Лекция № 1.1. Същност на ООП. Обектен модел.

Обектно ориентираното програмиране (ООП) е начин на програмиране, в голяма степен аналогичен на процеса на мислене при човека. В неговата основа е понятието обект като някаква структура, описваща нещо от реалния свят, неговото поведение. Задачата, решавана с помощта на ООП се описва в термините на обекти и операции върху тях, а програмата при такъв подход представлява набор от обекти и връзки между тях. С други думи може да се каже, че ООП е метод за програмиране, доста близък до човешкото поведение.

Дефиниция от Wikipedia: *ООП е **програмна парадигма**, при която основни градивни елементи са **обектите**. Те имат полета с **данни** (атрибути, описващи обекта) и свързани с тях **функции** (наричани методи). Изграждането на обектно ориентирана програма представлява проектиране и имплементиране на **взаимодействията** между обектите, които обикновено са инстанции на **класове***.

Необходимо е да се отбележи, че обектният подход е бил известен още на древногръцките философи. Те разглеждали света в термините на обекти и събития. В 17-ти век Р. Декарт отбелязал, че хората обикновено имат обектно ориентиран поглед към света. В 20-ти век тази тема намира отражение във философията на обективистката епистемология.

- **Обектен модел**

Обектно ориентираната технология е изградена на стабилна инженерна основа, чиито елементи взети заедно наричаме **обектен модел на разработка** или просто обектен модел. Обектният модел обхваща принципите на абстракция, капсулиране, модулност, йерархичност, типизираност, едновременност (паралелизим, *concurrency*) и устойчивост (*persistence*). Сам по себе си никой от тези принципи не е нов. Важното за обектния модел е, че тези елементи са обединени по синергетичен начин – действат съгласувано.

ООАД е подход фундаментално различен от традиционния структурен дизайн: той изисква различна гледна точка към декомпозицията и произвежда софтуерни архитектури, които далеч надхвърлят мащабите, характерни за структурния дизайн.

Методите на структурния дизайн еволюират в посока да направляват разработчиците, които се опитват да изграждат сложни системи, използвайки алгоритмите като фундаментални градивни блокове. По подобен начин се развиват и методите за обектно ориентиран дизайн – да помогнат на разработчиците да се възползват от изразителна сила на обектно базираните и обектно ориентираните езици за програмиране, които използват класове и обекти за градивни блокове.

Понятията клас и обект са толкова тясно свързани, че е невъзможно да се възприемат поотделно. Въпреки това е важно да се изясни съществената разлика между тях – докато обектът обозначава конкретна същност, определена във времето и пространството, класът определя само абстракция на съществената в обекта. Всеки обект е екземпляр на някой клас, а всеки клас може да поражда неограничен брой обекти.

ООП – дефиниция: *„ООП е метод на имплементиране, при който програмите са организирани като множества от сътрудничещи си обекти, всеки от които е инстанция на даден клас, а класовете са свързани от наследствени отношения в йерархия от класове.“*[1]

В тази дефиниция има 3 важни момента:

(1) ООП използва обекти, а не алгоритми като фундаментални логически градивни блокове;

(2) всеки обект е инстанция на някой клас;

(3) класовете могат да бъдат свързани помежду си от наследствени отношения.

[1] Grady Booch, Robert A. Maksimchuk, Michael W. Engel and Bobbi J. Young, „Object-Oriented Analysis and Design with Applications (3rd Edition)“, Addison-Wesley Professional, 2007, p. 41

Ако кой да е от тези елементи липсва, програмата не е обектно ориентирана. В частност, програмиране без наследяване не е ООП. То може да се определи като програмиране с абстрактни типове данни (АТД) или обектно базирано програмиране.

Елементи на обектния модел

Обектният модел има 4 основни елемента:

1. Абстракция (*abstraction*).

2. Капсулиране (*encapsulation*).

3. Модулност (*modularity*).

4. Йерархичност (*hierarchy*).

Тези основни елементи задължително трябва да присъстват, за да бъде моделът обектен.

Има и три допълнителни елемента:

5. Типизиране (*typing*).

6. Едновременност (паралелизъм, *concurrency*).

7. Устойчивост (запазване, постоянство, *persistence*).

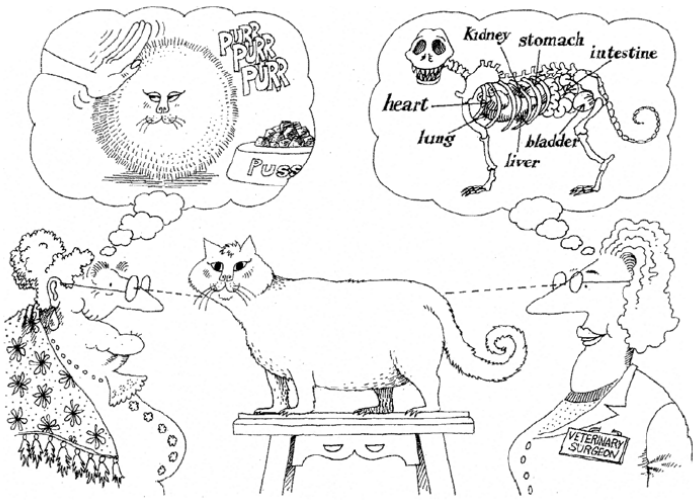
„Допълнителни“ означава, че всеки от тези елементи е полезен, но не е задължителна част от обектния модел.

Абстракция:

Абстракцията е един от основните инструменти на хората за справяне със сложността.

„Абстракцията произтича от разпознаването на приликите между някои обекти, ситуации или процеси в реалния свят и решението за момента да се концентрираме върху тези прилики и да игнорираме разликите.“

Дефиниция: *„Абстракцията обозначава съществените характеристики на даден обект, които го отличават от всички други видове обекти и по този начин осигурява отчетливо дефинирани концептуални граници, свързани с гледната точка на наблюдателя*“ – Grady Booch.



Фиг. 1 Различни гледни точки

Изборът на точния набор от абстракции за дадена проблемна област е централен проблем на ООД.

Можем да разделим абстракциите на следните видове:

- **абстракции на реални обекти** (*entity abstraction*): обект, който представлява полезен модел на реален обект от проблемната област или от областта на решението;
- **абстракции на действия** (*action abstraction*): обект, който предоставя обобщено множество от операции, всички изпълняващи един и същи вид функция;
- **виртуална машина** (*virtual machine abstraction*): обект, който групира операциите, използвани от някое по-горно ниво на контрол или операциите, използващи някое по-ниско ниво;
- **случайна абстракция** (*coincidental abstraction*): обект, който пакетира множество от операции без връзки помежду им.

Стремим се да изграждаме абстракции на реални обекти, защото те правят директен паралел с речника на проблемната област.

Всички абстракции имат както статични, така и динамични характеристики. Например, обект „файл“ заема точно определено количество място на дадено запомнящо устройство. Той има име и съдържание. Всички тези са статични характеристики. Стойностите на всяка от тези характеристики обаче е динамична величина и се променя по време на живота на обекта – файлът може да нараства или да намалява по големина; могат да се променят неговото име и съдържание.

Капсулиране

Абстракцията на даден обект трябва да предхожда решенията относно нейната имплементация. Щом веднъж бъде избрана имплементация, тя трябва да се третира като тайна на абстракцията и трябва да бъде скрита от повечето клиенти.

Абстракцията и капсулирането са допълващи се понятия: абстракцията фокусира върху видимото поведение на обекта, докато капсулирането фокусира върху имплементацията, която осигурява това поведение. Капсулиране се постига най-често чрез скриване на информация (*information hiding*). Тук ще спомена, че скриване на информация не е само скриване на данни (*data hiding*). Скриването на информация е процесът на скриване на всички тайни на обекта, които не допринасят за основните му характеристики. Обикновено скрита остава структурата на обекта и имплементацията на неговите методи.

Капсулирането позволява да са правят по-лесно промени в програмата, без това да засяга нейната надеждност.

Капсулирането осигурява ясно разграничаване между различни абстракции и по този начин води до ясно **разделение на отговорностите** им (*separation of concerns*).

За да работи абстракцията, имплементациите трябва да са капсулирани. На практика това означава, че всеки клас трябва да има две части: интерфейс и имплементация. **Интерфейсът** на класа обхваща само външния изглед, поведението на абстракцията, общо за всички инстанции на класа. **Имплементацията** на класа включва както представянето на абстракцията, така и механизмите с които се постига желаното поведение.

Накрая до обобщим:

Дефиниция: *Капсулирането е процесът на разделяне на елементите на абстракцията, които представляват нейната структура и нейното поведение. Капсулирането служи за отделяне на уговорения интерфейс на абстракцията и неговата имплементация*.

Модулност

Разделянето на програмата на индивидуални компоненти може да намали в известна степен нейната сложност, но по-обосновано е разделянето да се прилага с цел създаването на няколко добре дефинирани и документирани части в програмата. Тези части (или интерфейси) са неоценим помощник за разбирането на програмата.

В някои езици, например Smalltalk, няма понятие „модул“, така че класовете са единствената физическа форма на декомпозиция. В Java има пакети, които съдържат класове. В много други езици, включително Object Pascal, C++ и Ada, модулите са отделна езикова конструкция и това дава основания за отделен набор от дизайнерски решения. В тези езици класовете и обектите от логическата структура на системата се разполагат в модули, които създават физическата архитектура. Специално за големи приложения, в които можем да имаме стотици класове, използването на модули е съществен елемент за управлението на сложността.

Повечето езици, които поддържат модули като отделни концепции разграничават интерфейса от имплементацията на модула. Така може да се каже, че модулността и капсулирането вървят ръка за ръка.

Разработчикът трябва да балансира две конкуриращи се технически съображения: желанието да се капсулират абстракциите и необходимостта да се направят някои абстракции видими за други модули. Детайли на системата, за които е вероятно да се променят независимо трябва да останат тайни на отделните модули. Единствено елементи, за които е малко вероятно да бъдат променяни трябва да се появяват между модулите. Всяка структура от данни е частна за един модул. Тя може да бъде директно достъпвана от една или повече програми в същия модул, но не и от външни за модула програми. Всяка друга програма, която изисква информация, съхранена в структурите от данни на модула, трябва да я получи чрез извикването на програми (функции от интерфейса) на модула. С други думи, трябва да се стремим да създаваме модули, които са кохезивни (сплотени), чрез групиране на логически свързани абстракции и слабо свързани (*loosely coupled*), чрез минимизиране на зависимостите между модулите. От тази перспектива можем да дефинираме модулността така:

Дефиниция: *Модулността е свойство на система, която е декомпозирана на множество от кохезивни и слабо свързани модули*.

Така принципите на абстракция, капсулиране и модулност са синергични (действат съвместно). Обектът осигурява ясните очертания на единична абстракция, а капсулирането и модулността осигуряват бариерите около тази абстракция.

Йерархичност

Абстракцията е хубаво нещо, но във всички случаи, само с изключение на най-тривиалните приложения, можем да открием толкова много различни абстракции, че да не можем да ги възприемем и обхванем мислено по едно и също време. Капсулацията помага да се управлява тази сложност чрез скриването на вътрешността на абстракциите ни. Модулността също помага като ни дава възможност да клъстеризираме логически свързаните абстракции. Но това все още не е достатъчно. Множество от абстракции често формират йерархия и чрез идентифицирането на тези йерархии в нашия дизайн можем чувствително да опростим разбирането на проблема.

Можем да дефинираме йерархичността така:

Дефиниция: *Йерархията е ранкиране и подреждане на абстракциите*.

Двете най-важни йерархии в една сложна система са нейната структура от класове (“*is a*” йерархията) и структурата на обектите (“*part of*” йерархията).

Наследяването е най-важната “*is a*” йерархия и е съществен елемент от обектно ориентираните системи. Наследяването дефинира връзка между класове, където един клас споделя структура или поведение, дефинирано в един или няколко други класа. По този начин наследяването представя йерархия от абстракции, в която подклас (*subclass*) наследява от един или повече суперкласове (*superclasses*). Обикновено подкласът допълва или предефинира съществуващата структура и поведение на своите суперкласове.

В семантично отношение наследяването обозначава “*is a*” отношение. Например, мечката “е” вид бозайник; къщата “е” вид недвижим имот; quick sort “е” вид алгоритъм за сортиране. Вижда се, че наследяването предполага йерархия от вида обобщение/специализация (generalization/specialization), при което подкласът специализира по-общата структура или

поведение на своите суперкласове. Това може да се използва като лакмус за откриване на наследяване: Ако B не е вид A, то B не трябва да наследява от A.

Суперкласовете представят обобщени абстракции, а подкласовете представят специализации, в които са добавени, модифицирани или дори скрити полета с данни и методи от суперкласовете. Това ни позволява да опишем нашите абстракции с „икономичност на изразяване“ (*economy of expression*).

Докато “*is a*” йерархиите обозначават отношения на обобщаване/специализация, “*part of*” йерархиите описват отношения на **агрегация** (*aggregation*). Например цветята са част от градината. Агрегацията не е уникална за ООП концепция. Всеки език, който поддържа структури „запис“ поддържа агрегация. Комбинацията от наследяване и агрегация обаче дава силата на ООП – агрегацията позволява физическо групиране на логически свързани структури, а наследяването позволява тези общи групи да бъдат лесно повторно използваеми в различните абстракции.

Типизиране

Понятието „тип“ произлиза от теорията на абстрактните типове данни (АТД). Типът е точно описание на структурни и поведенчески свойства, които споделят група обекти (*entities*). За нашите цели можем да използваме термините „тип“ и „клас“ взаимнозаменяемо. Въпреки, че понятията са близки, включваме типизирането като отделен елемент на обектния модел, защото понятието „тип“ поставя много различен акцент върху значението на абстракцията. Можем да каже, че:

Дефиниция: *Типизирането е принуда (enforcement) на класа върху обекта, така че обекти от различни типове да не бъдат заменяни или поне тази замяна да може да става само по много рестриктивен начин*.

Езиците за програмиране могат да бъдат силно типизирани (*strongly typed*), слабо типизирани (*weakly typed*) и дори нетипизирани (*untyped*) и пак да са обектно ориентирани.

Идеята за съответствието (*conformance*) е централна при нотацията в типизирането. Ще дадем пример с мерните единици във физиката:

Когато делим разстояние на време очакваме стойност, означаваща скорост, а не тегло. Друг пример, ако делим сила на температура няма да получим смислен резултат, докато деленето на сила върху маса има смисъл. Това са примери на строго типизиране, където правилата на предметната област определят точните легални комбинации от абстракции.

Строгото типизиране ни позволява да използваме езика за програмиране, така че да форсира определени решения за дизайна, което има смисъл с нарастването на сложността на системата. Строгото типизиране си има и слаби черти. В частност, то въвежда семантични зависимости и дори малки промени в интерфейса на базов клас изискват прекомпилиране на всички подкласове.

Понятията строго и слабо типизиране са коренно различни от понятията статично и динамично типизиране. Силното и слабо типизиране се отнасят за консистенцията, докато статичното и динамично типизиране касаят времето, когато имената се свързват с типовете.

Статично типизиране (*static typing*, *static binding*, *early binding*) означава, че типовете на всички променливи и изрази са фиксирани по време на компилацията.

Динамично типизиране (*dynamic typing*, *late binding*) означава, че типовете на променливи и изрази не са известни преди стартирането на програмата (*runtime*). Един език може да бъде едновременно статично и строго типизиран (Ada), строго типизиран, но поддържащ динамично типизиране (C++, Java) или нетипизиран и поддържащ динамично типизиране (Smalltalk).

Полиморфизмът (*polymorphism*) е състояние, което съществува, когато си взаимодействат възможностите на динамичното типизиране и наследяването. Това е понятие от теорията на типовете и означава едно име (например декларация на променлива) да може да обозначава обекти на няколко различни класове, които са свързани чрез някой общ суперклас. Всеки обект, обозначен с това име, е в състояние да отговаря на някое общо множество от операции. Обратното

на полиморфизма понятие се нарича мономорфизъм (*monomorphism*) и е характерен за всички езици, които са едновременно строго и статично типизирани.

Полиморфизмът е може би най-мощният механизъм на обектно ориентираните езици, наред с абстракцията. Той е механизмът, който отличава ООП от традиционното програмиране с АТД.

Паралелизъм (едновременност, *concurrency*)

За определени задачи на автоматизираната система може да се наложи да обработва много различни събития едновременно. Други задачи пък може да изискват толкова много изчисления, че да надвишават капацитета на отделен процесор. Във всеки от тези случаи е естествено да се обмисли възможността от използване на разпределена (*distributed*) мрежа от компютри или от многозадачност (*multitasking*).

Всяка програма има поне една нишка (*thread*) на контрол, но система, включваща едновременност може да има няколко такива нишки – някои от тях са временни, а други съществуват през цялото време на работа на системата.

На високо ниво на абстракция ООП може да облекчи проблема с едновременността чрез скриването ѝ в повторно използваеми абстракции. Обектният модел е подходящ за разпределени системи, защото той имплицитно дефинира (1) единиците на разпределение и преместване и (2) обектите, които комуникират.

Докато ООП фокусира на абстракцията на данни, капсулирането и наследяването, едновременността фокусира върху абстракцията на процеси и синхронизацията. Обектът е понятие, което обединява тези две гледни точки: всеки обект (взет от абстракция на реалния свят) може да представя отделна нишка на контрол (абстракция на процес). Такива обекти се наричат **активни**. В система, базирана на ООД можем да концептуализираме света като множество от сътрудничещи си обекти, някои от които са активни и за това служат за центрове на независима активност. Като имаме това предвид, можем да дефинираме едновременността така:

Дефиниция: *Едновременността (concurrency) е характеристика, която разграничава активен обект от такъв, който не е активен.*

Независимо как ще е реализирана едновременността, факт е, че щом веднъж бъде въведена в системата, трябва да се грижим как активните обекти синхронизират своите действия, както помежду си, така и с изцяло последователни (*sequential*) обекти. Ако например два активни обекта опитват да изпратят съобщения да трети обект, трябва да сме сигурни, че използваме някои средства за взаимно изключване (*mutual exclusion*), така че състоянието на обекта, с който се работи, да не се повреди, когато и двата активни обекта се опитат да го променят едновременно. Това е мястото, където идеите на абстракция, капсулиране, и едновременност си взаимодействат. При наличието на едновременност не е достатъчно просто да се дефинират методите на даден обект. Ние също трябва да сме сигурни, че семантиката на тези методи се запазва в присъствието на множество нишки за контрол.

Устойчивост (запазване, *persistence*)

Става въпрос за континуум на живота на обекта. Най-популярните такива състояния са временни резултати при изчисляването на изрази, локални променливи при изпълнение на функции, собствени за езика променливи, глобални променливи и др. Тук се намесват и обектно ориентираните бази от данни и сега няма да се спираме подробно.

Дефиниция: *Устойчивостта е свойство на обекта, чрез което съществуването му надхвърля времето (т.е, обектът продължава да съществува след като неговият създател се отказва от него) и/или пространството (т.е., местоположението на обекта се премества от адресното пространство, в което е бил създаден)*.

Тоест, устойчивостта е свойство на обекта, чрез което той разширява времето и пространството на своето съществуване.

Предимства на обектния модел

1. Използването на обектния модел ни помага да се възползваме от изразителна сила на обектно ориентираните и обектно ориентираните езици за програмиране.
2. Насърчава повторното използване не само на софтуера, но и на цели дизайни, което води до създаването на повторно използваеми рамки за приложения (*application frameworks*). Това намалява както количеството код, така и разходите по създаването и поддръжката на софтуера.
3. Лесна еволюция на софтуерните системи.
4. Намалява рисковете, присъщите на разработката на сложни системи.
5. Обектният модел наподобява работата на човешкото съзнание и изглежда съвсем естествен дори за хора, които не са навътре със софтуерните технологии.