Развој софтвера 2

5. Дизајн при конструкцији

Неки аутори се слажу са ставом да дизајн у ствари није активност конструкције, али у малим пројектима се многе активности разматрају као део конструкције, што често укључује и дизајн. На неким већим пројектима, може се одредити да формална архитектура одговарати само на питања на нивоу система и да много од рада на дизајну намерно буде остављено за конструкцију. На другим великим пројектима се може одлучити да дизајн буде веома детаљан, тако да кодирање буде скоро па механичко, али је ретка ситуација да је дизајн толико комплетан, већ програмер обично (било уз званично одобрење, било без њега) дизајнира делове програма .

На малим, неформалним пројектима је много дизајна урађено док програмери седе за тастатуром. “Дизајн” може означавати то да се прво опише интерфејс класе пре него што се почнуи писати детаљи. Он може означавати цртање дијаграма за неколико класа пре него што се пређе на њихово кодирање. Дизајн може бити и питање за другог, искуснијег програмера који образац дизајна (енг. design pattern) њему изгледа као боље решење. И мали пројекти, исто као и велики, имају користи од пажљивог дизајна – без обзира на то како је тај дизајн реализован. Препознавање дизајна као експлицитне активности увећава корист коју пројекат добија од њега.

5.1. Изазови у дизајну

Фраза софтверски дизајн означава концепцију, проналазак или изум схеме за претварање спецификације за рачунарски програм у стварни, оперативни програм. Дизајн је активност која повезује захтеве са кодирањем и дебагирањем. Добар “одозго-наниже” дизајн обезбеђује структуру која са сигурношћу може садржавати већи број дизајна нижег нивоа. Добар дизајн је користан у малим пројектима и неопходан у великим пројектима.

Дизајн је такође обележен бројним изазовима који се постављају пред њега.

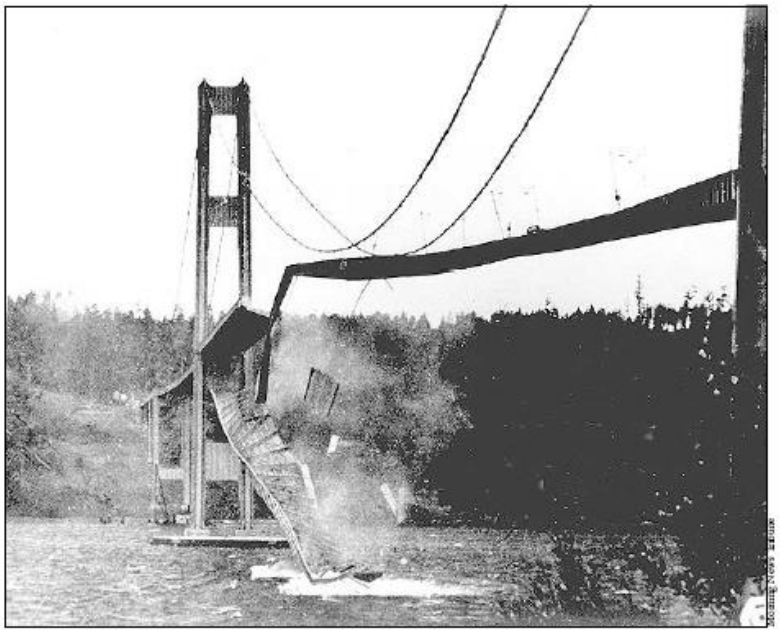
5.1.1. Дизајн је опак проблем

Рител и Вебер су 1973. године појмом “опак” (енг. wicked) проблем означили онај проблем који се може чисто дефинисати само решавањем тог проблема, или решавањем његовог дела. Овај парадокс, суштински посматрано, указује да би требало “решити” проблем да би он био јасно дефинисан и да га потом треба поново решавати како би се креирало решење које функционише. По речима Петерса и Трипа, овакав процес је практично основица развоја софтвера.

5.1. Изазови у дизајну

Драматичан пример таквог опаког проблема је био дизајн моста код Tacoma Narrows. У време градње моста, главна разматрања у дизајну моста су се односила на то да ли ће мост бити довољно јак да издржи планирано оптерећење. У случају моста код Tacoma Narrows, ветар је креирао неочекивано бочно хармонијско таласање. Једног ветровитог дана 1940. године, таласање је неконтролисано порасло све док се мост није срушио.

Ово је добар пример опаког проблема, зато што, све док се мост није срушио, инжењери нису знали да је до те мере потребно разматрати и аеродинамику. Само градњом моста (решавањем проблема) су могли да науче о додатним аспектима тог проблема, што им је омогућило да направе нови мост који се до сада није срушио.



Слика 5-1
Мост код Tacoma Narrows – пример опаког проблема

5.1. Изазови у дизајну

Једна од највећих разлика између програма који се праве у школи и програма које развијају професионалци је у томе што су проблеми који се развијају у школи веома ретко опаки.

Програмски школски задаци су осмишљени тако да се при раду директно крећемо од почетка према крају. Вероватно вам се не би свидео учитељ програмирања који постави задатак, па промени поставку пошто завршите дизајн, а онда је поново промени баш када сте скоро завршили ваш програм. Међутим, управо је такав процес свакодневна реалност у професионалном програмирању.

Слика по којој дизајнер софтвера изводи свој дизајн из захтева на рационалан начин и без грешака је сасвим ван реалности.

Ниједан систем није развијен на такав начин, а вероватно ниједан ни неће бити развијен на такав начин.

Ван реалности су чак и развоји малих програма који се приказују у уџбеницима и радовима. Ои су били прерађивани и дотеривани сва док аутор није показао како би он волео да је изгледао развој, а не како је развој стварно текао.

5.1. Изазови у дизајну

5.1.2. Дизајн је аљкав процес

Завршни софтверски дизајн треба да изгледа добро организовано и чисто, али процес који се користи при развоју таквог дизајна није ни изблиза тако чист као што је то случај са крајњим резултатом. Дизајн је аљкав зато што се предузимају многи погрешни кораци и што се иде кроз многе слепе улице – зато што се прави много грешака. Заиста, прављење грешака и јесте важно код дизајна, јер је јефтиније направити грешке и поправити дизајн, него што би било да се направи више истих таквих грешака, које се препознају нешто касније па се морају исправљати погрешке распршене кроз цео код. Дизајн је аљкав зато што се често добро решење разликује од лошег само у суптилним деловима.

Дизајн је аљкав и зато што је тешко знати када је креирани дизајн „довољно добар“. Колико детаља је довољно? Колики део дизајна треба да буде урађен коришћењем формалне нотације дизајнирања, а колико треба да остане да буде урађено за тастатуром?
Када сте готови? Будући да је дизајн са отвореним крајем, најчешћи одговор на то питање је: Када вам истекне време.

5.1. Изазови у дизајну

5.1.3. Дизајн је везан за баланс и за приоритете

У идеалном свету, сваки систем се може одмах покренути, не троши простор за складиштење, не заузима никакав пропусни опсег мреже, не садржи никакве грешке и његова изградња ништа не кошта.
У реалном свету, кључни део посла дизајнера је да извага супротстављене карактеристике дизајна и обезбеди равнотежу између тих карактеристика. Ако је брз одзив важнији од минимизирања време развоја, дизајнер ће креирати једну дизајн. Ако је минимизирање времена развоја важније, добар дизајнер ће оформити другачији дизајн.

5.1.4. Дизајн укључује ограничења

Кључни део дизајна је делом креирање могућности а делом ***ограничавање могућности***. Кад би људи имали бесконачно времена, ресурса и простора, видели бисмо невероватно проширене зграде са једном просторијом за сваку ципелу и са стотинама соба. Тако се и софтвер развија.
Ограничења ресурса за изградњу зграде терају на поједностављена решења, која ће у крајњем побољшати решење. И у софтверском дизајну, ситуација је иста.

5.1. Изазови у дизајну

5.1.5. Дизајн је недетерминистичан

Ако се тројици људи да да дизајнирају исти програм, лако се може догодити да се врате са три веома различита дизајна, где сваки од њих може да буде савршено прихватљив.
Може бити више од једног начина да се одере мачка, али обично постоје десетине начина да се осмисли рачунарски програм.

5.1.6. Дизајн је хеуристички процес

Како је дизајн недетерминистички, технике дизајна имају тенденцију да буду «хеуристика» – «правило палца» или «ствари које треба покушати а које понекад раде». Другим речима, технике дизајна нису понављајући процеси који гарантовано производе предвидљиве резултате.
Дизајн нужно укључује покушаје и грешке. Дизајнерски алати и технике која су добро функционисале на једном послу, или на једном аспекту посла можда неће радити добро на следећем пројекту. Ниједан од алата није погодан за све.

5.1. Изазови у дизајну

5.1.7. Дизајн је појавни

Бејн и Шаловеј су 1984. сумирали претходне атрибуте дизајна речима да је дизајн “појавни”. Дизајни не извиру директно из нечијег мозга као потпуно формиране целине. Они еволуирају и попревљају се кроз ревизије дизајна, неформалне дискусије, искуства писања самог кода, као и искуства ревизије самог кода.

Практично сви системи пролазе неки степен дизајнерских промена током свог иницијалног развоја, а онда се обично промене повећавају како се иде према каснијим верзијама. Степен у ком су промене корисне или прихватљиве зависи од природе софтвера који се гради.

5.2. Кључни појмови дизајна

Добар дизајн зависи од разумевања групе кључних појмова. Овај део презентације разматра улогу сложености, пожељне карактеристике дизајна и нивое дизајнирања.

5.2.1. Примарни технички императив код софтвера: Управљање комплексношћу

Да би разумели важност управљања комплексношћу, корисно је указати на веома значајан рад Фреда Брукса из 1987, под називом: „Нема сребрног метка“.

Успутне и суштинске потешкоће

Брук је истакао да је развој софтвера тежак из две различите групе проблема – ***суштинских (есенцијалних)*** и ***појавних***. При одређивању ова два појма, Брук се ослања на филозофску традицију која потиче од Аристотела.

У филозофији, есенцијалне особине су оне особине које појам мора имати да би био тај појам. Ауто мора да поседује мотор, точкове и врата да би био ауто. Ако не поседује неку од ових есенцијалних карактеристика, онда се заиста не ради о ауту.

5.2. Кључни појмови дизајна

Појавне особине су оне особине које појам има и од којих не зависи да ли је појам баш оно за шта се издаје. Аутомобил може да поседује V8 цилиндар, или 4 цилиндар са турбо пуњачем, или електромотор, а да опет буде аутомобил без обзира на тај детаљ. Аутомобил може да има двоје или четворо врата, обичне или спортске (тј. широке) точкове. Сви ови детаљи представљају успутне особине.

Брук је приметио да су најважније успутне потешкоће у софтверу одавно почеле да се решавају.

- Успутне потешкоће које су повезане са недовољно добром синтаксом језика су у великој мери елиминисане током еволуције од асемблерских језика према новим генерацијама програмских језика – од тада до данас је значај таквих потешкоћа у великој мери опао.
- Успутне потешкоће које су се односиле на недовољно интерактивне рачунаре су решене када су оперативни системи са дељењем времена заменили системе који су радили у пакетном моду.
- Интегрисано развојно окружење је додатно елиминисало проблеме у ефикасности рада програмера који произилазе из алата који су лоши у заједничком раду.

5.2. Кључни појмови дизајна

Брук је даље уочио да је много спорији напредак у превазилажењу есенцијалних потешкоћа. Разлог за то је што се, по својој суштини, развој софтвера састоји од рада на свим детаљима скупа веома компликованих, запетљаних, међусобно блокирајућих појмова. Еенцијалне потешкоће настају из потребе интеракције са сложеним и неуређеним реалним светом, из потребе да се потпуно и тачно идентификују зависности и случајеви са изузетцима, из потребе да се дизајнира решење које не сме бити само приближно тачно - већ мора бити потпуно тачно, итд.

Чак иако би могли да осмислимо програмски језик који користи исту терминологију као реални проблем који се решава, програмирање би и даље било тешко, зато што је веома изазовно прецизно одређивање како ради реални свет. Како софтвер решава све више великих реалних проблема, тако интеракције између ентитета реалног света постају све запетљаније, што заузврат повећава есенцијалне потешкоће развоја софтверског решења.

Корен есенцијалних потешкоћа сложеност, како есенцијална тако и успутна.

5.2. Кључни појмови дизајна

5.2.2. Значај управљања сложеношћу

Када прегледи софтверских пројеката означавају узроке пропасти пројекта, ретко се дешава да су технички разлози примарни узрок пропасти. Много чешће пројекти пропадају због лоших захтева, лошег планирања или лошег управљања.

Али, када се догоди да пројекат пропадне због примарно техничких разлога, тај разлог је веома често неконтролисана сложеност. Допуштено је да софтвер израсте толико комплексно да нико у ствари не зна шта он ради. Када пројекат достигне тачку у којој нико више не стварно не разуме утицај који ће промене у коду у једном делу програма имати на остале делове програма, тада се напредак успори до заустављања.

Управљање сложеношћу (енг. managing complexity) је најважнија технолошка тема у развоју софтвера. Могло би се чак рећи да је примарни технички императив у софтверу баш управљање сложеношћу.

5.2. Кључни појмови дизајна

Сложеност је нова карактеристика софтверског развоја. Пионир рачунарства Дијкстра је 1989. истакао да је рачунарство једина професија у којој је један мозак има обавезу да прошири растојање од бита до више стотина мегабајта, тј. Однос од 1 до 10^9. Он је то формулисао на следећи начин: У поређењу броја семантичких нивоа математике и рачунарства, просечна математичка теорија је скоро равна. Евоцирајући потребу за дубљим концептуалним хијерархијама, рачунар нас поставља пред радикално нови интелектуални изазов који нема преседана у историји.
Наравно, од 1989. до данас софтвер је постао много комплекснији, па Дијкстрин однос 1 према 10^9 данас лако може бити 1 према 10^{15}, или више.
Дијкстра је 1972. истакао да ничији мозак није толико моћан да обухвати целокупан модерни рачунарски програм. То значи да ми као програмери не треба да стрпамо одједном у главу цео програм, већ треба да покушамо да организујемо наш програм тако да се са сигурношћу у једном тренутку можемо фокусирати на један његов део. The goal is to minimize the amount of a program you have to think about at any one time. То се може видети као ментално жонглирање – ако је захтев да се што више менталних лопти истовремено држи у ваздуху, веће је вероватноћа да ће вам испасти нека од лопти, што доводидо грешке у дизајну и кодирању.

5.2. Кључни појмови дизајна

На нивоу софтверске архитектуре, сложеност проблема се редукује поделом система на подсистеме. Људима је лакше да разумеју неколико једноставнијих делова информације, него један сложенији део.

Циљ свих техника софтверског дизајна је да се сложени проблем разбије на једноставне делове. Што су подсистеми једноставнији, то се у датом тренутку сигурније можемо фокусирати на један аспект сложености. Пажљиво дефинисани објекти раздвајају аспекте проблема, па се у датом тренутку може фокусирати само на једну ствар. Пакети обезбеђују исту врсту користи на вишем нивоу апстракције.

Одржавање рутина кратким помаже у реудковању вашег менталног оптерећења. Исказивање програма у терминима домена проблема, а не у терминима имплементације ниског нивоа, као и рад на највишем нивоу апстракције доводи до умањења притиска на мозак.

Као закључак, програмери који компензирају (људима инхерентна) ограничења пишу програме који су и њима и другима лакши за разумевње и који имају мањи број грешака.

5.2. Кључни појмови дизајна

Како напасти сложеност

Постоје три извора за прескуп и неефикасан дизајн :

- Сложено решење једноставног проблема
- Једноставно, некоректно решење сложеног проблема
- Неадекватно, сложено решење сложених проблема

Као што је Дијкстра истакао, модеран софтвер се по својој природи комплексан, па ћете (без обзира колико се год трудили да то избегнете) на крају налетети на неки ново сложености који је иманентан самом реалном проблему. Ово сугерише двострани приступ у управљању сложеношћу:

- Минимизирање количине есенцијалне сложености такву да било чији мозак може да обради у датом тренутку.
- Спречавати непотребно повећање успутне сложености.

Када се једном схвати да су сви остали технички циљеви у софтверу секундарни у односу на управљање сложеношћу, тада многа разматрања у дизајну непосредно следе.

5.2. Кључни појмови дизајна

5.2.3. Пожељне карактеристике дизајна

Дизајн високог квалитета поседује неколико општих карактеристика. Ако се могу постићи ови циљеви, тада ће направљени дизајн заиста бити сматран веома добрим. Неки од циљева су у контрадикцији са другим, али то је изазов дизајна – креирање доброг скуша одлика које представљају баланс између циљева који се у некој мери искључују. Неке карактеристике квалитета дизајна су истовремено и карактеристике програма: читљивост, перформансе, итд. Друге карактеристике су унутрашње карактеристике дизајна.
Следи листа унутрашњих карактеристика дизајна:

Минимална сложеност
Примарни циљ дизајна треба да буде умањење сложености. Избегавајте прављење „паметног“ дизајна. Дизајн који је паметан је обично тежак за разумијевање. Умјесто тога, треба креирати „прост“ и „једноставан за разумијевање“ дизајн. Ако дизајн не допушта да се једноставно игнорише нејвећи део програма током концентрисања на један део, онда тај дизајн не ради свој посао.

5.2. Кључни појмови дизајна

Лакоћа одржавања

Лакоћа одржавања значи да се дизајнира тако да се има на уму програмер који одржава систем. Дакле, потребно је непрекидно замишљати питања о програму који се креира која би питао програмер за одржавање. Замислите да је програмер за одржавање ваша публика и дизајнирајте систем тако да буде самообјашњавајући.

Минимална повезаност

Минимална повезаност означава дизајнирање у ком се везе између различитих делова програма држе на минимуму. Користе се принципи јаке кохезије, слабог повезивања (енг. loose coupling) и сакривања информација како би се дизајнирали класе са што је могуће мање међувеза. Минимална повезаност минимизира рад на интеграцији, тестирању и одржавању.

Проширивост

Проширивост значи да се може проширивати систем а да то не оштети постојећу структуру. На тај начин се може мењати део система а да то не утиче на друге делове система. Највероватније промене треба да изазову најмању могућу трауму система.

5.2. Кључни појмови дизајна

Поновна искористивост

Поновна искористивост (енг. reusability) означава дизајнирање система тако да се могу поново искористити његови делови у дизајну других система.

Високи улазни ниво

Високи улазни ниво (енг. high fan-in) означава постојање великих броја класа које користе дату класу. Високи улазни ниво указује да је систем дизајниран тако да добро искористи помоћне класе из нижих нивоа система.

Мали или средњи излазни ниво

Мали или средњи излазни ниво (енг. low-to-medium fan-out) означава да дата класа користи мали или средњи број других класа. Високи излазни ниво (отприлике виши од седам) указује да класа користи велики број других класа и стога може бити превише сложена.

Истраживачи су открили да је принцип ниског излазног нивоа користан када се разматра број рутина које се позивају из дате рутине или из дате класе.

5.2. Кључни појмови дизајна

Преносивост

Преносивост означава дизајнирање система тако да се систем може лако пренети у друго окружење.

Мршавост

Мршавост (енг. leanness) означава дизајнирање система тако да он не садржи вишак делова. Филозоф Волтер је рекао да је књига завршена када се нема шта додати, али када се не може ништа ни одузети. То нарочито важи за софтвер, јер додатни код треба развити, прегледати, тестирати и разматрати приликом модификације осталог кода. Будуће верзије софтвера морају остати уназад компатибилне (енг. backward-compatible) са вишком кода.

Фатално је поставити питање: То је лако, па зашто онда да га не укључимо?

Стратификација

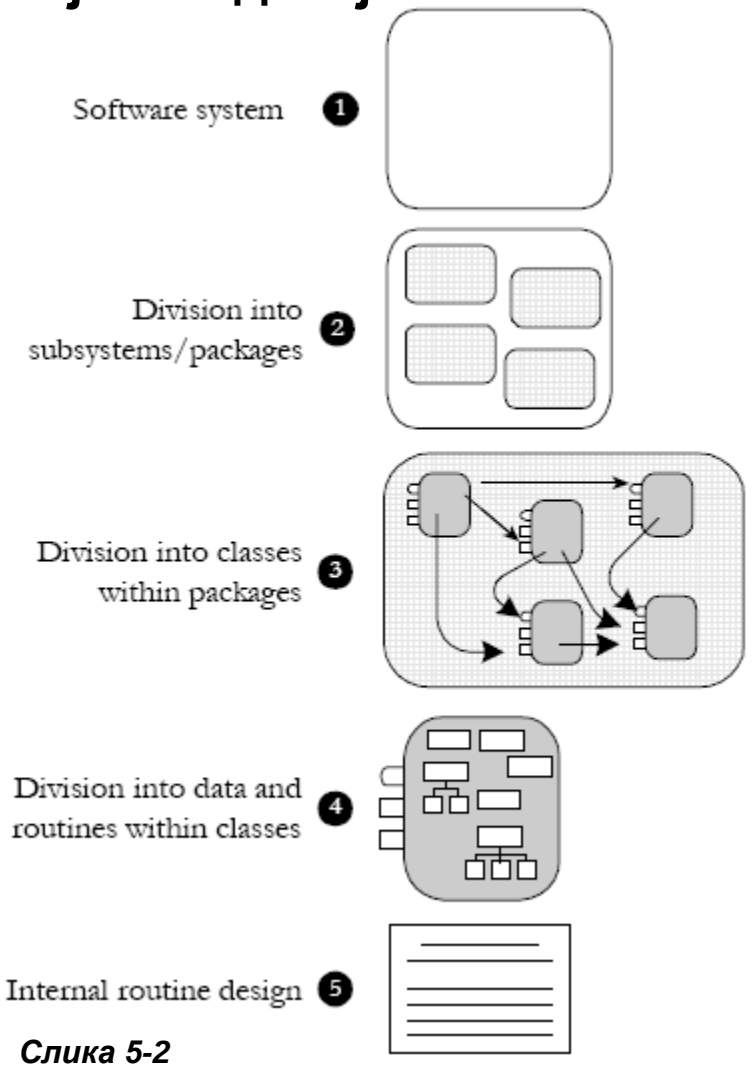
Стратификован дизајн означава покушаје да сви нивои декомпозиције буду стратификовани, тако да се цео систем може посматрати са једног нивоа и добити конзистентан поглед. Дакле, треба дизајнирати систем да се он може посматрати са једног нивоа без уласка у друге нивое.

5.2. Кључни појмови дизајна

5.2.4. Нивои дизајна

У сваком софтверском систему је потребно одредити дизајн на неколико нивоа детаљности.

Неке од техника дизајна се примењују на све нивое, а друге се примењују само на ниво или два. Слика 5-2 илуструје ове нивое.



Слика 5-2
Нивои дизајна у програму.
Систем (1) је прво организован у подсистеме (2). Подсистеми су даље подељени у класе (3), а класе су подељене у рутине и у податке (4). Такође се дизајнира и унутрашњост сваке од рутина (5).

5.2. Кључни појмови дизајна

Ако се пише модерни систем који треба да користи много старог, слабије дизајнираног кода, пожељно је написати слој новог система који је одговоран за интерфејс према старом коду. Тај слој се дизајнира тако да сакрива лош квалитет старог кода, представљајући конзистентан скуп сервиса новијим слојевима. Потом остатак система користи класе из новог слоја, а не стари код.

Корисни ефекти статификованог дизајна у оваквом случају су:

1. Он заобилази нејасноће и проблеме у старом коду
2. Ако икад дође до промена у старом коду, неће бити потребе да се модификује било шта у новом коду осим слоја одговорног за интерфејс.

Стандардне технике

Што се више систем ослања на егзотичне делове, то је више застрашен онај који први пут покушава да схвати како систем ради. Треба покушавати да се коришћењем стандардизованог, заједничког приступа целокупном систему даје познат осећај.

5.2. Кључни појмови дизајна

Ниво 1: Софтверски систем

Први ниво је целокупан систем. Неки програмери прескачу директно са системског нивоа на дизајнирање класа, али је обично корисно да се размисли о вишем нивоу комбинације класа, као што су подсистеми или пакети.

Ниво 2: Подела на подсистеме и пакете

Главни производ дизајна на овом нивоу је идентификација свих главних подсистема. Подсистеми могу бити велики – база података, кориснички интерфејс, пословна логика, интепретатор командне линије, подсистем за извештаје итд. Главна активност у дизајну на овом нивоу је одређивање како поделити програм у главне подсистеме, као и дефинисање на који се начин сваки од подсистема може користити од стране других подсистема. Подела на овом нивоу је обично потребна на сваком од пројеката који траје дуже од неколико недеља. Унутар сваког од подсистема, могу се користити различити методи дизајна, бирајући приступ који се најбоље уклапа за тај део система тј. за тај подсистем.

На овом нивоу су од највећег значаја правила о томе како могу комуницирати различити подсистеми. Ако сви подсистеми могу комуницирати са свим подсистемима, онад се губи предност њихове поделе.

5.2. Кључни појмови дизајна

Дати смисао систему тако што ће се ограничити комуникација.

Када нема правила, тада долази до изражаја други закон термодинамике и повећава се ентропија система. Један начин за повећање ентропије је да се, без икаквих ограничења у комуникацији међу подсистема, комуникација извршава неограничено.

У таквом случају, сваки од подсистема завршава тако што директно комуницира са свим другим подсистемима, што доводи до постављања следећих питања:

- Колико је различитих делова система потребно да бар мало разуме програмер, па да може нешто да мења у грфичком подсистему?
- Шта се догађа када покушавате да користите модул за финансијску аналитику у другом систему?
- Шта се догађа када желите да поставите нови кориснички интерфејс у систем, нпр. интерфејс командне линије ради тестирања?
- Шта се догађа када желите да поставите спремиште за податке на удаљену машину?

5.2. Кључни појмови дизајна

Сви претходно побројани проблеми могу да буду решени са мало додатног рада.

Треба допустити комуникацију између подсистема на бази „потребно је да се зна“ и истовремено мора да постоји добар разлог. Ако има дилема, лакше је ограничити комуникацију у раној фази, па је олакшати касније, него да се олакша у раној фази па потом да се покуша ограничити, онда када су написане на стотине позива између подсистема.

Код великих програма и код фамилија програма, дизајн на нивоу подсистема чини разлику. Ако сматрате да је ваш прогрем довољно мали да може да се прескочи дизајн на нивоу подсистема, бар ту одлуку донесите са потребном опрезношћу.

5.2. Кључни појмови дизајна

Заједнички подсистеми

Неке врсте подсистема се вишеструко појављују у разним системима. Следе неки од уобичајених.

Пословна логика

Пословна логика су закони, правила, политике и процедуре које се енкодирају у рачунарски систем. Ако пишете систем за обрачун плата, требаћ да се некдирају правила пореске управе о броју доступних улога и процени пореских стопа. Додатна правила у систему sa плате могу доћи из синдикалног уговора који одређује плаћање прековременог рада и рад током празника итд.

Кориснички интерфејс

Пожељно је да се креира подсистем ради изолације компоненти корисничког интерфејса, тако да кориснички интерфејс може да еволуира без оштећивања остатка програма. У највећем броју случајева, подсистем за кориснички интерфејс користи различите подређене подсистеме или класе за кориснички интерфејс, интерфејс командне линије, операције са менијем, управљање прозорима, систем помоћи итд.

5.2. Кључни појмови дизајна

Приступ бази података

Могуће је сакрити детаље имплементације који се односе на детаље приступа бази података, тако да највећи део програма не мора да води рачуна о прљавим детаљима манипулисања структурама ниског нивоа и да може да ради са подацима онако како се захтева на нивоу пословних захтева. Подсистеми који сакривају детаље имплементације обезбеђују вредан ниво апстракције који умањује сложеност програма. Они централизују операције над базом података на једном месту и умањују могућност грешке при раду са подацима. На тај начин они олакшавају промјену структуре базе података а да се не мења цели програм.

Системске зависности

Паковање зависности од оперативног система у посебан подсистем се врши из истих разлога због којих се пакују хардверске зависности. Ако се, на пример, развија програм за Microsoft Windows, зашто да се ограничимо само на Windows окружење? Може се одлучити да се Windows позиви изолују у подсистем за интерфејс са Windows-ом. Ако касније буде потребе да се програм помери на Macintosh или Linux, све што треба да се уради је да се промени подсистем за интерфејс.

5.2. Кључни појмови дизајна

Ниво 3: Подела на класе

На овом нивоу дизајн садржи идентификацију свих класа у систему. На пример, подсистему за интерфејс са базом података се може извршити даља подела на класе за приступ подацима, класе за перзистенцију и метаподатхе базе података. Ниво 3 са слике 5-2 показује како један од подсистема Нивоа 2 може бити подељен на класе. Ово указује да и остала три подсистема приказана на слици 5-3 у оквиру Нивоа 2 такође могу бити декомпоновани у класе.

Такође су специфицирани детаљи који се односе на начине на који свака од класа има интеракцију са остатком система – тј. на исти начин као што су класе специфициране. Конкретно, дефинише се интерфејс за класе. Главна активност на овом нивоу дизајна је обезбеђивање да сви подсистеми буду декомпоновани до нивоа детаљности који је довољно добар да се издвојени делови могу имплементирати као индивидуалне класе.

Подела подсистема на класе се обично захтева у пројектима који трају дуже од неколико дана. Ако је пројекат велики, подела се јасно разликује од поделе на Ниво 2. Ако је пројекат јако мали, може се директно са погледа на цели систем тј. Ниво 1 прећи на поглед класа тј. Ниво 3.

5.2. Кључни појмови дизајна

Класе насупрот објеката

Један од кључних концепата у објектно-орјентисаном дизајну је разликовање објеката и класа. Објекат је конкретан ентитет који постоји током извршавања програма. Класа је апстрактни ентитет који се налази у оквиру програма.

Другим речима, класа је статичка ствар која се може видети при гледању листинга програма, док је објекат динамичка ствар са специфичним вредностима и атрибутима која се може посматрати током извршавања програма.

На пример, може се дефинисати класа *Person* која има атрибуте име, узраст, пол итд. Током извршавања програма постојаће објекти nancy, hank, diane, tony итд. – то су конкретни примерци (инстанце) класе.

У наставку ће се појмови класа и објеката користити без истицања горње разлике.

5.2. Кључни појмови дизајна

Ниво 4: Подела на рутине

Дизајн на овом нивоу обухвата поделу сваке од класа на рутине. Интерфејс класе, дефинисан на Нивоу 3, ће обухватити дефинисање неких рутина. Дизајн на Нивоу 4 ће обухватити и приватне рутине класе. Када се испитују детаљи рутина унутар класа, уочава се да су многе међу њима врло једноставне, али да је мањи број композиција хијерархијски организованих рутина, што захтева додатно дизајнирање.

Рад на потпуном дефинисању рутина за класу често доводи до бољег разумевања интерфејса класе, што доводи до одговарајућих промена у интерфејсу, чиме се поново врши повратак на Ниво 3.

Овај ниво декомпозиције и дизајна се често оставља самом програмеру и он је неопходан за сваки пројекта који захтева више од неколико сати. Он не мора да се формално реализује, али захтева одређену менталну активност.

5.2. Кључни појмови дизајна

Ниво 5: Дизајн унутрашњих рутина

Дизајн на нивоу рутина се састоји од постављања детаљне функционалности за индивидуалне рутине. Дизајн унутрашњих рутина се обично оставља програмеру који ради на њима. Он се састоји од активности као што су писање псеудокода, избор алгоритма из референтних књига, одлучивање како организовати код унутар рутине и писања прогрмаског кода на изабраном програмском језику. Овај ниво дизајна се увек реализује, само што се понекад реализује лоше и непажљиво.

Дијаграм на слици 5-2 указује да је ниво на ком се дешавају ове активности Ниво 5.