София 1849, кв. Кремиковци„ тел./факс: +359 2 994 70 18, +359 882 82 66 83, +359 876 24 82 04
www.chrdri.net

ПРОЕКТИРАНЕ И КОДИРАНЕ

Дефиниции

Дейността проектиране започва след изготвянето на документа Спецификация. Тя е мост между заданието и крайното решение, което удовлетворява изискванията. Цел на проектирането е създаването на модел или представяне на системата - нарича се проект. Проектът се използува за изграждане на системата.

Ще считаме, че системата е множество от компоненти с ясно дефинирано поведение, които взаимодействуват помежду си по определен начин. Всеки компонент може да бъде изграден от своя страна от други компоненти. В една софтуерна система компонентът представлява софтуерен модул.

Процесът проектиране за софтуерните системи е на две нива. На първо ниво се решава от какви модули ще се състои системата, каква е тяхната спецификация, как те се свързват помежду си. Това е известно под името системно проектиране (top level design). На второ ниво се решава каква ще бъде вътрешната структура на отделните модули. Нарича се детайлно проектиране (logic design). При него се описват логиката на процедурите и структурите от данни. Може да се каже, че детайлното проектиране представлява разширение на системното проектиране.

Под методология на проектиране се разбира систематичен подход за създаване на проект чрез прилагането на множество от техники и ръководства. Повечето методологии се занимават със системното проектиране. Те не са формализирани и не представляват множество от стъпки, които да се следват от проектанта.

Вход на фазата проектиране е спецификацията на системата. Ето защо тя трябва да е с добро качество, недвусмислена и пълна. Изходът от системното проектиране е архитектура на системата, тя може да се създаде с или без използуването на методология на проектирането. Фазата завършва с верификация на проекта.

Проектът може да бъде обектно-ориентиран или функционално- ориентиран. При обектно- ориентираното проектиране модулите в проекта представляват абстракции на данни. При функционално- ориентираното проектиране проектът се състои от дефиниции на модули, като всеки модул изпълнява някаква абстрактна функция, свързана с естеството на разработвана система. Самата система се разглежда като съвкупност от функции, които преобразуват входа в желаните изходи. Цел на фазата проектиране е определянето на тези функции. В края на фазата проектиране стават известни всички основни модули на системата, как те се свързват помежду си и структурите от данни.

Принципи на проектирането

Това са общи принципи независимо от вида на проектирането. Един проект се нарича коректен, ако системата построена съгласно него удовлетворява изискванията. Цел на фазата проектиране е създаването на коректен проект и то най-добрия възможен проект.

Очевидно даден проект трябва да е пълен и лесно проверяем. Обаче най-важните негови свойства са ефективност (работоспособност) и простота. Ефективността на една система е свързана с подходящото използуване на ресурси от нея и оказва влияние върху цената. При

компютърните системи основните ресурси са процесорно време и памет, особено в миналото поради високата цена на хардуера. Понастоящем цената на хардуера непрекъснато пада в сравнение с тази на софтуера така, че тези съображения за ефективност вече не са основни. Единствено изключение от това правило са системите в реално време, при които има стриктно ограничение на времето за отговор.

Простотата е най-важния критерий за качеството на системата, защото влияе върху нейното поддържане. Поддържането на софтуера е скъпо, а проектът оказва най-голямо влияние върху него, защото този който ще поддържа системата най-напред трябва да разбере нейния проект, как модулите се свързват помежду си.

Тези критерии не са независими един от друг. Например един от триковете за повишаване на ефективността на системата е нейното усложняване. Затова трябва да се намери баланс между разгледаните критерии.

Създаването на прост и ефективен проект на големи системи е изключително трудна задача, защото проектирането е творческа дейност, която не може да се разбие на редица от стъпки. Могат само да се формулират принципи, които представляват основа на повечето методологии за проектиране.

Принципи на проектирането:

1.Разбиване на задачата на части и изграждане на йерархии. При големи проекти се спазва принципът "разделяй и владей", който се състои в това задачата да се разбие на малки подзадачи, които да се решават независимо. Тук трябва да се реши кои от модулите ще бъдат независими, т.е. могат да се променят без това да оказва влияние върху другите компоненти и кои модули са взаимно свързани. Разбирането на задачата води до създаването на йерархии между отделните компоненти на системата.

2.Абстракция. Абстракцията представлява метод за структуриране на данните, при който се разглеждат основни, общи свойства на дадено множество от обекти, като се изпускат несъществените детайли. Тя е важно средство, което позволява на проектантът да разглежда отделните компоненти на абстрактно ниво без да се занимава с подробностите по неговата реализация. Съществуват два механизма на абстракция при софтуерните системи:

- Функционална абстракция - модулът се задава чрез функциите, които изпълнява, например модулът за сортиране се задава чрез функцията sort, модулът за включване в системата - чрез функцията log. Функционалната абстракция е основа на разбиването на задачата при функционално-ориентираните подходи.
- Абстракция на данните - основа за нея са обектите или понятията от реалния свят, които се обработват с някакви предварително дефинирани операции върху тях. Вътрешността на обекта остава скрита за потребителя. Абстракцията на данните е основа на обектно-ориентираното проектиране.

3.Модулност - системата се счита, че е модулна, ако се състои от отделни компоненти, които могат да се реализират независимо като промяната в един от тях оказва минимално влияние върху останалите. Модулността е желателно свойство на всяка система, защото подпомага при debugging, при отстраняване на грешки и при изграждането ѝ.

4.Стратегии отгоре-надолу и отдолу-нагоре. Системата представлява йерархия от компоненти, като най-високо стоящия модул е самата система. Проектирането на йерархии се прави по два начина:

- Top-down - започва от компонентите на най-високото ниво и продължава надолу. Идентифицират се главните компоненти на системата, които след това се декомпозират – stepwise refinement, постъпково уточняване.
- Bottom-up - започва се от компонентите на най-ниско ниво и чрез тях се изграждат компонентите на по-високото ниво. Работи се с нива на абстракция.

Подходът отгоре-надолу е подходящ ако спецификацията на системата е пределно

ясна и системата се разработва за първи път. Комбинира се с каскадния модел на процеса.

Подходът отдолу-нагоре е подходящ, ако системата се изгражда като се има предвид вече съществуваща система, защото се започва от вече направени компоненти. Комбинира се с модел на процеса итеративно подобряване.

Обикновено двете стратегии не се използуват в чист вид, а се комбинират.

Функционално-ориентирано проектиране

Понятия, свързани с модулите

Модул това е логически обособена част от програма. Това е програмна единица, която може да се идентифицира по отношение на компилиране и свързване. В термините на език за програмиране програмата може да бъде макрос, функция, подпрограма или процедура, процес или пакет.

За да се получи модулен проект съществуват определени критерии за избор на модулите, за да могат те да поддържат добре дефинирани абстракции и да се променят поотделно. При системи използуващи функционални абстракции се два критерия за модулност:

1. Coupling - свързване.

Два модула се считат за независими, ако единият може да работи без другия. Обаче в една система не всички модули могат да бъдат независими един от друг, понеже трябва да взаимодействуват за да се получи външното поведение на системата. Понятието "свързване" се опитва да дефинира колко силно отделните модули взаимодействуват помежду си.

Свързване се нарича мярката за взаимна зависимост между модулите. Свързването между два модула може да бъде високо и слабо. Понеже модулите на софтуерната система се създават по време на проектирането й, свързването между модулите се решава основно по време на проектирането на системата и не може да се промени или намали по време на реализацията.

Свързването е абстрактно понятие, което не може лесно да се определи количествено. Няма формули за изчисляване на свързаността между два модула. Съществуват фактори, които й оказват влияние и те са:

- Вид на връзката между модулите - добре е информацията между модулите да се предава чрез параметри, не индиректно например чрез общи променливи.
- Сложност на интерфейса - той трябва да е възможно най-прост и малък, например не е нужно между два модула да се предава целия запис, а само необходимите полета.
- Тип на предаваната информация между модулите - между модулите се предава два типа информация: свързана с данни и свързана с контрола. Добре е между модулите да се предават само данни, защото тогава те могат да се разглеждат като прости входно- изходни функции, които трансформират данни. Интерфейси, свързани с контрола са с по-висока степен на обвързване.

2. Cohesion - сцепление.

Понятието обхваща връзката между отделните елементи на един и същ модул и определя доколко силно те са свързани помежду си. Свързването и сцеплението са се намират в обратно пропорционална зависимост.

Нотация и спецификация на проекта

Нотацията на даден проект е предназначена за самия проектант и се използува за представяне на решенията му така, че те да могат да се оценяват или променят. Обикновено нотациите са графични. След като проектантът завърши проекта, той се описва във вид на документ. За целта се използуват езици за спецификация. Разликата между нотация и спецификация на даден проект се състои в това, че нотацията се използува за да подпомогне разбирането на проекта - затова е графична, докато спецификацията го описва и има текстов

вид. Тя представлява основа за следващо развитие на проекта.

При функционално -ориентираното проектиране обикновено нотацията на даден проект се извършва чрез структурни диаграми (structure charts). Модулите се бележат с правоъгълници, а връзките между тях - чрез стрелки, като се означават какви параметри се предават. Модулите се разпределят в различни класове: входни, изходни, трансформиращи данните и координиращи - свързани с управлението. Бележат се по различен начин.

Структурните диаграми показват йерархията и връзката между модулите както и информацията, която се предава между тях. Те представляват компактна нотация за представяне на модела на системата, но не са подходящи за представяне на крайния проект, понеже не съдържат цялата информация - структури от данни, спецификация на всеки модул и др.

За да се избегнат проблеми, свързани с не еднозначното интерпретиране на структурните диаграми проектът трябва да се специфицира формално, т.е да се създаде документът Проект, който да се използува като "сламка" за детайлното проектиране и реализацията на системата.

Документът Проект съдържа следната информация:

1.Спецификация на задачата - преработва се документа Спецификация във вид, който е по-удобен за проектирането на системата.

2.Основни структури от данни - описват се основните структури от данни на разработвания софтуер.

3.Модули и спецификация на всеки от тях - основна част от спецификацията на системния проект. Описват се всички модули. За всеки модул се определя неговия интерфейс и абстрактно му поведение - какви функции ще изпълнява - какво ще прави, без други детайли свързани с реализацията.

4.Решения относно проекта - обясняват се избраните решения и съображенията за този избор.

Методология на структурното проектиране

Цел на всяка методология за проектиране е да осигури ръководство, което да подпомага проектанта при проектирането на системата. Методологията на структурното проектиране разглежда всяка софтуерна система като състояща се от някакви входове, които биват преобразувани в някакви изходи. Софтуерът се разглежда като трансформационна функция, която трябва да се проектира добре. Тази методология е функционално-ориентирана. Тя използува понятието структура на програмата. Целта е да се проектира система, като програмите, реализиращи проекта са йерархично структурирани с минимален брой връзки между тях и с функционално сцепление.

Обикновено при добре проектираните системи съществува един модул и няколко негови подчинени модули, които извършван изчисленията. Процесът на декомпозиция на даден модул до отделни атомарни модули, които да извършват изчисленията се нарича разбиване на системата.

Структурното проектиране обхваща следните стъпки:

1. Формулиране на проблема.
2. Идентифициране на входовете и изходите - цел е да се отделят вход и изход от трансформационната функция. Описва се как се конвертира входа преди за бъде подаден за трансформация и как се конвертира изхода преди да се изведе от системата.
3. Разбиване на първо ниво - определя се главния модул на системата, който координира всички дейности и неговите подчинени модули, които той извиква на първо ниво.
4. Разбиване на входа, изхода и клоновете извършващи трансформации - извършва се допълнително разбиване на модулите от първо ниво.

Евристики при проектирането

Разгледаните по-горе стъпки не свеждат процеса проектиране до набор от стъпки, които да се следват сляпо. Стратегията изисква от проектанта да проявява собствено виждане и да разглежда структурата получена след прилагането на структурното проектиране като начална структура, подлежаща на промени. За целта могат да се приложат следните евристики, които проектантът прилага или не в зависимост от разработваното приложение:

- Размер на модула - третира се като индикатор за сложността на модула; модули, които са много сложни може би не реализират единствена функция и е добре да се разбият на повече модули. От друга страна някои модули, които са много малки е добре да се комбинират. При всяко положение трябва да се отчита обвързването и сцеплението на модулите, което си остава основен водещ фактор. Обикновено се преглеждат модули, които са повече от 100 реда сорс код и по-малки от няколко реда.
- Брой вишестоящи модули (fan-in = брой стрелки влизащи в даден модул) и брой подчинени модули (fan-out = брой стрелки излизащи от даден модул). Максимизират се първите, минимизират се вторите - до 6 броя.

Основната идея на структурното проектиране е ясното разделяне на системата на три различни подсистеми - вход, изход, трансформации, които са независими една от друга и комуникират помежду си единствено чрез главния модул. Те могат да се проектират и разработват независимо една от друга. Не е необходимо да се прави диаграма на потока на данните винаги.

Другата основна идея е обработките да се структурират като се получи слабо и високо дърво, т.е. задачата да се разбие на по-малки задачи, всяка от които да се решава независимо.

Изходът от фазата системно проектиране трябва да се верифицира преди да се продължи процесът на разработване. Основните подходи за верификация са:

1.Преглед на проекта - целта е да се провери дали проектът удовлетворява изискванията и дали е с добро качество, защото при наличие на грешки цената за тяхното отстраняване е висока. Прегледът на проекта се осъществява от група хора. Най-често срещаните грешки са грешна интерпретация на дадено изискване или изпуснато изискване. Използването на списъци за проверка е много полезно. Например в тях може да се включат въпроси от вида:

- Взето ли е предвид всяко функционално изискване?
- Проектът модулен ли е?
- Форматът на данните отговаря ли на изискванията и др.?

2. Автоматичен cross-checking - чрез него се проверява да няма вътрешни противоречия в проекта, например при предаване на параметри между модулите, декларация на данните.

Съществуват формални езици за описване на проекти - например езикът PDL. Те са подходящи за машинна обработка и улесняват автоматичната проверка за вътрешни противоречия. Проектът се компилира и се откриват всички несъответствия.

Обектно - ориентирано проектиране

Дефиниции

Проектът може да бъде обектно-ориентиран или функционално- ориентиран. При функционално- ориентираното проектиране проектът се състои от дефиниции на модули, като всеки модул изпълнява някаква абстрактна функция, свързана с естеството на разработвана система. При обектно- ориентираното проектиране модулите в проекта представляват абстракции на данни. Самата система се разглежда като съвкупност от модули, които поддържат абстракция на данните. Цел на фазата проектиране е идентифицирането на обектите, които

изграждат системата, връзките и взаимодействията между тях. Преимущество на обектно-ориентирания подход е, че създаването на обектно- ориентиран модел представя най-точно проблемната област, което прави създаването на проект по-лесно.

Крайната цел на обектно-ориентираното проектиране е проектиране на модулна система, така че всеки от модулите да е лесен за разбиране и да може да се променя независимо от другите. За разлика от функционалните подходи се използува абстракция на данните, като това е ключовата разлика, която оказва влияние върху архитектурата на системата.

Понятия свързани с обектно - ориентираното проектиране.

1.Класове и обекти

Класовете и обектите са основни градивни блокове на обектно-ориентираното проектиране, подобно на функциите и процедурите при структурното проектиране.

Понятието обект произлиза от езика за програмиране Simula, разработен в Норвегия в средата на 60-те години за симулиране на различни процеси. Модулите в този език не са процедури, а физическите обекти, които се моделират при симулацията. От този език произлиза и съвременното понятие за обект като основна моделираща единица за всички елементи на реалния свят.

Обектите могат да се разглеждат като единици, които доставят някакви услуги на клиента, който може да бъде друг обект, програма или потребител. Основно свойство на всеки обект е капсулирането. Капсулирането е метод за структуриране на една система и представлява свързването в едно на данни и процедури за тяхната обработка. Съгласно този принцип системата се изгражда от модули, като достъпът до всеки модул става чрез добре дефиниран интерфейс. Прави се ясно разграничение между спецификацията на модула и неговата реализация чрез програмен код и структури от данни. Спецификацията може да се проектира и използува независимо от реализацията. Целта е да се ограничи и контролира достъпът до отделния модул.

От гледна точка на езиците за програмиране капсулирането се свързва с абстрактните типове данни. Всеки обектът се състои от две части:

- интерфейс - определя множеството от допустими над обекта операции и представлява единствената негова "видима" от потребителя част;
- реализация - състои се от:

структура данни за абстрактния тип, определяща състоянието на обекта;

функции (операции), които реализират интерфейса.

По този начин се получава ясно разграничаване между външния интерфейс към един обект и неговата реализация. Потребителите на обекти са хора или програми, които изпращат съобщения към тях. Те наблюдават поведението на обектите чрез техните операции и получените резултати - самите те обекти. Капсулирането означава, че реализацията не е видима или достъпна за потребителя на даден обект, т. е. той няма достъп до програмния код и структурите за достъп. Получава се обединяване на характеристиките на даден реален обект, описани чрез структури от данни с неговото поведение.

Капсулирането изолира приложенията от реализацията на обектите и позволява промяна на методите без да се правят промени в съответното приложение. Основно преимущество е, че потребителят не работи директно със съдържанието на обекта.

Обектите притежават състояние, поведение и идентитет. Всеки обект представлява уникално идентифицируема единица и се дефинира като множество от атрибути (instance variables) и действия (actions). Стойностите на атрибутите определят състоянието на всеки обект, което остава скрито за потребителя.

Обектите взаимодействуват помежду си като изпращат съобщения (message). При

получаване на съобщение обектът "получател" изпълнява някаква функция или операция по строго определен алгоритъм наречен метод (method).

Обобщавайки, обектите:

- са структури със собствено състояние и поведение;
- взаимодействуват помежду си изпълнявайки сложни задачи;
- комуникират чрез съобщения;
- притежават добре дефиниран интерфейс, който определя възприеманите от тях съобщения;
- имат скрито състояние.

Понятието "обект" може да се сравни с запис или структура в стандартните езици за програмиране. Съществената разлика е, че обектите могат да изпълняват операции.

Класът е съвкупност от обекти с еднаква структура и поведение. Класът може да се разглежда като шаблон-описание за множество от обекти с подобни характеристики. Атрибутите и свързаните с тях методи се дефинират еднократно за класа, а не поотделно за всеки обект. Обектите на даден клас се наричат състояние или екземпляри, като всеки от тях приема собствени стойности на атрибутите, но дели с останалите обекти имената им и методите. Класът дефинира всички съобщения, на които всеки обекта ще отговаря, както и начина на неговата реализация.

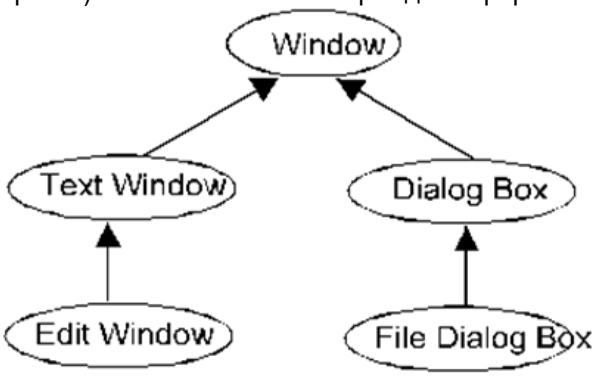
Класовете могат да се разглеждат като абстрактни типове данни. Разликата между клас и абстрактен тип на данните се състои в наследяването и полиморфизмът, които са присъщи на класовете.

2.Връзки между обектите

Обектът сам за себе си има ограничени възможности, понеже може да осигури само услугите, които са дефинирани за него. В една сложна система съществуват много обекти, които са групирани в класове. Някои обекти взаимодействуват помежду си чрез съобщения, т.е. са свързани помежду си. Два обекта са свързани (linked), ако единия от тях изисква услуга от другия. Тогава между тях съществува връзка - link. Връзка означава, че между двата обекта се предава съобщение. Два обекта могат да бъдат и агрегирани, т.е. единия да бъде част от другия.

3.Наследяване.

Това е механизъм за деление на поведение и атрибути между отделни класове, които се намират в определена връзка помежду си. Той позволява постепенното променяне на дефинициите на съществуващ клас и изграждането на негови подкласове. Новите подкласове наследяват структурите и поведението на родителя, като ги допълват или променят подходящо. Наследяването позволява даден клас да се дефинира като специален случай (подклас) на друг по-общ клас (суперклас). По този начин се изграждат йерархии от класове:



Класовете могат да се представят като върхове в ориентиран граф, чиито ребра дефинират пътищата на наследяване на свойствата. Подкласовете (извлечените класове) се наричат специализации на техните по-общи класове. Супер-класовете (основните класове) представляват обобщение на техните подкласове. Конфликтни дефиниции на поведение и

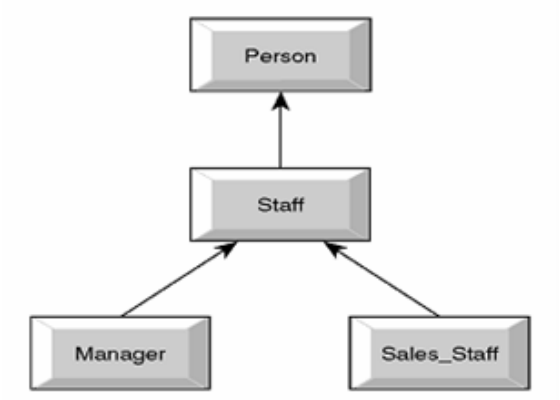
състояние могат също да се наследяват по пътищата на графа и се решават по системно - зависим начин. При създаването на подклас:

- атрибути и операции могат да бъдат добавяни към вече наследените от по-горните класове;

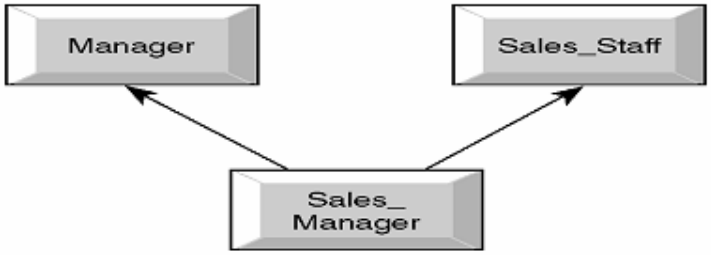
- съществуващите методи на суперкласа могат да бъдат препрограмирани за да се промени поведението на обекта; по този начин новите реализации на методите заместват (override) операциите, наследени от суперкласа.

Съществуват няколко форми на наследяването:

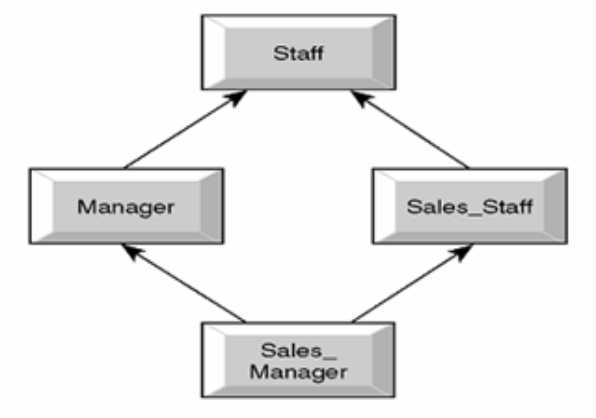
- единично наследяване (single inheritance) - всеки клас има точно един суперклас; класовете образуват йерархия



- множествено наследяване (multiple inheritance)- разрешено е един клас да има няколко суперкласа - фиг.21.6; то може да е твърде проблематично поради възможните конфликти, когато супер-класовете притежават едни и същи атрибути и методи; не се поддържа в някои обектно - ориентирани езици за програмиране и СУБД поради допълнителното усложняване



- повторено наследяване(repreated inheritance) - представлява специален случай на множественото наследяване, при който супер-класовете наследяват от общ суперклас; в примера това е суперкласа Staff; механизма за наследяване трябва да не допуска двукратно наследяване на свойствата от суперкласа Staff

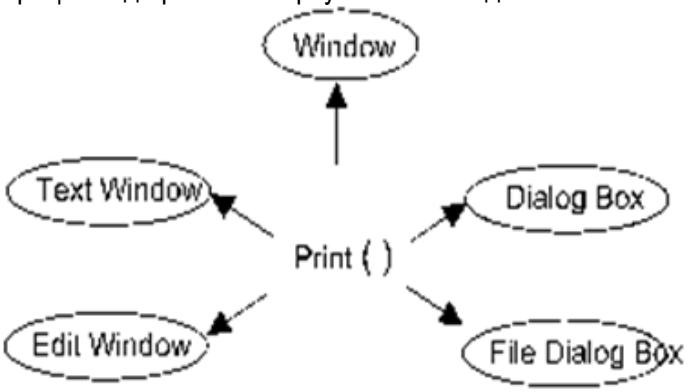


- селективно наследяване (selective inheritance) - позволява на подкласа да наследява ограничен брой свойства от суперкласа.

Наследяването допринася за повторното използуване (reuse) на обектите. То позволява даден клас да бъде дефиниран с помощта на друг клас, чрез добавянето на специфични атрибути и операции. Възможно е добавяне, заместване и премахване на наследени от суперкласа атрибути и операции. Наследяването представлява важна форма на абстракция, защото скрива детайлните характеристики на класовете и обединява само общите им свойства в суперкласа, което подпомага силно бързото и точно изграждане на приложения.

4. Полиморфизъм (polymorphism) и динамично свързване (binding).

Полиморфизъм означава възможността да се изпращат съобщения на различни обекти, които да ги интерпретират специфично в зависимост от вида си. С други думи това е възможността операциите да работят върху повече от един клас от обекти.



Свързване представлява избор между алтернативни реализации на една операция. Това означава избор на подходящ метод или методи в зависимост от вида на обекта, които да се изпълняват в отговор на подадено съобщение. Свързването на операция към метода за нейното изпълнение бива два вида:

- ранно свързване (early binding) - по време на транслация;
- късно свързване (late binding) - по време на изпълнение.

5. Специфични понятия

Разгледаните досега понятия са общи за обектно-ориентирания подход и представляват основните строителни блокове на проекта. Съществуват и специфични понятия, които се използуват за създаването на добър обектно-ориентиран проект, чийто основен компонент или модул се явява класът.

В един добър проект класовете трябва да притежават следните свойства:

- Скриване на информацията Това е метод на софтуерното проектиране, който

позволява да се скрие сложността и детайлите по реализацията на даден програмен модул от неговите потребители. При него се ограничава възможността за провеждане на изчисления над данните чрез ограничаване на достъпа до тях. Скриване на информацията означава отделяне на външните за даден обект аспекти, от вътрешните му детайли, които остават скрити и недостъпни. По този начин става възможно променянето на вътрешните за даден обект детайли без да се променят приложенията, които го използуват. Понятието е свързано с капсулирането и осигурява независимост на данните.

Абстракцията на данните и скриването на информацията опростяват разработването и поддържането на приложения чрез принципа на модулността. Обектът като елемент от класа се разглежда като черна кутия, която може да се конструира и променя независимо от останалата част на системата, при условие, че не се променя неговия интерфейс.

- Сцепление (cohesion) - това е свойство, което показва колко силно са свързани елементите на даден модул. Желателно е всеки клас да бъде със силно сцепление, т.е. всичките му елементи да поддържат добре дефинирана абстракция. Класът сам по себе си е компактен понеже данните и операциите са пакетирани заедно. Той трябва да отговаря на даден елемент от модела за да бъде с добро сцепление. Не бива да се правят класове, отговарящи на няколко понятия в модела.

Reuse. Повторната използваемост се осигурява най-вече чрез наследяването.

6.Правила за проектиране на добри и повторно използваеми класове:

- Дефиницията на данните не бива да бъде част от интерфейса на класа
- В класа се дефинират само нужните операции
- Обектите от даден клас не бива да взаимодействуват помежду си.
- Всяка операция в класа трябва да е такава, че да модифицира данните или да осигурява достъп до тях.
- Всеки клас трябва да зависи от минимален брой класове.
- Връзката между класовете трябва да е явна.
- Наследяването трябва да представя естествена йерархия между класовете.
- Броят на аргументите и размера на методите трябва да бъде малък.

Методологиите на обектно-ориентираното проектиране са много и няма да се разглеждат.

Спецификация на модулите при проектирането

Проектът може да бъде обектно-ориентиран или функционално- ориентиран. При функционално- ориентираното проектиране проектът се състои от дефиниции на модули, като всеки модул изпълнява някаква абстрактна функция, свързана с естеството на разработвана система. При обектно- ориентираното проектиране модулите в проекта представляват абстракции на данни. Самата система се разглежда като съвкупност от модули, които поддържат абстракция на данните.

Спецификацията на модулите трябва да притежава следните свойства:

- Пълнота - да описва цялостното поведение на модула
- Еднозначност - формалните спецификации са еднозначни за разлика от тези направени на естествен език
- Разбираемост - изисква се от практически съображения; за съжаление повечето формалните спецификации са трудни за разбиране и писане, което ги прави

рядко приложими

- Независимост от реализацията - спецификациите трябва да се задават абстрактно т.е. да не се използуват алгоритмични методи; трябва само да се описва външното поведение на модула. По това свойство няма съгласие между специалистите по софтуерно производство, защото никога не може да се опише модула така, че да липсват подробности от вътрешната му реализация.

При разработването на системния проект проектантът се концентрира върху спецификацията на модулите и начина, по който те комуникират помежду си. На отделния модул се дава някакво име, например Sort, което отразява неговата функционалност. В документа Проект се обяснява на естествен език какви функции се очаква да изпълнява даден модул. Тези неформални методи на спецификация могат да доведат до сериозни проблеми при кодирането, особено ако кодиращият е различен от проектанта човек. Причина за това е, че коректната реализация на модула зависи от интерпретацията на тази неформална спецификация. Дори и когато проектант и кодиращ са един и същ човек подобен проблем може да възникне, защото проектантът може да не си спомни какво е предвиждал като функции за модула.

Затова първа стъпка преди да се разработи детайлния проект или да се кодира модула е той да се специфицира прецизно.

Формалните методи за спецификация осигуряват точност на спецификациите и не позволяват модулите да се интерпретират различно. Независимо по кой от двата начина е направен проекта на системата (функционално или обектно проектиране), в крайна сметка се дефинират модули.

Спецификация на функционални модули

Най-абстрактният поглед върху един модул е той да се разглежда като черна кутия, която има някакви входове и произвежда някакви изходи. При повечето модули се налагат ограничения върху входните и изходните данни - например корен квадратен се смята от реални числа. Следователно за да се зададе външното поведение на един модул трябва да се опишат входовете, изходите и връзката между тях.

Един от методите за спецификация на модули е предложен от Hoare и използува т.нар. pre-conditions и post-conditions. При този метод ограниченията върху входа на даден модул се задават като логически израз върху входното състояние на модула наречен pre-condition, а изходите като логически израз върху изходното състояние на модула наречен post-condition. Не се задават явни връзки между входа и изхода.

Пример: да се сортира списък от L числа

Pre-condition: non-null L

Post-condition: for all k, 1 < k <size(L), L[k] < L[k+1]

Спецификацията показва, че ако входното състояние на модула sort е ненулев списък, то изходното състояние трябва да е такова, че елементите на L да са подредени нарастващо. Спецификацията не е пълна, тя например не указва, че сортирания списък трябва да съдържа всички елементи както и първоначалния. Интересно е, че тази спецификация се удовлетворява от модул, който взема първия елемент на списъка и го размножава във всички.

Спецификация на класове

Абстракцията на данните е една от най-важните характеристики на обектния подход. В езиците за програмиране тя се поддържа като абстрактни типове данни. Езици, които поддържат абстрактни типове данни са Ada и Simula. Да припомним, че абстрактен тип данни това е клас, при който няма наследяване.

Съществуват много методи за спецификация на абстрактни класове. Един от най-разпространените е аксиоматичната спецификация. При нея операциите върху класа се задават чрез аксиоми.

Детайлно проектиране

Разгледахме методите за създаване на системния проект. Повечето от тях в крайна сметка идентифицират основните модули и потокът на данни между тях. Един от начините за описание на системния проект е езикът PDL = Process Design language. Той се използува и при представянето на логическия проект на всеки модул.

Един от начините за представяне на даден проект е да се използува естествения език, но това често води до погрешното разбиране на някои от особеностите на проекта, което означава и грешно кодиране. Ако се използува формален език, като език за програмиране например, проектът се претрупва с подробности от реализацията на даден модул и това усложнява комуникирането между проектантите. Следователно е нужен език, който от една страна да не е двусмислен като естествения, а от друга - да не е толкова подробен, но лесно от него да се прави реализацията. Затова обикновено се използуват езици от типа на PDL.

PDL има синтаксис на структурен език за програмиране и същевременно използува думи от английския език.

Пример:

```
Minmax(infile)
ARRAY a
  DO UNTIL end of input
        READ an item into a
  ENDDO
  Max,min:= first item of a
  DO FOR each item of a
        IF max < item THEN set max to
        item
        IF min>item THEN set min to
        item

  ENDDO
END
```

PDL позволява да се зададе логиката на програмата с различно ниво на подробности. Най-напред се описва цялостното решение. Когато то се приеме от разработчиците, се преминава към описание на следващите нива. По този начин се прилага последователно уточняване на модулите и се избягва възможността за грешки. Структурата на PDL е такава, че неговите оператори директно се конвертират в оператори на езика за програмиране. Той съдържа всички конструкции на структурен език за програмиране, но без формално описание на условията.

Основна цел на детайлното проектиране е задаването логиката на отделните модули, които са описани в системния проект. Задаването на логиката изисква разработването на алгоритъм, който да реализира дадена спецификация.

Терминът алгоритъм е твърде общ и е приложим в много области. Характерното е, че един алгоритъм съдържа множество от стъпки, които водят до решаването на определена задача. Задачата не е непременно свързана с програмирането. От гледна точка на софтуерните технологии понятието "алгоритъм" се дефинира като недвусмислена процедура за решаването на дадена задача. Обикновено термините "процедура", "алгоритъм" и "логика" се използуват като синоними.

Няма ясна процедура как се проектират алгоритми, но обикновено процесът обхваща следните стъпки:

- Формулировка на задачата - използува се формулировката зададена в

системния проект.

- Разработване на математически модел - избират се математическите структури, които са подходящи за решаването на задачата. За целта се разглеждат други подобни задачи, които вече са решени.
- Проект на алгоритъма - определят се структурите от данни и структурата на програмата.

Обикновено се използува евристичното правило за постъпково уточняване (stepwise refinement technique), което по същество е top-down метод за разработване на детайлен проект. Обикновено се използува езикът PDL поради възможността да се задават операторите с различни нива на детайлност.

За обектно-ориентираното проектиране е необходимо да се опишат някои от особеностите на класовете. Предложения по-горе метод може да се използува за да се зададе логиката на методите в даден клас, но освен това трябва да се опише връзката между неговите обекти. Един от начините да се разбере поведението на класа е той да се разглежда като автомат с краен брой състояния!!

Верификация на проекта

Съществуват малко на брой техники за верифициране дали детайлният проект не противоречи на системния проект. Те са:

1.Design Walkthrough - това е ръчен метод за верификация и обикновено включва преглед на детайлния проект от работна група, състояща се от проектанта, друг проектант и ръководителя на групата. Преглежда се целия проект стъпка по стъпка, като се обяснява логиката на всеки модул и се прави дискусия.

2.Critical design review - целта се да се провери дали детайлния проект удовлетворява спецификациите, зададени по време на системното проектиране. Прави се от група, която освен проектанта включва проектанта на системния проект, бъдещия кодировчик и независим специалист по качеството. И това е ръчен процес, в който участвуват хора.

3.Consistency Checkers - това са компилатори, чиито вход е спецификацията на проекта направена например на PDL и които проверяват връзките между модулите, понеже техните интерфейси са зададени формално.

Кодиране и вътрешно документиране

Цел на фазата Кодиране е транслиране проекта на системата в код на избрания език за програмиране. Тази фаза оказва голям ефект върху тестването и поддържането на системата. Затова основна цел при кодирането е да се намали цената на тестването и поддържането, дори това да означава по-голяма цена на реализацията, т.е. повече работа за програмиста. Това съображение е особено важно, защото голяма част от програмистите бързат да си свършат работата без да се интересуват какво ще стане в следващите фази. При реализация винаги трябва да се изхожда от презумпцията, че програмите трябва да се конструират така, че не да са лесни са кодиране, а да са лесни за четене и разбиране.

Практики в програмирането

Цел на фазата Кодиране е транслирането на проекта в код, който е лесен за тестване, разбиране и промяна. Доброто програмиране е умение, което се добива с много практика, но все пак съществуват някои общи правила, независими от избрания език за програмиране, който могат да ръководят програмистите.

1.Имена - имената на променливите трябва да са говорещи и да отговарят на обектите, които представят. Имената на модулите трябва да отразяват тяхната функция. Лоша практика е да се използуват кодирани много къси имена, които не говорят нищо или да се използува едно и също име за много цели.

2.Конструкции за контрол - трябва те да имат единствен вход и единствен изход. Желателно е да се използуват еднотипни такива конструкции независимо, че з езика има голяма разнообразие.

3.Оператор go to - използуват се само когато трудно се моделират по друг алтернативен начин. Ако трябва да се напише go to то се предпочита скок напред в програмата, отколкото скок назад. Go to се използуват за излизане от цикъл и за извикване на процедури за управление на грешки.

4.Скриване на информацията - да се поддържа където е възможно.

5.Потребителски дефинирани типове - да се използуват възможностите на съответния език за описание на потребителски структури.

6.Влагане - да не се влагат много if-then-else конструкции, дори това понякога да не е толкова ефективно, но за сметка на това програмата е читабелна.

7.Размер на модула - да се изследват подпрограми с размер под 5 реда код и над 50 реда код. Големите модули често нямат нужното сцепление, а малките - водят до ненужно натоварване.

8.Интерфейс на модула - да не се пишат сложни интерфейси на модулите, т.е. с повече от 5 параметъра. Ако модулът е със сложен интерфейс да се намери възможност той да се разбие на части.

9.Структура на програмата - да се слагат празни места, скоби, отместване за да се подобри читабелността на програмата.

10.Странични ефекти - за страничен ефект се смята когато даден модул променя стойностите на глобалните променливи и подобни ефекти трябва да се избягват където е възможно.

Вътрешно документиране

Във фазата Кодиране изходен документ е самият код. Въпреки това той трябва да бъде вътрешно документиран за да може да бъде разбираем. Това се прави с помощта на коментари, като във всички езици за програмиране са предвидени средства за тяхното записване. Коментарите са незаменим помощник при поддържането на системата.

Целта на коментара не е да обяснява логиката на програмата. Той трябва да обяснява какво прави програмата, а не как това е реализирано. Следователно не е необходимо да се коментира всеки ред код, както това се прави от неопитните програмисти. Коментари се пишат за блокове от код и то за онези от тях, които са трудни за проследяване.

Най-полезно е да се осигури коментар за всеки от модулите, защото те представляват единицата за тестване, компилиране, верификация, и модифициране. Коментарите за даден модул често се наричат пролог на модула. Добре е да се указва следното:

- Функционалност на модула
- Параметри и цел
- Допускания за входа
- Глобални променливи, които се използуват или променят в модула

Може да се включи и друга информация, например име на автора, дата на компилиране и дата на последното обновяване.

Коментарите са полезни само ако се променят всеки път когато се променя и логиката на самия модул.