от които описваме като ниски. Архитектурата предоставя на строителите ограничения за изпълнението. Тя определя набор от ограничения върху изпълнението. Изпълнение се казва, че показва архитектура, ако тя съответства на структурните дизайнерски решения описани от архитектурата. Следователно изпълнението трябва да бъде разделено на предписаните компоненти, компонентите трябва да си взаимодействат помежду си в предварително и всеки компонент трябва да изпълнява отговорността си към другия компонент, както е продиктувано от архитектурата. Това ограничение на изпълнението е направено въз основа на системни и / или решения за разпределение за целия проект, които са невидими за изпълнителите, работещи по индивидуални компоненти и позволява разделяне на проблемите, което позволява на ръководството да вземе решение тези, които използват най-добре персонала. Конструкторите на компоненти трябва да владеят свободно уточняване на отделните им компоненти, но не и в системите за компромиси; архитектите не е необходимо да са експерти в дизайна на алгоритъма или тънкостите на програмен език. Архитектурата диктува организационната структура за развитие и Проекти за поддръжка. Не само архитектурата предписва структурата на системата, която се разработва, но тази структура се отразява в структурата на разбивка на работата и следователно присъща структура на проекта за развитие. Екипите комуникират помежду си от гледна точка на спецификациите на интерфейса към основните компоненти. Дейността по поддръжката, При стартиране ще отразява и софтуерната структура с екипи за поддръжка формирани за адресиране на специфични структурни компоненти .

Архитектурата позволява или изключва постигането на система Целеви качества Независимо дали системата може да покаже желаното (или задължително) качество. Атрибутите до голяма степен се определят от времето на избора на архитектурата. Атрибутите за качество могат да бъдат разделени на две категории. Първият включва тези, които могат да се измерва чрез стартиране на софтуера и наблюдение на неговите ефекти. Изпълнение, надеждност и функционалност попадат в тази категория. Вторият включва тези които не може да бъде измерено чрез наблюдение на системата, а по-скоро чрез наблюдение на развитието на дейности по поддръжка. Тази категория включва поддържаемост във всичките си различни видове. Адаптивност, преносимост, повторна употреба и други подобни. Модифицируемостта, например, зависи в голяма степен от модуларизацията на системата, която отразява стратегиите за капсулиране. Повторната употреба на компонентите зависи от това как тясно са свързани те са с други компоненти в системата. Производителността зависи до голяма степен от обема и сложността на междукомпонентната комуникация и координация, особено ако компонентите са физически разпределени процеси. Важно е обаче да се разбере, че архитектурата сама по себе си не може да гарантира функционалността или качеството, изисквани от системата. Лош дизайн или изпълнение надолу по веригата решенията за ментация винаги могат да подкопаят архитектурната рамка. Решения при всички етапи от жизнения цикъл - от проектиране на високо ниво до кодиране и внедряване качество на системата. Следователно качеството не е напълно функция на архитектурата дизайн. Необходима е добра архитектура, но недостатъчна, за да се гарантира качеството. Възможно е да се предскажат някои качества за системата, като се изследва нейната Архитектура. Ако архитектурата позволява или изключва атрибутите за качество на системата (но не може да гарантира за тях). Случайни архитекти-изборът на терапия би се извършил както и всеки друг метод. За щастие е възможно да се направи качествени прогнози за система, базирана единствено на оценка на нейните архитектитура. Техники за оценка на архитектурата като анализ на софтуерната архитектура метод (SAAM), предложен в Института за софтуерно инженерство (SEI), вземете отгоре надолу поглед към атрибутите на качеството на софтуерните продукти, които са активирани от специфични софтуерни архитектури. SAAM постъпва от изграждането на набор от сценарии, извлечени от домейн, които отразяват качествата, представляващи интерес за крайния продукт софтуер. Този набор включва директни сценарии (които упражняват необходимите софтуерни функциинационалности) и непреки сценарии (които отразяват нефункционални качества).

Направени между тези сценарии на домейни и кандидат архитектури, и резултатът е присвоена на степента, с която кандидат архитектурата удовлетворява очакванията на всеки сценарий. След това кандидат-архитектурите могат да бъдат противопоставени по отношение на тяхното изпълнение на сценарий-базирани очаквания от тях [4,5,6]. Архитектурата като преносим модел. Докато повторното използване на кода дава предимство, повторното използване на архитектурно ниво осигурява огромен лост за системи с подобни изисквания. Като архитектурно решение могат да бъдат използвани повторно в множество системи, всички въздействия на ранните решения, които ние просто описаните по-горе също се прехвърлят. Цели продуктови линии споделят обща архитектура. Продуктовите линии са получени от това, което Парнас посочва през 1976 г. като програмни семейства[7]. Струва си да се подреждат внимателно дизайнерските решения, които човек взема, така че тези най-вероятно да бъдат променени се случват най-късно в процеса. В архитектурно-базирано развитие-на продукта, архитектурата всъщност е сумата от тези ранни решения за проектиране и човек избира архитектура (или семейство от тясно свързани архитектури) които ще обслужват всички предвидени членове на продуктовата линия чрез вземане на дизайнерски решения които се отнасят за цялото семейство рано и като създават други, които се отнасят само за отделни хора членове със закъснение. Архитектурата определя какво е фиксирано за всички членове на продукта линия и какво е променливо. Дизайнерско решение за цялото семейство може да не е оптимално за всички производни системи, но е корпоративно решение, че качеството е известно, че е свързано с архитектурата и спестяванията на труд, спечелени чрез повторно използване на архитектурно ниво, компенсират загуба на оптималност в определени области. Архитектурата на продуктовата линия се превръща в развиване на основния актив на организацията, почти същия като другите капиталови инвестиции. Терминът специфична за домейн софтуерна архитектура се отнася за архитектури, предназначени да адресират известните архитектурни абстракции, специфични за дадени проблемни области. Примери за публикувани специфични за домейн софтуерни архитектури идват от ARPA. Специфична програма за домейн софтуерна архитектура (DSSA) [4]. Системите могат да бъдат изградени чрез импортиране на големи външно разработени компоненти, които са съвместими с предварително дефинирана архитектура. Докато предишните софтуерни парадигми са се фокусирали върху програмирането като първостепенна дейност, с напредък, измерен в редове код, развитие, базирано на архитектура често се фокусира върху съставянето или сглобяването на компоненти , които вероятно са били разработени отделно, дори независимо, един от друг. Този състав е тъй като архитектурата определя набора от компоненти, които могат да бъдат включени и оценени в системата. Архитектурата ограничава възможна подмяна (или допълнения).

По начина, по който те взаимодействат със заобикалящата ги среда, как получават и се отнасят да се откажат от контрола, данните, върху които работят и произвеждат и как имат достъп до тях, и протоколи, които използват за комуникация и споделяне на ресурси. Един ключов аспект на архитектурата е нейната организация на структурата на компонентите, интерфейсите, и оперативни концепции. Една съществена ценност на тази организация е идеята за взаимодействие и променливост. През 1793 г. масовото производство на мускети на Ели Уитни, основано на принтройка взаимозаменяеми части, обяви зората на индустриалната епоха. В дните преди физическите измервания да са били надеждни, това е било обезсърчително понятие. Днес в софтуер, докато абстракциите не могат да бъдат надеждно разграничени, понятието структурно взаимодействие променливостта е също толкова плашеща и също толкова важна. Търговски готови компонентите, подсистемите и съвместимите комуникационни интерфейси зависят от идеята за взаимозаменяемост. Все още има някои значителни нерешени проблеми, свързани със софтуерното развитие чрез композиция. Когато компонентите, които са кандидати за вносът и повторното използване са различни подсистеми, които са изградени с противоречиви архитектурни предположения, непредвидени усложнения могат да увеличат усилията необходими за интегриране на техните функции. Дейвид Гарлан е въвел термина „архитектурено несъответствие ", за да се опише тази ситуация [7]. Симптоми на архитектурно несъответствие са сериозните проблеми с интеграцията, които възникват, когато разработчиците на независими под-системите са направили архитектурни предположения, които се различават от предположенията на тези, които биха използвали тези подсистеми. За да разреши тези различия, Гарлан установява необходимостта да се направи изрична архитектуратурален контекст за потенциално многократно използваеми подсистеми. Някои дизайнерски практики, като скриването на информация, са особено важни за архитектурната последователност. Техники и инструменти за разработване на обвивки за преодоляване на несъответствия и принципи за композиране са необходими и софтуер. Най-елементарната нужда е от подобрени документи-практиките, включването на подробни предварителни условия за използването на интерфейси и конвенции за описване на типични архитектурни предположения.

Архитектурата позволява разделянето на функционалността на даден компонент от неговите компонентни механизми за взаимно свързване. Традиционните дизайнерски подходи се занимават предимно с функционалността на компоненти. Архитектурната работа се стреми да издигне връзките на компонентите с опаковката е малка част от софтуера, която осигурява по-използваем или подходящ интерфейс за софтуерна комбинация. Опаковката "скрива" (по-малко желания) интерфейс на компонента, така че само софтуерът за обвивка трябва да се справи с него.

Как компонентите си взаимодействат (координират, си сътрудничат, комуникират) се превръща в първокласно дизайнерско решение, където посочената цел е да се признае различни основни качества, придадени на системите чрез тези различни взаимовръзки-стратегии и за насърчаване на информиран избор. Обвързването на това рeшение може да бъде отложено и / или лесно променено. По този начин, двете системи се поддържат. Въпреки че привържениците на тази гледна точка говорят за „първо-клас конектори "[9], те всъщност правят възможно въпроса за съединители, които в много случаи се игнорират. Това контрастира с параграфа за програмиране digm, където механизмите за свързване са избрани много рано в цикъла на проектиране, те се обмисля много и е почти невъзможно да се промени. По-малкото е повече: изплаща се ограничаване на речника на алтернативните решения. Работата на Гарлан и Шоу за идентифициране на архитектурни стилове [8] ни учи на това въпреки че компютърните програми могат да се комбинират по повече или по-малко безкрайни начини, има нещо, което да бъде спечелено чрез доброволно ограничаване на относително малък набор от избор, когато става въпрос за програмно сътрудничество и взаимодействие. Предимствата включват подобрено повторно използване, по-способен анализ, по-кратко време за подбор и по-голямо взаимодействие -изтриваемост. Свойствата на софтуерния дизайн произтичат от избора на архитектурен стил. Архитектурните стилове са модели или идиоми на дизайна, които ръководят организацията на модулите и подсистеми в цялостни системи. Онези стилове, които са по-желани за част-ular проблемът трябва да подобри изпълнението на полученото дизайнерско решение, може би чрез разрешаване на по-лесен арбитраж на противоречиви дизайнерски ограничения, чрез увеличаване на прозрението в лошо разбран контекст на дизайна и / или като помага да се открият несъответствия в спецификациите на изискванията. Клиент-сървър 1 и тръбен филтър 2 са пример за архитектурни стилове Архитектурата позволява разработване на базирани на шаблони компоненти. Архитектурата олицетворява дизайнерските решения за това как компонентите си взаимодействат, макар да се отразява във всеки компонент на ниво код, може да бъде локализиран и написан просто веднъж. Шаблоните могат да се използват за улавяне на едно място на механизмите за взаимодействие при нивото на компонента. Например шаблон може да кодира декларациите за публичната област на компонента, където ще бъдат оставени резултати, или кодираните протоколи, които компонент използва, за да се ангажира с изпълнителния директор на системата. Пример за набор от фирми могат да бъдат архитектурни решения, позволяващи развитие на базирани на шаблони компоненти открити в подхода на структурното моделиране към софтуера [10]

Шаблонът е описание на набор от предварително дефинирани подсистеми. Шаблоните не се конструират от един-единствен човек: те отразяват опита на много разработчици. Те улавят съществуващи, добре доказани решения в разработването на софтуер и помощ за насърчаване на добри дизайнерски практики. Чрез писане на шаблон става по-лесно повторното използване на решението. Модели осигуряват общ речник и разбиране на дизайнерските решения. Имената на шаблони стават част от широко разпространен език за дизайн. Те премахват необходимостта да използвате дълго описание, за да обясните решение на определен проблем. Те помагат да се запази оригиналната визия, когато архитектурата се разширява и модифицира, или когато кодът се модифицира. Шаблоните поддържат изграждането на софтуер с дефинирани свойства. Когато проектираме приложение клиент-сървър, например сървърите трябва да се изграждат по такъв начин, че да инициира комуникация с своите клиенти. Шаблоните имат поне следните компоненти:

- контекст: ситуацията, която поражда проблем.

- проблем: повтарящият се проблем. Решение на проблема. Трябва да отговаря на изискванията, да обмисля ограниченията и да има желаните свойства.

- решение: доказано решение на проблема. Решението е дадено като структура с компоненти и взаимоотношения и като описание на поведението по време на изпълнение.

Архитектурата позволява разработване на базирани на шаблони компоненти. Архитектурата олицетворява дизайнерските решения за това как компонентите си взаимодействат, макар да се отразява във всеки компонент на ниво код, може да бъде локализиран и написан просто веднъж. Шаблоните могат да се използват за улавяне на едно място на механизмите за взаимодействие при нивото на компонента. Например шаблон може да кодира декларациите за публичната област на компонента, където ще бъдат оставени резултати, коитокомпонент се използва, за да се ангажира с изпълнителния директор на системата. Пример за набор от фирми могат да бъдат архитектурни решения, позволяващи развитие на базирани на шаблони компоненти открити в подхода на структурното моделиране към софтуера [10]. Всички модели са разработени отдолу нагоре това решението се записва под формата на шаблон. Шаблоните поддържат изграждането на софтуер с дефинирани свойства. Когато проектираме приложение клиент-сървър, например сървърът не трябва да се изгражда по такъв начин, че да инициира комуникация с своите клиенти.

Например моделът MVC (Model-View-Controller) поддържа променливост на потребителските интерфейси. Следователно моделите могат да се разглеждат като градивни елементи за по-сложен дизайн. Накратко, този модел означава, че концептуално различни проблеми са важн и се допълват отделно и че се използват слоеве с по-високо ниво на абстракция услуги с по-ниско ниво на абстракция, а не обратното. Този модел има следните предимства:- По-нисък слой може да се използва от различни по-високи слоеве. В TCP слой например от TCP / IP връзки, могат да бъдат използвани повторно без промени от различни приложения, като telnet или FTP .- Слоевете улесняват стандартизацията: ясно дефинирани и често срещани приетите нива на абстракция дават възможност за разработване на стандартни. Изградени задачи и интерфейси. Зависимостите се поддържат локални. Когато даден слой показва договореното между-лице към горния слой и очаква договорения интерфейс на слоя по-долу могат да се правят промени в слоя, без да се засягат други слоеве. Това означава, че разработчикът може да тества определени слоеве независимо внимателно от други слоеве и може да ги разработи и независимо: това подпомага развитието от екипи. Резултатът от горното е, че слоевете могат лесно да бъдат заменени с различни изпълнения.

Друг интересен модел е модела Client-Server сървърният компонент предоставя модел клиент-сървър услуги за множество клиентски компоненти. Искания на клиентски компонент услуги от сървърния компонент. Сървърите са постоянно активни за клиентите. Заявките се изпращат извън границите на процеса и машината. Това означава, че е необходим някакъв механизъм за комуникация между процесите: клиентите и сървърите могат да се намират на различни машини и следователно в различни входящи процеси. Всъщност можете да видите модела Client-Server като вариантна пластовия модел, пресичане на границите на процеса или машината: клиенти формират по-високо ниво и сървърът формира по-ниско ниво. Един от основните проблеми тук е че Заявките обикновено се обработват в отделни нишки на сървъра. Комуникацията между процесите причинява разходи. Данни за заявки и резултати често трябва да бъдат трансформирани или маршалирани, тъй като те имат различно представяне в клиент и сървър и защото има мрежа трафик.

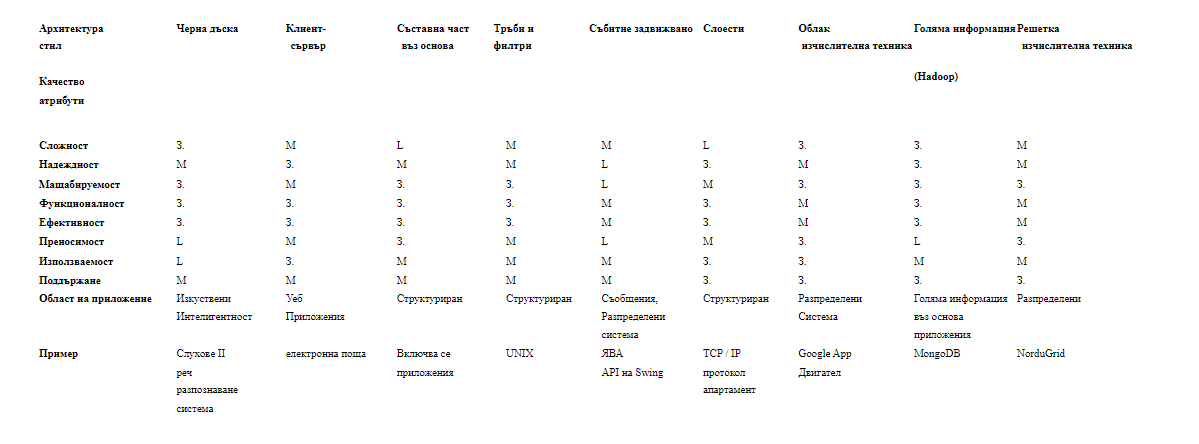
Моделът Master-Slave поддържа толерантност към. Образецът Master-Slave се прилага, например, при управление на процеса, вградени системи, в мащабни паралелни изчисления и в толерантни системи. Моделът Master-Slave е пример за принца „разделяй и владей“. В този модел аспектът на координация е отделен от действителна работа: опасенията са разделени. Робите са изолирани: няма споделена държава. Те работят паралелно. Латентността в комуникацията главен-подчинен може да бъде проблем, за например в системи в реално време: господарят и робите живеят в различни процеси.

Моделът Pipe-Filter осигурява структура за системи-образец на тръбен филтър, които произвеждат поток от данни. Всяка стъпка на обработка е капсулирана в филтърен компонент (представен с кръг на фигура 3.6). Данните се предават през тръби (представени със стрелките между съседни филтри). тръбите могат да се използват за буфериране или за синхронизация.

Шаблонът на Брокер се използва за структуриране на разпределени системи с отделяне. Брокерски модел представя компоненти, които си взаимодействат чрез отдалечени извиквания на услуги. Такива системите са много негъвкави, когато компонентите трябва да се познават взаимно местоположение и други подробности. Компонентът на брокера е повторно спонсориран за координацията на комуникацията между компонентите: той препраща заявки и предава резултати и изключения. Сървърите публикуват своите възможности (услуги и характеристики). Клиентите искат услуга от брокера и брокерът пренасочванасочва клиента към подходяща услуга от неговия регистър. Използването на модела на Брокер означава, че няма друг компонент освен брокерът трябва да се съсредоточи върху комуникацията между процесите на ниско ниво. Моделът на Брокер позволява динамична промяна, добавяне, изтриване и преместване на обекти и прави разпространението прозрачно за разработчика. Това изисква стандартизиране на описанията на услугите. Когато двама брокериси сътрудничат, може да са необходими мостове, за да се скрият подробности за изпълнението.

Моделът Event-Bus е модел, който се занимава със събития. Работи като Шаблон за събитие-автобус: източници на събития публикуват съобщения до определени канали в автобус за събития. Слушателите на събития се абонират за определени канали. Слушателите са Каналуведомени за съобщения, които се публикуват в канал, към който имат абонирани. Генерирането и уведомяването на съобщения е асинхронно: източник на събитие просто генерира съобщение и след това може да продължи да прави нещо друго. Не чака, докато всички слушатели на събития получат съобщението. Каналите могат да бъдат неявни, като се използва моделът на събитието, който е внедрен например в модела на събитията Java. Изричен канал означава, че абонатът се абонира директно за конкретен именен издател. Имплицитен канал означава, че абонатът се абонира за определен именен канал(към определен тип събитие в модела на събитията на Java) и не се нуждае да се знае кои производители произвеждат за този канал. Моделът Event-Bus се използва при мониторинг на процеси, в системи за търговия и в среди за разработка на софтуер. Моделът MVC улеснява получаването на множество изгледи на един и същ модел, който може да бъде свързан и изключен по време на изпълнение. То е възможно да се основава рамка за приложения на този модел. Моделът на MVC обаче увеличава сложността. Не всички видими еле-моментите се поддават на разделяне на модел, изгледи и контрол. Също,шаблонът потенциално води до много ненужни актуализации, където едно действието на потребителя води до различни актуализации. Изгледът и контролът са разделени, но са много тясно свързани. На практика по-често, те често се сглобяват. Изгледите и контролерите също са тясно свързани към модела. В уеб приложенията промяна в модела (за например, добавяне на имейл адрес към данни за лица) ще доведе до промяна в изгледа и контролера също (уеб сайтът ще трябва да покаже имота и възможността за промяна на имота трябва да бъде добавен също).

Моделът на Blackboard е полезен за проблеми, за които не е детерминиран. Сглобяват се няколко специализирани подсистеми съчетавайки знанията си, за да изградят евентуално частично или приблизително решение. Всички компоненти имат достъп до споделено хранилище за данни, черна дъска. Компонентите могат да създават нови обекти от данни, които се добавят към черната дъска. Компонентите търсят конкретни видове данни на черната дъска и може да ги намери чрез съвпадение на шаблона.

 Примери за проблеми, при които моделът на Blackboard може да бъде полезен прилагат се разпознаване на реч и извод за 3D структурата на молекулата. В следната таблица съм показал видовете шаблони техните предимства недостатъци и примери.

Системи Tuple Space, като JavaSpaces ,формират друг пример за този модел. Добавянето на нови приложения е лесно. Разширяване на структурата на данните пространство също е лесно, но модифицирането на структурата на пространството за данни е трудно, тъй като всички приложения са засегнати. Освен това процесите трябва да се споразумеят за структурата на споделеното пространство за данни. Може да има нужда за синхронизация и контрол на достъпа Мери Шоу и Пол Клементс са предложили класификация на стиловете, въз основа на съставните части (компоненти и съединители), контролът е съдебни дела и проблеми с данните. Тази класификация е следната: - Взаимодействащите процеси имат своя собствена нишка за контрол. Взаимодействащи процеси може да има формата на асинхронно предаване на съобщение, имплицитно извикване, чрез събития или отдалечени извиквания на процедури. Когато събитието Шаблонът на шината се реализира с независими процеси или проекти със собствена нишка на контрол, този модел е пример за този стил. Шаблонът клиент-сървър и модел Peer-to-Peer са били достатъчно.

- В стила на потока от данни, данните преминават от един компонент към друг, Шаблонът на филтъра на тръбата принадлежи към този стил. Някои случаи на шаблонът клиент-сървър също принадлежи към този стил, например когато клиент се използва за получаване и показване на поточно предаване на аудио или видео сървър. Стилът, ориентиран към данните, включва централно съхранение на данни. Пример Дантов центриран стил от този стил е шарката на Blackboard. Отново случаи на Клиент-Сървърният модел може да принадлежи на този стил. Например, ако основната функцията на сървъра е да управлява база данни и клиентите са свикнали с достъп до тази база данни. В Йерархичния стил системата е разделена на подсистеми Йерархичен стилс ограничено взаимодействие. Модели в този стил са интерпретаторът модел и шарката на слоевете.- Стилът на обаждане и връщане кара процеса на извикване да изчака връщането. Стил на обаждане и връщане на заявка. Моделите в този стил са Master-slave и, отново,шарката на слоевете. Обектно-ориентираните системи без нишки също са пример за този стил. Кой архитектурен модел е най-подходящ за дадена ситуация зависи от това кои изисквания имат най-висок приоритет, като например: Поддържане: колко лесно или трудно е да се добави допълнителна прокомпонент за прекратяване, например за филтриране на определени думи? Колко лесно или е трудно да се промени входният формат, чрез добавяне на номера на редове, например? В модела Pipe-Filter например добавянето на филтър е много лесно, но промяната на входния формат може да е трудно. - Многократна употреба: могат ли да се използват повторно отделни компоненти в други системи? В този случай моделът Pipe-and-Filter подобрява повторната употреба. - Изпълнение: достатъчно ли е времето за реакция? Е цялостен ресурс поведение (използване на памет в този пример) приемливо? Модели, които използваме паралелизъм, като модел на Pipe-Filter и Event-Шаблонът на шината ще има по-добра производителност. - Изричност: възможно ли е да се предостави обратна връзка на потребителя? На етап?Това не е възможно в модела Pipe-Filter, например.- Толерантност към повреди: за примера на KWIC няма разлика междуразличните решения, но толерантността към повреди би била подобрена ако е приложен шаблон Master-slave. Списъкът с изисквания и техните приоритети ще варира за всяка система. Няма налични твърди насоки, които да ви кажат кой модел ще бъде най-добре във всеки случай. Много зависи и от прилагането на модел. Независими процеси, например, могат да бъдат приложени чрез нишки или използване на процеси на различни машини. Балансът е-между комуникацията и изчисленията, капацитетът на процесорите включени и скоростта на комуникация между машините, сред други неща, ще решат кое изпълнение ще има най-добри резултати форма. И така, както "структурата на сграда" има много значения в зависимост от нечий мотив и гледна точка, софтуерът показва много структури и не можем да общуваме смислено за даден софтуер, освен ако не е ясно коя структура описваме. Освен това, когато се проектира софтуерът за голяма, сложна система, той ще бъде необходим да разгледаме повече от една структурна перспектива, както и връзките между тях. Въпреки че често се мисли за структурата от гледна точка на функционалността на системата-в допълнение към функционалността има системни свойства, като физическа дистрибуция, комуникация на процеси и синхронизация, които трябва да бъдат обосновани в архитектурно ниво. Тези други свойства са адресирани в множество структури често наричани архитектурни възгледи. Всеки различен възглед отразява определен набор от опасения, които представляват интерес за дадена група от заинтересовани страни в системата. Следователно възгледите са абстракции, всяка с по отношение на различни критерии. Всяка абстракция „кипи“ подробности за софтуера които са независими от загрижеността, адресирана от абстракцията. Всеки изглед може да бъде считан за софтуерен план и всеки може да използва своя собствена нотация, може да отрази свой собствен избор на архитектурен стил и може да определи какво се има предвид в неговия случай чрез ком-компоненти, взаимовръзки, обосновка, принципи и насоки. Мненията обаче не са напълно независими. Елементите в едно могат да се отнасят до елементи в друг, така че, докато е полезно да се разглежда всеки поотделно, също така е необходимо да следи строго относно взаимовръзките на тези възгледи. Изгледите могат да бъдат категоризирани като следва: Дали структурите, които представляват, се различават в системата време на изпълнение. Например, програмите съществуват по време на изпълнение; човек може да определи повикванията структура на системата чрез наблюдение на изпълнението на системата. Модули, обаче изчезват; модулите са чисто статично (преди изпълнение). Дали структурите описват продукта, процеса на изграждане на продукт или процеса на използване на продукта за решаване на проблем. Всички гледки обсъдени в този раздел са на продукта. Модел на взаимодействие с потребителя представя друг поглед върху архитектурата на системата и е типично представени чрез диаграми на обект-връзка. Други възгледи моделират проблема област или домейн. Някои автори разграничават възгледите според това каква информация показват. Например, Буджен прави разлика между функционално, поведенческо, структурно и моделиране на данни гледни точки [12]. Всички те се отнасят горе-долу към предишните две категории. Имайте предвид, че всеки изглед може да се тълкува или като описание на системата, която има или рецепта, създадена за постигане на съответното качество атрибути. Докато някои застъпват поетапния подход към архитектурния дизайн [14], опитите ce показа, че развитието на архитектурата е силно итеративно и изисква някои прототипиране, изпитване, измерване и анализ [13]. Текущата работа в Института по софтуерно инженерство твърди, че архитектите имат влияние определяно от фактори в три области [8]: изисквания (включително необходимите атрибути за качество) на системата или системите в процес на разработка. Изисквания, наложени (може би имплицитно) от организацията, изпълняваща развитие. Например, може да има изисквания или насърчение, за използване на хранилище на компоненти, обектно-ориентирана среда или предишни проекти, в които организацията е инвестирала значително.

Резултатите от предишни решения, независимо дали са били успешни, напълно пагубен или някъде между тях, ще повлияе на това дали архитектът използва повторно тези стратегии. Брукс твърди силно и красноречиво, че концептуалната цялост е ключът към звука системния дизайн и че концептуалната цялост може да бъде постигната само от много малък брой на умовете, които се събират, за да проектират архитектурата на системата [12]. Представяне и комуникация на архитектурата за да може архитектурата да бъде ефективна като гръбнак на дизайна на проекта, тя трябва да се съобщава ясно и недвусмислено на всички заинтересовани страни. В разработване операторите трябва да разбират работните задачи, които изисква от тях; тестерите трябва да под-издържат структурата на задачите, която им налага; ръководството трябва да разбере последствия от планирането. Следователно представителният носител трябва да бъде информативни и недвусмислиците трябва да бъдат прочетени от много хора с различни фонове. Самите архитекти трябва да се уверят, че архитектурата ще отговаря на поведението, Основни изисквания, производителност и качество на системата, които ще бъдат изградени от архитектурата. Следователно има предимство, ако представителният носител може да служи, като принос към техники за формален анализ като изграждане на модели, симулация, проверка фикция. За тази цел представителната среда трябва да бъде официално и пълно. Възможностите за анализ, които много ADL предлагат на масата, са ценни, но са склонни да се концентрират се върху свойствата на изпълнението на системата - нейното изпълнение, нейното поведение, нейните комуникационни модели и други подобни. По-малко представени са техниките за анализ на оценка архитектурата от гледна точка на атрибутите за качество, които не са по време на изпълнение придава на система. Главен сред тях е поддържаемостта, способността за подкрепа и промяна. Поддържането има много вариации: преносимост, повторна употреба, адаптивност, разтегливост; всички са специални перспективи за способността на системата да поддържа промяната. Появява се консенсус относно стойността на оценката, базирана на сценарии, за да се прецени архитектурата по отношение на атрибутите за качество извън изпълнение [13], [12]. Тази дейност се занимава с гарантиране, че разработчиците се придържат към структуритеи протоколи за взаимодействие, продиктувани от архитектурата. Архитектурна среда или инфраструктурата би била от полза тук. Работата в тази област обаче все още е доста безсмислена тура. Нуждата от софтуерната архитектура е предизвикана от необходимостта за солидна основа при големите и сложни системи. Тя е гаранцията за дълъг живот на системата. Ако не са предвидени различните сценарии и начини на използване на системата, общите проблеми и начините на справяне с тях, целите в дългосрочен план, то тогава софтуерният продукт е в риск. А това струва време и пари! Разбиране на системата от всяко заинтересовано лице, нейното структуриране на елементите е друга необходимост. Софтуерната архитектура играе ролята на инструмент за комуникация, обосновка при вземане на решения, средство за анализ и развитие на системата.

Областта на софтуерната архитектура е тази, която е постигнала значителен растеж през последното десетилетие и обещава да продължи този растеж в близкото бъдеще. Тъй като архитектурният дизайн разцъвтява в инженерна дисциплина, които са универсално признати и практикувани, има редица значителни предизвикателства, които ще трябва да бъдат отстранени. Много от решенията на тези предизвикателства с авероятно да възникне като естествена последица от разпространението и съзряване на архитектурните практики и технология, за която не знаем днес. Други предизвикателства възникват,причина за променящия се пейзаж на изчисленията и нуждите за софтуера В близкото бъдеще софтуерната архитектура ще изисква значителни нови иновации.

**Литература:**

**[1]** Abowd, G .; Бас, L .; Хауърд, L .; & Northrop, L. *Структурни*

*Моделиране: рамка за приложения и разработка*

*cess forFlight Simulators.*(CM U / SEI-93-TR-14, ADA 271348)

Питсбърг, Пенсилвания: Институт за софтуерно инженерство, Карнеги Мел

lon University, 1993

**[2]** Dijkstra, EW "Структурата на мултипрограмирането" THE "Система. " Комуникации на ACM 18, 8 (1968): 453-457.

**[3]** Александър, Кристофър; Ишикава, Сара; & Силвърщайн, Мъри.Езиков модел. Ню Йорк: Oxford University Press,1977 г

**[4]** Abowd, G .; Бас, L .; Казман, R .; & Webb, М. "SAAM: A

Метод за анализ на свойствата на софтуерната архитектура

tures, "81-90. *Сборник от 16-та международна конференция*

*енция по софтуерно инженерство.*Соренто, Италия, 16-21 май,

1994. Лос Аламитос, Калифорния: IEEE Computer Society Press, 1994

**[5]** Clements, P .; Бас, L .; Казман, Ж; & Abowd, G. "ПрогнозиранеКачество на софтуера чрез оценка на ниво архитектура, "485-498. Сборник, Пета международна конференция по софтуерКачество. Остин, Тексас, 23-26 октомври 1995 г. Остин, Тексас: американскиОбщество за контрол на качеството, Софтуерен отдел, 1995.

**[6]** Abowd, G .; Бас, L .; Казман, R .; & Webb, М. "SAAM: AМетод за анализ на свойствата на софтуерната архитектураtures, "81-90. Сборник от 16-та международна конференцияенция по софтуерно инженерство. Соренто, Италия, 16-21 май,1994. Лос Аламитос, Калифорния: IEEE Computer Society Press, 1994

**[7]** Парнас, Д. "За проектирането и развитието на програматаСемейства. " IEEE транзакции на софтуерно инженерство SE-2,1 (1976): 1-9.

**[8]** Браун, А.; Carney, D .; & Clements, P. "Казус в As-оценка на поддръжката на големи, софтуерно интензивни системиtems, "240-247. Сборник, Международен симпозиум иСеминар по системно инженерство на компютърно базирани системитемс. Tucson, AZ, 6-9 март 1995 г. Salem, MA: IEEE Com-puter Society Press, 1995.

**[9]** Парнас, Д. "За критериите за разлагане на системи вМодули. " Комуникации на ACM 15,12 (декември1972): 1053-1058.

**[10]** Парнас, Д. "Проектиране на софтуер за лесно разширяване иСвиване. " Транзакции на IEEE за софтуерно инженерствоSE-5, 2 (1979): 128-137

**[11]** Saunders, TF et al. Нов процес за придобиване на софтуерАрхитектура. Хартия на MITER Corporation, M92B-126. Бедфорд,MA, ноември 1992 г

**[12]** Шоу, М. "Концептуална основа за софтуерна архитектура." Покана-беседа, Първа международна работна среща по архитектура за мекиизделия системи. Сиатъл, Вашингтон, април 1995 г

**[13]** Сони, D .; Nord, R .; & Hofmeister, C. "Софтуерна архитектура вПромишлени приложения. "196-210. Сборник, 17-ти Международеннационална конференция по софтуерно инженерство. Сиатъл, Вашингтон, април23-30, 1995. Ню Йорк: Асоциация за изчислителни машиниери, 1995

**[14]** Witt, BI; Бейкър, FT; & Merritt, EW Софтуерна архитектураи Принципи на проектиране, модели и методи. Ню Йорк, НюЙорк: Ван Ностранд Рейнхолт, 1994