# РАЗВИТИЕ НА ПРОГРАМИРАНЕТО. ПРЕДПОСТАВКИ, ВЪЗНИКВАНЕ И ОСНОВНИ ПОНЯТИЯ НА ОБЕКТНО ОРИЕНТИРАНОТО ПРОГРАМИРАНЕ

*Ивайло Дончев*

**Въведение**

И до днес е широко разпространена методиката на преподаване на програмиране, предлагаща от начало усвояване на структурния подход и чак след това да се прави преход към изучаване на други парадигми в програмирането: функционално, логическо и обектно ориентирано. Такава методика се е оформила исторически (8). Тя има определени предимства и недостатъци пред другите методики. Един от забележимите обективни проблеми е трудността за обучаваните да направят прехода от станалата вече привична структурна парадигма в програмирането към друга, особено към обектно ориентираната (6). Ако във функционалната и логическата парадигми се използват специфични програмни езици, в една или друга степен изискващи промяна в мисленето на програмиста, то всички разпространени работещи езици за ООП в своята основа имат езикови конструкции от другите парадигми, най-вече от структурната. Езиковите конструкции на ООП са като надстройка над старите езикови конструкции. Това обстоятелство позволява да се пише на обектно ориентиран език, използвайки старите подходи към програмирането. У начинаещия обектен програмист е голямо изкушението да разработи програмата в обичайните за него структурни понятия. Критерии за самооценка на свършената работа са коректността, устойчивостта и бързодействието на написаната програма. Като резултат се получава, че в обучавания се формира недоразбиране на обектно ориентирания подход, възникват трудности с възприемането на новите идеи в ООП, изгражда се основата на бъдещите проблеми с проектирането на крупни програмни системи, а това е проблем на професионалното израстване на обучавания.

Затова на студентите, изучавали структурно и процедурно програмиране трябва да се обърне внимание на различията между традиционните и съвременните възгледи за приложния софтуер, да се покаже ролята на обектно ориентирания подход в програмирането, историческата необходимост от ООП в разработката на големи програмни системи.

## 1. Понятието „компютърна програма” в процеса на развитие на идеите и парадигмите за програмиране

Един от ключовите стереотипи на мисленето на програмиста е смисълът на понятието “компютърна програма”, отговорът на въпроса “какво е програма?”. На практика програмистите често не се замислят над разкриването на съдържанието на тази категория. Тя им се струва очевидна. Но, както ще покажем по-долу, съществуват принципно различни подходи към понятието програма. Неразбирането на съществуването на такива различия води до недоразумения и грешки при проектирането на приложен софтуер.

Ще отбележим основните гледни точки по въпроса за понятието “програма” и ще покажем принципната разлика между програмиране по структурна и по обектно ориентирана парадигма. Исторически първото централно понятие в програмирането е бил **алгоритъмът** – строга последователност от операции, в процеса на изпълнение на които входните данни могат да бъдат преобразувани в резултат. Програмата в съзнанието на програмиста се оформя като реализация на някакъв алгоритъм. Такава гледна точка е най-популярна до края на 1980-те години. След появата на компютри, съхраняващи програмата в своята памет и след поставянето на задачата за разработка на езици за програмиране от високо ниво, се появява възможността за прилагане в програмирането на теорията на формалната логика, например ламбда-изчисленията на Чърч и изчисляването на предикати (10). Така последователно се появяват езици за програмиране с използването на функционален и логически подход. Функционалният подход изисква запис на програмата във вид на композиция на функции. При използването на логическия подход програмистът записва множество факти и правила на логически извод. Принципното различие на функционалните и логическите програми от програмите, реализиращи алгоритми се заключава в това, **какво** тези програми описват. Програмите, реализиращи алгоритми, **определят** какви действия да извърши компютърът и затова се наричат *императивни*. Те “обясняват” на компютъра **как** да извърши изчисленията, при това **какво** се изчислява съставлява входните данни и резултата. Функционалните и логическите програми **описват** някакви специални логически системи, обикновено построени в рамките на някакви предметни области и затова се наричат *декларативни*. Те “обясняват” на компютъра **кое** е предмет на изчислението. При това компютърът отпреди “знае” **как** да изпълни изчисленията.

Императивните програми пряко се свеждат до последователности от машинни инструкции и се изпълняват от компютъра. Декларативните програми се поддържат от програма-интерпретатор, която може да изпълнява операцията редукция над елементите на програмната библиотека. Така се осъществява изчислението: опитвайки се да приведе изходните описания към някакъв логически коректен в рамките на описаната логическа система резултат.

От периода на масовото разпространение на персонални компютри през 1980-те години, когато работата със сложна изчислителна техника започват да усвояват широки слоеве от служители в най-различни области на човешката дейност, все по-остра става потребността от някакво друго централно понятие в програмирането. Преди компютърът се е използвал основно като инструмент за решаване на изчислителни задачи, бил е скъпо и сложно устройство, с което са работили само специалисти по изчислителна техника. Малко внимание се е отделяло на въпросите за взаимодействието между човека и машината. С течение на времето приложението на компютъра се разширява. Той се превръща в универсално информационно устройство, можещо не само да изчислява, но и нагледно да въвежда, преобразува, съхранява, търси и визуализира най-разнообразна информация, устройство, с което може да работи всеки човек. Компютърът престава да се възприема в качеството му на обикновен **изчислител** и се превръща в **среда за решаване** на различни, в общия случай неизчислителни приложни задачи. Заедно с промяната на ролята на компютрите в живота и дейността на хората започва да се променя и съдържанието на категорията „компютърни програми". Задачите по организиране на просто и удобно взаимодействие човек – машина, разработката на шелове ("обвивки") на операционните системи и прилагането на графичния интерфейс, създаването на многомашинни изчислителни комплекси пораждат редица трудности в програмирането. Старите стереотипи на мислене вече са неприложими. В процеса на решаване на възникналите трудности в 1990-те години протича масов преход към обектно ориентиран подход.

В началото на 1980-те години в Япония е започнат проект за разработване на ЕИМ от 5-то поколение, предполагащо широко приложение на изкуствения интелект: повсеместно използване на бази от знания, експертни системи, общуване с компютъра на естествен език. Като основна парадигма на програмирането е избран логическият подход, получил развитие в рамките на този проект. Но в процеса на реализация

на този проект очакваните резултати не са постигнати – най-вече заради "задънената улица" в областта на изкуствения интелект.

Най-приемливо решение на натрупалите се проблеми се оказава използването на новото понятие *информационен модел* – множество обекти (структури от данни и правила за тяхното преобразуване), които манипулира потребителят. Информационният модел описва физическа или социална реалност или се явява алтернативно представяне на някакъв друг информационен модел. Информационният модел предоставя на потребителя *среда от възможности*, в която той сам формира и сам решава възникващите пред него задачи. Понятието информационен модел изпълва новият смисъл на категорията компютърна програма.

Между алгоритъм и информационен модел съществува определена взаимовръзка. От една страна, досега на физическо ниво компютърът си остава същият изпълнител на елементарни инструкции, какъвто е той от момента на изобретяването си. Съответно, произволна програма за компютър, която ще бъде сведена към машинен език, може да бъде записана във вид на алгоритъм, съставен от достъпни за изпълнение машинни инструкции, макар че на практика написването на ръка на сложна съвременна програма на машинен език е много трудоемка задача. От друга страна, решаването на задачи от потребителя в рамките на някой информационен модел въпреки всичко оставя потребност от преобразуване на данните по определени правила. Такива преобразувания често изискват съставяне на техни алгоритми. В същото време информационният модел не е някъв усъвършенстван метод за записване на алгоритми. Той има същата декларативна природа, както и функционалните и логически програми. По-точно информационният модел съвместява в себе си декларативността на описанието на елементите на модела – обектите и императивността на описанието на операциите, изпълнявани над обектите – обектните методи.

Може да се открие известно сходство на обектите и методите в информационния модел с предложените от Мински (7) фреймове и техните демони, използвани за описание на знания. Информационните модели описват нашите знания в различни предметни области. Тези знания са персонифицирани в особеностите на структурите и поведението на моделираните системи. Информационните модели имат същата рекурсивна структура, както и фреймовите мрежи. Това позволява моделът на някоя система да се конструира от моделите на нейни подсистеми чак до елементарни обекти.

През 1990-те години се измени мисленето на много програмисти, появи се нова визия за проблемите и, съответно, принципно нови постановки на задачи в програмирането. Конкретният алгоритъм представлява процес на решаване на някаква конкретна задача. Бидейки записан във вид на програма, той фиксира и гледната точка върху проблема, и постановката за решение на проблема, и избрания метод на решение. След написването на програмата нейният потребител не може да направи нищо, освен да предаде на компютъра входните данни и да получи от него резултата от изчисленията. В този случай се казва, че **алгоритъмът управлява данните**. При работата с информационен модел на някоя проблемна област потребителят е ограничен само от гледната точка, към която се е придържал съставителят на модела. Както и при логическите програми постановката на задачата може да се избира от потребителя по негово усмотрение. За разлика от логическите програми потребителят на информационния модел е свободен да избере също и методите за решение на поставените задачи. Обикновено работата по конструиране на решението на задачите в рамките на информационния модел протича в процеса на диалог с машината. Например, в текстов процесор може в **произволен** ред да се изпълняват всички допустими функции по обработка на документа: вмъкване и замяна на текст, блокови операции, стилово оформление, форматиране на таблици и др. , например, в текстовия процесор MS Word всички изпълнени от потребителя операции могат да бъдат съхранени в макрос – те ще се запишат като програма, изпълнението на която може да се повтори в бъдеще. От гледна точка на потребителя се изпълнява някаква работа, заключаваща се в манипулация на частите на информационния модел за получаване на определен резултат (например, подготовка на документ). Всички манипулации се извършват от потребителя на понятен за него език – езикът на предметната област. В същото време от гледна точка на компютърната програма всички тези действия представляват алгоритъм за решение на задачата от потребителя. Този алгоритъм се формира динамично в процеса на диалог с него, той не е заложен предварително в програмата. Навярно този алгоритъм няма да бъде ефективен, ще съдържа различни излишни действия, всички етапи на обмисляне, съмнения и колебания на ползвателя в процеса на изработване на решението. Но все пак той си остава алгоритъм – един от възможните способи за получаване на желания от потребителя резултат. Новите компютърни програми се състоят не от готови алгоритми, а от възможности за конструи-

рането им или от ползвателя, или от друг програмист. В този случай казваме, че **данните управляват алгоритъма**.

Изхождайки от описаното различие може да се формулира разликата в процеса на разработка на програми в миналото и сега. В миналото, във времето на царстване на структурната парадигма за програмиране, разработчикът е имал на входа постановка на конкретна задача (често от изчислителен характер) и е построявал решение на тази задача по пътя на разбиването и на по-малки подзадачи до достигане на прости операции на компютъра. Така в крайна сметка се е получавало решението на изходната задача. Естествено изискване за такъв процес се явява яснотата и неизменността на постановката на задачата. Всички останали действия, както и подготовката на входните данни и интерпретацията на резултата се извършват от потребителя, извън компютъра. В настоящето, при използването на обектно ориентираната парадигма, разработчикът създава такъв информационен модел и изчислителна среда, в която изходната задача лесно може да се разширява или от самия разработчик, или от потребителите на програмата по пътя на манипулирането на обектите от създадения информационен модел. В резултат се получава набор от възможности, приложими както за решението на изходната задача, така и за решаване на други задачи от общата предметна област. При такъв подход не се изисква яснота и непроменяемост на постановката на конкретната задача. Достатъчно е само изискването за неизменимост на модела. Всички действия в допълнение към решението на задачата потребителят може да изпълнява на компютъра в рамките на същия информационен модел, в който е планирал да реши първоначалната си задача: получава възможност да изследва проблема, да експериментира с данните, да формулира постановката на задачата в процеса на диалог с машината, да избира методи за решението.

Би било неправилно да се твърди, че описаният по-горе нов подход към програмирането е възможен единствено в рамките на ООП. Отделните елементи на този подход са възможни и при използването на други парадигми на програмирането. В края на краищата всяка програма за съвременен компютър се свежда до машинни инструкции, към някакъв неизвестен от по-рано алгоритъм. Но определено това, че **разбирането** на новия подход към програмирането се разпространи широко се дължи именно на развитието на ООП. Основните понятия на тази парадигма определят новия начин на мислене на програмиста. Използването на обектно ориентиран подход позволи да се решат редица проблеми на

индустрията за разработка на програмно осигуряване. Преди много програми се създаваха "от нулата". Повторното използване на библиотеки подпрограми се затрудняване от еднозначността на записаните в тях алгоритми, понякога дори се използват само еднократно. Такъв подход е скъпо струващ. Много от създадените вече разработки не могат да се прилагат в нови проекти. Методиката на ООП позволява да се създават достатъчно универсални библиотеки класове от обекти, че да могат многократно и по различен начин да се използват при решението на много задачи. Това води до снижаване на себестойността на разработката, ускоряване излизането на пазара на нови програмни продукти, създаване на по-сложни и устойчиви програмни системи и изобщо бурно развитие на отрасъла.

**2. Категории в качеството на програмното осигуряване. Възникване и развитие на основните понятия в ООП**

С изменението на понятието компютърна програма все повече и повече се изпълват със съдържание категориите, отнасящи се към качествените свойства на програмите.

**2.1. Сложност**

Програмирането, както и много други видове дейности, насочени към създаване на нови неща, има ограничения, определяни от работната среда. При програмирането тези ограничения са били свързани основно с физическите възможности на машините и понятието **сложност** на програмирането се е отнасяло по-скоро към експлоатацията на машините, а не толкова към вътрешния смисъл на програмата. Но след много кратко време, когато развитието на техниката съществено надскочи физическите ограничения и програмистите вече могат да създават програми с по-голям размер, а в наше време – големи програмни системи, се разразява "криза на програмирането", перманентно продължаваща и до наши дни (3) (5).

Под криза на програмирането обикновено се разбира такова състояние, при което програмистът или организиран колектив от програмисти, без да има съществени ограничения на машините, не може да създаде за разумно време не съдържаща грешки програма, решаваща голяма и сложна задача. Причините за кризата са няколко и една от тях е сложността на създаваната програма.

Сложността на решаваната задача е субективно усещане на човека. Обикновено това усещане възниква тогава, когато човек не знае начин за решение на задачата. Колкото по-малко очевиден е начинът за решение, толкова по-сложна изглежда задачата. Категорията сложност в

програмирането може да се разглежда от две позиции – теоретическа и инженерна. От гледна точка на теорията сложността е свързана с търсенето на ефективен алгоритъм за решението на задачата и построяването на адекватни информационни модели. Теоретическата сложност се отнася непосредствено към процеса на програмиране. Сложността от гледна точка на инженерната работа обикновено се свързва с количеството елементи в програмата и способността на програмиста да задържа в мозъка си тези елементи и връзките между тях. Ще разгледаме сложността само от инженерна гледна точка. Под елементи на програмата ще разбираме нейни части, атомарни на определено ниво на абстракция.

В първите езици за програмиране от средно ниво целите програми представляват линейни последователности от оператори. Пределният размер на такива програми от гледна точка на сложност е от порядъка на 1000 реда. Да се работи ефективно (да се модифицират и разширяват) с програми с по-голям размер, реализирани по "линейния" подход е практически невъзможно, защото програмистът не е в състояние да запомни всички използвани променливи, а логическите части на програмата могат да се търсят само по коментарите към кода или по номерата на редовете.

Когато отрасълът се сблъсква със задачи, изискващи разработване на по-крупни програми се прилага известният от древността метод за управление на сложността "разделяй и владей". Преди линейно записваните оператори започват да организират в йерархия от обръщения към процедури и функции. Появява се понятието област на видимост на променливите. Променливите се разделят на глобални и локални, организират се в структури от данни – **структурен подход** към програмирането. Сега програмистът може да мисли решението на задачата като йерархия от сравнително кратки алгоритми, състоящи се от обръщения към процедури и функции, съдържанието на които се загатва от заглавията им. Благодарение на такова инженерно решение границата на сложност на програмите се отмества с два порядъка и се оценява на 100'000 реда код на език от високо ниво.

Независимо от факта, че извикванията на процедури и функции са йерархически организирани, самите процедури и функции в много езици за структурно програмиране представляват еднорангови множества и на по-високо ниво на абстракция могат да се разглеждат като елементи на програмата. Когато възниква потребността да се създават програми,

изискващи написването на хиляди процедури и функции, които човешкият ум не е в състояние да обхване в цялост, кризата в програмирането минава на следващ етап – отново се изисква по някакъв начин да се организират програмните елементи. Развитие на структурния подход става **модулният подход**, при който процедурите и функциите на големите програми се групират в модули, пакети или библиотеки, често разработвани от различни програмисти. Границата на сложност на програмите се увеличава и в момента оценката за нея е неизвестна, защото по редица причини единни програмни решения с размер над няколко милиона реда не се създават – вместо такива решения се създават програмни комплекси.

Принципите на декомпозиция на програмния код на модули могат да бъдат най-различни. Един такъв принцип е функционалната декомпозиция – всеки модул на програмата решава отделна задача или съдържа набор процедури, функции и данни, които в съвкупност предоставят някаква услуга. Благодарение на функционалната декомпозиция на програмната система на модули, отделният модул може да се счита за програмен елемент на по-високо ниво на абстракция. Интересен ефект се наблюдава в програмните системи, работещи като множество паралелно изпълнявани процеси. Във всеки процес модулът може да има независими стойности за глобалните променливи, може независимо да се инициализира. Фактически в програмната система може да съществуват няколко автономни един от друг екземпляра на един и същ програмен модул. Ако доведем принципа на модулността до логически завършек, всеки минимално възможен функционален елемент на програмата да се реализира като отделен модул, а също така да се позволява на модулите в рамките на един процес да имат няколко автономни екземпляра, ние достигаме до тъждественост между понятието модул и **понятието клас в обектно ориентираното програмиране**. Екземплярите на класа – обектите са атомарни програмни елементи на системата, създадени с помощта на обектно ориентирания подход. Всеки клас съдържа в себе си структура от данни, а също и процедури и функции, логически свързани с функционалното предназначение на класа. Модулите обикновено имат две части – интерфейс и реализация. Процедурите, функциите и структурите от данни, описани в интерфейсната част са достъпни за други модули, а в частта на реализацията могат да се намират процедури, функции и данни, достъпни само в рамките на модула. Така в модулното програмиране се реализира **принципът на капсулиране**, пренесен в последствие в обектно ориентирания подход. Благодарение на принципа

на капсулация към всеки модул или обект на програмата съществуват две гледни точки – отвън и отвътре.

Отвън обектът изглежда като „черна кутия" с известни входове и изходи. Освен това програмистът знае функционалното предназначение на обекта, което обикновено е загатнато и от названието на класа обекти. Тези сведения са напълно достатъчни, за да може програмистът да използва модула или обекта. Взаимодействието на обекта с обкръжаващата го среда протича чрез интерфейса. Едни обекти играят ролята на клиенти, „поръчват" някакви услуги, изпълнението на които е поверено на други обекти – сървъри.

Във вътрешността си обектът има някаква реализация на своя интерфейс. Когато програмистът разработва нов обект, той разглежда обектния интерфейс като множество възложени на обекта задължения да изпълнява услуги или да решава задачи, необходими на външния за обекта свят. Как именно ще бъдат изпълнени тези задачи за външния свят не е толкова важно (макар от гледна точка на цялата система това да е важен въпрос). Всеки обект или модул в своята реализация може да се обръща към други обекти или модули, но благодарение на принципа на капсулиране тази информация остава скрита от клиентските обекти.

Описаната организация на програмния код във вид на йерархия от модули или система от обекти към дадения момент е достатъчно удобно решение за изграждането на програмни системи с произволни размери. Всяка задача вътре в програмната система се решава чрез организиране на отговорна за решението на задачата подсистема, състояща се от неголям брой програмни обекти. Благодарение на свойството капсулиране програмистът може безопасно да се абстрахира от вътрешното устройство на обектите, концентрирайки вниманието си изцяло върху решаваната от него задача.

В днешно време за големите програмни системи проблемът за сложността преминава от сферата на програмирането към сферата на проектирането (моделирането) (1). Именно на етапа на проектирането е необходимо да се премахнат противоречията в изискванията към програмната система.

**2.2. Универсалност**

Другата важна категория, отнасяща се към качествените свойства на програмите е **универсалността**. Под степен на универсалност на програмното осигуряване разбираме степента на неговата приложимост в различните изчислителни среди за решаването на различни задачи. Разработката на първите програми за първите компютри обикновено се

характеризират с латинския израз „ad hoc", обозначаващ в програмирането пълна привързаност на решението на задачата към конкретната изчислителна машина (или както в наши дни е прието да се казва – към конкретната платформа). Подобен метод на разработка има своите положителни и отрицателни страни. От една страна, програмите, разработени с помощта на такъв подход най-пълноценно използват възможностите на машината (или платформата), за която са били написани и най-икономично използват изчислителни ресурси. От друга страна, в случай на смяна на платформата, се налага да се пренаписват всички вече създадени решения, което е доста трудоемко. По такъв начин „ad hoc" програмирането притежава най-ниска степен на универсалност, в сравнение с другите методи.

С увеличаването на възможностите на изчислителните машини изискването за най-ефективно използване на възможностите на машината или платформата за решаването на задачи от необходимо става желателно. Също така става възможно понижаването на разходите за труд по разработването на програмно осигуряване за сметка на преносимостта на програмите между машини и платформи, за което допринасят унификацията на машините и удачните архитектурни решения. Появява се програмно осигуряване, работещо не по оптимален начин на всяка конкретна машина, но за сметка на това работещо с минимални модификации на множество различни машини. Възможността да се създават първите универсални от гледна точка на апаратната част програми възниква с появата на езици за програмиране от средно ниво, в които особеностите на устройството на машината (такива като адресация на паметта, регистрите на процесора, системните машинни команди) са много или малко скрити от програмиста. Например, компилатори за езика Fortran са били реализирани на най-различни платформи и алгоритми, записани на този език с незначителни промени (или изобщо без промени) се изпълнявали на различните машини.

Като развитие на идеята за разработката на независим от машината програмен код се явява концепцията за **виртуалните машини**, в по-старо време прилагана към апаратното ниво, а в по-ново – на програмно ниво (например платформата Java). Виртуалните машини се създават за превод от някой общ за всички изчислителни среди език на език на конкретната изчислителна среда. Описаното по-горе е само един аспект на категорията универсалност, свързана с универсалността на работата на програмния код в различните изчислителни среди. Това е самоуниверсалност на програмите по форма. Друг аспект е универсалността по съдържание – пригодност на програмата към решаване на широк клас задачи. От гледна

точка на алгоритмичния подход към програмирането, универсалността на програмата зависи от универсалността на този алгоритъм, който реализира програмата. Универсалните алгоритми позволяват да се решават по-широк клас задачи и обикновено това свойство на алгоритъма се придобива за сметка на отказа от оптимизация за всеки специфичен подклас задачи. Алгоритъмът обикновено е тясно свързан със структурите от данни. При програмиране на „старите" езици често възниква необходимостта да се дублират едни и същи алгоритми за различни структури от данни. Например реализация на дърво с различни типове възли изисква програмиране за всеки отделен тип. Такъв метод не може да се приеме за удачен: първо, той е крайно трудоемък. Второ – всяка реализация на алгоритъма трябва да се проверява за грешки. Трето – модификацията на алгоритъма изисква пренаписване на всичките му реализации в програмния код. Проблемът се решавал по пътя на използване на нетипизирани указатели към записаните във възлите на структурата данни. Но и такъв подход е неудобен, защото не може да предпази програмиста от грешки по неправилната интерпретация на данните, скрити зад нетипизираните указатели.

Развитието на абстрактните типове данни в езиците за програмиране доведе до създаването на **параметризирани типове** данни и цели програмни модули (например, в езика Ada). При програмирането на алгоритми разработчикът може да се възползва от неопределени в дадения момент типове данни, явяващи се параметри на модула или структурата от данни. Да се работи с такива параметри е възможно само, ако се абстрахираме от съдържанието им. Например може да се разработи модул за операции над стек с неизвестни елементи. Модулът за работа с неизвестни елементи и елементите от конкретен тип за този модул са разделени помежду си със „стена" на абстракция – добре капсулирани. За да може да се използва модула е необходимо програмистът да му направи „специализация" – да зададе на параметрите на модула конкретния тип данни. Тогава вече взетият за пример модул за работа със стек може да се превърне в модул за работа със стек от например цели числа или низове, или дефинирани от потребителя структури.

Компилатори на езици, поддържащи подобни конструкции, обработват специализираните модули като различни варианти на изходния модул, опитвайки се да прилагат всички операции, изпълними над типовете-параметри или параметризираните типове, към конкретните типове, които са стойности на праметрите. Ако това стане – специализацията се счита за успешна. В противен случай компилаторът съобщава за синтактична грешка.

Например операцията деление на две променливи от неизвестен тип-параметър се специализира успешно с числови типове променливи и довежда до грешка при опит да се зададат като фактически параметри низове.

Параметризацията на модули и структури от данни като цяло решава проблема с контрола на типовете и избавя програмиста от необходимостта многократно да програмира едни и същи алгоритми. Такова решение не води до неуправляемо нарастване на сложността на програмата (11). При разработването на параметризиран модул програмистът не може (или не е длъжен) да прави догадки за вариантите за специализация на модула. От друга страна, при специализация на модул програмистът не е длъжен да прави предположения за особеностите по реализацията на предоставения му параметризиран модул. Въпреки това често възникват задачи, когато такива предположения е необходимо да се правят. Проблемът може да се реши с въвеждането на йерархия от последователно уточнявани структури от данни и модули, където всяка нова структура или модул по пътя от най-абстрактните към конкретните решения съдържа в себе си частична специализация на стоящите в йерархията над нея решения. Такава организация на кода често е свързана с необходимост да се задават повече параметри. Остава и проблемът с ефективността на реализацията. Най-универсалните алгоритми за обработка на данни не са най-ефективни в конкретните решения. Често се налага за конкретни специализации на типове данни да се използват конкретни варианти на алгоритми за тяхната обработка. Например сортировка в случая на структури от данни с произволен достъп до елементите ефективно се реализира с алгоритъма на Хоор или Шел. Тези алгоритми, обаче са неефективни при структури с последователен достъп.

В ООП съществува собствен начин на изразяване на абстракцията в езиците за програмиране. Между класовете може да се дефинира отношение на обобщение, при което от няколко класа, наречени наследници, общите части се отделят в отделен клас, наречен предшественик. При това наследниците наследяват от предшественика цялостно или частично неговата вътрешна структура от данни и съществуващите реализации на неговите методи. Благодарение на механизма на **наследяване** програмистът се освобождава от необходимостта да дублира код на алгоритмите, съкращават се разходите за труд и се повишава надеждността на създаваните приложения.

В редица езици за програмиране се среща **множествено наследяване** – един наследник обединява в себе си свойствата на множество предшественици. Трябва да се отбележи, че свойствата на наследяване не се огра-

ничават с автоматично пренасяне на свойствата от класовете-предшественици към класовете-наследници. Важно е това, че цялата верига наследявани класове съхранява свойството на капсулиране на своите вътрешни структури и реализацията на своите методи като цяло – скрита от невлизащите в тази верига класове. Няма необходимост от предоставяне на общ достъп до тези части на класовете-предшественици, които ще се използват от класовете-наследници.

Запазването на свойството капсулиране на класовете при изнасяне на общите части в отделни класове отсъства в модулния подход и не може да бъде реализирано с помощта на механизма на параметризация. Това свойство на класовете позволява по-добре да се управлява сложността на разработката при достигане на някакво ниво на универсалност на създаваните решения, отколкото това е възможно в модулното програмиране.

Механизмът на наследяване при ООП е тясно свързан с още едно свойство – **полиморфизъм** на типове и функции. И полиморфизмът на типовете, и този на функциите не се ограничава в рамките на ООП. Полиморфизъм на типовете, например, е добре реализиран в езика Ada (9): всички типове данни са построени йерархично по нива на абстракция и програмистът може, например, на основата на целочислен тип данни да определи собствен тип с по-ограничен интервал на стойностите от изходния тип. Полиморфизмът при процедурите и функциите е известен под името „предефиниране" – програмистът може да реализира няколко варианта на една и съща функция с различен набор параметри, типове на параметрите и връщани от функцията резултати, или да определи алтернативни варианти на реализациите на функциите в различните модули.

В ООП свойството полиморфизъм е изведено на качествено по-високо ниво. Тъй като класовете обединяват в себе си и данните и методите за тяхната обработка, свойството полиморфизъм на класовете обединява в себе си полиморфизма на типовете и на функциите. От гледна точка на полиморфизма на типовете, някой обект едновременно се явява екземпляр както на своя собствен клас, така и на предшествениците на този клас от всяко ниво на генерализация. Благодарение на това един и същ обект може да се използва в работата на алгоритми на различни нива на абстракция. Благодарение на капсулирането на особеностите на обекта при разработването на всеки алгоритъм можем да се абстрахираме от несъществените за

алгоритъма детайли на класа и да не се грижим за конкретните обекти по време на изпълнение на програмата.

Заедно капсулирането, наследяването и полиморфизмът позволяват да се програмират решения с такава степен на абстракция, каквато се предоставя от механизма на параметризация на модули и структури от данни. Така полиморфизмът на класовете позволява удобно да се реши проблема за ефективната специализация на алгоритмите за обработка на абстрактни данни, което е трудно за реализиране само с помощта на маханизма на параметризация. Всеки клас-наследник може да предефинира методите от своя предшественик, като отчита особеностите на собствената си реализация. При това използването на механизма на **виртуалните функции** гарантира, че при работата на програмата ще се извика метод на конкретния клас, а не метод на класа, от чийто тип е обекта, чрез който е извикан методът. По такъв начин в ООП се размиват понятията изпращане на съобщение към обект и извикване на метод на обект. Към някой обект от някакъв тип, който не е конкретизиран, се изпраща съобщение. Обработката на това съобщение ще поеме една от предефинираните функции в наследствената йерархия от реализации на метода. Изборът на конкретна реализация става автоматично и зависи от конкретния клас на обекта. Най-активно такъв подход се използва в интерпретируемите или динамично компилируемите нетипизирани езици за ООП. Въвеждането на механизъм за явно изразяване на абстракцията в езиците за програмиране позволи да се създават програмни решения с нужната степен на универсалност, значително понижавайки разходите за програмистки труд и повиши качеството на работа на програмистите.

**2.3. Гъвкавост**

В наши дни, в голяма степен благодарение на ООП, се натрупа такъв обем вече разработени програмни системи, библиотеки, компоненти. От тях са събрани такива масивни и сложни програмни комплекси, че в много случаи да се постави задача за разработка на принципно нови програмни решения е икономически нецелесъобразно. Често се поставят задачи за приспособяване на налични разработки към решаването на нови проблеми и адаптиране на налични програмни системи към решаване на допълнителни задачи. Класическата задача за написване на компютърна програма се е превърнала основно в задача за събиране и преправяне на налични компоненти, готови библиотечни модули, обекти, функции, съединяване заедно на различни компоненти и услуги за решаване на произволен приложен проблем. Отчитайки смяната на съдържанието на задачата на програмирането, разработчикът трябва да преразгледа своите

възгледи за качествените характеристики на разработваните от него програмни решения, да отдели внимание на **гъвкавостта** на програмите.

Под гъвкавост на програмното осигуряване се разбира организирането на такава вътрешна структура на програмата, която позволява да се модифицира програмата с минимални разходи за труд. Трябва да обърнем внимание на факта, че гъвкавостта не е моментна, а постоянна характеристика на програмното решение (4).

Първо, гъвкавото решение притежава такава структура, която осигурява минимизация на разходите за труд при произволно (в това число и от преди известно) изменение на програмата. Второ, всяко изменение трябва да се прави така, че да не намалява гъвкавостта на програмата. Гъвкавото решение е дългосрочно решение. Ако в процеса на модификацията на съществуващата програма разработчикът достигне до извода, че е по-изгодно да се пренапише всичко от начало, отколкото да се преработва, може да се констатира краят на жизнения цикъл на тази програма. Такава програма може да се нарече „морално остаряла" – не съответстваща на новите изисквания към нея, възникнали по време на експлоатацията ѝ. Да удовлетвори новите изисквания може само ново програмно решение. Ако всяко ново изискване към някоя програмна система е възможно с незначителни преобразувания, жизненият цикъл на такава програмна система продължава. Вложените усилия и труд в нея не пропадат.

Определящи за важността на гъвкъвостта на програмното решение за разработчика са условията за експлоатация на това решение, както и съдържанието на потока нови изисквания към експлоатираното решение. По такъв начин, ако се пише „еднократна" програма, не предполагаща многократни стартирания, продължителна експлоатация, или ако се пише много специфична програма, силно свързана към хардуера, не предполагаща повторно използване на нейни части в други проекти, гъвкавостта на такава програма не е важна характеристика за разработчика. За програмни системи, предназначени за активна работа с потребители, за масови продукти, гъвкавостта често е толкова важна характеристика, че става по-приоритетна даже от въпросите на алгоритмическата ефективност и икономичността на използваните изчислителни ресурси.

Средство за осигуряване на гъвкавост на програмната система е използването на операциите абстрахиране и прилагане (аплициране) при синтеза и анализа на проектните решения. Достатъчно гъвкава е

структура на програмно решение, състояща се от напълно разделени на различни нива на абстракция общи и специални решения. Такива нива образуват архитектурата на програмната система. В преобладаващото мнозинство от случаи става дума за функционални нива на архитектурата, като тази функционалност се разбира двузначно. В единия случай се наблюдава подчинение на някакви посистеми на обща за тях суперсистема. В този случай суперсистемата встъпва в ролята на регламентиращ програмен елемент, определящ реда и функционалния състав на своите подсистеми. В друг случай се наблюдава само капсулация на реализацията на възложените на някоя система функции. Такива системи обикновено са самостоятелни компоненти и услуги, или цели библиотеки, готови за включване в различни комплекси и изпълняващи разнообразни функции.

Фактически тези две гледни точки определят двата различни подхода към построяването на архитектурата. Единият е проектиране отгоре надолу – от концептуалните проблеми, решавани на високите нива на абстракция с последователно разкриване на съдържанието на решението в по-специализирани и конкретни подсистеми. Другият подход е проектиране отдолу нагоре – на базата на наличните компоненти се създават програмни системи, решаващи приложни задачи. И двата подхода имат своите предимства и недостатъци от гледна точка на гъвкавостта на получаваните решения (2).

*Първият подход* има предимството, че получаваното с негова помощ съдържание на архитектурно ниво е ограничено до най-необходимото за решаването на задачата. Това със сигурност влияе както на сроковете за разработката, така и на обема на полученото решение. Недостатък на подхода е рискът от бърза загуба на гъвкавост на програмата при преждевременното използване на създадените програмни елементи в конкретната текуща задача, стояща пред разработчика.

При *втория подход* достойнствата и недостатъците са огледално противоположни. Преимущество тук е активното повторно използване на по-рано разработен програмен код. При това разработеният код в голяма степен е абстрахиран от решението на конкретната задача, стояща пред разработчика и е универсален. Недостатъкът се състои в това, че често разработчиците на компоненти ги претрупват с функционални възможности за всички възможни и невъзможни житейски ситуации, тъй като не знаят предварително в какви задачи ще се прилагат тези компоненти. Това значително увеличава кода и себестойността на ком-

понентите с неблагоприятно влияние върху качеството на предлаганите решения.

Отделно трябва да се отбележи процедурата по стандартизиране на програмните библиотеки, компоненти и методи за прилагането им – това, което днес се нарича технология на програмирането. Очевидно преимущество на стандартизацията е във взаимозаменяемостта на стандартизираните компоненти, в унификацията на интерфейсите и методите за тяхното свързване. Недостатък е опасността при стандартизацията от подмяна на решаваната задача: вместо опит за адаптиране на стандартните решения към поставената задача често се среща опит да се адаптира задачата към стандартните методи за решение. В резултат процесът на разработката може напълно да деградира: разработчиците вместо действително решаване на стоящите пред тях задачи обявяват за невъзможно решението, поради ограниченията на стандартните методи за решение, или преформулират задачата така, че да я „пригодят" към наличните стандарти. В резултат на това се получава решение не на изходната задача, а на някоя друга.

На другия полюс е пълното игнориране на стандартите: разработчиците престават да следват правилата при решаването на стоящата пред тях задача.

И двата подхода показват крайностите на парадоксалното съдържание на категорията гъвкавост на програмната система. Първият подход намалява сумарно разходите за труд на отделно взета програмна система, но за сметка на по-бързата деградация на гъвкавостта на създаваната система в процеса на адаптирането и към нови изисквания. Вторият подход акцентира вниманието на високата адаптивност на отделно взета компонента към различните области на нейното приложение, но за сметка на по-големите разходи за труд. Да се реши този парадокс е възможно чрез придържане към „средния път". От една страна, е необходимо да се откажем от практиката да се разработват „тежки" компоненти, претрупани с функции, повече от които обикновено не се използват в конкретните решения. От друга страна, да се откажем от практиката на преждевременно частично решение на конкретна задача, вместо решението на задачата в нейния обобщен, абстрахиран от частната формулировка вид, ако се предполага продължителен жизнен цикъл на разработваната програмна система. В резултат активно и в пълна степен повторно използваеми се оказват най-абстрактните „полуфабрикати" програмни компоненти – скелета на програмната система. Адаптацията на тези „полуфабрикати" към конкретната

задача, привеждането им до вид готов за използване от потребителя е същността на конкретната работа.

По такъв начин двата подхода се сливат в едно цяло. От една страна, разработчикът с помощта на операцията „прилагане" повторно използва компоненти, съдържанието на които е абстрахирано от конкретната задача. От друга страна, цялата специфика на своята конкретна задача разработчикът реализира сам, съгласно изискванията – той не е предварително ограничен от неявни конкретни решения и методи. От трета страна, нито разработчикът на приложни програмни системи, нито разработчикът на компоненти изпълняват излишна работа, от която никой няма да се възползва повторно. Ако разработчикът на приложни програми усети, че използваните от него абстрактни компоненти не му позволяват да реши стоящите пред него задачи, следва той да се откаже от някои компоненти, пренасяйки решението на проблема на по-абстрактно ниво. Когато на високите нива на абстракция възникващите проблеми са успешно решени, на по-ниските нива става възможно решаването на приложната задача. Ако, отказвайки се от едни компоненти, разработчикът не е намерил по-подходящи за решението алтернативи, той може да ги реализира самостоятелно. На този етап се откриват предимствата на обобщения подход: не се налага да се създават трудоемки завършени компоненти и в същото време не е нужно веднага да се създават конкретни решения. Необходимо е да се разработват „полуфабрикати" компоненти и чак след това да се пристъпи към прилагането им към конкретната приложна задача. Разделянето на общото и частното решение протича по линия, отделяща компонента от кода на неговото прилагане за решението на задачата.

Могат да се открият примери за подобна практика. Гениалността на изобретяването на такъв клас приложно програми като табличния процесор (електронната таблица) потвърждава широкото приложение на тези програми в най-различни приложни области. Гениалността се състои още и в разбираемите принципи на работа на табличния процесор даже за не много добре подготвени потребители. Лежащата в основата на табличния процесор абстракция от гледна точка на програмирането е извънредно проста. В същото време прилагането (аплицирането) на тази абстракция е най-разнообразно. Потребителят сам изпълнява операциите по абстрахиране и прилагане при работа с таблици. Той формира структурата на конкретната таблица на основа на обобщено решение. Той включва в тази таблица тези данни и формули,

които са му необходими. От гледна точка на програмата (от гледна точка на програмираната от разработчика абстракция) всичко, което се въвежда от потребителя при създаването на произволна конкретна таблица е не повече от матрица от обекти на различни достатъчно прости типове (низове, числа, дати, формули и др.). От гледна точка на потребителя всяка конкретна таблица е очевиден за него резултат от аплицирането на абстракцията към конкретна приложна задача. Колкото по-обобщена е абстракцията, толкова повече възможности има за нейното аплициране, толкова по-голяма е вероятността за повторното и използване в бъдеще. Колкото по-висока е вероятността за аплициране на абстракцията, толкова по-голямо е количеството на разнообразните изисквания, които тя трябва да удовлетворява. Следователно толкова по-устойчива е тя в процеса на еволюция на конкретната програмна система.

Овладяването на методите на откриване и явно отделяне на нивата на абстракция дава възможност на разработчика да избира най-гъвкавото програмно решение от наличните алтернативи.

От съществуващите парадигми в програмирането само обектно ориентираният подход предоставя удобни и „естествени" средства за явно описание на ниво програмен код на структури с различна степен на абстрактност на основа на йерархия от наследявани обекти и поведение с различна степен на абстрактност на основа на полиморфизма на обектите.

**Заключение**

Обектно ориентираната парадигма осигурява мощен набор от концепции, напълно абсорбирани в културата на софтуерните разработки от 90-те на миналия век до наши дни, точно както методиката на структурното програмиране през 70-те и 80-те. Доказателство за това е изобилието от средства, поддържащи всички аспекти на разработването на софтуер, следвайки тази парадигма. При нея фазата на проектиране е много тясно свързана с фазата на имплементиране, тъй като дизайнерите използват подобни абстрактни концепции (като класове и обекти), както и програмистите.

Историята показва, че обектно ориентираното програмиране се комбинира успешно с други подходи. То е повлияно, както и то влияе на други идеи. След повече от 30 години, откакто бе представен първият обектно ориентиран език за програмиране, дебатът за преимуществата на новата парадигма продължава. Няма съмнение обаче, че мнозинството софтуерни системи са и ще бъдат обектно ориентирани.

## ЛИТЕРАТУРА


1. *Alphonce, Carl., Ventura, Phil.* Object orientation in CS1-CS2 by design, Proceedings of the 7th annual conference on Innovation and technology in computer science education, SESSION: Objects, Pages: 70–74, Aarhus, Denmark, 2002.

2. *Booch, Grady.* Object-Oriented Analysis And Design, With applications, second edition, Addison Wesley l5th Printing, 1998.

3. *Buhrer, Hans.* Software development: what it is, what it should be, and how to get there, ACM SIGSOFT Software Engineering Notes, Volume 28 , Issue 2 (March 2003), ACM Press. New York, NY, USA.

4. *Eden, Amnon H., Mens Tom.* Measuring Software Flexibility, IEE Software, Vol. 153, № 3 (Jun. 2006), pp. 113–126. London, UK: The Institution of Engineering and Technology.

5. *Kraut Robert, E.* Coordination in software development, Communications of the ACM, Volume 38, Issue 3 (March 1995), pages 69–81, ACM Press. New York, NY, USA.

6. *Lister, R., Berglund, A.* Research perspectives on the objects-early debate, WORKSHOP SESSION: ITiCSE-2006 , ACM Press. New York, NY, USA.

7. *Minsky, M. A.* Framework for Representing Knowledge, Wiston P. (ed.) *The Psychology of Computer Vision*, New York: McGraw-Hill, 1975.

8. *Robert, W Floyd.* Turing Award Lecture, 1978: "The Paradigms of Programming," Communications of the ACM, Vol. 22: 8, August 1979, pp. 455–460. ©1979 Association for Computing Machinery, Inc.

9. *Seidewitz, Ed.* How much object-orientation in Ada?, Proceedings of the conference on TRI-ADA '90, Baltimore, Maryland, United States, 1990.

10. *Turner, David.* Church's Thesis and Functional Programming, from "Church's Thesis after 70 Years" ed. A. Olszewski, Logos Verlag. Berlin, 2006.

11. *Wegner, P.* Concepts and paradigms of object-oriented programming, ACM SIGPLAN OOPS Messenger, Volume 1, Issue 1 (August 1990) Pages: 7–87.

# РАЗВИТИЕ НА ПРОГРАМИРАНЕТО. ПРЕДПОСТАВКИ, ВЪЗНИКВАНЕ И ОСНОВНИ ПОНЯТИЯ НА ОБЕКТНО ОРИЕНТИРАНОТО ПРОГРАМИРАНЕ

ИВАЙЛО ДОНЧЕВ

Резюме

Обосновава се необходимостта от изучаване на ООП. Проследява се историческото развитие на техниките за програмиране. Изяснява се съдържанието на основните категории, свързани с разработването на софтуер и тяхната интерпретация в различните програмни парадигми. Формулира се разликата в процеса на разработка на програми в миналото и сега. Подчертават се предимствата на обектно ориентирания подход в проектирането и програмирането на сложни информационни системи.

# PROGRAMMING DEVELOPMENT. PREMISES, ORIGIN AND BASIC CONCEPTS OF OOP

IVAYLO DONCHEV

Summary

Abstract: This paper discusses the necessity of learning OOP. It traces the historical development of OOP techniques. Clarificates the contents of basic categories related to software development and their interpretation in different programming paradigms. The differences in software development process at present and in the past are formulated. The advantages of OO approach in design and development of complex information systems are emphasized.