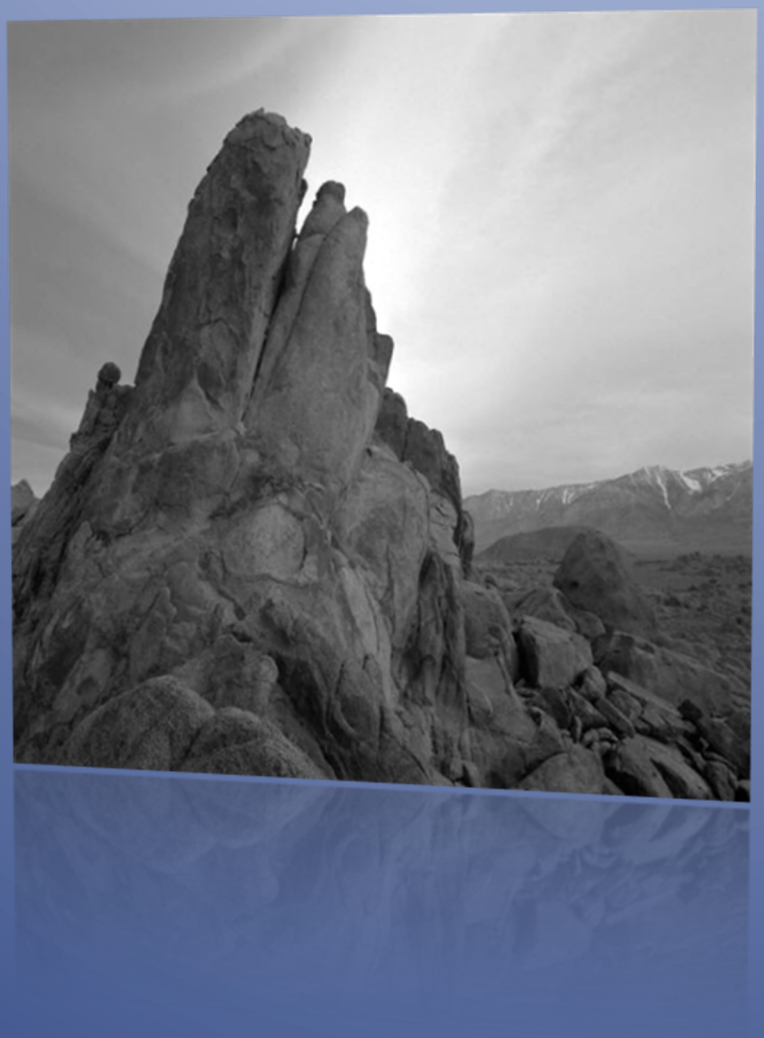
2015.



Драган Ћајић

Објектно-оријентисани софтверски инжењеринг

Object-Oriented Software Engineering using UML and Java

Software and Software Engineering, Object Orientation, UML, Java, Object-Oriented Software Engineering (OOSE): Practical Software Development using UML and Java

Садржај:

1. Увод

Технике софтверског инжењеринга су потребне зато што велике системе не може у потпуности да разумије једна особа.

Тимски рад и координација су потребни у изради великих софтверских система високог квалитета.

Кључни изазов је расподијелити посао и осигурати да дијелови система раде коректно у цјелини, заједно.

Крајњи производ који планирамо да правимо мора бити задовољавајућег квалитета.

Написати програм за пет минута који рачуна нешто, то није инжењеринг, јер није потребно примијенити инжењерске принципе за његову израду.

Инжењерске принципе је потребно примијенити онда када пређемо границу тривијалности, и већина софтвера данас захтијева инжењерски приступ у његовој изради.

Можда можете написати десет хиљада линија кода, али ће неко вјероватно касније хтјети да направи неке измјене у коду, дакле имате тим људи и тада су потребни инжењерски принципи да би људи у тиму могли да комуницирају и пројектују ефикасно.

UML није програмски језик, него то је језик за визуелно пројектовање.

2. Осврт на објектну оријентацију

2.1. Шта је објектна оријентација?

Процедурална парадигма: позовемо процедуру, унесемо податке, добијемо резултате, а не занима нас шта се дешава „изнутра", унутар програма. Софтвер је организован око нотације процедура.

Процедурална апстракција функционише све док су подаци прости.

Процедурално програмирање је добро када радите с простим подацима, бројевима, карактерима и слично, дакле при нумеричким калкулацијама/израчунавањима, гдје имамо нумеричке резултате, па онда на примјер исписујемо те податке. Дакле, подаци су једноставни, мада сама израчунавања могу бити сложена.

Додавање апстракције података у контексту процедуралне парадигме, групише у цјелину дијелове података који описују неки ентитет и тако помаже у смањењу сложености система. Појављују се сложенији подаци типа запис (енгл. *record(s)*) и структура (енгл. *structure(s)*).

Међутим, подаци настављају да постају све сложенији, комплекснији и у једном тренутку људи схватају да раде над овим апстракцијама података из различитих мјеста у програму, што постаје прави кошмар, права збрка у раду, главни програм, један потпрограм, па други потпрограм, итд. Настаје права збрка у приступи подацима с различитих мјеста у програму.

Настаје прекретница и идеја организовања програма око података и то је поријекло концепта класе, оранизовање програма у класе, гдје свака класа представља одређене податке и убацивање процедура у класу. Дакле, имамо сада процедуру, која оперише над подацима, у класи. То омогућава поједностављивање цјелокупне структуре система и омогућава изградњу још сложенијих система, које је још увијек могуће разумјети и лако одржавати.

Објектно-оријентисана парадигма постаје доминантна у свијету у развоју софтвера, али не треба заборавити чињеницу да је велики проценат укупог развоја софтвера још увијек у бити процедуралне основе.

Објектно-оријентисана прадигма је један приступ у рјешавању проблема у којем су сва израчунавања (рачунарска обрада, енгл. *computations*) извршена у контексту објеката

који су инстанце класа, док су класе апстракције података које садрже апстракцију процедура које врше операције над објектима.

Покренут програм може се посматрати као колекција објеката који међусобно сарађују на извршавању датог задатка.

2.2. Класе и објекти

Објекат је цјеловит дио (компактан дјелић) структурисаног податка у софтверском систему који је у радном стању. Посједује својства, која представљају његово стање, и понашање, које одређује како објекат дјелује и реагује, а може и да симулира понашање неког објекта у стварном свијету.

У раним фазама развоја софтвера не радимо пуно са објектима, углавном радимо с класама из којих правимо објекте.

Класа је јединица апстракције у објектно-оријентисаном (ОО) програму. Репрезентује сличне објекте, инстанце те класе. Врста је софтверског модула који описује структуру својих инстанци (== објеката), то јесте њихова својства (енгл. *properties*) и садржи методе које имплементирају понашање инстанци (енгл. *methods*), које ће бити извшене над тим објектима током рада програма.

Термини су јасни већини људи, али када се дође до објектно-оријентисане анализе праве се грешке шта је објекат а шта је класа!

Нешто треба да буде класа ако може да има инстанце, а нешто друго треба да буде инстанца ако је то без сумње један члан скупа дефинисаног класом.

Потребно је у току објектно-оријетисане анализе и дизајна размислити и разумјети шта је класа, а шта инстанца.

Препоруке за именовање класа су:

- ✓ употреба великог почетног слова; за више ријечи у имену класе употреба *Upper CamelCase* (тзв. *PascalCase*) нотације,
- ✓ користити именице у једнини – не употребљавати множину за именовање класа, јер уноси забуну, нпр. инстанце класе *Accounts* се могу погрешно протумачити као да су скуп рачуна (енгл. *Set of Accounts*), а ми желимо да објекат представља једну ствар,
- ✓ потребно је изабрати одговарајући ниво уопштености (енгл. *level of generality*) при именовању класа, нпр. при изради система за обрачун пореза (на имовину) *Naselje* или *NaseljenoMjesto* би било прихватљивије име за класу него име *Grad* –

не живе сви становници у (великим) градовима, дакле користићемо израз који је одговарајућег новоа уопштености / општости да бисмо обухватили све инстанце,

- ✓ бити сигуран да име класе има само једно значење, дакле без двосмислености, треба изабрати добра име за име класе.

2.3. Варијабле

Постоје различите врсте варијабли. У ранијим фазама програмирања постојале су локалне варијабле унутар процедура, функција, касније метода. У неким програмским језицима постоје глобалне варијабле – варијабле које су изван било које процедуре, било које класе, саме су и доступне свугдје.

Глобалне варијабле не употребљавамо у Јави, али имамо локалне варијабле, и унутар класа можемо да дефинишемо два типа варијабли: варијабле које припадају инстанци класе (енгл. *instance variables*, *fields* или *member variables*) [атрибути и асоцијације] и варијабле које припадају класи (енгл. *class variables*), такође познате као статичке варијабле (енгл. *static variables*), са *static* кључном ријечју.

Битна разлика између варијабле и објекта: варијабла је мјесто гдје смјештамо објекат (енгл. *placeholder*), па се каже да варијабла референцира објекат, то је показивач, референца на објекат, објекат је дио меморије (енгл. *chunk of memory*), а варијабла је мјесто гдје смјештамо референце на објекат.

Држаћемо се Јава концепта варијабле, концепт у C++ је сложенији – можемо имати трајно смјештене дијелове меморије унутар варијабле без постојања референци, али у Јави имамо посла с референцама.

На објекат може да показује више варијабли, типично можемо имати варијаблу инстанце која има референцу на неки објекат и локалну варијаблу која манипулише, тј. контролише неки алгоритам и има референцу на исти објекат.

Објекат има класу, варијабла има тип. Тип одређује која врста објеката може бити смјештена у варијаблу тог типа (нпр. варијабла типа *Kvadrat* и варијабла типа *Krug*).

Јава је строго типизиран језик, тако да имамо строга ограничења који објекат може варијабла да садржи.

Варијабле класе и статичке варијабле су синоними у овом контексту, а то значи да је вриједност варијабле класе **дијељена** од стране свих инстанци класе, док је варијабла инстанце доступна само једној инстанци класе, то је разлика; варијабла инстанце за једну инстанцу, варијабла класе доступна свим инстанцама.

Када се C++ појавио, коришћена је кључна ријеч *static* да означи варијабле класе, па је то преузето у Јаву.

Суштина је, ако једна инстанца промијени вриједност статичке варијабле, онда све остале инстанце виде исту промијењену вриједност.

Желимо да користимо статичке варијабле веома пажљиво и не тако често. Типична употреба је за чување константи – вриједности које вјероватно не желимо да мијењамо, као нпр. вриједност броја π у некој математичкој класи, или за *lookup* табеле гдје имамо стандардан алгоритам који ће радити неку врсту израчунавања користећи исте вриједности, али уколико нису у питању ове ситуације, правило је: „Не користите статичке варијабле!". Видјећете много лоших пројеката у индустрији гдје су у употреби статичке варијабле, али то није разлог да помислите да треба и ви да их употребљавате!

Static не треба мијешати са *const(ant)* и *final* (*const* не постоји у Јави, само *final*), то су тотално различити концепти!

2.4. Методе, операције и полиморфизам

Операција је апстракција скупа метода, дакле процедурална апстракција вишег нивоа која спецификује тип понашања, а независна је од било којег кода који имплементира то понашање.

Нпр. једна апстрактна операција је „израчунати површину", а „израчунати површину круга" или „израчунати површину правоугаоника" су методе како израчунати површину и представљају различите алгоритме рачунања површине раванске фигуре (слике) научене још у основној школи. Дакле, начин на који ћемо израчунати површину ће се разликовати од класе до класе (у хијерархији класа), тј. различит алгоритам ће бити примијењен, али је једна апстрактна операција „израчунати површину"; једна операција, а више метода да се то уради.

Метода је процедурална апстракција коришћена да имплементира понашање класе.

Неколико различитих класа може имати методе истих имена, које имплементирају апстрактну операцију на начин подесан за сваку класу.

Полиморфизам је својство објектно оријентисаног софтвера помоћу којег нека апстрактна операција може бити извршена на различите начине у различитим класама.

Полиморфизам је једна од кључних ствари која дефинише објектно оријентисани систем, тј. ако намамо полиморфизам у програмском језику – не можемо га назвати објектно оријентисаним. Захтијева да буде више метода истог имена, а избор која ће

бити извршена зависи током извршења (енгл. *run-time*) од тога која тачно одређена врста/класа објекта се налази у датој варијабли.

Ако немате хијерархију (насљеђивања) класа, него једну усамљену класу онда не можете имати полиморфизам, јер можете декларисати варијаблу те класе и знате да ће та варијабла увијек имати/садржавати објекат те класе.

Дакле, имате хијерархију насљеђивања класа, имате мноштво подкласа, направите варијаблу суперкласе, и не знате која ће се од њених подкласа појавити током извршења у варијабли, и онда током извршавања систем провјери и види која класа је у варијабли и онда изабере одговарајућу методу да покрене, нпр. ако је *Krug* онда се покреће верзија методе за рачунање површине круга, ако је *Trougao* онда се бира њена метода, итд.

Предност коришћења полиморфизма је да не морате да пишете: ако је *Krug* уради ово, ако је *Trougao* уради нешто друго, ако је *Pravougaonik* уради ... дакле смањује се потреба програмера да пише пуно *if-else* или *switch* исказа у коду. Другим ријечима, ефективан полиморфизам у контесту хијерархије класа поједностављује кодирање.

2.5. Организовање класа у хијерархије насљеђивања

Суперкласа садржи својства заједничка за скуп подкласа, тј. информација коју посједује налазиће се у свим њеним поткласама.

Хијерархије насљеђивања показују односе између суперкласа и поткласа.

Насљиђивања је, уз полиморфизам, један од кључних концепата објектно оријентисане парадигме.

Насљеђивање је имплицитно посједовање, од стране свих поткласа, својстава/одлика дефинисаних у њиховим суперкласама.

Како можемо да провјеримо да ли смо добро направили хијерархију класа?

- ✓ Користећи *ISA* правило, које мора увијек да важи, јер су поткласе специјалне врсте суперкласе. Ако то није случај – хијерархија није добра!

Ово зна уносити забуну и бити чест извор грешака! *ISA* правило нам не гарантује да смо направили добар пројекат, али ако га прекршимо можемо бити сигурни да имамо лош пројекат/дизајн.

Сваки пут када направимо поткласу, треба да провјеримо у мислима *ISA* правило.

Сљедеће што је потребно провјерити је смисао, тј. да ли све што имамо у суперкласи има смисла у поткласама.

Потребно је омогућити лако одржавање нашег система и поновну употребљивост.

Двије намјене постојања апстрактне класе су:

- ✓ омогућава нам да декларишемо варијабле, које можемо послије да ставимо у било коју класу нижег новоа,
- ✓ можемо да декларишемо апстрактне методе које морамо да имплементирамо у стварни метод на нижим нивоима хијерахије; до краја хијерархије морамо имати конкретну имплементацију свих апстракних метода.

О много пројектантских одлука и алтернатива треба размислити и то је класични случај размишљања о пројекту/дизајну, алтернативама које нам дају најбољи квалитет по питању лаког одржавања (колико се добро носимо са измјенама), једноставности и поновне употребљивости (кроз насљеђивање).

2.6. Насљеђивање, полиморфизам и варијабле

Од апстрактне класе немогуће је креирати њене инстанце, јер у тој класи није садржан довољан број информација које су потребне да би класа била у могућности да уради нешто смислено са инстанцама те класе. Једино што можемо да урадимо јесте да правимо инстанце од класа које се налазе ниже у хијерархији, а то су конкретне класе.

Не морају све конкретне класе да буду на најнижим позицијама хијерархије класа, некада се конкретне класе налазе на вишим позицијама у хијерархији – то је могуће, али што можемо засигурно рећи јесте да су све класе на дну хијерархије конкренте класе, јер другачије не бисмо могли да направимо инстанце тих класа и хијерархија (или њен дио) би била неупотребљива.

Апсктрактне методе морају бити имплементиране негдје до дна хијерархије, односно до тренутка када дођемо до подножја хијерархије, што значи да можемо креирати варијаблу типа апстрактне класе у коју ћемо ставити инстанце било које од конкретних класа и онда можемо позивати операције над том варијаблом и то апстрактне методе које су декларисане у апстрактној (супер)класи. Дакле, сваки објекат који се налази у тој варијабли моћи ће да изврши операције апсктрактне класе, то јесте њене апсктрактне методе.

Апсктрактне класе и методе

Операција треба да буде декларисана да постоји у класи која је на највишој позицији у хијерархији, гдје то има смисла, а имплементираћемо је прије него стигнемо до неке од класа „листова". Дакле, ако има смисла ставити операцију на виши ниво – онда је треба ставити на виши ниво хијерахије. Треба да будемо сигурни да сва насљеђена својства из суперкласе имају смисла у поткласама.

Ако у класи имамо апстрактну операцију, онда класа мора бити апстрактна, читава мора бити проглашена апстрактном, као цјелина. Значи довољна је само једна апстрактна операција декларисана у класи да би читава класа постала апстрактна. Другачије речено, ако имамо конкретну класу све њене методе морају бити имплементиране, морају имати имплементацију, тј. не могу се налазити/мијешати апстрактне операције у конкретној класи са операцијама које посједују имплементацију.

Ако суперкласа има неку апстрактну операцију, онда њене потклассе на неком нивоу морају имати конкретну методу за ту операцију, тако да када стигнемо до класа „листова" имамо конкретне методе за све операције. Значи, класе „листова" хијерахије морају имати или наслиједити конкретне методе за све операције, и класе „листова" хијерархије морају бити конкретне класе. Нема смисла да имамо апстрактне класе за класе „листова", јер су неупотребљиве. Да би класа била употребљива, корисна, морамо бити у могућности да направимо инстанце те класе, или те класе или класа испод ње у хијерахијском ланцу насљеђивања.

Већина објектно-оријентисаних система има неке од хијерархија класа велике, али већина кода у системима је , у ствари, релативно мала у погледу величине хијерархије насљеђивања. Типична „дубина" хијерархије насљеђивања класа за типично одсредње класе је један или два (1-2) нивоа. Веома је уобичајено, у односима који нису типа суперкласа-потклаcа, да имате класу која је сама и ради нешто релативно просто и нема генерализације. То је сасвим уобичајено, зависно од система може бити чак и до 50% класа које немају суперкласа-потклаcа однос/релацију. Дакле, биће значајан проценат хијерахија са бар једним или два нивоа, и у сваком великом систему вјероватно ће се налазити неколико пристојно великих хијерархија класа.

Што више имате класа у хијерархији, то бољу поновну употребљивост имате, бољи полиморфизам добијате и бољу поновну употребљивост суперкласе; насљеђивање метода је моћан концепт, полиморфизам такође, па је корисно да размотрите ситуације у којима можете креирати хијерархије, и наравно апстрактне класе и методе. Али не претјерујте с тим, не форсирајте хијерархије насљеђивања у системима гдје то, у ствари, није неопходно, јер постоји одређени степен комплексности да се реализују претходно споменуте ствари/концепти, да се „упосле"/угради апстрактне класе, насљеђивање итд.

Дакле, корисно је са аспеката поновне употребљивости, полиморфизма итд., али не треба претјерати са форсирањем хијерархија насљеђивања тамо гдје то није потребно; постоји одређени степен комплексности упослити апстрактне класе, насљеђивање, и слично.

Преклапање метода (енгл. *Overriding*)

„Преклапање" метода је концепт, гдје можете имати класу која има методу, а онда можете имати и потклассу која има методу истог имена, и у ствари потклаcа преузима функционалност на нижем новоу.

Бројни су разлози зашто бисте жељели „преклапање" метода. Један од разлога може бити рестрикција/ограничење – то називамо преклапање ради ограничења. Када се наруше та ограничења „бацамо" изузетак. Видјећемо касније да морате бити опрезни око тога, јер када почнете са превише бацања изузетака у одређеним случајевима, добијете/стварате додатну сложеност вашем систему, а то је најбоље избјегавати.

Али, без обзира на то, можда ће бити нешто што ће требати у одређеним случајевима увођење ограничења коришћењем преклапања метода. У основи, преклопљена метода ће бацити изузетак у одређеном случају, док ће у неком другом случају метода радити без ограничења – на подразумијевани начин (енгл. *default*).

Други разлог за преклапањем метода јесте да имате додатне информације, додатне ствари у потклаcи, додатна обиљежја – то називамо преклапање ради проширења (енгл. *extension*).

Са аспекта клијент нису битни разлози за преклапањем метода – он то не примијети, али нама је битно да се припремимо за могућност појаве изузетака и примијенити *try/catch* блокове у Јави.

Трећи разлог за преклапањем метода је оптимизација – преклапање ради оптмизације. Концепт у овом случају је да метода ради у потпуности исти посао, без проширења, без ограничења, дакле као насљеђена метода, али то ради на ефикаснији начин; можда користи мање меморије, можда употребљава мање процесорског времена. То је чест случај, да алгоритми који раде одређеној класи ради ефикасније, јер имају више информација, више правила у тој класи. Нпр. метода за рачунање обима круга је значајно једноставнија, него метода за рачунање обима елипсе. Метода на страни круга је оптимизованија. Са аспекта клијента то је иста ствар – рачунање обима раванске фигуре.

Стандардан начин писања кода када радите преклапање јесте да често позовете *super* верзију методе која ће бити аутоматски пронађена на вишим нивоима хијерархије, након што сте урадили дио посла, позовете *super* верзију да преузме контролу и уради остатак посла. То је веома чест програмерски идиом/стил.

Потребно је запамтити, када је у питању преклапање метода, да концептуално, све имплементације исте апстрактне операције треба да раде логички исту ствар – не би требало произвољно проширивати операцију да ради логички потпуно другачију ствар, нити да је ограничимо тако да операција постане толико различита да не спада у исти скуп понашања као и остале имплементације исте операције.

Непромјењиви објекти (енгл. *immutable objects*)

Концепт који може бити користан у неким ситуацијама јесте концепт непромјењивих објеката. Непромјењиви објекти су објеки који када су једном креирани, користећи конструктор, нису предвиђени да буду промијењени од тренутка конструисања до краја

њиховог животног вијека. Понекад желите да пројектујете хијерархије класа, гдје су објекти непромјенљиви из разних разлога. У том случају, ако неко позове метод који ће покушати да направи измјене над објектом који је *immutable*, метод ће вратити нови објекат који ће рефлектовати/одражавати измјене, а оригинални објекат ће остати исти.

Уопштено говорећи, ако одлучите да све операције враћају нови објекат са измјенама, радије него да мијењају оригинални објекат – тада требате бити конзистенти/досљедни свуда и поштовати тај шаблон (енгл. *pattern*) свугдје – сви објекти су непромјенљиви, или донијети одлуку да свуда промјене над објектима буду дозвољене. У супротном, добићете комплексност, јер људи неће моћи схватити да ли је објекат дефинисан као непромјенљив или није.

Дакле, желите да будете досљедни сво вријеме у било којој врсти дизајна/пројектовања. Конзистентност је једно од правила доброг дизајна.

Основни разлог за примјену правила у пројектовању јесте да би (други) људи били у могућности да разумију ваш дизајн/пројекат. Нека правила се примјењују из разлога ефикасности, али велики проценат њих се примјењује зато што ви треба да се трудите да будете сигурни да је ваш систем што једноставнији, што разумљивији и да се лако прилагођава промјена, колико је то могуће.

Можемо ми писати шпагети кôд (енгл. *spaghetti*) и можемо стварати велику, непријатну збрку, ако желимо да правимо програме које ниједно људско биће неће бити у стању разумјети, јер ће бити у потпуности нечитљиви. Међутим, ми желимо управо супротно – ми желимо да наш дизајн буде лако разумљив, па тако мијешање непромјенљивих објеката и промјенљивих би било тешко за разумијевање, потребно би било вријеме за тумачење, зато желимо да изаберемо стил један или други, и да се држимо тога.

Дакле, концепт непромјенљивих објеката јесте да након конструисања објекта, не могу се мијењати варијабле инстанце, оне су фиксиране и ако постоји метода која прави измјене или покушава да прави измјене, враћена вриједност ће бити нови „облик" који има логично измијењено име, али је то нови објекат.

Концептуални алгоритам за одлуку коју методу покренути:

1. Процес почиње позивом *main* функције над варијаблом, и тада објекат у варијабли треба да реагује на тај позив; први корак је да објекат буде у варијабли – то може бити локална варијабла, варијабла инстанце, може бити и варијабла класе/статичка варијабла (енгл. *static*), могуће је.

   Систем провјерава објекат у тој варијабли, који може бити од класе којом је та варијабла декларисана (типа те класе) или од било које њене поткласе.

   Ако постоји конкретна метода за операцију у текућој класи -> покрени методу

2. Ако не постоји метода у текућој класи, провјери у непосредној наткласи, и ако постоји у њој конкретна метода -> покрени методу
3. Понављај претходни корак 2, тражећи у вишим узастопним суперкласама конкретну методу док је не пронађеш, и -> покрени методу
4. Ако не постоји конкретна метода, онда је то грешака, и за Јаву и C++ компајлер би требало да јави да не постоји конкретна имплементација метода у хијерархији и не би требало да можемо да дођемо до овог корака 4.

Овај процес је апстрактан, не дешава се овако у стварности, јер у стварности има много уграђене оптимизације, ту су *look-up* табле уграђене у извршно окружење (енгл. *run-time environment*), тако да је имплементација овог процеса изведна на много ефикаснији начин и то зависи од специфичне верзије извршног окружења језика.

Динамичко везивање (енгл. *Dynamic binding*)

Динамичко везивање је још један кључни концепт објектне-оријентације који слиједи из концепта полиморфизма и насљеђивања, и то је у основи оно што се дешава у посљедњем алгоритму. Претходни алгоритам је процес динамичког везивања, то је процес динамичког одлучивања коју методу покренути, и динамичко везивање је потребно само у одређеним условима, то јесте морају бити испуњена ова два услова:

- ✓ постоји варијабла која је типа суперкласе, јер ако имамо варијаблу типа класе са дна хијерархије, онда засигурно знамо који ће метода бити позвана, а то је метода који припада тој класи „листу" – та ће бити позвана, уз све оне методе које су наслијеђене, али сво вријеме тачно знамо које су то методе за ту варијаблу
- ✓ ако декларишемо варијаблу која је типа класе вишег ранга у хијерархији, онда у ту варијаблу можемо смјестити објекате различитог типа, тако да тачно не знамо која метода ће бити покренута, па је други услов који мора бити истинит јесте да мора постојати више од једне раположиве полиморфне методе, коју је могуће покренути поред типа варијабле и њених поткласа.

Оба услова морају бити испуњена/истинита.

2.7 Концепти који дефинишу објектну оријентацију

То су ствари које су неопходне да буду у систему, да би систем могао да се назове објектно оријентисаним. То се такође односи и на програмски језик, а то су принципи:

- ✓ објект има јединствен идентитет

У традиционалним програмским језицима и релационим базама података посебно, постоји мишљење да ако два дјелића података изгледају исто – онда су исти, док у објектно оријентисаном моделу имамо објекте, који могу да садрже исте податке – али

ипак немају исти идентитет. Дакле, објекти имају идентитет који иде изнад тога које податке садрже објекти. То је први концепт који треба да имамо.

- ✓ кôд је организован употребом класа, а свака класа описује скуп објеката, тј. класа представља шаблон или модел (енгл. *template*) за креирање објеката/инстанци (класе)
- ✓ насљеђивање
- ✓ полиморфизам.

Ако имамо уграђене ове концепте, онда можемо рећи за систем или програмски језик да је објектно оријентисан. Као додатак овим, постоје и други кључни концепти, који су у вези са претходним:

- ✓ апстракција

Постоји више различитих врста апстракције. Апстракција је општи инклузивни (укључује више појмова, ствари, посебно, укључује своја ограничења, границе појма) концепт могућности посматрања нечега без потребе сагледавања свих детаља, могућност погледа који је простији, поглед вишег нивоа гдје детаљи нису представљени да нас не збуњују. Апстракција може бити и концепт представљања једне или више ствари, које су више конкретне у неког погледу. Апстракција је темељни принцип свих израчунавања (енгл. *computing*) и постоји од првих почетака, нпр. дјелић података у систему је апстракција стварног објекта којег тај податак представља.

Објекат – ствар која постоји у свијету, може да репрезентује нешто што постоји физички или да буде нека логичка ствар. Често се ограничавамо на поглед да су објекти подаци у рачунару, који представљају ствари које постоје у стварном свијету. Та два погледа је лијепо разграничити.

- ✓ модуларност
- ✓ инкапсулација

Апстракција и инкапсулација заједно чине принцип скривања података.

Основе програмског језика Јава

Софтверско инжењерство покрива све нивое апстракције, од захтјева високог нивоа до конкретног програмирања, јер у пракси, софтверски инжењери раде и програмирање.

Постоје схватања и погледи да софтверски инжењери раде само пројектни менаџмент, архитектура софтвера и повезане активности високог нивоа, и заиста неки то само и раде, али професија софтверског инжењера као цјелина укључује програмирање, и треба да укључује.

Софтверски инжењери треба да буду добри програмери, јер треба да доносе добре одлуке које се тичу захтјева, софтверске архитектуре и изазова пројектног менаџмента.

У *hi-tech* фирмама, великим *IT*, телекомуникационим и владиним компанијама, људи који развијају софтвер – програмирају, баве се захтјевима, пројектују и узимају учешће у пројектном менаџменту.

Најбољи пројект менаџери имају програмерску позадину/предзнање.

Преглед програмског језика Јава

Сваки пут када намјеравате да презентујете информације крајњем кориснику, увијек користите пуни скуп карактера и Јава стрингове (*Unicode* подршка), јер ако користите низове бајтова (енгл. *byte arrays*) и бајтове (енгл. *bytes*) не омогућавате оговарајућу интернационализацију, већ се ограничавате на подскуп изворних (енгл. *native*) језика, што се не сматра пожељним у данашњем свијету.

`Character` је класа која представља *Unicode* карактере, а `char` је примитивни тип података који садржи *Unicode* карактер. Потребно је водити рачуна о томе, да не дође до забуне и мјешања то двоје. Класа је корисна када желимо да позовемо метод, а примитивни тип је користан када желите неку врсту процесирања више ефикасности.

`String` је специјална класа уграђена у Јава језик (енгл. *built-in*) која садржи секвенце карактера, али је дубоко уграђена у виртуелну машину која покреће Јаву, и `+` оператор је специјално „преоптерећен" да ради са стринговима. Знак `+` је конкатенација за стрингове.

`Array` је још једана од специјалних *built-in* класа, која је пројектована да може да манипулише секвенцама произвољно изабраних (енгл. *arbitrary*) Јава типова, објеката или примитивних типова. Фиксне је величине, што треба имати на уму, јер ако их употребљавамо при развоју софтвера – стварамо могућност за нефлексибилност, односно уводимо ограничење за флексибилност. Огроман посао одржавања се ради у свијету, јер су оригинално пројектовани системи гдје су употребљени низови, тако да се трудимо да их избјегавамо, гдје је то могуће.

Зато је блага препорука користити уграђене класе које су мање ефикасне у многим случајевима, али дају више флексибилности – често се флексибилност више исплати него што кошта ефикасност.

Постоје двије класе, класа `Vector` и класа `ArrayList`; то су класе које бисте требали користити умјесто `Array` класе, увијек када мислите да постоји могућност да не желите фиксну величину нечега (број елемената или максимални број елемената). `Vector` и `ArrayList` објекти ће проширивати своју величину по потреби, а то је флексибилније.

Када манипулишете објектима који су колекције података, потребно је да можете да пролазите кроз њихове елементе. Можете да користите `for` петљу, `while` петљу или да користите специјалне класе, `Iterator` и `Enumeration`. Са `Iterator` класом имате мање шансе да направите грешке (енгл. *bugs*), не требате да имате логику која каже

„мање или једнако од максимума“ и могућност појаве *Overflow* колекције, па добијате мању ефикасност, али већу флексибилност.

Дакле, желимо да користимо класе попут `Vector` и `ArrayList` ради флексибилности, и желимо да користимо *Iterator* ради више флексилности, за разлику од *hard coding* петљи над `Array` објектима.

Кастовање је концепт који је битан *C/C++* девелоперима, а посебно у *C++* језику.

Изузеци су још једна битна ствар; постоји читава хијерархија изузетака вишег и нижег нивоа.

Класа може да наслиједи само једну класу, али зато може да имплементира произвољан број интерфејса, што значи да интерфејси имају моћ коју саме класе немају.

Можете декларисати варијаблу да буде типа интерфејса, што значи да можете смјестити у ту варијаблу инстанце било које класе која имплементира тај интерфејс; једини недостатак је то што тада можете позвати само методе те варијабле које се налазе у дефиницији тог интерфејса.

Битни уграђени интерфејси у Јави: `Runnable`, `Collection`, `Iterator`, `Comparable`, `Cloneable`.

наставиће се ...

to be continued ...