МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Український державний морський технічний університет
імені адмірала Макарова

А.Ю. Гайда

Розробка програмного забезпечення

Методичні вказівки до курсового проектування

Рекомендовано Методичною радою УДМТУ

Миколаїв 2003

УДК 004.4'2(076)

Гайда А.Ю. Розробка програмного забезпечення: Методичні вказівки до курсового проектування. – Миколаїв: УДМТУ, 2003. – 44 с.

Кафедра інформаційних управляючих систем та технологій

Викладені основи розробки програмного забезпечення з прикладами на *C/C++* і вимоги до розробки курсового проекту та виконання лабораторних робіт.

Призначено для студентів напряму "Комп'ютерні науки".

Рецензенти: д-р техн. наук, проф. К.В. Кошкін
канд. техн. наук, доц. С.В. Суслов

1. Ï Ð Î ÃÐÀ Ì ÓÂÀ Í Í ß ßÊ ÂÈÄ Ä²ßËÜ Í Î ÑÒ²

Програмування – один з видів людської діяльності. Між програмуванням та іншими видами діяльності, наприклад будівельною справою, багато спільного. При програмуванні, як і при будівництві будинку, необхідно виконати передпроектні дослідження, проектні роботи, створення і перевірку модулів, компонування, налагодження, супроводження. На кожному з цих етапів повинні бути враховані вимоги замовника, визначені будівельні матеріали та інструменти, технології та архітектура, вимоги безпеки, дизайн.

Складність створення великих проектів у програмуванні має таку саму природу, що і в будівельній справі, яка полягає в необхідності:

на кожному з етапів робіт оперувати інформацією про весь проект і його оточення або їх значну частину (системний підхід);

забезпечити погоджену взаємодію частин, з яких складається проект (інтерфейси);

передбачити можливості розширення сфери використання результатів проекту (масштабування), його можливу інтеграцію з іншими проектами;

забезпечити можливість застосування знайдених рішень і створених складових частин в інших умовах (переносимість);

забезпечити функціональність продукту протягом деякого проміжку часу (стійкість).

Так само, як і в будівельній справі, при створенні програмних продуктів існують прийоми, що дозволяють перебороти складність проблеми. Найбільш загальними з них є:

поділ проблеми на більш прості складові частини – елементи.

Це дозволяє при розробці окремих елементів зосередитися над розв'язанням частини загальної проблеми (декомпозиція);

повторне використання складових частин (складання). При цьому найчастіше можна не знати, як побудовані елементи, важливо лише знати їх властивості, поведінку і засоби взаємодії з іншими елементами;

документування. Добре документоване рішення дозволяє швидше одержувати якісні відповіді про будівлю, функціонування і властивості продукту або його частин, що важливо як на етапі його розробки, так і на етапі розвитку.

Процес програмування може бути складнішим, ніж діяльність у сфері матеріального виробництва, наприклад такій, як будівельна справа, тому, що загальні принципи, вироблені протягом тривалої історії матеріального виробництва, не завжди враховуються програмістами. Цьому є кілька основних причин:

результати праці будівельника матеріалізовані, тому він змушений більш прискіпливо стежити за дотриманням загальноприйнятих технологій. Починаючи нову справу, жоден будівельник не витратить години часу, попередньо не виконавши передпроектні дослідження (огляд ділянки, аналіз ґрунту і т.д.), проектування (ескіз, розмітка і т.п.), перевірку результатів. Програміст може недооцінити значення етапу проектування, розпочинаючи процес створення програми з написання безпосередньо програмного коду;

будівельнику недоступні засоби створення нових матеріалів, тому він змушений використовувати існуючі матеріали з відомими властивостями. Цим визначається передбачуваність результатів його роботи. Програміст під час реалізації проекту може визначати методи і принципи організації його складових частин. Зловживаючи цією можливістю, програміст уподоблюється будівельнику, що в процесі виконання роботи намагається створювати для себе нові будівельні матеріали;

велика частина роботи програміста прихована від користувача. При цьому користувач може оцінити тільки дизайн і функціональність програми, а не архітектуру і технології, що використовувались при її створенні. Це ускладнює об'єктивну оцінку якості програмних продуктів;

між програмуванням і будівельною справою є істотна відмінність – кожну програму необхідно проектувати окремо, у той час, коли за одним проектом може бути побудовано кілька будинків.

З цих причин проекти з розробки програмних продуктів закінчуються невдачею набагато частіше, ніж проекти в будівництві.

Виявлення загальних принципів організації проекту в програмуванні і таких традиційних галузях людської діяльності, як, наприклад, будівельна справа, дозволяє застосовувати в програмуванні апробовані рішення з цих галузей.

2. ÅÒÀÏ È ÐÎÇÐÎÁÊÈ ÏÐÎÃÐÀÌ Í ÎÃÎ ÇÀÁÅÇÏ Å×Å Í Íß

2.1. Аналіз

Проекту передує аналіз, у результаті якого повинні бути визначені доцільність його виконання, границі розв'язуваної проблеми, сформульовані задачі і вимоги для подальшого проектування. Необхідно аналізувати:

напрямок і галузь застосування розробки; предмет, мету і задачі розробки; актуальність і значущість теми; існуючі в даній галузі рішення та інструментальні засоби; технічні вимоги та експлуатаційні характеристики, обмеження; доцільність розробки.

Якщо результати аналізу підтверджують можливість і доцільність реалізації проекту, формується технічне завдання, яке визначає основні вимоги до проекту, функціональні та експлуатаційні характеристики, архітектуру тощо.

2.2. Проектування

Створення моделі. Проект починається з розробки моделей. Модель дозволяє простими способами ще до реалізації проекту отримати наочну інформацію про необхідні складові проекту і відносини між ними, їх властивості і поведінку. Модель будується з абстракцій шляхом виділення істотних для моделі понять і відносин між ними і приховання несуттєвих. У свою чергу, для отриманих складових проекту можуть бути розроблені уточнення – нові моделі. Абстракції, впорядковані за рівнем уточнення, утворюють ієрархію абстракцій. Відносини між окремими абстракціями реалізуються за допомогою інтерфейсів [3]. Інтерфейс – це набір операцій, що використовується для специфікації послуг, якими користуються абстракції, тому абстракції повинні бути сумісні з

одним набором інтерфейсів і забезпечувати реалізацію будь-якого іншого [4].

Проектування – це процес розробки, аналізу і опису моделей з метою подання проблеми з області знань, пов'язаних з розв'язуваною задачею предметної області, найбільш зрозумілими і близькими розробнику засобами.

Відповідно до поняття ієрархії, належність абстракції до визначеного рівня в ієрархії абстракцій визначає ступінь деталізації задач, пов'язаних з елементом, для якого створена абстракція. Чим вище розташована абстракція в ієрархії абстракцій, тим більш загальні задачі вона має розв'язувати.

Послідовність уточнень абстракцій дозволяє:

скоротити кількість інформації для аналізу, тобто сформулювати задачу тільки з необхідними і достатніми для її розв'язання початковими умовами;

розглядати кожну абстракцію як незалежну проблему, для якої можуть бути отримані нові абстракції;

розглядати кожну проблему окремо, що дозволяє удосконалювати її розв'язання незалежно від інших проблем.

Виділення складових проблеми і розробка для них абстракцій як правило відокремлені від процесу їх застосування при складанні, тому важливо забезпечити такий внутрішній склад абстракцій, для якого на етапі реалізації абстракцій може бути виконаний максимально можливий обсяг робіт, а на етапі складання – мінімальний. Іншими словами, при виділенні абстракцій необхідно забезпечити такий набір складових у кожній з них, що зв'язки для складових окремої абстракції між собою (з'вязаність) відносно сильні, а зв'язки між окремими абстракціями (зчеплення) відносно слабкі.

Виконання цих двох умов дозволяє одержати відповідно цілісні та відокремлені абстракції, забезпечити між ними найбільш прості (вузькі) інтерфейси і обмежити необхідну функціональність для кожної окремо взятої абстракції. У свою чергу, це призводить до завершеності і самодостатності абстракцій, відсутності або зменшення їх небажаного взаємного впливу (побічних ефектів) і збільшенню можливостей повторного застосування.

Графічно ієрархія абстракцій може бути зображена у вигляді дерева, вузлами і листям якого є отримані абстракції, а гілками – зв'язки між ними (рис. 2.1). Застосування механізму уточнень при

одержанні ієрархії абстракцій визначає напрямок "зверху – донизу" (від загального до частини).

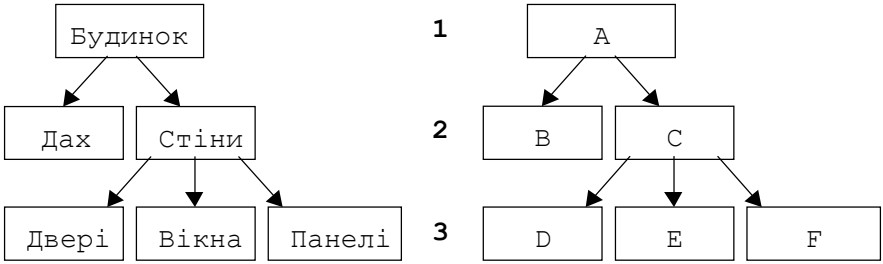
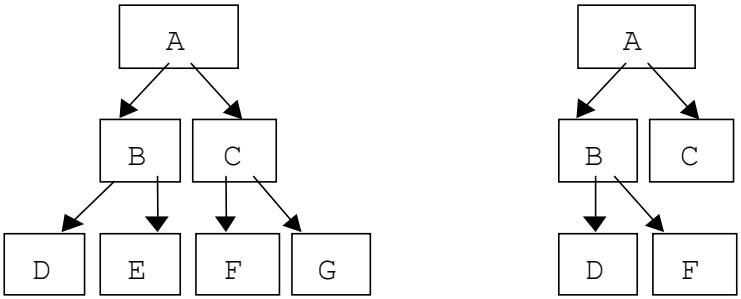


Рис. 2.1. Діаграма абстракцій

Виділення абстракцій з окремої проблеми і визначення зв'язків між ними є найбільш складною задачею проектування в тому, що необхідно визначити ступінь розподілу і ті суттєві ознаки, за якими окремі складові загальної проблеми можуть бути об'єднані в абстракції. Розподіл проблеми на більше число дрібних частин дозволяє одержати сильну зв'язаність для окремих абстракцій і слабке зчеплення з іншими, але призводить до збільшення кількості абстракцій та інтерфейсів між ними. Наочною ілюстрацією труднощів виділення абстракцій є електронні прилади: виріб на основі одного технологічного елементу (плати) є більш простим, але його налагодження і розширення функціональності майже завжди будуть складнішими, ніж для багатоелементних приладів.

У зв'язку з тим, що розподіл проблеми на абстракції виконується з метою полегшення сприйняття проблеми людиною, необхідно враховувати особливості людського мислення. Дослідження показують, що людський мозок при розв'язанні незалежної проблеми досить просто може оперувати в середньому сімома різними поняттями (тобто, у даному контексті, людський мозок можна розглядати як сім незалежно мислячих агентів, кожний з яких зайнятий розв'язанням власної проблеми). Виходячи з цього, середня кількість абстракцій одного рівня (ступінь розпаралелювання), що складають окремо взяту проблему, повинна бути близько семи, а для проблеми, що вимагає для свого повного розв'язання N абстракцій, буде потрібно близько $k = \log_7 N$ рівнів абстракцій. Це значення буде справедливим, якщо при розподілі проблеми на абстракції може бути забезпечена відносно однакова складність окремих абстракцій. При такому розподілі дерево, що представляє діаг-

рамму абстракцій, буде збалансованим (симетричним і насиченим) і найнижчим із усіх можливих (рис. 2.2).



а ***б***

Рис. 2.2. Приклад збалансованого (***а***) і незбалансованого (***б***) дерев абстракцій

Діаграму абстракцій (див. рис. 2.2,***а***) можна також розглядати:

як карту розподілу відповідальності (керування). Абстракції *B* і *C* належать до зони відповідальності абстракції *A* і самі несуть відповідальність за підпорядковані їм абстракції;

як схему інформаційних потоків. З цієї схеми видно, що інформаційний потік у зоні відповідальності будь-якої абстракції може забезпечуватися тільки цією абстракцією і характеризує підпорядкування однієї абстракції іншій;

як схему побудови абстракцій.

Діаграма абстракцій не відображає з'вязаність абстракцій (складові *B* і *C* абстракції *A*; складові *D* і *E* абстракції *B* і т.п.) і зчеплення між ними, тому що не несе інформації про потужність інформаційних потоків між абстракціями та їх складовими.

Взаємні зв'язки між абстракціями і місце абстракцій у дереві ієрархії визначають перелік задач, розв'язання яких вони повинні забезпечувати: аналізувати стан і керувати підлеглими абстракціями; надати інформацію для аналізу керуючим абстракціям; забезпечити, якщо це необхідно, інформаційні потоки для взаємодії підлеглих абстракцій; виконувати, якщо це необхідно, деякі дії, спрямовані на розв'язання задач, що не стосуються підпорядкованих абстракцій. Для абстракцій, які не мають підпорядкованих, ця задача є головною.

Як випливає з визначення функціональності абстракцій, засоби забезпечення інформаційних потоків (інтерфейси) повинні бути присутніми у всіх абстракціях, а для деяких абстракцій забезпечення таких засобів може бути головною чи навіть єдиною метою їх створення. Використання на різних рівнях ієрархії абстракцій, що не мають підпорядкованих, призводить до незбалансованості дерев абстракцій (див. рис 2.2,***б***). Інтерфейси забезпечують керування і передачу даних між абстракціями і реалізуються через функції: вхідний потік – через виклик функції і передачу аргументів, а вихідний – через значення, яке повертає функція.

Встановлення безпосереднього зв'язку між абстракціями, відмінного від зв'язку підпорядкування, суперечить принципам, викладеним вище, призводить до збільшення кількості необхідних інтерфейсів і тому є небажаним. Можливості використання і принципи організації інтерфейсів між абстракціями, відмінних від зв'язку підпорядкування, будуть розглянуті нижче.

Декомпозиція та ієрархічне впорядкування абстракцій є потужними засобами подолання складності проблем. Такі додаткові витрати для їх забезпечення, як виділення абстракцій і розробка інтерфейсів, поліпшують розуміння сутності проекту і проблем, що його складають. Модель не розв'язує проблем, а є засобом, який дозволяє одержати якісне рішення з меншими витратами.

Процедурний і об'єктно-орієнтований підходи. При розв'язанні окремої проблеми можуть бути обрані два полярні і практично взаємовиключні підходи (технології). Можна розв'язувати проблему самостійно (для цього необхідно визначити перелік дій, які необхідно виконати і пасивні сутності, над якими або за допомогою яких ці дії повинні бути виконані) або запросити групу фахівців (агентів) та відкрити кожному з них суть розв'язуваної ним проблеми, а також сформулювати задачі, вимоги й обмеження.

Перший підхід обирають при розв'язанні простих проблем ("простіше зробити самому, ніж пояснити"). При його реалізації оперують наказами (дієсловами), а одиницею поділу проблеми (втіленням абстракції) є функція. Це процедурний підхід.

Другий підхід обирають при розв'язанні складних проблем. При його реалізації оперують іменами агентів (іменниками), а одиницею поділу проблеми є об'єкт. Це об'єктно-орієнтований підхід.

Навіть таке поверхневе визначення розбіжностей у процедурному (ПП) й об'єктно-орієнтованому (ООП) підходах дозволяє

зробити висновки про можливості їх використання у світлі повторного застосування результатів розробки. Витрати на навчання агентів будуть виправдані тільки при необхідності багаторазового розв'язання аналогічних проблем. У протилежному випадку застосування ООП може тільки уповільнити процес розробки.

Обрана технологія визначає принципи, за допомогою яких буде відбуватися виділення абстракцій із складної проблеми, і тому розходження для різних технологій виявляються, насамперед, на етапі проектування. На етапі реалізації проекту ці розходження виявляються слабкіше, а на рівні обраних інструментів вони малопомітні. Так відбувається тому, що в будь-якому випадку рішення проблеми полягає в послідовному виконанні деяких корисних дій, що можуть бути реалізовані тільки через накази (функції).

Як критерій оцінки ефективності кожного з підходів можуть розглядатися різні характеристики процесу реалізації проекту і володіння отриманими результатами, але в остаточному підсумку найбільш важливим критерієм є час, необхідний для одержання, супроводження і розвитку кінцевого продукту, відшкодування витрачених на його створення ресурсів. Тому витрати часу на реалізацію проекту і його супровід повинні, в першу чергу, визначати вибір на користь тієї чи іншої технології розробки.

Алгоритми і організація даних. Як і в інших сферах діяльності, у програмуванні для досягнення мети можуть бути обрані різні методи і засоби. Методи визначають необхідні алгоритми, а засоби – організовані дані. Задача програміста – знайти ефективні шляхи розв'язання проблеми, тому вибір алгоритмів і організація даних є важливим етапом розробки програми.

Програміст рідко стикається з необхідністю розробки нових алгоритмів: як правило, подібні задачі вже розв'язувалися, і для них були знайдені ефективні методи розв'язання. При виборі алгоритму необхідно оцінити час його виконання, кількість споживаної пам'яті і складність структур даних. Принципи розробки нових алгоритмів і аналіз їх ефективності виходять за рамки даного матеріалу [2, 5, 6, 8].

Дані, що використовуються в програмі, характеризуються:

типом – множиною допустимих значень і набором операцій, що можуть бути виконані над ними. Тип повинен бути найбільш простим з можливих, у цьому випадку над даними буде допустиме виконання найменшого набору операцій;

часом життя – терміном, протягом якого значення даних, отримані на деякому етапі виконання програми можуть бути використані. Значення даних, отримані під час виконання програми, будуть загублені після її завершення. Якщо необхідно забезпечити більший час життя для отриманих значень, дані можуть бути збережені в файлах або базах даних. Час життя повинен бути найменшим із усіх можливих, у цьому випадку в кожний момент виконання програми будуть існувати тільки ті дані, які необхідні, і не більше;

областю видимості – частиною програми, в якій значення даних можуть бути використані. Область видимості повинна бути найменшою з усіх можливих, у цьому випадку в кожний момент виконання програми будуть досяжні тільки ті дані, що необхідні, і не більше;

складністю. У програмі рідко зустрічаються дані примітивних типів. Такі дані звичайно використовують для лічильників при організації циклів або збереження деякого логічного стану програми для наступного аналізу (прапорів). Складність даних повинна відповідати складності сутностей або понять, що представляють дані, – інформація про одну сутність або поняття повинна бути згрупована в одному екземплярі даних.

Групування даних є ключовим моментом їх організації і дозволяє забезпечити вузькі інтерфейси між елементами програми і взаємну несуперечність даних. Вузькі інтерфейси формуються завдяки можливості передавати у функції меншу кількість аргументів при використанні більш складних типів даних, а несуперечність визначається такою організацією даних, коли будь-яка інформація про деякий екземпляр сутності або поняття в програмі зберігається в єдиному екземплярі даних. Принципи, на основі яких відбувається групування даних, залежать від алгоритмів, що застосовуються для їх обробки, і, в свою чергу, самі можуть визначати вибір тих чи інших алгоритмів. Дані, що представляють набір однотипних сутностей або понять, повинні бути організовані у вигляді масиву. Цей принцип організації даних дозволяє значно скоротити розмір програмного коду і підвищити його ясність за рахунок використання циклів. Дані, що описують складні сутності або поняття повинні бути організовані в структури мови *С* для ПП. При ООП дані повинні бути організовані в структури, якщо об'єкти чи поняття, що представляються ними, не мають власної

поведінки, і в класи, якщо така поведінка може бути визначена. Варіантні дані, тобто такі, які можуть мати різні типи або інтерпретацію в залежності від контексту програми, повинні бути організовані в об'єднання.

Як ілюстрацію вищесказаного розглянемо приклад – точку на площині, координати якої задані в декартовій системі координат.

У зв'язку з тим, що дві координати належать сутності, що не має власної поведінки, визначимо структуру:

```
struct Point {
    int x;       //координата по x
    int y;       //координата по y
};
```

Для визначення трьох незалежних точок використовуємо масив:

```
struct Point point[3];
```

Трикутник визначається структурою:

```
struct ThreeAngle {
    struct Point a;  //вершина A
    struct Point b;  //вершина B
    struct Point c;  //вершина C
};
```

У залежності від контексту, три точки можна розглядати як незалежні або як вершини трикутника:

```
union ThreePoint {
    struct Point point[3];          //бачимо три точки
    struct ThreeAngle threeangle;   //бачимо трикутник
};
```

Засобом реалізації алгоритмів у *C/C++* є керуючі структури, які можна розглядати як іменовані (для функцій) чи безіменні (для блоків коду) фрагменти програмного коду, для яких визначене єдине місце входу і єдине місце виходу. Над керуючими структурами допустимі три види дій: конкатенація, вибір і повторення. У зв'язку з тим, що для оператора *goto* може бути визначено мітку переходу в будь-якому місці програмного коду функції, його застосування порушує визначені вище вимоги і тому неприпустиме. Функції при ПП визначають дії над даними, а при ООП – поведінку об'єктів.

Викладені вище підходи і принципи, що використовуються при виділенні абстракцій, визначають наступні вимоги до функцій:

для ПП функції, що використовують дані, повинні нести відповідальність за їх коректність. Засобом забезпечення захисту даних є приховання даних від сторонніх функцій. При ПП єдиним засобом приховання даних є розміщення їх у функціях;

для ООП функції, що визначають поведінку об'єктів, повинні застосовуватися тільки в контексті цих об'єктів. При ООП єдиним засобом забезпечення контексту є розміщення функцій в об'єктах.

На підставі вищесказаного можна зробити висновок: ніяка програма не повинна містити загальнодоступні (глобальні) дані, а програма, написана з застосуванням ООП не повинна містити загальнодоступні (вільні) функції. Застосування макровизначень і константних виразів не суперечать наведеному положенню, тому що їхні значення не можуть бути змінені в ході виконання програми. Взагалі, додавання однієї глобальної змінної еквівалентне додаванню інтерфейсів, кількість яких дорівнює кількості абстракцій, в області видимості яких знаходиться глобальна змінна. У зв'язку з тим, що ці інтерфейси не можуть бути описані – транслятор не може забезпечити контроль за їх використанням. З іншого боку, використання глобальних змінних дозволяє звузити існуючі інтерфейси. Глобальні змінні можуть бути присутніми у значних за обсягом проектах як змінні оточення, розгляд прийомів їх використання виходить за рамки даних методичних вказівок.

Ще одним засобом забезпечення приховання для даних і контексту для функцій є модульна побудова програми, але у зв'язку з відсутністю однозначних принципів побудови модулів даний засіб розмежування повноважень використовується рідко і, як правило, тільки при реалізації значних за обсягом проектів.

Важливим наслідком наведених положень є можливість формування такої структури програми, коли додавання нових складових програми при її подальшому розвитку не буде впливати на існуючі абстракції або такий вплив може бути зведений до мінімуму.

Процедурний підхід. При проектуванні програми з застосуванням процедурного підходу необхідне виділення абстракцій за функціональними ознаками на основі аналізу послідовностей дій (алгоритмів), які необхідно виконати для досягнення мети (алгоритмічна декомпозиція). Поетапне уточнення цих дій дозволяє виді-

лити необхідні для розв'язання задачі керуючі структури (процедури і функції), і пасивні сутності (структури даних). Такий підхід у проектуванні одержав назву "структурне проектування зверху-донизу" (рис. 2.3).

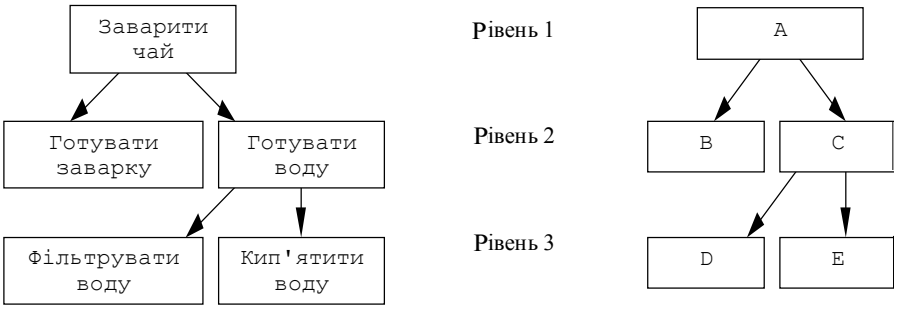


Рис. 2.3. Уточнення дій при структурному проектуванні

Існує досить вузьке і специфічне коло задач, де можуть бути використані елементи "структурного проектування знизу-догори". Такий підхід, як правило, застосовують у тому випадку, якщо від можливості реалізації деякої обов'язкової складової частини проекту залежить доля всього проекту. Наприклад, при розробці менеджера файлів з керуванням голосом, найскладнішим елементом (на поточний час) буде звуковий аналізатор. Очевидно, що починати проект необхідно з розробки саме цього елемента. Після його успішної реалізації можна використовувати підхід "структурне проектування зверху–донизу".

На основі принципів, визначених у підрозділі "Алгоритми і організація даних" (див. с. 10) і прикладу моделі, наведеної на рис. 2.3, можуть бути визначені вимоги до локалізації структур даних в обробляючих їх функціях:

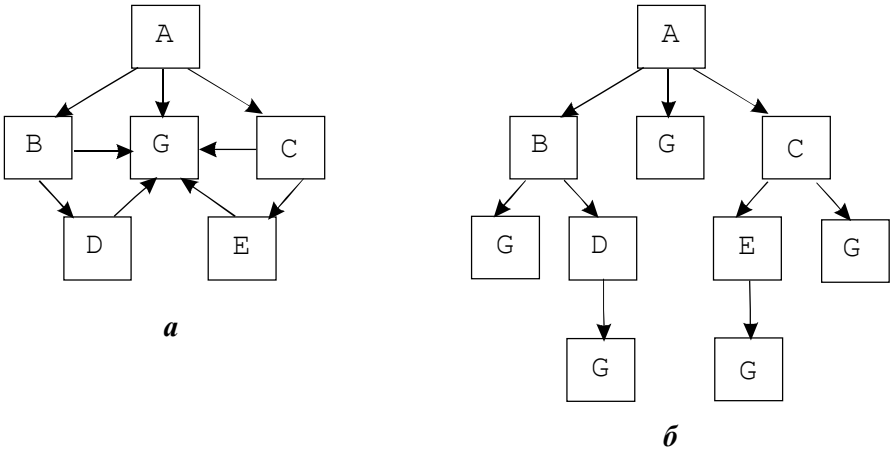
1. Як відповідальна за зону використання даних повинна вибиратися абстракція, яка визначає необхідну зону найменшого розміру. Таке розміщення відповідає вимогам найменшої області видимості (використання). Наприклад, якщо дані повинні використовуватися абстракціями *D* і *E* (див. рис. 2.3), відповідальною за зону повинна бути призначена абстракція *C*, а не *A*;

2. Екземпляри даних, що використовуються в зоні відповідальності деякої абстракції, можуть бути визначені:

в цій абстракції. Для доступу до екземпляра даних його копія у вигляді аргумента функції (при доступі для читання) або адреси (при доступі для запису) передаються в підпорядковані абстракції;

в одній із підпорядкованих абстракцій. У цьому випадку абстракція, яка містить ці дані, повинна надати їхню копію (для читання) або адресу (для запису) абстракції, що відповідає за зону. Екземпляри даних повинні мати статичний клас пам'яті або зберігатися в динамічно розподіленій пам'яті ("купі").

Виділені в результаті алгоритмічної декомпозиції абстракції можуть бути повторно використані на різних рівнях уточнення, що дозволяє значно скоротити час на розробку програмних продуктів. З іншого боку, повторне використання абстракцій ускладнює одержання ієрархічно впорядкованих відносин між абстракціями і розмежування рівнів абстракцій (рис. 2.4) (абстракція *G* використовується на всіх рівнях, тому не може бути віднесена до жодного з них).



а ***б***

Рис. 2.4. Дерево абстракцій при алгоритмічній декомпозиції

Складності опису інформаційних потоків деякою мірою можуть бути переборені розмежуванням рівнів абстракцій (див. рис. 2.4,***б***), але отримана таким чином модель стає досить складною для розуміння.

Цими причинами визначаються труднощі або повна неможливість застосування структурного підходу для реалізації складних програмних проектів.

***Об'єктно-орієнтований підхід* (*ООП*)** дозволяє перебороти проблеми проектування, характерні для процедурного підходу, завдяки наступним нововведенням: структури даних і функції для їх обробки інтегровані в новій концепції об'єкта; повторне використання функцій і структур даних не породжує проблеми розмежування абстракцій завдяки введеному механізму успадкування.

Ці нововведення як концепції об'єктно-орієнтованого підходу забезпечуються такими базовими механізмами мов об'єктно-орієнтованого програмування, як об'єкт, інкапсуляція, успадкування і поліморфізм.

Об'єкт можна розглядати як поняття чи сутність, яким можна оперувати [1, 7]. Дані, що складають об'єкт, визначають його стан, а функції – поведінку. Кожний об'єкт відноситься до певного класу об'єктів, за допомогою якого визначається структура і поведінка об'єкта. Об'єкт повинний поводитися певним чином, може зберігати або змінювати свій стан, керувати або бути керованим іншими об'єктами.

Інкапсуляція є засобом приховання внутрішньої будови об'єкта від сторонніх об'єктів з метою надання тільки необхідної та істотної на даному рівні ієрархії інформації (досягається формуванням найбільш вузьких інтерфейсів) та забезпечення цілісності і несуперечності даних, що входять до об'єкта (досягається захистом даних від використання з боку сторонніх функцій і визначенням спеціальних функцій для створення і руйнування об'єкта – конструкторів і деструктора).

Склад об'єкта може бути визначений з ієрархії абстракцій, що визначає підпорядкування одних об'єктів іншим – ієрархії об'єктів (рис. 2.5).

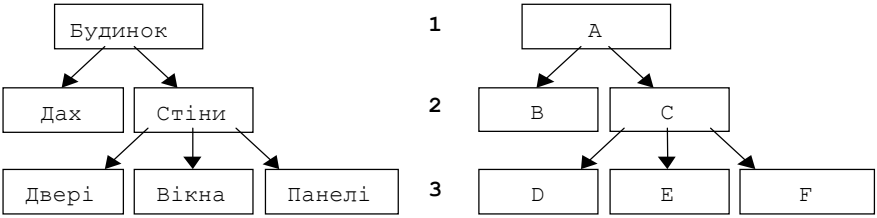


Рис. 2.5. Ієрархія об'єктів

Аналіз цих положень і положень, наведених у підрозділі "Алгоритми і організація даних" (див. с. 10) дозволяє зробити наступні висновки:

у відкритій частині класу повинні знаходитися тільки функції, що визначають виключно інтерфейс класу;

у захищеній частині класу мають знаходитися тільки ті функції, які повинні бути успадковані, і дані, що використовуються цими і інтерфейсними функціями;

у приватній частині класу повинні знаходитися інші функції (що визначають специфіку класу) і дані, що використовуються цими функціями;

локальними у функції повинні бути дані, які використовуються тільки в цій функції.

Успадкування є механізмом, за допомогою якого більш спеціалізовані типи елементів запозичують структуру і поведінку загальних (базових) типів елементів. Підвищення спеціалізації відбувається шляхом послідовності уточнень, який може бути поданий у вигляді ієрархії успадкування. Базовий клас називають предком або надкласом, а класи, що успадковуються від нього, – нащадками або підкласами. На відміну від ієрархії об'єктів, що відбиває відносини між об'єктами (абстракціями), ієрархія успадкування відбиває відносини між їх типами. Графічно ієрархія успадкування може бути зображена у вигляді дерева (рис. 2.6) і отримана в такий спосіб:

1. З ієрархії об'єктів як початкового набору абстракцій (див. рис. 2.6,***а***) визначається перелік абстракцій, що не повторюються (див. рис. 2.6,***б***). Кожній з неповторних абстракцій привласнюється тип об'єкта. Групи повторюваних об'єктів будуть належати до одного типу.

2. Отриманий перелік типів об'єктів аналізується на спільність поведінки і властивостей. На основі аналізу виділяються групи родинних типів об'єктів (див. рис. 2.6,***б***).

3. Для кожної отриманої групи родинних типів об'єктів створюється базовий тип, який визначає поведінку і властивості, загальні для групи типів об'єктів (див. рис. 2.6,***в***).

4. Отримані типи об'єктів зазнають повторного групування (п. 2) і типізації (п. 3) доки можуть бути виділені нові типи (див. рис. 2.6,***г***).

Визначення родинних відносин між різними типами об'єктів може бути отримане таким чином:

1. Для кожного класу необхідно скласти перелік даних і функцій (як спрощений варіант – тільки інтерфейсних функцій);

2. Групувати в родинні типи такі класи, для яких множини даних і функцій перетинаються найбільшою мірою (рис. 2.7).

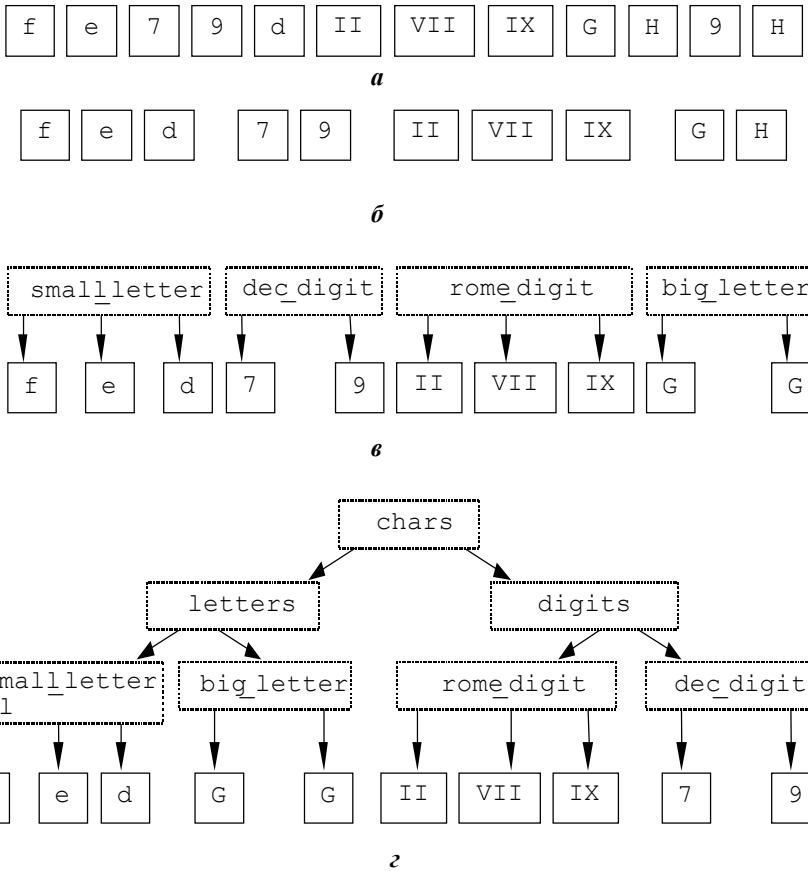


Рис. 2.6. Схема одержання ієрархії успадкування (типів)

Застосування механізму узагальнень при одержанні ієрархії типів визначає напрямок "знизу–догори" (від часткового до загального). Ієрархія успадкування відбиває два способи одержання типів об'єктів:

уточнення (показане у визначенні механізму успадкування). використовується в тому випадку, коли програміст має у своєму розпорядженні готові типи об'єктів і на їхній основі створює похідні типи для наступного використання в проекті;

узагальнення (показане в способі одержання ієрархії успадкування) використовується в тому випадку, якщо програміст самостійно проектує необхідні типи, можливо з обмеженим використанням готових типів.

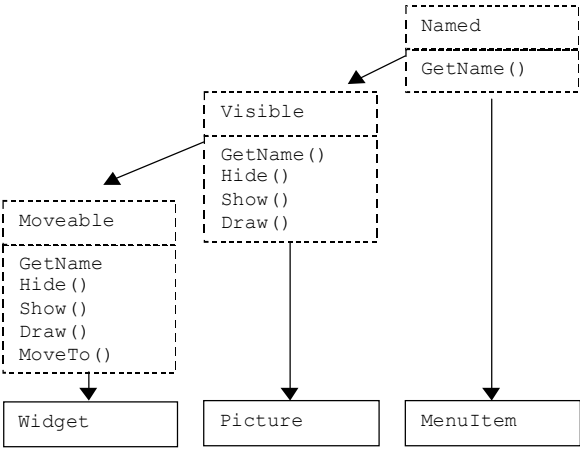


Рис. 2.7. Схема визначення родинних відносин

У результаті узагальнень будуть отримані нові типи, деякі з яких не повинні бути реалізовані у вигляді екземплярів об'єктів (абстрактні типи). Абстрактні типи повинні містити чисті віртуальні функції для поведінки, що має бути присутньою у всіх похідних типах, але не може бути визначена в базовому типі.

В *C++* може бути застосоване множинне успадкування (рис. 2.8), але його застосування вимагає високої акуратності при програмуванні і пов'язане з можливими помилками побудови класів (рис. 2.9).

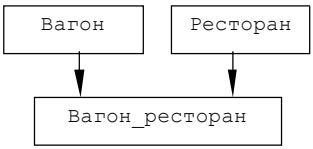


Рис. 2.8. Приклад множинного успадкування

Поліморфізм – здатність об'єкта динамічно змінювати свою поведінку під час виконання програми. Поліморфізм у *C++* реалізується за допомогою механізму віртуальних функцій, який гарантує, що у випадку створення покажчика на деякий базовий тип і перенастроювання його на об'єкт похідного типу буде викликана функція для того об'єкта, на який вказує покажчик. Використання віртуальних функцій дозволяє уникнути введення в об'єкт змінної для збереження типу об'єкта та її наступного аналізу, тому, по можливості, бажано робити віртуальними всі функції класу.

Інкапсуляція і успадкування як механізми ООП дозволяють одержати задану властивість чи поведінку об'єкта різними способами: при інкапсуляції – за рахунок включення даної властивості до об'єкта як у контейнер (агрегування); при успадкуванні – за рахунок їх визначенням у базовому класі.

Приклад: автомобіль є транспортним засобом і має мотор (див. рис. 2.9). Як у першому (див. рис. 2.9,***а***), так і в другому (див. рис. 2.9,***б***) випадках автомобіль має властивості і функції транспортного засобу і мотора. У першому випадку ці властивості успадковуються, і тут криється помилка – автомобіль має мотор, але не є мотором. У другому випадку автомобіль агрегує мотор.

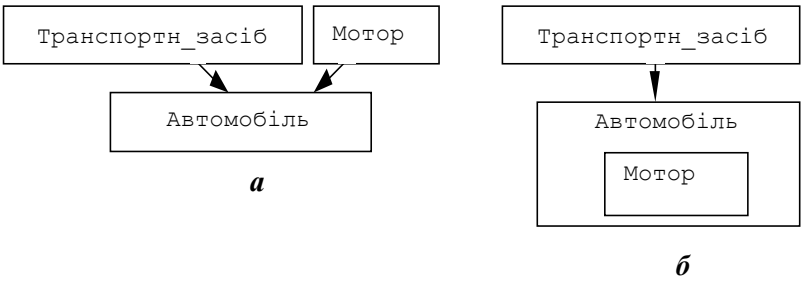


Рис. 2.9. Приклад помилкової (***а***) і вірної (***б***) діаграми успадкування

Як видно з прикладу, при визначенні механізму одержання властивості і функціональності необхідно зробити вибір між тим, що об'єкт має (агрегує) і чим він є (успадковує).

Базові концепції об'єктно-орієнтованих мов програмування дозволяють створювати програмний код у найбільшій відповідності до розроблених моделей. Як було показано в підрозділі "Створення моделі" (див. с. 5), одним з головних результатів декомпо-

зиції є розподіл відповідальності між абстракціями. Засобом розподілу відповідальності для класу є надання (делегування) класу всіх прав по забезпеченню свого стану і поведінки. При цьому клас повинний мати можливість маніпулювати тільки своїми функціями та їх аргументами, інкапсульованими даними і об'єктами.

Об'єктно-орієнтовані методології є найбільш сучасним напрямком у розробці програмного забезпечення, тому що дозволяють створювати моделі, найбільш наближені до реальності, і реалізувати найбільш наближений до розроблених моделей програмний код.

Взаємодія абстракцій. Використання як абстракцій функцій для ПП і класів для ООП має принципові відмінності:

функція не може інкапсулювати інші функції і тому може використовувати будь-який інтерфейс (звертатися до будь-якої функції) в межах модуля;

об'єкт класу може інкапсулювати інші об'єкти і тому має звертатися (використовувати інтерфейс), лише до тих об'єктів, що входять до його складу.

У свою чергу, ці відмінності позначаються на етапі розробки програмного коду: при ПП відстеження зв'язків між елементами покладено на програміста; при ООП відстеження зв'язків між елементами покладено на транслятор.

Для ПП у порівнянні з ООП витрати на написання тексту програми (кодування) скорочуються, але програміст змушений оперувати відносно більшою кількістю інформації. Для ООП повідомлення транслятору інформації про відносини між абстракціями на етапі написання тексту програми пов'язане з додатковими витратами, але це звільнює розробника від контролю цих відносин і дозволяє йому зосередитися над розв'язанням проблеми. При цьому переваги ООП і недоліки ПП виявляються тим значніше, чим більший обсяг проекту.

Як було показано в підрозділі "Створення моделі" (див. с. 5), абстракція повинна створити інформаційні потоки в межах своєї зони відповідальності для керування підпорядкованими абстракціями і надання інформації керівним (вищого рівня). Керування підпорядкованими абстракціями забезпечується через виклик відповідних функцій, а надання інформації – через значення, що повертаються функціями, для чого для кожної абстракції повинен бути створений і специфікований відповідний інтерфейс –

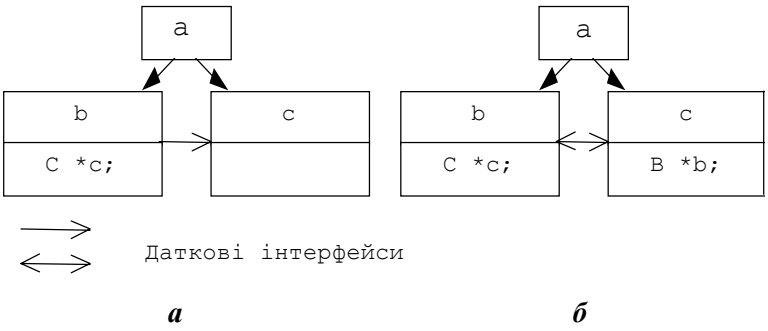
оголошена функція (у випадку ПП) чи набір функцій (у випадку ООП).

При розв'язанні окремих задач можуть бути отримані моделі, які вимагають встановлення безпосередніх зв'язків між абстракціями, відмінних від зв'язків підпорядкування. Створення додаткових інтерфейсів, що реалізують подібні зв'язки призводить, з одного боку, до збільшення кількості необхідних інтерфейсів і ступеня зчеплення для абстракцій, що використовують такі інтерфейси, з іншого боку – дозволяє зменшити інформаційні потоки через абстракції вищого рівня, а отже і складність цих абстракцій. Додатковий інтерфейс може бути встановлений тільки абстракціями вищого рівня, в області видимості яких знаходяться одержувачі інтерфейсу. При ПП до таких засобів може бути віднесена можливість маніпулювання покажчиком на функцію з метою збереження в структурі даних для обробки деякої події та безпосереднього виклику зазначеної функції.

При ООП додатковими засобами забезпечення взаємодії абстракцій є:

надання керівним об'єктом свого покажчика підпорядкованим об'єктам. Завдяки цьому можлива реалізація активної поведінки, при якій з підпорядкованого об'єкта можуть бути викликані методи об'єкта вищого рівня;

передача в об'єкт *B* покажчика на деякий об'єкт *C*, завдяки чому між об'єктами *B* і *C* формується додатковий інтерфейс (рис. 2.10). Цей інтерфейс буде двостороннім, якщо покажчики присутні з кожної сторони інтерфейсу.



а ***б***

Рис. 2.10. Створення додаткового інтерфейсу між абстракціями

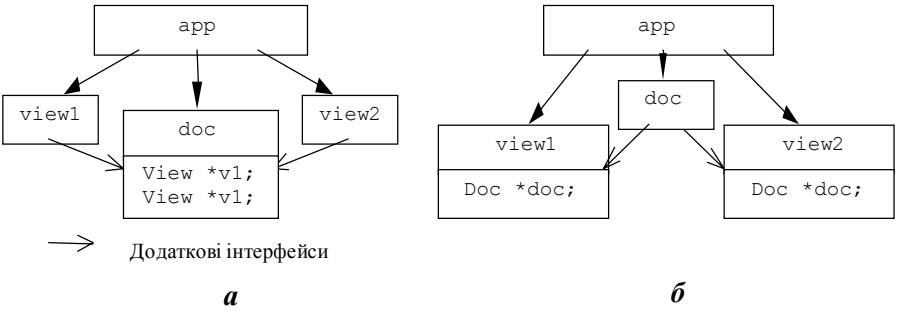
Однією з поведінок об'єкта є надання інформації про свій стан – формування відображення. Відображення об'єкта залежить від власного стану, що знаходиться в його зоні відповідальності (очевидно, що в межах своїх повноважень об'єкт сам повинен формувати своє відображення); стану і можливостей спостерігача (відображення об'єкта повинне формуватися засобами спостерігача, що перетворюють інформацію про стан об'єкта. Різні спостерігачі можуть одержувати різне відображення для одного стану об'єкта).

У залежності від ступеня реалізації цих положень, Б. Страуструп [7] визначає три рівні розв'язання даної задачі:

1. Об'єкт відображає себе сам за допомогою відповідних функцій. Засоби, що формують відображення, інкапсульовані в об'єкт. Це найпростіший спосіб відображення, який не враховує спостерігача і не використовує засобів взаємодії між абстракціями;

2. Об'єкт *doc* (рис. 2.11,***а***) одержує інформацію про спостерігача (спостерігачів) *view* і на її основі формує своє відображення за допомогою засобів спостерігача. Це більш складний спосіб відображення, що базується на засобах взаємодії об'єктів (використовується в продуктах фірми Microsoft);

3. Спостерігач *view* (рис. 2.11,***б***) одержує інформацію про об'єкт *doc* і на основі отриманої інформації і своїх можливостей формує відображення об'єкта *doc*. Даний спосіб також базується на засобах взаємодії об'єктів (використовується в продуктах, які розроблюються в рамках проекту GNU і ОС Linux).



а ***б***

Рис. 2.11. Архітектура "Додаток – Документ – Відображення"

Останній варіант (п.3) має ряд переваг перед варіантами, коли засоби формування відображення використовуються в контексті об'єкта, для якого необхідно отримати відображення (пп.1, 2):

винесення засобів формування відображення за межі об'єкта *doc* призводить до спрощення класу, до якого належить об'єкт;

відсутність засобів формування відображення в об'єкті *doc* дозволяє розвивати проект, додаючи нові абстракції у вигляді спостерігачів *view*, що будуть формувати своє бачення існуючих об'єктів *doc*. Вносити зміни до об'єктів *doc* при цьому не потрібно;

відповідає реальному положенню речей. Наприклад: винахід рентгенівського апарату не призвів до внесення змін в існуючі об'єкти спостереження; спостерігач може знаходитися поза межами області видимості об'єкта, що спостерігається, але не навпаки і т.п.

Припущення, що додавання спостерігача в проект не спричинить змін в об'єктах спостереження, відповідає ідеальному випадку повної відповідності програмних моделей реальним об'єктам. У зв'язку з тим, що програмні моделі є спрощеним зображенням реальних об'єктів, доробка об'єктів спостереження може знадобитися в тому випадку, якщо спостерігач запитує інформацію, яка не міститься в даному об'єкті або для якої не реалізовані інтерфейси. *Приклад*: хмара може мати різні відображення на фотографії і екрані радара. Додавання спостерігача у вигляді термометра (температура – відображення хмари на термометр) може вимагати доробки класу "хмара" (при відсутності властивості "температура"), але не тому, що додано нового спостерігача, а тому, що необхідна програмна модель хмари не відповідає реальному об'єкту.

Концепції, викладені в пп. 2 і 3 визначають архітектуру проекту "Додаток – Документ – Відображення" (див. рис. 2.11), у якій додаток складається з документів, кожен з яких може мати одне чи кілька відображень. Як приклад розглянемо файловий менеджер з розподілом ролей відповідно до даної архітектури: *app* – є власне менеджером файлів; *doc* – містить список файлів на диску; *view* – відображає список файлів.

Відображеннями списку файлів можуть бути назви файлів, кількість файлів, займаний простір і т.ін. Зміна правил відображення (впорядкувати за ...) не змінює порядок файлів на диску, тому не повинна змінювати зміст списку (об'єкт *doc*), а виконуватися на рівні відображення (об'єкт *view*). Виходячи з вимог забезпечення несуперечності даних, об'єкт *view* повинен не дублювати дані з *doc*, а використовувати для маніпулювання масив покажчиків на об'єкти списку, що містяться в *doc*.

Як видно з прикладу, основні переваги такої архітектури полягають у наступному: зміни в *doc* будуть вноситися при маніпуляції файлами, а не їх відображенням; при відкритті одного списку в різних вікнах буде створене ще одне відображення *view* для одного документа *doc*.

До супутніх переваг можна віднести наступне: маніпуляція покажчиками на об'єкти списку вимагає менше ресурсів, ніж маніпуляція безпосередньо об'єктами; кількість об'єктів у списку *doc* може значно перевищувати необхідну для відображення кількість покажчиків у списку *view*.

Реалізація класу для даної архітектури (див. рис. 2.11) у програмному коді може бути отримана в такий спосіб:

```
class App {
public:
    App();
protected:
    Doc *doc;
    View *view1;
    View *view2;
};
```

Реалізація конструктора для рис. 2.11,***а***:

```
App::App()
{
    doc = new Doc;
    view1 = new View;
    view2 = new View;
    doc->AddView(view1);
    doc->AddView(view2);
}
```

Реалізація конструктора для рис. 2.11,***б***:

```
App::App()
{
    doc = new Doc;
    view1 = new View(doc);
    view2 = new View(doc);
}
```

Функція як конвеєр обробки даних. Будь-яка функція програ-

ми містить у собі деяку послідовність операцій над даними і/або викликів функцій, тому її можна розглядати як конвеєр. Використовуючи аналогію з матеріальним виробництвом, визначимо основні принципи ефективної організації такого конвеєра:

конвеєр повинен однозначно визначати порядок виконання одних операцій за іншими з метою забезпечення можливості використання в наступних операціях результатів попередніх;

всі операції на конвеєрі і використовувані дані повинні служити одній меті: одержанню кінцевого результату певного типу. З цього положення випливає, що операції, які складають конвеєр, повинні бути зв'язані через оброблювальні дані;

конвеєр повинен бути досить довгим для того, щоб на ньому можна було виконати всі необхідні операції, і досить коротким для того, щоб бути простим і надійним. З цього положення випливає, що для надійної роботи конвеєра на нього необхідно подавати деталі достатньої складності (на головний конвеєр при складанні автомобіля необхідно подавати кузов і двигун, а не їх складові).

При розв'язанні окремих задач буває необхідно забезпечити декілька паралельно працюючих конвеєрів (потоків виконання). Ця можливість може бути забезпечена на рівні операційної системи, бібліотек, прикладної програми і мови програмування. Мови програмування *C* и *C++* не містять вбудованих механізмів підтримки паралелізму. Розв'язання цієї проблеми на рівні операційної системи, бібліотек або програми виходить за рамки даного матеріалу.

Поняттю безперервного конвеєра відповідають принципи структурного програмування, які визначають керуючі структури (*if-else, do, while, for, switch-case*), достатні для побудови програми будь-якої складності. Довільне переміщення по конвеєру суперечить принципам організації конвеєра, тому використання оператора переходу *goto*, що виконує таке переміщення, недопустиме. Принципи організації функції як конвеєра можуть бути застосовані як при ПП, так і ООП. Конвеєрна модель, що відбиває послідовність взаємодій між абстракціями (чи їх складовими), може бути відображена за допомогою діаграми взаємодії. Можлива реалізація діаграми для прикладу з рис. 2.3 наведена на рис. 2.12. Час на діаграмі рухається зверху-вниз, стрілками позначені дії з передачі керування від однієї функції до іншої (тоб-

то стрілки і є стрічкою конвеєра). Аналогічні діаграми можуть бути побудовані і для моделей, розроблених з використанням ООП.

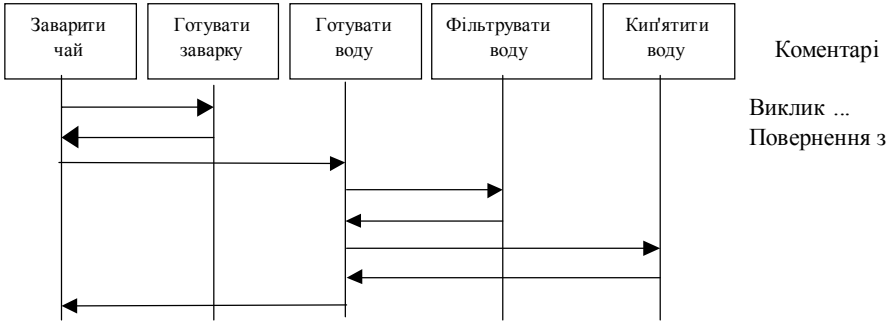


Рис. 2.12. Діаграма взаємодії

2.3. Реалізація

Прийоми безпечного програмування. На етапі реалізації програмного продукту програміст взаємодіє з засобами програмування. Незнання тонкощів мови програмування, недбалість, неправильне розуміння розроблених моделей призводять до появи в програмному коді помилок. Синтаксичні помилки виявляються компілятором, логічні (семантичні) помилки можуть бути виявлені тестовими засобами. Основний принцип безпечного програмування полягає в наступному: якщо компілятор у стані "знайти невідповідність", йому необхідно надати таку можливість. Для цього необхідно:

використовувати прототипи функцій. На основі цієї інформації компілятор може визначити вірність виклику функції;

створювати ідентифікатори з найменшими з усіх можливих часом життя й областю видимості;

використовувати покажчики лише при необхідності зміни значень;

покажчику, що не використовується, завжди привласнювати значення NULL, перевіряти значення покажчика перед використанням;

перевіряти коректність розподілу динамічних ресурсів (відкриття файлів, виділення пам'яті тощо) і зовнішніх даних (дані з клавіатури, файлів і т.ін.);

визначати розмір даних і об'єктів за допомогою оператора *sizeof*, а не числових значень;

використовувати специфікатор *const* скрізь, де це можливо;

оголосити віртуальний деструктор при використанні віртуальних функцій;

визначати класи, які не повинні бути реалізовані в екземплярах як абстрактні;

створити конструктор за умовчанням, конструктор копіювання і оператор присвоювання.

Правильно спроектована програма при порушенні специфікації абстракцій повинна породжувати помилки компіляції, а не помилки виконання.

Функція main визначає місце входу в програму і в програмах з використанням ПП і ООП розв'язує різні задачі:

при ПП функція *main* є головним конвеєром обробки, як складова знаходиться на найвищому рівні абстракції і повинна вміщувати розв'язання головної проблеми в загальному вигляді;

при ООП функція *main* є єдиною функцією, що не може бути інкапсульована в об'єкт. Застосування вільних функцій суперечить принципам ООП, тому функція *main* повинна лише створити об'єкт найвищого рівня в ієрархії об'єктів.

Розглянемо можливі варіанти функції *main* на прикладі найпростішого текстового редактора.

При ПП:

```
/*деякі попередні оголошення*/
int main()
{
     struct Doc *doc=NULL;
     char *menu[]={
          "New",
          "Open"
     };
     int choice;
     InitEditor();
     choice=InitMenu(menu,sizeof menu/sizeof(char *));
     switch(choice)
     {
          case NEW: doc=CreateDocument(); break;
```

```
        case OPEN: doc=OpenDocument(); break;
    }
    if(doc && EditDocument(doc)) SaveDocument(doc);
    CloseEditor();
    return !doc;
}
```

При ООП:

```
/*деякі попередні оголошення*/
int main()
{
    Application app;
    return app.Run();
}
```

Кодування. Застосування розглянутих у підрозділі "Створення моделі" (див. с. 5) принципів проектування (абстракція, модель, декомпозиція і т.д.) буде найбільш ефективним у тому випадку, якщо ці ж принципи будуть застосовані і на етапі створення програмного коду. Для цього на етапі програмування, як і на етапі проектування, необхідне застосування механізму послідовних уточнень (макетування) з одержанням працюючих макетів на кожному кроці уточнень. При цьому процес кодування необхідно організувати таким чином, щоб постійно мати можливість не тільки компілювати, але і виконувати програму. Послідовність кодування подібна до розробки моделей і полягає у наступному.

1. Розробити функцію *main*. На першому етапі її складові повинні бути замінені програмними заглушками (шаблонами). Для прикладу функції *main* на *C++*, розглянутого вище, це може мати наступний вигляд:

```
//файл project.h
class Application {
public:
    Application() {
        cout << "Create application" << endl;
    }
    virtual ~Application() {
        cout << "Delete application" << endl;
    }
```

```
    virtual int Run() {
        cout << "Run application" << endl;
        return 0;
    }
protected:
private:
};
```

2. Розробити шаблони для наступного рівня ієрархії абстракцій (об'єктів). Змінити вміст шаблонів попереднього рівня таким чином, щоб використовувалися знов створені складові. Перейти до наступного пункту.

3. Тестувати. Якщо у функцію передаються аргументи, можна вивести їх значення в тексті повідомлень із шаблонів. Після виправлення можливих помилок, перейти до п. 2.

Головна перевага подібного покрокового підходу полягає в тому, що на кожному кроці розробки є працездатна програма. Такий підхід дозволяє уникнути налагодження значних фрагментів коду, тому що після кожного циклу розробки і тестування складових формується ядро програми, яке розширюється і для якого тестування вже виконане (метод розкручування), і на його основі виконується подальша розробка. Важливим є і те, що при такому підході і на рівні тестування будуть повторені етапи, пройдені при проектуванні, а для налагодження буде активно використовуватися компілятор. Застосування механізму уточнень при реалізації програмного коду визначає напрямок "зверху–донизу" (від загального до частки).

Тестування функцій, об'єктів, модулів і програми в цілому повинно вестися протягом всього процесу роботи над проектом. Для цього необхідно підготувати достатню кількість тестових прикладів, виконувати ці тести і фіксувати результат. Тестування повинні зазнавати: дані – для перевірки і визначення діапазону допустимих значень; інтерфейси – для перевірки коректності взаємодії абстракцій; алгоритми – для оцінки їх вірності та ефективності.

Для полегшення тестування може бути розроблений командний файл, який виконує запуск програми і формування для неї тестових завдань. Тестові завдання і результати тестування для подальшого використання можуть бути розміщені в файлах і зв'я-

зані з програмою за допомогою механізму перенаправлення вводу–виводу. Це дозволить досить просто додавати нові завдання і повторно виконувати існуючі.

Останнім часом набули поширення програми з використанням графічних користувацьких інтерфейсів і маніпулятора миша. При тестуванні таких програм необхідно приступати до реалізації графічних користувацьких інтерфейсів лише після того, як буде налагоджена логіка роботи програми в текстовій консолі. Якщо програму спроектовано вірно, то засоби відображення даних будуть локалізовані в окремих функціях, замінити які надалі не складе значних труднощів; призначити діям миші функціональні клавіші чи escape-послідовності і перевірити однозначність їх відповідності діям з мишею. Така можливість, корисна сама по собі, крім того дозволяє описати маніпуляції з мишею в тестовому файлі.

3. ÇÀÃÀËÜ Í² ÂÈ Ì ÎÃÈ ÄÎ ÎÔÎÐÌ ËÅ Í Íß ÏÐÎÃÐÀ Ì È

3.1. Правила іменування

При спільній роботі над одним проектом програмісти повинні дотримуватися одного, заздалегідь обраного, стилю. При виборі імен функцій, типів і т.ін. найбільш загальноприйнятим у поточний час вважається такий стиль:

імена типів і функцій, а також кожне нове слово починаються з великої літери, – `ThisIsAType`;

імена макросів препроцесора записуються великими літерами з роздільником слів '_' `THIS_IS_A_MACRO`;

всі інші імена – з маленької літери, кожне нове слово – з великої літери – `thisIsAValue`.

Загальноприйяті префікси для імен функцій: Get... – одержати елемент (наприклад, GetCount()); Set... – встановити значення елемента (наприклад, SetCount()); Has... – перевірити наявність чогонебудь (наприклад, HasLink()); Can... – перевірити присутність чого-небудь (наприклад, CanDraw()); Is... – перевірити умову (наприклад, IsValid()); To... – перетворити (наприклад, ToRectangle()); Run... – активізувати об'єкт (наприклад, car.Run()); Make... – сконструювати об'єкт (наприклад, flag.Make()); Init... – ініціювати об'єкт (screen.Init()).

3.2. Файли заголовків

Файли заголовків визначають специфікацію інтерфейсів програми. Для різних мов програмування файли заголовків повинні містити:

оголошення типів функцій і даних для мови *C*. Файли заголовків не повинні містити оголошення змінних (при включенні файла в текст програми змінні виявляться глобальними) і визначення функцій (окремі середовища розробки не "зважають" на зміни у функціях, визначених у файлі заголовків);

оголошення типів функцій і даних, визначення невеликих (*inline*) функцій для мови *C++*. У файлі заголовків має сенс визначати тільки функції довжиною не більше двох-трьох операторів, що виконують просту дію (налагодження *inline*-функцій утруднене, а часте коригування файлів заголовків небажане).

Для невеликих проектів зручно використовувати єдиний файл заголовків, який повинен містити всі інструкції включення файлів заголовків для необхідних бібліотек, макровизначення і опис типів. Для значних за обсягом проектів кількість заголовної інформації може бути також значною, тому зручно використовувати кілька файлів заголовків.

Включення в модуль будь-якого файлу заголовків у будь-якому порядку не повинне призводити до повідомлень про помилку при компіляції. Приклад, що показує як цього досягти, наведений нижче.

```
// ---- файл first.h --------//
#ifndef __FIRST_H
#define __FIRST_H
.....
#endif

// ---- файл some.h ---------//
#ifndef __SOME_H
#define __SOME_H
#include "first.h"
....
#endif

// ---- файл module.cpp -----//
#ifndef __SOME_H
```

```
#include "some.h"
#endif
#ifndef __FIRST_H
#include "first.h"
#endif
```

Без інструкцій умовної компіляції включення одного з файлів заголовків в інший файл заголовків завжди призводить до помилки компіляції модуля *module.cpp*, тому що один з файлів заголовків буде включений у модуль двічі: у самому модулі і через другий файл заголовків.

3.3. Стиль оформлення програми

Для отримання тексту програми з чіткою структурою бажано розміщувати один оператор-вираз в окремому рядку; записувати всі оператори блоку, починаючи з однієї позиції рядка; формувати відступ від лівої границі табуляціями, а не пробілами. При вході в новий блок бажано:

лічильник табуляцій збільшувати на одиницю, при виході з блоку – зменшувати;

ставити парні фігурні дужки однією вертикальною лінією –

```
while(value)
{
     if(something)
     {
          ....
     }
     else
     {
          ....
     }
}
```

виконувати опис структур, об'єднань і класів в наступному стилі –

```
struct Point {
     int x;
     int y;
};
```

розпочинати опис класів з визначення послідовно *public-, protected-, private-* секцій. Конструктори і деструктор варто визначати на початку секції. При передачі аргументів у функції-члени класу імена аргументів і відповідних властивостей класу можуть бути співзвучними, у цьому випадку імена аргументів бажано розпочинати з великої літери –

```
class Menu : public Widget {
public:
    Menu(char *Menu[], int Count);
protected:
    int count;
    char **menu;
};
```

При розміщенні пробілів у виразах необхідно дотримуватися єдиного стилю (наприклад, ставити пробіли після кожної лексеми або не ставити взагалі):

```
for ( int i = 0 ; i < 15 ; ++ i ) ...
for(int i=0;i<15;++i) ...
```

При виборі імен для ідентифікаторів необхідно дотримуватися наступних правил:

імена функцій повинні містити дієслово та іменник, які лаконічно описують дію та об'єкт, над яким виконується дія;

імена змінних повинні бути іменниками, які лаконічно описують призначення змінних;

імена локальних змінних для невеликих блоків повинні бути досить короткими (не довшими семи-восьми символів).

Середній розмір функцій повинен знаходитися в діапазоні від 5 до 20 рядків програмного коду. Функції розміром понад 50 рядків практично не сприймаються як цілісні керуючі структури. Це призводить до помилок при програмуванні і стримує розвиток програмного продукту. По можливості, функція повинна бути тим коротша, чим складніший її алгоритм. Кількість локальних змінних у блоці не повинна перевищувати семи-дев'яти.

Якщо імена ідентифікаторів обрані вдало, текст програми практично не потребує коментарів. Не потрібно коментувати, як працює код, краще описати, що він робить. Приклади коментарів:

```
//Марний коментар:
a = 0; // привласнимо нуль змінній a
```

```
//Корисний коментар:
a = 0; // встановимо початковий стан прапора
//Надлишковий коментар:
flag = 0; // встановимо початковий стан прапора
```

Описи логічних частин, з яких складається текст програми, повинні бути поєднані в групи (група макровизначень, група описів, функції) відокремлені одна від одної порожнім рядком:

```
#include <stdio.h>
#include ....
//порожній рядок
#define N 10
#define ....
//порожній рядок
int Function1()
{
    ....
}
//порожній рядок
int Function2()
{
    ....
}
```

4. ÊÓÐÑÎÂÈÉ ÏÐÎÅÊÒ

4.1. Опис і аналіз предметної області

Аналіз предметної області виконується з метою формування технічного завдання на проектування і повинен включати наступні етапи:

1. Визначення цілей і призначення розробки, виявлення предметної області проекту.

2. Збір інформації за темою проекту, аналіз існуючих у даній області рішень з метою виявлення їх основних властивостей і характерних рис, застосованих технологій і архітектури, дизайну і т.ін.

3. Вибір і обґрунтування критеріїв ефективності і якості проекту, формування вимог і обмежень.

4. Визначення переліку рішень, технологій чи властивостей, що є присутнім у готових продуктах, які можуть бути застосовані чи повторені в даній розробці.

5. Визначення напрямків досліджень для реалізації рішень, які не можуть бути запозичені.

На основі аналізу предметної області можуть бути визначені можливості застосування визначеної технології проектування. При цьому повинні бути враховані масштаби проекту, кількість учасників, архітектурні і функціональні особливості, перспективи подальшого розвитку, вимоги до дизайну. У випадку, якщо передбачається тісна інтеграція проекту з іншими проектами або продуктами – на етапі аналізу можуть бути визначені технологія проектування та інструментальні засоби (мова програмування, бібліотеки тощо).

4.2. Постановка задачі

На основі результатів аналізу предметної області виконується розробка технічного завдання – постановка задачі. У технічному завданні повинні бути чітко сформульовані: галузь застосування, призначення і мета проекту; підстави для розробки проекту; вимоги до архітектури проекту, технологій та інструментів розробки; вимоги до функціональних, технічних і експлуатаційних характеристик об'єкту розробки; визначення структури вхідних і вихідних даних; зв'язок проекту з іншими продуктами, характер використання результатів; вимоги до документації, основні етапи і терміни розробки; методи тестування.

4.3. Проект програми

Етапи виконання проекту. Виконання проекту передбачає наступні етапи: *ескізний проект*, на якому приймаються загальні рішення обрання інструментів розробки, побудування моделей і визначення засобів взаємодії продукту з оточенням (користувачем, обладнанням); *технічний проект*, на якому приймаються спеціальні рішення з деталізації алгоритмів, документування алгоритмів та інтерфейсів; *робочий проект*, на якому проводиться кодування програми.

Вибір інструментів розробки. На основі технічного завдання і аналізу вимог до технічних і експлуатаційних характеристик об'єкта розробки необхідно визначити технологію проектування.

Після вибору технології проектування, повинні бути визначені інструментальні засоби створення даного проекту. До них можуть бути віднесені:

мова програмування;

транслятор, який повинен відповідати сучасному стану мови програмування і, по можливості, мати реалізацію для різних платформ, що дозволить використовувати отримані результати на цих платформах;

бібліотеки. В даний момент є широкий вибір альтернативних бібліотек для реалізації різних задач. При виборі бібліотек необхідно враховувати стабільність, переносимість, фактор розвитку, функціональність, наявність додаткових інструментів і т.ін. Невдалий вибір бібліотеки може призвести до необхідності розробки засобів, уже реалізованих в інших бібліотеках;

середовище розробки. Середовище розробки повинне бути інтуїтивно зрозумілим, зручним і швидким у роботі. Бажана наявність інтегрованих у середовище розробки системи довідки, тестової оболонки, засобів документування. До додаткових можливостей можна віднести наявність колоризації синтаксису, засобу контролю версій і т.ін.;

засоби документування.

Алгоритмічна декомпозиція в ПП. Цей розділ виконується в тому випадку, якщо був обраний ПП. На цьому етапі повинні бути визначені алгоритми і структури даних, та на основі їх аналізу створена ієрархія абстракцій, яка відображає взаємодію функцій і необхідні інтерфейси. Алгоритми (рис. 4.1) і отримані в результаті алгоритмічної декомпозиції ієрархії абстракцій (рис. 4.2) повинні бути зображені графічно у вигляді діаграм.

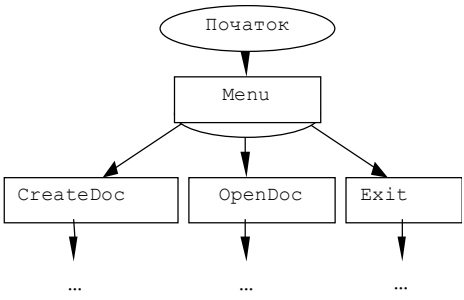


Рис. 4.1. Алгоритм програми. Фрагмент блок-схеми

Ієрархія об'єктів в ООП. Цей розділ виконується при виборі ООП. На цьому етапі в результаті декомпозиції початкової проблеми повинна бути створена ієрархія об'єктів, що відбиває склад і структуру необхідних для реалізації проекту абстракцій. Отримана ієр-

рархія об'єктів повинна бути описана з визначенням специфікацій для всіх об'єктів та ілюстрована графічно (рис. 4.3,***а***). Окремим об'єктам на графічному зображені повинні відповідати окремі прямокутники, у яких указуються назви відповідних об'єктів і назви (прототипи) інтерфейсних функцій (рис. 4.3,***б***).

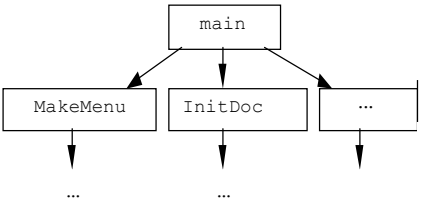


Рис. 4.2. Структурна схема програми.

Інтерфейси підпорядкування між об'єктами зображуються у вигляді ліній або стрілок, які вказують напрямок поширення керуючих впливів. Керуючі впливи завжди поширюються зверху вниз, але при реалізації механізму активної поведінки об'єктів (розглянуті в підрозділі "Взаємодія абстракцій" (див. с. 21)) керуючі впливи можуть поширюватися від підпорядкованих об'єктів до керуючих. Інтерфейси, що реалізують додаткові засоби взаємодії, графічно повинні бути зображені відмінними від інтерфейсів підпорядкування засобами з обов'язковою вказівкою напрямку поширення керуючих впливів.

Ієрархія успадкування в ООП. Цей розділ виконується при використанні ООП. На основі ієрархії об'єктів, отриманої в попередньому розділі, і методики одержання ієрархії класів, наведеної в підрозділі "Об'єктно-орієнтований підхід" (див. с. 16) розробляється ієрархія успадкування. Опис ієрархії успадкування повинен включати специфікації класів та інтерфейсів. Результати проектування класів повинні бути подані графічно з виділенням абстрактних (графічно зображуються у вигляді прямокутників з переривчастими контурами) і кінцевих (графічно зображуються у вигляді прямокутників із суцільними контурами) класів. В прямокутниках необхідно вказати назви відповідних класів і назви интерфейсных функцій (рис. 4.4).

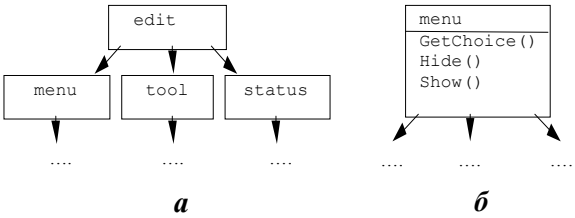


Рис. 4.3. Ієрархія об'єктів (включення). Фрагмент блок-схеми

Опис класів в ООП, функцій і даних в ООП і ПП. Цей розділ

повинен вміщувати деталізацію алгоритмів і технічну документацію на складові частини програми з докладним описом інтерфейсів, властивостей і поведінки класів, принципів організації даних і алгоритмів функцій з обґрунтуванням прийнятих рішень. Суттєві для проекту алгоритми необхідно ілюструвати за допомогою блок-схем або діаграм взаємодії. Якщо при розробці застосовувалися готові рішення сторонніх розробників (бібліотеки, алгоритми і т.ін., необхідно навести короткі описи цих рішень з обґрунтуванням доцільності їх вибору.

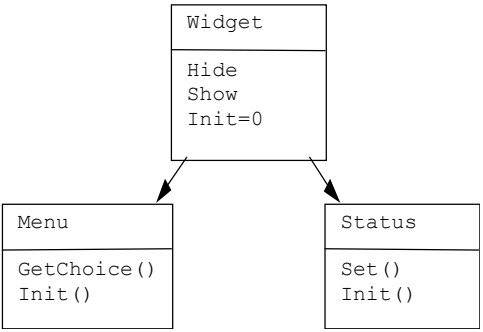


Рис. 4.4. Ієрархія класів (успадкування). Фрагмент блок-схеми

Реалізація і тестування. Розділ повинен включати опис основних етапів реалізації програми і використані на кожному етапі методики тестування, контрольні приклади і результати їх виконання, особливості застосування інструментальних засобів. Розробка програми найчастіше ведеться в умовах, відмінних від умов експлуатації, тому при остаточному тестуванні готового програмного продукту необхідно виявити можливий діапазон умов, при яких програмний продукт відповідає визначеним у технічному завданні вимогам і обмеженням.

Якщо при тестуванні окремих елементів програми використовувалися додаткові програми (тестери), необхідно навести вихідний код цих програм.

4.4. Аналіз отриманих результатів

Аналіз отриманих результатів виконується розроблювачем після завершення робіт над проектом чи його частиною. Основне призначення такого аналізу полягає у визначенні, в якій мірі результати виконаної роботи відповідають початковим вимогам до програмного продукту і поточного стану справ у даній предметній області, та формуванні на його основі планів і перспективних напрямків розвитку проекту чи його частини. Виконання аналізу

необхідно з наступних причин: в результаті роботи над проектом функціональність або властивості деяких його частин були реалізовані частково або з обмеженнями; в результаті роботи над проектом були виявлені можливості розширення його функціональності або поліпшення якості; під час роботи над проектом у предметній області відбулися якісні зміни.

Формування планів і визначення перспективних напрямків відповідає поетапному принципу розвитку складних виробів, що визначає для розроблювача можливість випуску стабільних проміжних програмних продуктів (версій).

На закінчення розкривається значущість розглянутих питань, наводяться головні висновки, які характеризують у стислому вигляді підсумки виконаної роботи і можливості подальшого вдосконалення проекту.

4.5. Технічний опис та інструкція з експлуатації

У даному розділі в частині технічного опису вказуються принципи функціонування програмного продукту, пропозиції і рекомендації щодо використання отриманих результатів; вимоги до програмного і апаратного забезпечення, мінімальна конфігурація технічних і програмних засобів, що забезпечують коректну роботу і комфортні умови експлуатації програми.

У частині експлуатаційної документації повинні бути наведені методика установки програмного продукту і перевірки правильності його функціонування; опис сценаріїв роботи і послідовності настроювання програмного продукту; опис методики виконання контрольних завдань.

4.6. Список літератури

Впорядкований список літератури повинен бути пронумерований арабськими цифрами з крапкою. Основні вимоги до списку літератури:

відповідність темі курсового проекту і повнота відображення всіх аспектів його розробки;

різноманітність видів видань (нормативні, довідкові, навчальні, наукові та ін.);

бібліографічні описи документів розташовують за абеткою за першими їх елементами – прізвищами та ініціалами авторів або за назвами;

бібліографічні описи на різних мовах групуються (спочатку на мовах з кирилицею, потім на мовах з латиницею).

До списку літератури можуть бути включені посилання на ресурси Internet.

4.7. Додатки

Тексти програм повинні починатися з найменування програмних модулів (файлів), кожен з яких – з нової сторінки. На початку визначення кожного модуля чи функції класу бажано розміщувати короткий коментар з указівкою призначення і принципів організації даного коду. Текст програми повинен бути відформатований програмістом. При написанні текстів програм необхідно враховувати максимальну довжину рядка в роздрукованому документі (неприпустиме використання документів, в яких засобами друку розірвані рядки програмного коду).

Формати збереження даних. Якщо програма використовує файли для збереження даних між сеансами своєї роботи або вхідні чи вихідні дані програми містяться в файлах або вводяться користувачем, необхідно навести структуру цих файлів з указівкою призначення, типів, допустимих значень, позиції і (або) розмірів для кожного поля даних. На етапі розробки програмного забезпечення бажано забезпечити логічну і зрозумілу структуру файлів зовнішніх даних з використанням текстового формату збереження даних.

Інтерфейс користувача. Якщо для програми розроблявся графічний інтерфейс користувача або результати роботи програми можуть бути подані на паперовому носії, то в даному додатку повинні бути наведені копії найбільш значущих екранних форм і (або) результатів на пристрої друку.

ÂÈÊ Î ÐÈÑÒÀ Í À Ë²ÒÅÐÀÒÓÐÀ

1. *Бадд Т.* Объектно-ориентированное программирование в действии. – С.-Пб.: Питер, 1997. – 464 с.

2. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы. – М.: Символ Плюс, 1999. – 304 с.

3. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – 2-е изд. – М.: "Издательство БИНОМ", С.-Пб.: "Невский Диалект", 1999. – 560 с.

4. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. – М.: ДМК, 2000. – 432с.

5. *Павловская Т.А.* C/C++. Программирование на языке высокого уровня. – С.-Пб.: Питер, 2001. – 464 с.

6. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ/ Структуры данных/Сортировка/Поиск. – К.: Издательство "Диасофт", 2001. – 688с.

7. *Страуструп Б.* Язык программирования C++. – 3-е изд. – С.-Пб.; М.: "Невский Диалект" – "Издательство БИНОМ", 1999. – 991 с.

8. *Элджер Дж.* C++: библиотека программиста – С.-Пб.: Питер, 2000. – 320 с.

Ç Ì ²ÑÒ

Анатолій Юліанович ГАЙДА

Розробка програмного забезпечення

Методичні вказівки до курсового проектування

Видавництво УДМТУ, 54002, м. Миколаїв, вул. Скороходова, 5

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ДК № 1150 від 12.12.2002 р.

Редактор І.Ю. Цицюра
Комп'ютерна правка та верстка Ю.В. Зайцева
Коректор Н.О. Шайкіна

Підписано до друку 04.09.03. Формат 60×84/16. Папір офсетний.
Ум. друк. арк. 2,5. Обл.-вид. арк. 2,7. Тираж 200 прим.
Вид. № 8. Зам. № 314. Ціна договірна.