**5. Обектни, потокови и контекстни модели на софтуерната архитектура**

**ОО принципи:** капсулиране – осигурява видимост на функциите и прозрачност за имплементацията . (Например скрит вътрешен контекст и процедури. Частните променливи в класовете са неустойчиви, а публичен интерфейс е устойчив.)

наследственост – осигурява адаптивност на кода чрез наследяване и допълване на спецификациите – т.е. от общо (родителски клас) към частно (наследен клас, дериват)

полиморфизъм – осигурява адаптивна функционалност чрез развитие на наследяването. Има два вида полиморфизъм – вертикален (отмяна и предефиниране на атрибути в дериватите) и хоризонтален (презареждане на нов контекст за същия клас).

**АДТ:** Математически модел за определен клас от структури от данни , които имат подобно поведение. АДТ се определя косвено, само от операции, които могат да се извършват върху него и от математически ограничения на ефектите (а вероятно и разходите) на тези дейности. Класовете са имплементации на АТД с публичен интерфейс от атрибути и операции, а обектите са имплементации на класове, които се явяват техни «типове».

– UML-спецификация на клас с +/- модификатори на достъпността на атрибутите и операциите – (*фиг. 5.1*).

**Видове отношения между класовете:** статични (1.конструкция на комплексни класове от класове (композиция и наследяване); 2.статична консистентност (т.е. логичност) на зависимите класове – като при базите данни (агрегация и асоциация)) и динамични (примерно, обмен на съобщения).

Композицията е дефиниране на клас като съставен от други класове. Компонентите са активни, докато е активен и съставния клас и не се включват в други класове. (пресилено ограничение за garbage collection – чрез конструктурите и деструктурите на класовете); в UML – плътен ромб към главния клас с етикети на мощността - (фиг.*5.2)*

Агрегацията е аналогично отношение на класовете, но без изброените ограничения – *(фиг. 5.3)*

Асоциацията е обобщена композиция – *(фиг. 5.4)* ; характеризира се с:

1) име (етикет), което отразява свързващата функционална логика – напр.

«Customer places an/some Orders», 2) мощностите на асоцииране, 3) 2 асоциативни типа на връзката между двата класа (задават тип композиция към иницииращия клас)

- навигационната посока към инициирания клас – т.е. указателите на асоциираните класове са налични като атрибути в иницииращия клас (плътна линия)

- зависимост посока към зависимия клас – зависимия клас извиква операция

на асоциирания клас или променя негов атрибут (пунктир)

4) иницииращият клас може да асоциира повече от един класове

Наследяване и полиморфизъм: наследяването отразява взаимстване на повтарящите се

атрибути – деривата наследява всички публични атрибути (без частните). Полимофизмът е механизъм за диверсификация на дериватите при изпълнение

*(фиг. 5.5)*. В UML наследявянето се означава с триъгълна стрелка към основния клас.

В примера двата деривата се различават по методите на идентификация. Клиентът зарежда cookie в браузъра си, регистрираният потребител изпраща парола и ползва отстъпка (и двете функционалности отсъстват в базовия клас)

Наследяване и композиция: и двете черти поддържат взаимстването на атрибути между

класовете (reuse), но с различен обхват на приложение съгласно принципите:

наследяване се прилага при is-a отношение между деривата и базовия клас

композиция (или агрегация) се прилага когато отношението е has-a (Пример: базови класове Person и University, класът Student може да бъде дериват на двата класа или да има атрибути с указатели към двата класа или комбинация от двата подхода

Student IS-A Person → Student e уместно да бъде наследствен дериват на Person

Student HAS-A University → Student e уместно да има атрибут с указател към University)

Наследяването е противопоказно за капсулацията (локалността) на кода, тъй като промяна на атрибут в базовия клас предизвиква каскадни промени в дериватите -

пример *(фиг. 5.6)* - Student и Professor като деривати на Person (легитимно но

ниска капсулация) и като агрегиращи PersonalHandler (с прозрачна конверсия на обръщението към атрибути).

**ОО анализ:** анализът предхожда проектирането и имплементацията и се състои в

структуриране на предметната област и представянето й като набор класове с определена функционалност. Обикновено се състои в описание на потребителския сценарий чрез диаграма на случаите, от която се извлича и аналитичната (или

принципна) клас-диаграма. 1) Диаграма на случаите (Ivar Jacobson 1987) – пример за OPS (Order Processing System) *(фиг. 5.7)* : определя типовете потребители на системата (напр. клиент, счетоводство,доставка); определят се основните случаи, които ще се детайлизират като [една или повече] операции в етапа на проектирането (напр. случая добавяне на изделие в пазарската количка би изисквал и операция със складовата БД).

2) Принципна клас-диаграма: е абстрактно описание на класовете на системата – по-близко до сценариите и функционалността, отколкото до имплементацията (не отчита производителност на модулите, технологии и технологичност на проектирането и експлоатацията). Състои се от гранични, същностни и контролни класове (boundary,

entity, control). Граничните класове се извличат от интерфейсните случаи и са ориентирани към имплементация с GUI (Web форми, прозорци, браузър-плугини) или

като междинни интерфейси (middleware wrappers) към други системи. Същностните класове отразяват информационния слой (напр. клиентската или продуктова идентичност са същностни класове). Контролните класове отразяват отделните случаи т.е. операциите, които свързват граничните и същностните класове (пример *(фиг. 5.8)* – принципна КД на OPS.

**ОО проектиране:** проектирането е самостоятелна фаза в развоината дейност на

разпределените системи (Може да се приложи подход, различен от този на фазата на анализа – потоков(event driven), контекстен (data driven), структурен (с функции))

Целта на проектирането е декомпозиция на системата на технологични модули – (класовете се описват с техния интерфейс т.е. публичните им атрибути и операции, и се специфицират след това на фазата на имплементацията). Различават се високо и ниско ниво на проектирането. Високото ниво идентифицира класовете напр. с приложение на

CRC-карти и клас-диаграми за статичните отношения (specification/compile time) между класовете. Ниското ниво детайлизира проектираните класове и тяхното динамично взаимодействие (run time) с диаграми за взаимодействието (най-често с диаграми на последователността или на комуникациите) и на машината на състоянията (state machine) – като се използват диаграмите на случаите от фазата на анализ.

**1 стъпка** - прилагат се CRC карти (Class-Responsibility-Collaborator – Kent

Beck & Ward Cunningham, 1989) и/или клас диаграми за пълно (а не принципно като при анализа) описание на класовете. CRC картата на всеки клас съдържа таблица с описание на името, функционалните задължения (responsibility – заданията които изпълнява + контекста им) и списък клобориращи класове за изпълнение на тези задължения (пример за OPS от *(фиг. 5.9):* RegistrationPage и RegistrationController – *(фиг. 5.10).*

**2 стъпка** - описва се взаимодействието между обектите от стъпка 1 и се прилагат диаграми на последователността или на комуникациите. Моделът се състои от последователни стъпки, описани чрез обмен на съобщения (пример – диаграма на последователността за случая Registration – описва обмена между класовете RegistrationPage и RegistrationController – (фиг. 5.11): В горната част на диаграмата са взаимодействащите обекти – с означения <object\_name>:<class\_name> (името на обект може да отсъства). Връзките отразяват дейностите на съответните обекти и носят

съответните етикети – включително new за създаване на обект от клас-колаборатор. В примера само обектите successPage и failurePage са именувани – за разлика от останалите класове – тъй като се предават алтерантивно от RegistrationController към RegistrationPage)

**3 стъпка** - описва динамичното поведение на по-сложните класове за целия им цикъл на живот (напр. контролните класове) – с диаграми на машината на състоянието. ДМС се извлича от диаграмите на случаите, в които участва дадения клас. В ДМС отделните състояния означават стабилност на колекцията от променливи на средата и от вътрешни променливи на класа. Вътрешните променливи на класа обикновено задават граничните стойности, с които се сравняват променливи на средата (условие за

преход между състоянията на класа) и евентуално се изпълнява преход в друго състояние. За по сложните класове ДМС е съставна – включва и sub-state диаграми, но:

Сложният клас е желатeлно да се представи от няколко класа, ако логическата му функционалност не се описва от едно просто изречение; това се отразява обратно и в СRC-модела.

**4 стъпка** – представлява подробно описание на интерфейсите на всеки клас – изброяват се атрибутите и операциите и тяхната публичност (с + и - в UML). Публичната част от интерфейса е фиксирана и не трябва да се променя в следващата след проектирането фаза – имплементацията. Публичният интерфейс се състои главно от дефинирани константи и операции: Операциите в публичния интерфейс са 4 категории:конструктор, деструктор, аксесор и мутатор. Определянето на публичните атрибути (константи) се базира на следните фактори: 1) Какви са външните стойности, които класът използва в своите операции – от CRC-диаграмата – напр. класът RegistrationPage използва Име и Парола *(фиг. 5.12)*. 2) Какви са възможните състояния на класа от ДМС – те се включват като атрибути (но обикновено частни). 3) От мощността на асоциациите: 1..1 асоциация изисква скаларен атрибут-указател към асоциирания клас, а 1..\* асоциация – атрибут-колекция (вектор). 4) Други атрибути, необходими за изпълнение на операциите – обикновено са локални.

**Предимства и недостатъци на ОО архитектурите**: предимства: непосредствена връзка с потребителските сценарии и проблемната област; взаимстване (reuse) и капсулиране на имплементацията; лесно допълване чрез полиморфизма и класовете-деривати; устойчивост на системата, поради защитеност на локалните атрибути; удобен преход към други модели и най-вече към компонентна архитектура. Възможни проблеми: непредвидени странични ефекти при взаимодействието на много обекти, включително при асоциации 1..\*; интерфейсите и вътрешната имплементация на класовете – макар и пордукт на отделни фази – не са толкова разграничени, колкото при компонентните архитектури. Обикновено се разработват съвместно, което снижава нивото на абстракция (и сложност) на цялата архитектура, а също обичайно води до по фина грануларност в сравнение с компонентните архитектури; наследствеността между класовете често води до грешки в спецификацията и следва да се прилага много внимателно.

**Потокови (Data Flow) архитектури:** представят обработката като последователност от трансформации (т.е. групи операции) върху последователност от набори структурирани еднотипни данни. Системата се декомпозира на функционални модули или подсистеми (паралелизъм по управление - аналогия с [нелинейните] конвейри). Интерфейсът между модулите може да е във формата на потоци (streams), файлове, канали (pipes, асинхронни потоци) и др. Основният паралелизъм е по данни, тъй като ритъмът на

обработка се задава от наличието на данни за обработка. По тази причина – отсъствието или минимизирането и импликацията на контролния поток – ПА са подход и стил, приложим предимно при автоматизирани процеси на обслужване – напр. езикови компилатори, автоматизирани системи с пакетно обслужване като разпределените транзактивни системи, вградените системи.

Топологията на пренос на данните между модулите се задава експлицитно с блок-диаграми *(фиг. 5.13)*. Обработката е асинхронна. Модулите поддържат само интерфейс по данни, не и контролен интерфейс и не се адресират взаимно – адресацията е само

чрез предаваните данни. По механизма на свързване между модулите (т.е. на обмен) се разграничават: пакетна обработка (Batch Sequential), филтрирани канали (Pipe & Filter) и контролни процеси (Process Control).

**1.Пакетна обработка (Batch Sequential)**: Това е най-старият модел на СА за обслужване в транзактивни системи и класическите ОС със стандартен файлов IO и редиректори. Приложението е скрипт с команди за изпълнение на съответните модули в UNIX, DOS, Tcl/Tk – напр. myShell.sh

exec searching kwd <inpitFile >matchedFile

exec counting <matchedFile >countedFile

exec sorting <countedFile >reportFile

Този стил е приложим и в съвременните ОО езици, където отделните обработващи модули, входът и изходът се представят като методи и атрибути на класа. Приложимост: 1) Данните (включително междинните резултати!) са оформени в пакети – файлове, т.е. с последователен достъп. 2) Модулите се представят като програми, които се активират със скрипт или като резидентни модули, които сканират входните си файлове. 3) Неприложима СА за интерактивен интерфейс. 4) Широко приложение за асинхронни паралелни процеси – данните се декомпозират като множество входни файлове, а обработващите модули се репликират в множество възли (принцип на обслужване в пакетната фонова обработка – Condor, Boinc).

**2.Филтрирани канали (Pipe & Filter)**: приложението се декомпозира на източник на данните, филтри, канали(pipes) и консуматор на данните (sink). 1) Данните са последователни FIFO потоци (буфери, опашки) от байтове, символи или записи, които представят в последоветилен вид всички структури – вкл. и по-сложни, които се сериализират – (в ОС marshalling/unmarshalling). 2) Филтрите трансформират потока данни – без необходимост да изчакват готовност на целия пакет за разлика от пакетната обработка! Те записват изходните данни в канал, който ги предава на друг асинхронно работещ филтър. Има 2 типа филтри: 1\*) активен филтър – изпълнява операциите pull/push върху пасивни канали – каналите осигуряват съответните операции, а инициативата е на филтъра. В Java PipedWriter и PipedReader класовете предоставят този интерфейс за канали 2\*) пасивен филтър – предоставя push/pull интерфейси на каналите. 3) Каналите преместват, а по същество съхраняват, потока данни, които се

обменят между два филтъра.

Клас-диаграма на СА с филтри и канали *(фиг. 5.14)* – активният модул е с плътни интерфейсни линни. Филтърът е свързан с до 3 класа – източник на данните, консуматор и канал.

(Блокова и последователностна диаграма на ФКСА – *(фиг. 5.15)*. ФКСА се организира лесно в пакетните ОС (напр. в Unix who | wc -l означава пасивен канал между две операции – в случая who генерира списък от потребителите, wc брои думите в списъка

(спрямо стандартни разделители); поддържат се канали с имена, а филтри могат да са

произволни процеси в основен и фонов режим (fore- и background)))

Макар, че управлението е по данни, паралелизма е управление и архитектурата е приложима, когато обработката може да се раздели на асинхронни модули. Реализира се модела производител/консуматор. Не се поддържа динамичен и интерактивен интерфейс – ограничение, което е предимство при дадени приложения. Приложението се ограничава от формата на данните в каналите – обикновено се използва ASCII код.

**3.Контролни СА**: Прилагат се при вградените системи (ВАС) – компютърно контролиране на процеси в реално време с или без човеко-машинен интерфейс. При вградените системи управлението е на база на сканиране на променливи на средата, извличани като поток данни от сензори и управляващо въздействие чрез компютърно контролирани актуатори (напр. автомобилен ABS – *(фиг. 5.16)*).

При КСА процесът се разделя на няколко модула, но те са от 2 типа: 1) контролни модули – за следене и манипулиране на променливите на средата и състоянието. 2) изпълнителски модули – за управление на актуаторите. (Връзките между модулите са чрез поточни данни).

Типове контролни потоци при КСА: 1) контролирани променливи – характеристики на ВАС (сила на ток, налягане и др.), физически контроли на изпълнителните актуатори – те се измерват текущо от сензорите и се съпоставят с контролните константи т.е. целевите стойности. 2) входни променливи според проблемната област (скорост, налягане, температура, влажност, GPS координати).

**Контекстни архитектури (Data Centric)**: характеризират се с централизирано хранилище на данните, които садостъпни за всички компоненти на системата, така че

декомпозицията е на модул за управление на достъпа до данните и агенти, които извършват операции върху тях. Интерфейсът между агентите и данните може да е явен (напр. RMI или RPC) или имплицитен (напр. транзактивен). В чист вид КАрх не предвиждат преки комуникации между информационните агенти – *(фиг. 5.17)*.

Модулът данни изпълнява операции по извличане или регистриране и промяна на записи – по 2 въможни модела: 1) хранилище (repository) (с активни (инициативни) агенти). Хранилището е обикн. Е организирано като СУБД, CORBA, UDDI или Web-услуги. 2) черна дъска (с инициатива на модула данни). Агентите са абонати за събития

(event listeners), които настъпват при промяна в данните и на които абонатите отговарат реактивно (често при AI-разпределени приложения, охранителни системи за разпознаване на звук и образ, системи за управление на бизнес ресурси – складове, транспорт).

**1. Контекстни архитектури с хранилище**: макар и с управление по данни, за разлика от потоковите архитектури за пакетно обслужване на транзакции, тези архитектури поддържат интерактивните UI.

Пример: клас-диаграма на университетска информационна система – *(фиг. 5.18)*.

Класът Collector поддържа вектор на колекция от студентски записи и затова агрегира клас Student, като поддържа UI за извличане, добавяне и промяна на записите за студентите. Класът Student е интерфейс към таблицата на студентите, чиито инстанции представят по един запис (т.е. ред) в нея. Диаграмата на последователността *(фиг.5.19)* представя споделянето на данните чрез класа Student между няколко клиенти.

Релационните СУБД са обичайната платформа за имплементация на тези архитектури, тъй като поддържат свързаност (консистентност) на разпределения достъп до данните, както и множество системни средства за операции, базирани на метаданни.

За по-висока отказоустойчивост и защита на данните се прилагат разпределени хранилища.

Основен недостатък е статичната структура на данните – еволюция в

структурата на релационните таблици се прилага трудно, струва скъпо и

надеждността й се проверява трудно.

**2. Контекстни архитектури с черна дъска**: ориентирани са главно към проблеми, решими с методите на AI – най-вече разпознаване на шаблони в различни области (първите приложения от края на 1970те са експертни системи в метеорология, изображения, звук, молекулярна химия). Те декомпозират решаването на проблеми на два[+] дяла: 1) черна дъска, съхраняваща данни – факти и хипотези т.е. еволюционни модели над фактите 2) източници на знания – паралелно работещи агенти, които съхраняват различни страни (данни, организирани като знания) от проблемната област. Всеки ИЗ капсулира специфичен аспект от проблема и е отговорен за частни хипотези и решения като част от общото решение 3) [контролер – система за начално зареждане и управление на разпределеното приложение].

При контекстните арх. се запазва блок-д-та от *(фиг. 5.20)*, но контролният поток е само от ЧД към ИЗ: Имаме неявни (имплицитни) обръщения към регистрираните в ЧД агенти-източници. Обръщенията възникват при промени в данните и се предават към абонираните за тези промени ИЗ, които изпълняват реактивно заложените в тях логически правила за извод. Този асиметричен механизъм на обмен е известен като модел publish/subscribe (pub/sub) в общите комуникации.

Контекстните арх. с черна дъска се класифицират като слабо-свързана (loosely coupled) РС поради асинхронния комуникационен модел с обмен на публикувани съобщения към абонатите (за разлика от силно свързаните (tightly coupled) системи с хранилища, където транзактивното обслужване е свързано със заключване на данните за конкурентен достъп).

(Клас-диаграма на такава архитектура –> *(фиг. 5.21)*). Класовете-източници KnowledgeSource съхраняват специфичните правила за логически изводи, регистрират се в съответната ЧД, абонират се за оповестяване на промени в данните на ЧД и евентуално генерират реакции с изменения в локалния си или общ (ЧД) контекст; форматът на знанията и правилата за всеки ИЗ може да е специфичен. ЧД управлява общия котекст, регистрира промените в него, оповестява абонатите и регистрира евентуалните реакции, както и съхранява крайното решение. Контролерът инициира ЧД, множеството на ИЗ, инспектира състоянието им и публикува крайното решение.

(Последователностна диаграма на архитектурата –> *(фиг. 5.22)*).

(Блок-диаграма на КАЧД на система за туристически консултации –> *(фиг. 5.23)*).

Обединява множество резервационни агенции – пътни, хотелски, за атракции, за коли под наем, кредитни и т.н. Клиентските заявки се публикуват на ЧД и се оповестяват съответните агенти, чрез реакциите, на които се изготвят един или повече планове за туристическо пътуване и съответното финансиране. Всички операции се инициират по данни, а се поддържа и UI. Типично за КАЧД клиентският интерфейс през контролера е минимален. Примерно еднократен, но интерефейсът за управление на агентите може да е итеративен.

КАЧД е подходяща архитектура за комплексни неизследвани и особено мултидисциплинарни проблеми, които са без детерминистично решение и с представяне на котекста във форматите на AI, както и неподходящи за търсене на решение с пълно обхождане на проблемния домен, поради изчислителната сложност или непълнота/неточности в данните. Може да се генерират оптимално или няколко субоптимални решения или решения на частни подпроблеми. Подходяща е за разпределена обратка с умерена скалируемост поради централизирания контекст. Проблем е еволюцията в структурата на контекста, поради обвързаност с агентите на знания. Отсъствието на междуагентни комуникации води до необходимост от

централизираната им синхронизация (например приоритетна) на достъпа до общия контекст. Трудно се формулира условие за край на обработката, поради недетерминистичния характер на проблемите.