

Зміст

| | |
|--|----|
| 1.Технологія програмування. Етапи розвитку технології програмування. Перший етап - «стихийне» програмування. | 3 |
| 2.Другий етап – структурний підхід до програмування (60-70-і роки ХХст.)..... | 4 |
| 3.Третій етап – об’єктний підхід до програмування (із середини 80-х до кінця 90-х років ХХст.)..... | 5 |
| 4.Четвертий етап – компонентний підхід й CASE-технології (із середини 90-х років ХХст. До нашого часу) | 6 |
| 5.Технологія COM та її розвиток. Технологія COBRA..... | 7 |
| 6.CASE-технології (Computer-Aided Software/System Engineering – розробка програмного забезпечення/програмних систем з використання комп’ютерної підтримки) | 8 |
| 7. Розробка складних програмних систем. Життєвий цикл розробки програмного забезпечення. Проблеми розробки складних програмних систем (СПС)..... | 9 |
| 8. Блочно-ієрархічний підхід до створення складних систем. Декомпозиція, абстракція та ієрархічне впорядкування розробки СПС. Принципи проектування при блочно-ієрархічному підході. | 10 |
| 9. Життєвий цикл й етапи розробки програмного забезпечення (ПЗ). Еволюція моделей життєвого циклу програмного забезпечення. | 11 |
| 15.Поняття технологічності програмного забезпечення. Модулі і їхні властивості..... | 13 |
| 16.Зчепленість за зв’язаність модулів..... | 14 |
| 17.Зчеплення модулів: за даними, за зразком, по керуванню, по загальній області даних, за вмістом. | 16 |
| 18. Зв’язність модулів: функціональна, послідовна, інформаційна(комунікативна), процедурна, тимчасова, логічна, випадкова. | 17 |
| 19. Бібліотеки ресурсів. | 21 |
| 20. Спадна та висхідна розробка програмного забезпечення. Структурне й «не структурне» програмування. | 22 |
| 21. Засоби опису структурних алгоритмів. Стель оформлення програми. Ефективність і технологічність. | 25 |
| 22. Програмування «із захистом від помилок». Наскрізний структурний контроль | 27 |
| 23. Визначення вимог до програмного забезпечення й вихідних даних для його проектування. Класифікація програмних продуктів по функціональній ознаці. | 28 |
| 24. Основні експлуатаційні вимоги до програмних продуктів. Перед-проектні дослідження предметної області. Розробка технічного завдання. Принципові рішення початкових етапів проектування..... | 30 |
| 25. Аналіз вимог і визначення специфікацій програмного забезпечення при структурному підході. | 33 |
| 26. Специфікації програмного забезпечення при структурному підході.Специфікації. | 35 |
| 27. Діаграми переходів станів. Функціональні діаграми..... | 36 |
| 28. Діаграми потоків даних. Структури даних і діаграми відносин компонентів даних. Математичні моделі завдань, розробка або вибір методів рішення. | 39 |

| | |
|---|----|
| 29. Використання методу покрокової деталізації для проектування структури програмного забезпечення. Структурні карти Константайна. | 42 |
| 30. Проектування структур даних. Проектування програмного забезпечення, заснованого на декомпозиції даних..... | 45 |
| 32. Розробка користувальницьких інтерфейсів. Типи користувальницьких інтерфейсів й етапи їхньої розробки. Психофізичні особливості людини, пов'язані зі сприйняттям, запам'ятовуванням й обробкою інформації. Користувальницька й програмна моделі інтерфейсу..... | 46 |
| 33. Класифікації діалогів і загальні принципи їхньої розробки. Основні компоненти графічних користувальницьких інтерфейсів. Реалізація діалогів у графічному користувальницькому інтерфейсі. Користувальницькі інтерфейси прямого маніпулювання і їхнє проектування. Інтелектуальні елементи користувальницьких інтерфейсів. | 48 |
| 34.Тестування програмних продуктів. Види контролю якості розроблюваного програмного забезпечення. Формування тестових наборів. Ручний контроль програмного забезпечення | 49 |
| 36.Складання програмної документації. Види програмних документів. Пояснювальна записка. Керівництво користувача. Основні правила оформлення програмної документації. Вимоги до оформлення розрахунково-пояснювальної записки при курсовому програмуванні. | 52 |

1. Технологія програмування. Етапи розвитку технології програмування. Перший етап - «стихійне» програмування.

технології програмування – сукупність виробничих процесів, що приводить до створення необхідного програмного забезпечення, а також опис цієї сукупності процесів. В технології програмування акцент робиться саме процесах розробки програм (технологічних процесах) у порядку їх проходження. Слід зазначити, що для однієї методології може існувати декілька технологій програмування.

Як і будь-яка інша технологія, технологія програмування являє собою набір технологічних інструкцій, що включають:

- вказівку послідовності виконання технологічних операцій;
- перерахування умов, при яких виконується та чи інша операція;
- описи самих операцій, де для кожної операції визначені вихідні дані, результати, а також інструкції, нормативи, стандарти, критерії та методи оцінки і т. п.

Структурний підхід до програмування являє собою сукупність технологічних прийомів, що охоплюють виконання всіх етапів розробки програмного забезпечення. Так процес розробки імперативних програм включає наступні етапи:

1. постановка задачі;
2. її формалізація;
3. вибір методу вирішення задачі;
4. розробка алгоритму програми;
5. розробка тексту програми;
6. тестування і налагодження програми.

Перший етап - «стихійне» програмування. Цей етап охоплює період від моменту появи перших обчислювальних машин до середини 60-х років XX ст. Перші програми мали найпростішу структуру. Вони склалися з власне програми на машинній мові і оброблюваних нею даних.

На початку 60-х років XX ст. вибухнув «криза програмування». Він висловлювався в тому, що фірми, які взялися за розробку складного програмного забезпечення, такого, як операційні системи, зривали всі терміни завершення проектів. Проект застарівав раніше, ніж був готовий до впровадження, збільшувалася його вартість, і в результаті багато проектів так ніколи і не були завершені.

Об'єктивно все це було викликано недосконалістю технології програмування. Перш за все стихійно використовувалася розробка «знизу-вгору» - підхід, при якому спочатку проектували і реалізовували порівняно прості підпрограми, з яких потім намагалися побудувати складну програму.

2. Другий етап – структурний підхід до програмування (60-70-і роки ХХст.)

Другий етап - структурний підхід до програмування (60-70-ті роки ХХ ст.). Структурний підхід до програмування являє собою сукупність рекомендованих технологічних прийомів, що охоплюють виконання всіх етапів розробки програмного забезпечення. В основі структурного підходу лежить декомпозиція (розбиття на частини) складних систем з метою подальшої реалізації у вигляді окремих невеликих (до 40 - 50 операторів) підпрограм.

Проектування, таким чином, здійснювалося «зверху вниз» і мало на увазі реалізацію загальної ідеї, забезпечуючи опрацювання інтерфейсів підпрограм. Одночасно вводилися обмеження на конструкції алгоритмів, рекомендувалися формальні моделі їх опису, а також спеціальний метод проектування алгоритмів - метод покрокової деталізації.

Подальше зростання складності і розмірів розроблюваного програмного забезпечення зажадав розвитку структурування даних. Як наслідок цього в мовах з'являється можливість визначення користувацьких типів даних. Одночасно посилилося прагнення розмежувати доступ до глобальних даних програми, щоб зменшити кількість помилок, що виникають при роботі з глобальними даними. В результаті з'явилася і почала розвиватися технологія модульного програмування.

Модульне програмування передбачає виділення груп підпрограм, які використовують одні і ті ж глобальні дані в окремо компільовані модулі (бібліотеки підпрограм). Зв'язки між модулями при використанні даної технології здійснюються через спеціальний інтерфейс, в той час як доступ до реалізації модуля (тіл підпрограм і деяким «внутрішнім» змінним) заборонений. Цю технологію підтримують сучасні версії мов Pascal і C (C ++), мови Ада і Modula.

Використання модульного програмування істотно спростило розробку програмного забезпечення кількома програмістами. Тепер кожен з них міг розробляти свої модулі незалежно, забезпечуючи взаємодія модулів через спеціально обумовлені міжмодульні інтерфейси. Крім того, модулі надалі без змін можна було використовувати в інших розробках, що підвищило продуктивність праці програмістів.

3.Третій етап – об'єктний підхід до програмування (із середини 80-х до кінця 90-х років ХХст.)

Третій етап - об'єктний підхід до програмування (з середини 80-х до кінця 90-х років ХХ ст.). Об'єктно-орієнтоване програмування визначається як технологія створення складного програмного забезпечення, яка базується на уявленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного типу (класу), а класи утворюють ієрархію з успадкуванням властивостей. Взаємодія програмних об'єктів в такій системі може передбачати надсилання повідомлень.

Основною перевагою об'єктно-орієнтованого програмування в порівнянні з модульним програмуванням є «більш природна» декомпозиція програмного забезпечення, яка істотно полегшує його розробку. В результаті істотно збільшується показник повторного використання кодів і з'являється можливість створення бібліотек класів для різних застосувань.

Бурхливий розвиток технологій програмування, заснованих на об'єктному підході, дозволило вирішити багато проблем. Так були створені середовища, підтримують візуальне програмування, наприклад, Delphi, C ++ Builder, Visual C ++ і т. Д.

4. Четвертий етап – компонентний підхід й CASE-технології (із середини 90-х років ХХст. До нашого часу)

Четвертий етап - компонентний підхід і CASE-технології (з середини 90-х років ХХ ст. До нашого часу). Компонентний підхід передбачає побудову програмного забезпечення з окремих компонентів фізично окремо існуючих частин програмного забезпечення, які взаємодіють між собою через стандартизовані виконавчі інтерфейси. На відміну від звичайних об'єктів об'єкти-компоненти можна зібрати в динамічно викликаються бібліотеки або виконувати файли, поширювати в двійковому вигляді (без вихідних текстів) і використовувати в будь-якій мові програмування, що підтримує відповідну технологію.

Компонентний підхід лежить в основі технологій, розроблених на базі COM (Component Object Model - компонентна модель об'єктів), і технології створення розподілених додатків CORBA (Common Object Request Broker Architecture - загальна архітектура з посередником обробки запитів об'єктів). Ці технології використовують подібні принципи і розрізняються лише особливостями їх реалізації.

Відмінною особливістю сучасного етапу розвитку технології програмування, крім зміни підходу, є створення та впровадження автоматизованих технологій розробки і супроводу програмного забезпечення, які були названі CASE-технологіями (Computer-Aided Software / System Engineering - розробка програмного забезпечення / програмних систем з використанням комп'ютерної підтримки) . Без засобів автоматизації розробка досить складного програмного забезпечення на даний момент стає важко здійсненою: пам'ять людини вже не в змозі фіксувати всі деталі, які необхідно враховувати при розробці програмного забезпечення. На сьогодні існують CASE-технології, що підтримують як структурний, так і об'єктний (в тому числі і компонентний) підходи до програмування.

5.Технологія COM та її розвиток. Технологія COBRA.

COM (Component Object Model – компонентна модель об'єктів), і технології створення розподілених додатків CORBA (Common Object Request Broker Architecture – загальна архітектура з посередником обробки запитів об'єктів). Ці технології використовують схожі принципи і розрізняються лише особливостями їх реалізації.

По технології COM додаток надає свої служби, використовуючи спеціальні об'єкти – *об'єкти* COM, які є екземплярами *класів* COM. Об'єкт COM так як і звичайний об'єкт має поля і методи, але на відміну від звичайних об'єктів кожний об'єкт COM може реалізовувати декілька інтерфейсів, що забезпечують доступ до його полів і функцій. Це досягається за рахунок організації окремої таблиці адресів методів для кожного інтерфейсу (по типу таблиць віртуальних методів). При цьому інтерфейс звичайно об'єднує декілька однотипних функцій. Крім того, класи COM підтримують унаслідування інтерфейсів, але не підтримують *наслідування реалізації*, тобто не наслідують код методів, хоча при необхідності об'єкт дочірнього класу може викликати метод батьківського.

Технологія CORBA, розроблена групою компаній OMC (Object Management Group – група впровадження об'єктної технології програмування), реалізують підхід, аналогічний COM, на базі об'єктів та інтерфейсів CORBA. Програмне ядро CORBA реалізовано для всіх апаратних і програмних платформ, і тому цю технологію можливо використовувати для створення розподіленого програмного забезпечення в гетерогенному (різномірному) обчислювальному середовищі. Організація взаємодії між об'єктами клієнта і сервера в CORBA здійснюється за допомогою спеціального посередника названого VisiBroker, та іншого спеціалізованого програмного забезпечення.

6. CASE-технології (Computer-Aided Software/System Engineering – розробка програмного забезпечення/програмних систем з використання комп'ютерної підтримки)

Особливістю сучасного етапу розвитку технології програмування, крім зміни підходу, є створення та супроводження програмного забезпечення, які були названі CASE-технології (Computer-Aided Software/System Engineering – розробка програмного забезпечення програмних систем з використанням комп'ютерної підтримки). Без засобів автоматизації розробка достатньо складного програмного забезпечення на сучасний момент стає важко реалізованою задачею: пам'ять людини вже не в стані фіксувати всі деталі, які необхідно враховувати при розробці програмного забезпечення. На сьогодні існують CASE-технології, які підтримують як структурний, так і об'єктний (у тому числі і компонентний) підходи до програмування.

Поява нового підходу не означає, що все програмне забезпечення буде створюватись з програмних компонентів, але аналіз існуючих проблем розробки складного програмного забезпечення показує, що він буде використовуватись досить широко.

Однією з сучасних засобів розробки ІС є **CASE-технологія** (CASE – Computer-Aided System Engineering) – програмний комплекс, автоматизує весь технологічний процес аналізу, проектування, розробки і супроводження складних програмних систем.

Засоби CASE-технологій діляться:

- на вбудовані в систему реалізації – всі рішення по проектуванню і реалізації прив'язки до вибраної СУБД;
- незалежні від системи реалізації – всі рішення по проектуванню орієнтовані на уніфікацію (визначення) початкових етапів життєвого циклу програми і засобів їх документування, забезпечують велику гнучкість у виборі засобів реалізації.

Основна перевага CASE-технології – це підтримка колективної роботи над проектом за рахунок можливості роботи в локальній мережі розробників, експорту (імпорту) довільних частин проекту, організованого управління проектами.

В деяких CASE-системах підтримується кодогенерація програм – створення каркасу програм і створення повного продукту.

7. Розробка складних програмних систем. Життєвий цикл розробки програмного забезпечення. Проблеми розробки складних програмних систем (СПС).

Світовий досвід розробки ПЗ дозволив виділити декілька загальноприйнятих моделей створення складних програмних систем. Ці моделі призначені для встановлення чіткої регламентації етапів і змісту робіт на кожному етапі, методів і процедур виконання самих робіт, складу і змісту розроблюваної документації. У результаті вдається помітно знизити витрати на розробку програмного виробу і підвищити якість продукту.

Одним з базових понять методології проектування інформаційної системи є поняття життєвого циклу її ПЗ. Життєвий цикл ПЗ (ЖЦПЗ) – це безперервний процес, який починається з моменту прийняття рішення про необхідність створення ПЗ і закінчується в момент його повного вилучення з експлуатації.

Розробка включає в себе всі роботи по створенню ПЗ і його компонентів відповідно до заданих вимог, включаючи оформлення проектної і експлуатаційної документації, підготовку матеріалів, необхідних для перевірки працездатності і якості програмних продуктів, матеріалів, необхідної для організації навчання персоналу і т.д

Експлуатація включає в себе роботи по впровадженню компонентів ПЗ в експлуатацію, в тому числі конфігурацію бази даних, і робочих місць користувачів, забезпечення експлуатаційною документацією, проведення навчання персоналу, і безпосередньо експлуатацію, модифікацію ПЗ, підготовку пропозицій до вдосконалення, розвитку і модернізації системи.

Управління проектом пов'язане з питаннями планування і організації робіт, створення колективів розробників і контролю за термінами і якістю виконуваних робіт. Технічне і організаційне забезпечення проекту включає вибір методів і інструментальних засобів для реалізації проекту, визначення методів опису проміжних станів розробки, розробку методів і засобів випробувань ПЗ, навчання персоналу. Забезпечення якості проекту пов'язане з проблемами перевірки і тестування ПЗ. У процесі реалізації проекту важливе місце займають питання ідентифікації, опису і контролю конфігурації окремих компонентів і всієї системи загалом.

Управління конфігурацією є одним з допоміжних процесів, який підтримує основні процеси життєвого циклу ПЗ, передусім процеси, розробки і супроводу ПЗ.

Кожний процес характеризується певними задачами і методами їх вирішення, початковими даними, отриманими на попередньому етапі, і результатами. Результатами аналізу, зокрема, є функціональні моделі, інформаційні моделі і відповідні їм діаграми.

8. Блочно-ієрархічний підхід до створення складних систем.

Декомпозиція, абстракція та ієрархічне впорядкування розробки СПС. Принципи проектування при блочно-ієрархічному підході.

Прогрес науки і техніки призводить до появи все більш складних технічних об'єктів, а саме складних систем, які складаються з великої кількості взаємодіючих елементів. При застосуванні систем автоматизованого проектування використовується блочно ієрархічний підхід до проектування складних систем.

При блочно ієрархічному підході процес проектування і представлення самого об'єкта розбивається на рівні. На вищому рівні використовується найменш деталізоване представлення, яке відображає тільки самі загальні риси і особливості системи, що проектується. На кожному новому наступному рівні розробки, ступінь деталізації зростає. При цьому система розглядається не в цілому, а окремими блоками. Такий підхід дозволяє на кожному рівні формувати і вирішувати задачі невеликої складності. Розбиття на блоки повинно бути таким, щоб документація на блок будь-якого рівня була зрозумілою і сприймалася однією людиною. Перевага блочно ієрархічного підходу полягає в тому, що складна задача великої розмірності розбивається на послідовно розв'язувані задачі малої розмірності. Недоліком блочно ієрархічного підходу є те, що на кожному рівні робота ведеться з не до кінця визначеними об'єктами.

В якості елементів на k -тому рівні використовується достатньо складні об'єкти, які будуть розглядатися як системи на наступному $k+1$ рівні. На k -тому рівні ці елементи ще не визначені, так як структура k -того рівня складної системи формується до того, як будуть спроектовані елементи. Відповідно рішення приймаються в обстановці неповної інформації, тобто без сурового обґрунтування. В умовах блочно-ієрархічного проектування на кожному рівні є свої представлення про систему і елементи. Те, що на найбільш високому k -тому рівні називається елементом, стає системою на наступному $k+1$ рівні.

Елементи з самого низького з рівнів, на якому ведеться розгляд називають базовим елементами або компонентами. Існуючи по ЕСКД ділення схем на принципові функціональні структурні відображають принципи блочно-ієрархічного проектування. Функціональні схеми пояснюють протікання визначених процесів у виробі або його частинах, тобто дають представлення про функціонування об'єкту з врахування тільки суттєвих факторів і функціональних частин.

Структурні схеми дають найбільш загальні і найменш деталізоване представлення про об'єкт, визначаючи основні функціональні частини виробу, їх призначення і взаємозв'язки.

На найнижчому рівні схеми виділяються логічні елементи, які є найпростішими. На наступному рівні – функціональні схеми а далі – структурні схеми.

9. Життєвий цикл й етапи розробки програмного забезпечення (ПЗ).

Еволюція моделей життєвого циклу програмного забезпечення.

Необхідність визначення етапів життєвого циклу (ЖЦ) ПЗ обумовлена прагненням розробників до підвищення якості ПЗ за рахунок оптимального управління розробкою і використання різноманітних механізмів контролю якості на кожному етапі, починаючи від постановки завдання і закінчуючи авторським супроводом ПЗ. Найбільш загальним представленням життєвого циклу ПЗ є модель у вигляді базових етапів - процесів, до яких відносяться:

- системний аналіз і обґрунтування вимог до ПЗ;
- попереднє (ескізне) і детальне (технічне) проектування ПЗ;
- розробка програмних компонент, їх комплексування і відлогдження ПЗ в цілому;
- випробовування, дослідна експлуатація та тиражування ПЗ;
- регулярна експлуатація ПЗ, підтримка експлуатації та аналіз результатів;
- супровід ПЗ, його модифікація і вдосконалення, створення нових версій.

Дана модель є загальноприйнятою і відповідає як вітчизняним нормативним документам у області розробки програмного забезпечення, так і зарубіжним. З погляду забезпечення технологічної безпеки доцільно розглянути детальніше особливості представлення етапів ЖЦ в зарубіжних моделях, оскільки саме зарубіжні програмні засоби є найбільш вірогідним носієм програмних дефектів диверсійного типу.

Спочатку була створена каскадна модель ЖЦ, в якій крупні етапи починалися один за одним з використанням результатів попередніх робіт. Найбільш специфічною є спіралевидна модель ЖЦ. У цій моделі увага концентрується на ітераційному процесі початкових етапів проектування. На цих етапах послідовно створюються концепції, специфікації вимог, попередній і детальний проект. На кожному витку уточнюється зміст робіт і концентрується зовнішність створюваного ПЗ.

Стандартизація ЖЦ ПЗ проводиться по трьох напрямках. Перший напрям організовується і стимулюється Міжнародною організацією зі стандартизації (ISO - International Standard Organization) і Міжнародною комісією з електротехніки (IEC - International Electro-technical Commission). На цьому рівні здійснюється стандартизація найбільш загальних технологічних процесів, що мають значення для міжнародної кооперації. Другий напрям активно розвивається в США Інститутом інженерів електротехніки і радіоелектроніки (IEEE - Institute of Electrotechnical and Electronics Engineers) спільно з Американським національним інститутом стандартизації (American National Standards Institute-ANSI). Стандарти ISO/IEC і ANSI/IEEE в основному мають рекомендаційний характер. Третій напрям стимулюється Міністерством оборони США (Department of Defense-DOD). Стандарти DOD мають обов'язковий характер для фірм, що працюють за замовленням Міністерства оборони США.

Для проектування ПЗ складної системи, особливо системи реального часу, доцільно використовувати загальносистемну модель ЖЦ, засновану на об'єднанні всіх відомих робіт в рамках розглянутих базових процесів. Ця модель призначена для використання при плануванні, складанні робочих графіків, управлінні різними програмними проектами.

Сукупність етапів даної моделі ЖЦ доцільно ділити на дві частини, що істотно розрізняються особливостями процесів, техніко-економічними характеристиками і впливаючими на них чинниками.

У першій частині ЖЦ проводиться системний аналіз, проектування, розробка, тестування і випробовування ПЗ. Номенклатура робіт, їх трудомісткість, тривалість та інші характеристики на

цих етапах істотно залежать від об'єкту і середовища розробки. Вивчення подібних залежностей для різних класів ПЗ дозволяє прогнозувати склад і основні характеристики графіків робіт для нових версій ПЗ.

Друга частина ЖЦ, що відображає підтримку експлуатації і супроводу ПЗ, відносно слабо пов'язана з характеристиками об'єкту і середовища розробки. Номенклатура робіт на цих етапах стабільніша, а їх трудомісткість і тривалість можуть істотно змінюватися, і залежать від масовості застосування ПЗ. Для будь-якої моделі ЖЦ забезпечення високої якості програмних комплексів можливо лише при використанні регламентованого технологічного процесу на кожному з цих етапів. Такий процес підтримується CASE засобами автоматизації розробки, які доцільно вибирати з тих, що є або створювати з урахуванням об'єкту розробки і адекватного йому переліку робіт.

15. Поняття технологічності програмного забезпечення. Модулі і їхні властивості.

Під технологічністю розуміють якість ПЗ, від якого залежать трудові та матеріальні витрати на його створення і наступні модифікації.

Хороший проект швидко і легко кодується, тестується, налагоджували і модифікується.

Технологічність ПЗ визначається

- пропрацьованістю його моделей;
- рівнем незалежності модулів;
- стилем програмування;
- ступенем повторного використання кодів.

Для забезпечення необхідної технологічності застосовують спеціальні прийоми і методики, що включають в себе методи і правила: декомпозиції, проектування, програмування, контролю якості, які під загальною назвою «структурний підхід до програмування» були сформульовані ще в 60-х роках ХХ ст.

В його основу було покладено такі основні концепції:

- спадна розробка «зверху вниз»;
- модульне програмування;
- структурний стиль програмування;
- наскрізний структурний контроль.

Модулі

Модулем називають автономно компільовані програмну одиницю. Термін «модуль» традиційно використовується в двох сенсах. Спочатку, коли розмір програм був порівняно невеликий, і всі підпрограми компільовалися окремо, під модулем розумілася підпрограма, тобто послідовність пов'язаних фрагментів програми, звернення до якої виконується по імені. Згодом, коли розмір програм значно зріс, і з'явилася можливість створювати бібліотеки ресурсів: констант, змінних, описів типів, класів і підпрограм, термін «модуль» став використовуватися і в сенсі автономно компільований набір програмних ресурсів.

Дані модуль може отримувати і / або повертати через загальні області пам'яті або параметри. Спочатку до модулів (ще розуміється як підпрограми) пред'являлися наступні вимоги:

- окрема компіляція;
- одна точка входу;
- одна точка виходу;
- відповідність принципу вертикального управління;
- все одно можна зателефонувати інших модулів;
- невеликий розмір (до 50-60 операторів мови);
- незалежність від історії викликів;
- виконання однієї функції.

16.3чепленість за зв'язаність модулів.

Зв'язність модулів – міра надійності функціональних та інформаційних об'єктів всередині одного модуля. Розміщення сильно зв'язаних елементів в одному модулі зменшує між модульні зв'язки, в той час як переміщення сильно зв'язаних елементів в різні модулі не тільки посилює між модульні зв'язки, але й ускладнює розуміння їх взаємодії. Об'єднання слабо зв'язаних елементів також зменшує технологічність модулів, роблячи їх складнішими для розуміння.

Розрізняють наступні види зв'язності (в порядку зменшення рівня) [1]:

- функціональна;
- послідовна;
- інформаційна (комунікативна);
- процедурна;
- часова;
- логічна;
- випадкова.

Приведемо алгоритм визначення рівня зв'язності модуля.

1. Якщо модуль — одинична проблемно-орієнтована функція, то рівень зв'язності — функціональний; кінець алгоритму. Інакше перейти до пункту 2.
2. Якщо дії усередині модуля зв'язані, то перейти до пункту 3. Якщо дії усередині модуля ніяк не зв'язані, то перейти до пункту 6.
3. Якщо дії усередині модуля зв'язані даними, то перейти до пункту 4. Якщо дії усередині модуля зв'язані потоком управління, перейти до пункту 5.
4. Якщо порядок дій усередині модуля важливий, то рівень зв'язності — інформаційний. Інакше рівень зв'язності — комунікативний. Кінець алгоритму.
5. Якщо порядок дій усередині модуля важливий, то рівень зв'язності — процедурний. Інакше рівень зв'язності — тимчасовою. Кінець алгоритму.
6. Якщо дії усередині модуля належать до однієї категорії, то рівень зв'язності — логічний. Якщо дії усередині модуля не належать до однієї категорії, то рівень зв'язності — по збігу.

Кінець алгоритму.

Можливі складніші випадки, коли з модулем асоціюються декілька рівнів зв'язності. У цих випадках слід застосовувати одне з двох правил:

1. правило паралельного ланцюга. Якщо всі дії модуля мають декілька рівнів зв'язності, то модулю привласнюють найсильніший рівень зв'язності;

2.правило послідовного ланцюга. Якщо дії в модулі мають різні рівні зв'язності, то модулю привласнюють найслабкіший рівень зв'язності.

Наприклад, модуль може містити деякі дії, які зв'язані процедурно, а також інші дії, зв'язкові із збігу. В цьому випадку застосовують правило послідовного ланцюга і в цілому модуль вважають зв'язним по збігу.

Зчеплення модуля – це міра його залежності по способу передачі даних від інших модулів. Чим слабше зчеплення модуля з іншими модулями, тим сильніше його незалежність від інших модулів. Для оцінки ступені зчеплення існує шість видів зчеплення модулів по:

- даним;
- зразку;
- управлінню;
- зовнішнім посиланням;
- загальної області даних;
- за змістом.

17.Зчеплення модулів: за даними, за зразком, по керуванню, по загальній області даних, за вмістом.

Зчеплення модуля – це міра його залежності по способу передачі даних від інших модулів. Чим слабше зчеплення модуля з іншими модулями, тим сильніше його незалежність від інших модулів. Для оцінки ступені зчеплення існує шість видів зчеплення модулів по:

- даним;
- зразку;
- управлінню;
- зовнішньому посиланню;
- загальної області даних;
- за вмістом.

Найгіршим видом зчеплення модулів є *зчеплення за вмістом*. Таким є зчеплення двох модулів, коли один із них має пряме посилання на вміст іншого модуля (наприклад, на константу, яка знаходиться в іншому модулі). Таке зчеплення модулів недопустиме.

Не рекомендується використовувати також *зчеплення по загальній області* – це таке зчеплення модулів, коли декілька модулів використовують одну і ту ж область пам'яті.

Зчеплення по зразку припускає, що модулі обмінюються даними, об'єднані в структури. Цей тип забезпечує непогані характеристики порівняно з попередніми. Недолік полягає в тому, що конкретні дані, які передаються, «заховані» в структури, і тому зменшується «прозорість» зв'язку між модулями. Крім того, при зміні структури даних необхідно модифікувати всі модулі, які її використовують.

При *зчепленні по управлінню* один модуль посилає іншому деякий інформаційний об'єкт (прапорець), призначений для управління внутрішньої логіки модуля. Таким способом часто виконують настройку режимів роботи програмного забезпечення. Подібні настройки також знижують наглядність взаємодії модулів і тому забезпечують не кращі характеристики технологічності розроблюваного програмного забезпечення.

Зчеплення по зовнішнім посиланням припускає, що модулі засилаються на один і той же глобальний елемент даних.

Єдиним видом зчеплення модулів, який рекомендується для використання сучасною технологією програмування, є *зчеплення по даним* (параметричне зчеплення) – це випадок, коли дані передаються модулю або при зверненні до нього як значення його параметрів, або як результат його звернення до іншого модуля для обчислення деякої функції. Такий вид зчеплення модулів реалізується на мовах програмування при використанні звернень до процедур (функцій).

18. Зв'язність модулів: функціональна, послідовна, інформаційна(комунікативна), процедурна, тимчасова, логічна, випадкова.

Зв'язність модуля (Cohesion) — це міра взаємної залежності його частин. Зв'язність — внутрішня характеристика модуля. Чим вище зв'язність модуля, тим кращим є результат проектування, тобто тим «чорнішим» є його ящик, тим менше «ручок управління» знаходиться на ньому, і тим простішими є ці «ручки».

Для вимірювання зв'язності використовують поняття сили зв'язності (СС). Існує 7 типів зв'язності:

1.**Зв'язність по збігу(випадкова)** ($Сс=0$). У модулі відсутні явно виражені внутрішні зв'язки.

2.**Логічна зв'язність** ($Сс=1$). Частини модуля об'єднані за принципом функціональної подібності. Наприклад, модуль складається з різних підпрограм обробки помилок. При використанні такого модуля клієнт вибирає тільки одну з підпрограм.

Недоліки:

о складне сполучення;

овелика вірогідність внесення помилок при зміні сполучення ради однієї з функцій.

3.**Часова(тимчасова) зв'язність** ($Сс=3$). Частини модуля не зв'язані, але необхідні в один і той же період роботи системи.

Недолік: сильний взаємний зв'язок з іншими модулями, звідси — сильна чутливість внесенню змін.

4.**Процедурна зв'язність** ($Сс=5$). Частини модуля зв'язані порядком виконуваних ними дій, що реалізують деякий сценарій поведінки.

5.**Комунікативна зв'язність** ($Сс=7$). Частини модуля зв'язані по даним (працюють з однією і тією ж структурою даних).

6.**Інформаційна (послідовна) зв'язність** ($Сс=9$). Вихідні дані однієї частини використовуються як вхідні дані в іншій частині модуля.

7.**Функціональна зв'язність** ($Сс=10$). Частини модуля разом реалізують одну функцію.

Відзначимо, що типи зв'язності 1,2,3 — результат неправильного планування архітектури, а тип зв'язності 4 — результат недбалого планування архітектури застосування.

Загальна характеристика типів зв'язності представлена в табл. 4.1.

Таблиця 5.1. Характеристика зв'язності модуля

| Тип зв'язності | Супроводжуваність | Роль модуля |
|----------------|-------------------|--|
| Функціональна | Краща | «Чорний ящик» |
| Інформаційна | | Не зовсім «чорний ящик» |
| Комунікативна | | «Сірий ящик» |
| Процедурна | Гірша | «Білий» або такий, що «просвічує ящик» |
| Часова | | «Білий ящик» |
| Логічна | | |
| По збігу | | |

8.1. Функціональна зв'язність

Функціонально зв'язний модуль містить елементи, що беруть участь у виконанні одній і лише одного проблемного завдання. Приклади функціонально зв'язних модулів:

Обчислювати синус кута;

Перевіряти орфографію;

Читати запис файлу;

Обчислювати координати мети;

Обчислювати зарплату співробітника;

Визначати місце пасажира.

Кожен з цих модулів має одиничне призначення. Коли клієнт викликає модуль, виконується тільки одна робота, без залучення зовнішніх обробників. Наприклад, модуль Визначати місце пасажера повинен робити тільки це; він не повинен роздруковувати заголовки сторінки.

Деякі з функціонально зв'язних модулів дуже прості (наприклад, Обчислювати синус кута або Читати запис файлу), інші складні (наприклад, Обчислювати координати мети). Модуль

Обчислювати синус кута, очевидно, реалізує одиничну функцію, але як може модуль Обчислювати зарплату співробітника виконувати тільки одну дію? Адже кожен знає, що доводиться визначати нараховану суму, вирахування по розстрочках, прибутковий податок, соціальний податок, аліменти і т. д.! Річ у тому, що, не дивлячись на складність модуля і на те, що його обов'язок виконують декілька під-функцій, якщо його дії можна представити як єдину проблемну функцію (з погляду клієнта), тоді вважають, що модуль функціонально зв'язний.

Застосування, побудовані з функціонально зв'язних модулів, найлегше супроводжувати. Спокусливо думати, що будь-який модуль можна розглядати як одно-функціональний, але не треба помилятися. Існує багато різновидів модулів, які виконують для клієнтів перелік різних робіт, і цей перелік не можна розглядати як єдину проблемну функцію. Критерій при визначенні рівня зв'язності цих нефункціональних модулів — як зв'язані один з одним різні дії, які вони виконують.

8.2. Інформаційна зв'язність

При інформаційній (послідовній) зв'язності елементи-обробники модуля утворюють конвеєр для обробки даних — результати одного обробника використовуються як початкові дані для наступного обробника.

Приведемо приклад:

Модуль Прийом і перевірка запису

прочитати запис з файлу

перевірити контрольні дані в записі

видалити контрольні поля в записі

повернути оброблений запис

Кінець модуля

У цьому модулі 4 елементи. Результати першого елементу (прочитати запис з файлу) використовуються як вхідні дані для другого елементу (перевірити контрольні дані в записі) і так далі Супроводжувати модулі з інформаційною зв'язністю майже так само легко, як і функціонально зв'язні модулі. Правда, можливості повторного використання тут нижче, ніж у разі функціональної зв'язності. Причина — сумісне застосування дій модуля з інформаційною зв'язністю корисно далеко не завжди.

8.3. Комунікативна зв'язність

При комунікативній зв'язності елементи-обробники модуля використовують одні і ті ж дані, наприклад зовнішні дані.

Приклад комунікативно зв'язного модуля:

Модуль Звіт і середня зарплата

використати Таблиця зарплати службовців

згенерувати Звіт по зарплаті

обчислити параметр Середня зарплата

повернути Звіт по зарплаті. Середня зарплата

Кінець модуля

Тут всі елементи модуля працюють із структурою Таблиця зарплати службовців.

З погляду клієнта проблема застосування комунікативно зв'язного модуля полягає в надмірності отримуваних результатів. Наприклад, клієнтові потрібний тільки звіт по зарплаті, він не потребує значення середньої зарплати. Такий клієнт буде вимушений виконувати надмірну роботу — виділення в отриманих даних матеріалу звіту. Майже завжди розбиття комунікативно зв'язного модуля на окремі функціонально зв'язні модулі покращує супроводжуваність системи.

8.4. Процедурна зв'язність

Досягши процедурної зв'язності ми потрапляємо в прикордонну область між хорошою супроводжуваністю (для модулів з вищими рівнями зв'язності) і поганий супроводжуваністю (для модулів з нижчими рівнями зв'язності). Процедурно зв'язний модуль складається з елементів, що реалізують незалежні дії, для яких заданий порядок роботи, тобто порядок передачі управління. Залежності по даним між елементами немає.

8.5. Часова зв'язність

При зв'язності за часом елементи-обробники модуля прив'язані до конкретного періоду часу (з життя програмної системи).

Класичним прикладом тимчасової зв'язності є модуль ініціалізації.

Модуль із зв'язністю за часом зазнає ті ж труднощі, що і процедурно зв'язний модуль. Програміст спокушається можливістю сумісного використання коду (діями, які зв'язані тільки за часом), модуль стає важко використовувати повторно.

Так, за бажання ініціалізувати магнітну стрічку 2 в інший час ви зіткнетеся з незручностями. Щоб не скидати всю систему, доведеться або ввести прапорці, вказуючі ініціалізовану частину, або написати інший код для роботи із стрічкою 2. Обидва рішення погіршують супроводжуваність.

Процедурно зв'язні модулі і модулі з тимчасовою зв'язністю дуже схожі. Ступінь їх непрозорості змінюється від темного сірого до світло-сірого кольору, оскільки важко оголосити функцію такого модуля без перерахування її внутрішніх деталей. Відмінність між ними подібно до відмінності між інформаційною і комунікативною зв'язністю. Порядок виконання дій важливіший в процедурно зв'язних модулях. Крім того, процедурні модулі мають тенденцію до сумісного використання циклів і розгалужень, а модулі з тимчасовою зв'язністю частіше містять лінійний код.

8.6. Логічна зв'язність

Елементи логічно зв'язного модуля належать до дій однієї категорії, і з цієї категорії клієнт вибирає виконувану дію.

Розглянемо наступний приклад:

Модуль Пересилка повідомлення
переслати по електронній пошті
переслати факсом
послати в телеконференцію
переслати по ftp-протоколу
Кінець модуля

Як видно, логічно зв'язний модуль — набір доступних дій. Дії вимушені спільно використовувати один і той же інтерфейс модуля. У рядку виклику мод для значення кожного параметра залежить від використовуваної дії. При виклику окремих дій деякі параметри повинні мати значення пропуску, нульові значення і так далі (хоча клієнт все ж таки повинен використовувати їх і знати їх типи).

Дії в логічно зв'язному модулі потрапляють в одну категорію, хоча мають не тільки схожість, але і відмінності. На жаль, це примушує програміста «зав'язувати код дій у вузол», орієнтуючись на те, що дії спільно використовують загальні рядки коду. Тому логічно зв'язний модуль має:

- потворений зовнішній вигляд з різними параметрами, що забезпечують, наприклад, чотири види доступу;
- заплутану внутрішню структуру з безліччю переходів, схожу на чарівний лабіринт.

У результаті модуль стає складним як для розуміння, так і для супроводу.

8.7. Випадкова Зв'язність

Елементи модуля, з випадковою зв'язністю взагалі не мають ніяких відношень один з одним:

Модуль Різні функції (якісь параметри)
привітати з Новим роком (...)
перевірити справність апаратури (...)
заповнити анкету героя (...)
зміряти температуру (...)
вивести собаку на прогулянку (...)

запаситися продуктами (...)
придбати «ягуар» (...)

19. Бібліотеки ресурсів.

Бібліотека (від англ. library) — збірка об'єктів чи підпрограм для вирішення близьких за тематикою задач. У залежності від мови програмування бібліотеки містять об'єктні модулі чи сирцевий код та дані, допоміжні для задіяння та інтеграції нових можливостей в програмні рішення.

Бібліотека може означати те саме, що модуль, або декілька модулів.

З точки зору комп'ютерних наук бібліотеки діляться на статичні та динамічні.

- **Статичні бібліотеки**

Можуть бути у вигляді початкового тексту, що підключається програмістом до своєї програми на етапі, або у вигляді об'єктних файлів, що приєднуються (лінкуються) до виконуваної програми на етапі компіляції (у Microsoft Windows такі файли мають розширення .lib, у UNIX-подібних ОС — зазвичай .a). В результаті програма включає всі необхідні функції, що робить її автономною, але збільшує розмір.

- **Динамічні бібліотеки**

Також називаються розподілюваними бібліотеками (англ. shared library), або бібліотеками, що динамічно підключаються (англ. Dynamic Link Library, DLL). Це окремі файли, що надають програмі набір використовуваних функцій для завантажування на етапі виконання при зверненні програми до ОС із заявкою на виконання функції з бібліотеки. Якщо необхідна бібліотека вже завантажена в оперативну пам'ять, програма використовуватиме завантажену копію бібліотеки. Такий підхід дозволяє зекономити час і пам'ять, оскільки декілька програм використовують одну копію бібліотеки, вже завантажену в пам'ять.

При написанні програми програмістові досить вказати транслятору мови програмування (компілятору або інтерпретатору), що слід підключити таку-от бібліотеку і використовувати таку-от функцію зі вказаної бібліотеки. Ні початковий текст, ні виконуваний код функції до складу програми не входить.

20. Спадна та висхідна розробка програмного забезпечення.

Структурне й «не структурне» програмування.

Одна з основних ідей, покладених в більшість відомих технологій програмування, - спадна розробка програмного забезпечення (Top-Down Programming - програмування "зверху вниз"). Існують також інші назви: "метод покрокової деталізації", "систематичне програмування", "ієрархічне програмування". Його принцип полягає в тому, що спочатку визначаються основні функції, які повинні бути забезпечені розроблюваною програмою, а потім вони конкретизуються за допомогою набору додаткових функцій. Наприклад, основна функція - обробка файлу. Функції деталізації - відкрити файл, обробити всі записи, закрити файл.

У методі **спадного** проектування спочатку пишеться основна програма, використовуючи засоби виклику підпрограм, причому в якості підпрограм спочатку вводяться "заглушки" виду: "Викликали підпрограму номер ...". Потім, будучи впевненим у правильності логічного побудови основної програми, пишеться кожна підпрограма, викликаючи в міру необхідності підпрограми нижчого рівня. Цей послідовний процес триває, поки не буде завершено процес програмування і тестування.

При іншому методі, **висхідному** проектуванні (програмуванні "від низу до верху"), спочатку пишуться підпрограми нижнього рівня, ретельно тестуються та відлагоджуються. Далі додають підпрограми більш високого рівня, які викликають підпрограми нижнього рівня, і так до тих пір поки не буде досягнута програма самого верхнього рівня. Метод проектування "знизу вгору" придатний при наявності великих бібліотек стандартних підпрограм.

Іноді кращим є гібрид двох методів. Однак в обох випадках кожна підпрограма повинна бути невеликою, так щоб можна було охопити одним поглядом всю її логіку.

Суть структурного програмування полягає в простій ідеї: програма повинна використовувати керуючі конструкції з одним входом і одним виходом. Така конструкція являє собою блок коду, в якому є тільки одне місце, де він може починатися, і одне - де може закінчуватися. У нього немає інших входів і виходів. Структурне програмування - це не те ж саме, що і структурне проектування зверху вниз. Воно відноситься тільки до рівня кодування.

Структурна програма пишеться в упорядкованій, дисциплінованій манері і не містить непередбачуваних переходів з місця на місце. Ви можете читати її зверху вниз, і практично так само вона виконується. Менш дисципліновані підходи призводять до такого вихідного коду, який містить менш зрозумілу і зручну для читання картину того, як програма виконується. Менша читабельність означає гірше розуміння і врешті гіршу якість програми.

Модульне програмування

Реалізація принципу структурного програмування здійснюється з використанням макрокоманд і механізмів виклику підпрограм. Ці ж механізми підходять і для реалізації модульного програмування, яке можна розглядати як частину структурного підходу.

Необхідно розрізняти використання слова модуль, коли йдеться про одиницю дроблення великої програми на окремі блоки (які можуть бути реалізовані у вигляді процедур і функцій) і коли мається на увазі синтаксична конструкція мов програмування (unit в Object Pascal).

Модульне програмування - це організація програми як сукупності незалежних блоків, що називаються модулями, структура і поведінка яких підкоряються певним правилам.

Концепцію модульного програмування можна сформулювати у вигляді таких понять і положень:

1. великі завдання розбиваються на ряд більш дрібних, функціонально самостійних підзадач - модулів, які пов'язані між собою тільки вхідними і вихідними даними;
2. модуль являє собою "чорний ящик" з одним входом і одним виходом. Це дозволяє безболісно проводити модернізацію програми в процесі її експлуатації, полегшує її супровід, а також дозволяє розробляти частини програмного проекту різними мовами програмування;
3. у кожному модулі повинні здійснюватися ясні завдання. Якщо призначення модуля незрозуміле, то це означає, що декомпозицію на модулі було проведено недостатньо якісно. Процес декомпозиції потрібно продовжувати до тих пір, поки не буде чіткого розуміння призначення усіх модулів та їх оптимального поєднання;
4. вихідний текст модуля повинен мати заголовок та інтерфейсну частину, де відображаються призначення модуля й усі його зовнішні зв'язки;
5. у ході розробки модулів програми слід передбачати спеціальні блоки операцій, що враховують реакцію на можливі помилки в даних або в діях користувача.

Велике значення в концепції модульного програмування надається організації керуючих та інформаційних зв'язків між модулями програми, що спільно розв'язують одну або декілька великих задач.

Об'єктно-орієнтований підхід

При об'єктно-орієнтованому підході в якості будівельних блоків використовуються об'єкти, що містять свої власні коди і дані. Структура програм при об'єктно-орієнтованому підході представляється графом взаємодії об'єктів, а не деревом ієрархії, як це має місце в структурному проектуванні.

Об'єктно-орієнтоване програмування (object-oriented programming) - це технологія реалізації програм, заснована на представленні програми у вигляді сукупності об'єктів, кожний з яких є екземпляром певного класу, а класи утворюють ієрархію спадкування.

Поняття об'єктно-орієнтованого програмування визначає три основні концепції, при дотриманні яких програма буде об'єктно-орієнтованою:

- об'єктно-орієнтоване програмування використовує в якості базових елементів класи, які породжують об'єкти;
- у процесі виконання програми може одночасно використовуватися кілька об'єктів, породжених від одного класу (примірників реалізації класу);
- класи організовано ієрархічно (ієрархія означає "бути частиною").

Клас - представляє собою об'єднуючу концепцію набору об'єктів, що мають спільні характеристики. Клас також визначає інтерфейс із навколишнім світом, за допомогою якого здійснюється взаємодія з окремими об'єктами.

Клас є описом того, як буде виглядати і вести себе його представник. Тому клас проектується як утвір, що відповідає за створення своїх нових представників (примірників або об'єктів). Створення об'єктів та їх знищення здійснюється за допомогою особливих методів - так званих конструкторів і деструкторів.

Об'єкт - це структурована змінна типу клас, що містить всю інформацію про деякий фізичний предмет або поняття, яке реалізується в програмі. Всі об'єкти - представники даного класу аналогічні один одному в тому сенсі, що вони мають один і той же набір операцій - методів.

Об'єкт, як логічна одиниця, має такі дані та операції (методи з кодом алгоритму) в окремій ділянці пам'яті:

- поля об'єкта (або атрибути вихідних даних), значення яких визначають поточний стан об'єкта;
- методи об'єкта, які реалізують дії (виконання алгоритмів) у відповідь на їх виклик у вигляді відданого повідомлення;
- властивості - частина методів, які визначають поведінку об'єкта, тобто його реакцію на зовнішні впливи.

При оголошенні класів визначаються описані вище три характеристики об'єктів: поля, методи і властивості, а також вказується предок даного класу.

Об'єкти у програмах відтворюють усі відтінки явищ реального світу: "народжуються" і "вмирають"; змінюють свій стан; запускають і зупиняють процеси; "вбивають" і "відроджують" інші об'єкти.

21. Засоби опису структурних алгоритмів. Стиль оформлення програми. Ефективність і технологічність.

Алгоритми представляють за допомогою конкретних образотворчих засобів, склад і правила вживання яких утворюють конкретні способи або форми запису алгоритмів. Існує декілька таких способів:

- словесний;
- словесно-формульний;
- графічна схема;
- блок-схема;
- операторна схема;
- HIPO-схема;
- таблиця рішень, тощо.

Словесна форма. Це словесно сформульована послідовність правил перетворення інформації. При цьому формальні правила перетворення інформації формулюються, нумеруються та вказується послідовність їх виконання. Така форма є прийнятною для досить простих або, навпаки, складних задач, розв'язання яких можна скласти з готових блоків (процедур обробки інформації), а за допомогою словесного опису вказати порядок їх виклику.

Словесно-формульна форма. Це поєднання формул перетворення інформації та словесного визначення послідовності їх виконання. Використовує загальноприйняті математичні позначення, коментарі до них, що пояснюють дії, та їх послідовність, яка визначається за допомогою міток.

Граф-схеми. Граф-схема — це представлення алгоритму у вигляді системи точок, кожна з яких визначає дію, та стрілок, які вказують перехід від однієї дії до іншої.

Блок-схеми. Це форма представлення, при якій процес розв'язання задачі поділяється на окремі етапи (або операції), які представляються у вигляді спеціальних блоків, конфігурація яких вказує тип дій. Зв'язки між блоками визначають послідовність цих дій.

Операторні схеми. Це послідовність операторів певних типів (всі перетворення інформації подають у вигляді послідовності допустимих операцій).

HIPO-схеми (Hierarchy Input Process Output). Це таблиця, кожний рядок якої містить вказівки щодо вхідної інформації, дії над нею та вихідної інформації.

Таблиці рішень. Це табличне представлення алгоритму прийняття рішень у процесі перетворення інформації. Такий вид представлення алгоритмів обробки інформації найчастіше використовується у тих випадках, коли на кожному кроці перетворення інформації потрібно проводити аналіз стану системи (із скінченної множини станів) та вхідної інформації, і кожна з комбінацій результатів аналізу спричиняє перехід у новий стан та певну вихідну інформацію. Побудувати такий процес з багатьма розгалуженнями у вигляді блок-схеми неможливо.

З точки зору технологічності хорошим вважають стиль оформлення програми, який полегшує її сприйняття як самим автором, так і іншими програмістами, яким, можливо, доведеться її перевіряти або модифікувати.

Саме виходячи з того, що будь-яку програму неодноразово доведеться переглядати, слід дотримуватися хорошого стилю написання програм.

Стиль оформлення програми включає:

- правила іменування об'єктів програми (змінних, функцій, типів, даних і т. П.);

- правила оформлення модулів;
- стиль оформлення текстів модулів.

Правила іменування об'єктів програми. При виборі імен програмних об'єктів слід дотримуватися наступних правил:

- ім'я об'єкта має відповідати його змісту

якщо дозволяє мову програмування, можна використовувати символ «_» для візуального поділу імен, що складаються з декількох слів

Традиційно ефективними вважають програми, що вимагають мінімального часу виконання і / або мінімального обсягу оперативної пам'яті. Особливі вимоги до ефективності програмного забезпечення пред'являють при наявності обмежень (на час реакції системи, на обсяг оперативної пам'яті і т. П.). У випадках, коли забезпечення ефективності не вимагає серйозних тимчасових і трудових витрат, а також не призводить до істотного погіршення технологічних властивостей, необхідно цю вимогу мати на увазі

Частково проблему ефективності програм вирішують за програміста компілятори. Засоби оптимізації, використовувані компіляторами, ділять на дві групи:

- машинно-залежні, т. е. орієнтовані на конкретний машинний мову, виконують оптимізацію кодів на рівні машинних команд, наприклад, виключення зайвих пересилань, використання більш ефективних команд і т. п .;
- машинно-незалежні виконують оптимізацію на рівні вхідного мови, наприклад, винесення обчислень константних (незалежних від індексу циклу) виразів з циклів і т. П.

Природно, не можна втрутитися в роботу компілятора, але існує багато можливостей оптимізації програми на рівні команд.

22. Програмування «із захистом від помилок». Наскрізний структурний контроль

Будь-яка з помилок програмування, яка не виявляється на етапах компіляції і компоновки програми, в кінцевому рахунку може проявитися трьома способами: привести до видачі системного повідомлення про помилку, «зависання» комп'ютера і отримання невірних результатів.

Програмування, при якому застосовують спеціальні прийоми раннього виявлення і нейтралізації помилок, було названо захисним іліпрограмуванням із захистом від помилок. При його використанні істотно зменшується ймовірність отримання невірних результатів. Детальний аналіз помилок і їх можливих ранніх проявів показує, що доцільно перевіряти:

- правильність виконання операцій введення-виведення;
- допустимість проміжних результатів (значень керуючих змінних, значень індексів, типів даних, значень числових аргументів і т. Д.).

Перевірки правильності виконання операцій введення-вивода Причинами невірного визначення вихідних даних можуть бути, як внутрішні помилки-помилки пристроїв вводу-виводу або програмного забезпечення, так і зовнішні помилки - помилки користувача. При цьому прийнято розрізняти:

- помилки передачі - апаратні засоби, наприклад, внаслідок несправності, спотворюють дані;

помилки перетворення - програма невірно перетворює вихідні дані з вхідного формату у внутрішній;

- помилки перезапису - користувач помиляється при введенні даних, наприклад, вводить зайвий або інший символ;
- помилки даних - користувач вводить невірні дані. Помилки передачі зазвичай контролюються апаратно.

Наскрізний структурний контроль являє собою сукупність технологічних операцій контролю, що дозволяють забезпечити якомога більш раннє виявлення помилок в процесі розробки. Термін «наскрізний» в назві відображає виконання контролю на всіх етапах розробки. Термін «структурний» означає наявність чітких рекомендацій по виконанню контролюючих операцій на кожному етапі.

Наскрізний структурний контроль повинен виконуватися на спеціальних контрольних сесіях, в яких, крім розробників, можуть брав участь спеціально запрошені експерти. Час між сесіями визначає обсяг матеріалу, який виноситься на сесію: при частих сесіях матеріал розглядають невеликими порціями, при рідкісних - істотним фрагментами. Матеріали для чергової сесії повинні видаватися учасникам заздалегідь, щоб вони могли їх обдумати.

23. Визначення вимог до програмного забезпечення й вихідних даних для його проектування. Класифікація програмних продуктів по функціональній ознаці.

Процеси встановлення вимог

- Визначення та розробка вимог до ПЗ;
 - Визначення вимог до інтерфейсу
 - Становлення пріоритетів та інтеграція вимог до ПЗ

Вхідні дані:

- Вимоги, що пред'являються для установки;
- Обмеження системи (системні обмеження)
- Функціональні вимоги до ПЗ системи.

Вихідні дані:

- Попередні (первинні) вимоги до ПЗ
- Вимоги, що пред'являються для установки

Призначення:

- Планування проекту
- Встановлення вимог

Вимоги до ПЗ, включаючи обмеження, повинні бути отримані із вхідних документів і результатів моделювання, створення прототипів.

Використовуючи зазначену вхідну інформацію, розробник має аналізувати функціональні та експлуатаційні вимоги до ПЗ, з метод визначення простежуваності, ясності, достовірності, тестованості, безпеки та будь-яких інших проектно-специфічних характеристик.

Такі методи, як структурний аналіз, моделювання, прототипування є корисними в цьому процесі.

Попередні (первинні) вимог до ПЗ і вимоги. Що пред'являються для установки повинні включати розглянуті обмеження системи, такі як: термі, розмір, мова, маркетингові обмеження і технології.

Кожен програмний продукт призначений для виконання певних функцій. За призначенням усі програмні продукти можна розділити на три групи: системні, прикладні та гібридні.

До **системних** зазвичай відносять програмні продукти, що забезпечують функціонування обчислювальних систем (як окремих комп'ютерів, так і мереж). Це - операційні системи, оболонки і інші службові програми (утиліти).

Прикладні програми і системи орієнтовані на рішення конкретних для користувача завдань. Розрізняють користувачів:

- розробників програм;
- непрограмістів, що використовують комп'ютерні системи для досягнення своїх цілей.

Гібридні системи поєднують в собі ознаки системного та прикладного програмного забезпечення. Як правило, це великі, але вузькоспеціалізовані системи, призначені для управління технологічними процесами різних типів у режимі реального часу. Для підвищення надійності і зниження часу обробки в такі системи зазвичай включають програми, що забезпечують виконання функцій операційних систем.

До кожного з перерахованих вище типів програмного забезпечення при розробці, крім функціональних, зазвичай пред'являють ще й певні експлуатаційні вимоги.

24. Основні експлуатаційні вимоги до програмних продуктів. Перед-проектні дослідження предметної області. Розробка технічного завдання. Принципові рішення початкових етапів проектування.

Визначення вимог до програмних продуктів

Експлуатаційні вимоги визначають характеристики програмного забезпечення, які проявляються в процесі його використання. До таких характеристик належать:

- **правильність** – функціонування у відповідності з технічним завданням. Ця вимога є обов'язковою для всякого програмного продукту, але оскільки ніяке тестування не дає гарантії 100%-ої правильності, мова може йти про деяку ймовірність наявності помилок. Ймовірність збою системи управління космічними польотами повинна бути близька до нуля;
- **універсальність** – забезпечення правильності роботи при довільних допустимих даних і захист від неправильних даних. Так як в попередньому випадку, довести універсальність програми неможливо, тому є зміст говорити про степінь її універсальності;
- **надійність** – забезпечення повної повторюваності результатів, тобто забезпечення їх правильності при наявності різного роду збою. Джерелами завад можуть бути технічні і програмні засоби, а також люди, які працюють з цими засобами. Зараз існує достатня кількість способів запобігти втрат інформації при збоях. Наприклад, прийом «створення контрольних точок», при якому зберігаються проміжні результати, що дозволяє після збою програми продовжити роботу з даними, записаними в останній контрольній точці. Можливо також зменшити кількість помилок, використовуючи дублювання систем чи введення надлишкової інформації;
- **перевірюваність** – можливість перевірки результатів. Для цього необхідно документально фіксувати початкові дані, встановлені режими та іншу інформацію, яка впливає на результати. Особливо це проявляється, коли сигнали поступають безпосередньо від датчиків;
- **точність результатів** – забезпечення похибки результатів не вище заданої. Величина похибки залежить від точності початкових даних, степені адекватності моделі, точності вибраного методу і похибки виконання операцій на комп'ютері. Жорсткі вимоги до точності пред'являють системи навігації (наприклад, система стиковки космічних апаратів) і системи управління технологічними процесами;
- **захищеність** – забезпечення конфіденційності інформації. Найбільш жорсткі вимоги пред'являються до систем, в яких зберігається інформація, пов'язана з державною і комерційною таємницею. Для забезпечення захисту інформації використовують програмні, криптографічні, правові та інші методи;
- **програмна сумісність** – можливість сумісного функціонування з іншим програмним забезпеченням. Частіше всього в даному випадку мова йде про функціонування програми

під управлінням заданої операційної системи. Однак може бути потрібний обмін даними з деякою іншою програмою. У цьому випадку точно обговорюється формат даних.

- апаратна сумісність – можливість сумісного функціонування з деяким обладнанням. Цю вимогу формують у вигляді мінімально можливої конфігурації обладнання, на якому буде працювати дане програмне забезпечення. Якщо припускається використання нестандартного обладнання, то для нього повинні бути описані інтерфейси;
- ефективність – використання мінімально можливої кількості ресурсів технічних засобів (наприклад, часу мікропроцесора, об'єму оперативної пам'яті, об'єму зовнішньої пам'яті, кількості зовнішніх пристроїв та ін.). Ефективність оцінюється по кожному ресурсу окремо, тому вимоги ефективності часто суперечать одна одній. Наприклад, щоб зменшити час виконання програми, необхідно збільшити об'єм оперативної пам'яті;
- адаптованість – можливість швидкої модифікації з ціллю пристосування до змінних умов функціонування. Оцінити цю характеристику кількісно практично неможливо. Можливо тільки констатувати, що при розробці даного ПЗ використовувались прийоми, які полегшують його модернізацію;
- повторне входження – можливість повторного виконання без пере загрузки з диску. Дана вимога зазвичай пред'являється до програмного забезпечення, резидентно завантаженому в оперативну пам'ять (наприклад, драйвери);
- ресентерабельність – можливість «паралельного» використання декількома процесами. Щоб задовольнити цю вимогу, необхідно створювати копію даних, що змінюються програмою, для кожного процесу.

Технічне завдання

Технічне завдання являє собою документ, в якому сформульовані основні цілі розробки, вимоги до програмного продукту, визначено терміни та етапи розробки та регламентований процес приймально-здавальних випробувань [33]. У розробці технічного завдання беруть участь як представники замовника, так і представники виконавця. В основі цього документа лежать вихідні вимоги замовника, аналіз передових досягнень техніки, результати виконання науково-дослідних робіт, передпроектних досліджень, наукового прогнозування і т. п.

Основні чинники, що визначають характеристики майбутнього програмного забезпечення – це:

- вихідні дані і необхідні результати, які визначають функції програми або системи;
- середовище функціонування (програмне та апаратне) - може бути задане, а може вибиратися для забезпечення параметрів, зазначених у технічному завданні;
- можливу взаємодію з іншим програмним забезпеченням або спеціальними технічними засобами - також може бути визначено, а може вибиратися виходячи з набору виконуваних функцій.

Розробка технічного завдання виконується в такій послідовності. Перш за все, встановлюють набір виконуваних функцій, а також перелік і характеристики вхідних даних. Потім визначають перелік результатів, їх характеристики і способи подання. Далі уточнюють середовище функціонування програмного забезпечення: конкретну комплектацію і параметри технічних засобів, версію

операційної системи і, можливо, версії і параметри іншого встановленого програмного забезпечення, з яким може взаємодіяти майбутній програмний продукт.

Відповідно до цього стандарту технічне завдання повинне містити наступні розділи:

- вступ;
- підстави для розробки;
- призначення розробки;
- вимоги до програми або програмного виробу;
- вимоги до програмної документації;
- техніко-економічні показники;
- стадії і етапи розробки;

25. Аналіз вимог і визначення специфікацій програмного забезпечення при структурному підході.

Аналіз вимог.

Аналіз вимог полягає в визначенні потреб та умов, які висуваються щодо нового, чи зміненого продукту, враховуючи можливо конфліктні вимоги різних замовників, таких як користувачі чи бенефіціари.

Аналіз вимог є критичним для успішної розробки проекту. Вимоги мають бути задокументованими, вимірними, тестовними, пов'язаними з бізнес-потребами, і описаними з рівнем деталізації достатнім для конструювання системи. Вимоги можуть бути архітектурними, структурними, поведінковими, функціональними, та не функціональними.

Аналіз вимог включає три види діяльності:

- Виявлення вимог: задача комунікації з користувачами для визначення їх вимог. Також це називають збором вимог.
- Аналіз вимог: виявлення недоліків вимог (неточностей, неповноти, неоднозначностей чи суперечностей) і їх виправлення.
- Запис вимог: Вимоги можуть документуватись в різних формах, таких як опис звичайною мовою, прецедентами, користувацькими історіями, чи специфікаціями процесу.

Структурний підхід.

Сутність структурного підходу полягає в декомпозиції програми або програмної системи за функціональним принципом. Усі пропонувані методи декомпозиції використовують інтерфейси найпростішого типу: примітивні інтерфейси і традиційні меню, і розраховані на аналіз та проектування як структур даних, так і програм, що їх оброблюють. Процес проектування складного програмного забезпечення починають з уточнення його структури, тобто визначення структурних компонент та зв'язків між ними. Результат уточнення структури може бути представлений у вигляді структурної та / або функціональної схем і опису (специфікацій) компонентів.

Специфікації.

Специфікації є *повним і точним* описом функцій і обмежень майбутнього програмного забезпечення [31]. При цьому одна частина специфікацій (*функціональні*) описує його функції, а інша частина (*експлуатаційні*) визначає вимоги до технічних засобів, до надійності, інформаційної безпеки і інш. Стосовно функціональних специфікацій мають на увазі наступні вимоги:

- вимога *повноти* означає, що специфікації повинні містити всю істотну інформацію, де нічого важливого не було б упущено, і відсутня неістотна інформація, наприклад деталі реалізації, щоб не перешкоджати розробникові у виборі найбільш ефективних рішень;
- вимога *точності* означає, що специфікації повинні однозначно сприйматися як замовником, так і розробником.

Останню вимогу виконати досить складно внаслідок того, що природна мова для опису специфікацій не підходить: навіть докладні специфікації на природній мові не забезпечують необхідної точності. Точні специфікації можна визначити, тільки розробивши деяку *формальну модель майбутнього* програмного забезпечення.

Формальні моделі, що використовуються на етапі визначення специфікацій можна розділити на дві групи: моделі, *залежні від підходу до розробки* (структурного або об'єктно-орієнтованого), і моделі, *не залежні* від нього. Так, діаграми переходів станів, які демонструють особливості поведінки майбутнього програмного забезпечення, при отриманні тих або інших сигналів ззовні, і математичні моделі наочної області використовують при будь-якому підході до розробки.

В рамках структурного підходу на етапі аналізу і визначення специфікацій використовують три типи моделей: орієнтовані на функції, орієнтовані на дані і орієнтовані на потоки даних. Кожну модель доцільно використовувати для свого специфічного класу програмних розробок.

26. Специфікації програмного забезпечення при структурному підході. Специфікації.

Специфікації є *повним* і *точним* описом функцій і обмежень майбутнього програмного забезпечення [31]. При цьому одна частина специфікацій (*функціональні*) описує його функції, а інша частина (*експлуатаційні*) визначає вимоги до технічних засобів, до надійності, інформаційної безпеки і інш. Стосовно функціональних специфікацій маються на увазі наступні вимоги:

- вимога *повноти* означає, що специфікації повинні містити всю істотну інформацію, де нічого важливого не було б упущено, і відсутня неістотна інформація, наприклад деталі реалізації, щоб не перешкоджати розробникові у виборі найбільш ефективних рішень;
- вимога *точності* означає, що специфікації повинні однозначно сприйматися як замовником, так і розробником.

Останню вимогу виконати досить складно внаслідок того, що природна мова для опису специфікацій не підходить: навіть докладні специфікації на природній мові не забезпечують необхідної точності. Точні специфікації можна визначити, тільки розробивши деяку *формальну модель майбутнього* програмного забезпечення.

Формальні моделі, що використовуються на етапі визначення специфікацій можна розділити на дві групи: моделі, *залежні від підходу до розробки* (структурного або об'єктно-орієнтованого), і моделі, *не залежні* від нього. Так, діаграми переходів станів, які демонструють особливості поведінки майбутнього програмного забезпечення, при отриманні тих або інших сигналів ззовні, і математичні моделі наочної області використовують при будь-якому підході до розробки.

В рамках структурного підходу на етапі аналізу і визначення специфікацій використовують три типи моделей: орієнтовані на функції, орієнтовані на дані і орієнтовані на потоки даних. Кожну модель доцільно використовувати для свого специфічного класу програмних розробок.

27. Діаграми переходів станів. Функціональні діаграми.

Діаграма переходів станів (SDT) демонструє поведінку майбутньої програмної системи, при отриманні керуючих дій (ззовні).

Під *керуючими діями* або сигналами розуміють керуючу інформацію, яку отримує система ззовні. Наприклад, керуючими діями вважають команди користувача і сигнали давачів, підключених до комп'ютерної системи. Отримавши таку керуючу дію, система повинна виконати певні дії і, або залишитися в тому ж стані, або перейти в інший стан взаємодії із зовнішнім середовищем.

Умовні позначення, які використовуються при побудові діаграм переходів станів, показані на малюнку

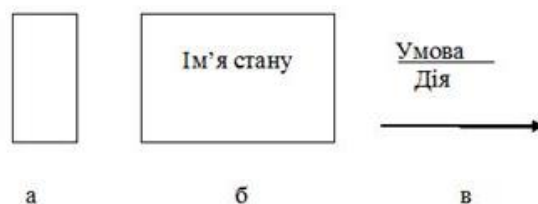


Рисунок 12.1 – Умовні позначення діаграм переходів станів:

- а-термінальний стан;
- б – проміжний стан;
- в - перехід

Якщо програмна система в процесі функціонування активно не взаємодіє з навколишнім середовищем (користувачем або давачами), наприклад, використовує примітивний інтерфейс і виконує деякі обчислення за заданими початковими даними, то діаграма переходів станів зазвичай інтересу не представляє. В цьому випадку вона демонструє тільки послідовно виконувані переходи: з початкового стану в стан введення даних потім після виконання обчислень - в стан виводу і, нарешті, в стан завершення роботи



Рисунок 12.2 - Діаграма переходів станів програмного забезпечення, яке активно не взаємодіє з навколишнім середовищем

Для інтерактивного програмного забезпечення з розвиненим призначенням для користувача інтерфейсом основні керуючі дії - команди користувача, для програмного забезпечення реального часу — сигнали від давачів і/або оператора виробничого процесу. Загальним для цих типів програмного забезпечення є наявність стану очікування, коли програмне забезпечення припиняє

працювати до отримання чергової керуючої дії. Для інтерактивного програмного забезпечення найбільш характерне отримання команд різних типів, а якщо це ще і програмне забезпечення реального часу - однотипних сигналів (або від багатьох давачів, або потребуючих тривалої обробки).

На відміну від інтерактивних систем для систем реального часу зазвичай встановлено більш жорстке обмеження на час обробки отриманого сигналу програмного забезпечення. Таке обмеження часто вимагає виконання додаткових досліджень поведінки системи в часі, наприклад, з використанням мереж Петрі або марківських процесів.

Функціональними називають діаграми, які в першу чергу відображають взаємозв'язок функцій взаємозв'язку функцій програмного забезпечення. Вони створюються на ранніх етапах проектування систем, для того щоб допомогти проектувальнику виявити основні функції і складові частини системи і, по можливості, знайти і виправити суттєві помилки. Сучасні методи структурного аналізу і проектування надають розробнику визначені синтаксичні і графічні засоби проектування функціональних діаграм інформаційних систем. В якості прикладу розглянемо методологію SADT, запропоновану Дугласом Россом. На основі неї розроблена, відома методологія IDEF() (Icam DEFinition). Методологія SADT представляє собою набір методів, правил і процедур, призначених для побудови функціональної моделі об'єкту деякої предметної області. Функціональна модель SADT відображає функціональну структуру об'єкту, тобто виконуваним ним дії і зв'язки між цими діями. Основні елементи цієї методології ґрунтуються на наступних концепціях:

- графічне представлення блочного моделювання. На SADT-діаграмі функції представляються у вигляді блоку, а інтерфейси входу-виходу – у вигляді дуг, які відповідно входять у блок і виходять з нього. Інтерфейсні дуги відображають взаємодію функцій одна з одною;
- строгість і точність відображення.

Правила SADT включають:

- унікальність міток і найменувань;
- обмеження кількості блоків на кожному рівні декомпозиції;
- синтаксичні правила для графіки;
- зв'язність діаграм;
- відділення організації від функції;
- розділення входів і керувань.

Методологія SADT може використовуватись для моделювання і розробки широкого кола систем, які задовольняють визначені вимоги і реалізують потрібні функції. У вже розроблених системах методологія SADT може бути використана для аналізу виконуваних ними функцій, а також для вказування механізмів, засобами яких вони здійснюються.

Діаграми – головні компоненти моделі, всі функції програмної системи та інтерфейси на них представлені як блоки і дуги. Місце з'єднання дуги з блоком визначає тип інтерфейсу. Дуга, яка означає керування, входить в блок зверху, в той час як інформація, яка підлягає обробці, представляється дугою з лівої сторони блоку, а результати обробки – дугами з правої сторони. Механізм (людина чи автоматизована система), який здійснює операцію, представляється у вигляді дуги, яка входить у блок знизу



Рис. 3.22. Функциональный блок и интерфейсные дуги

Блоки на діаграмі розміщуються по «сходинковій» схемі у відповідності з послідовністю їх роботи чи *домінуванням*, яке розуміється як вплив одного блоку на інші. У функціональних діаграмах SADT розрізняють п'ять типів впливів блоків один на одного:

- вхід-вихід блоку подається на вхід з меншим домінуванням, тобто наступного
- управління. Вихід блоку використовується як управління для блоку з меншим домінуванням
- зворотній зв'язок по виходу. Вихід блоку подається на вхід блоку з більшим домінуванням
- зворотній зв'язок по керуванню. Вихід блоку використовується як керуюча інформація для блоку з більшим домінуванням
- вихід-виконавець. Вихід блоку використовується як механізм для другого блоку

28. Діаграми потоків даних. Структури даних і діаграми відносин компонентів даних. Математичні моделі завдань, розробка або вибір методів рішення.

Діаграми потоків даних дозволяють специфікувати як функції програмного забезпечення, що розробляється, так і дані, що обробляються потоками. При використанні цієї моделі систему представляють у виді ієрархії діаграм потоків даних, що описують асинхронний процес перетворення інформації з моменту введення в систему до видачі користувачу. На кожному наступному рівні ієрархії відбувається уточнення процесів, поки черговий процес не буде визнаний елементарним. В основі моделі лежать поняття зовнішньої сутності, процесу, сховища (накопичувача) даних і потоку даних.


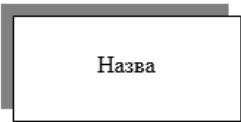


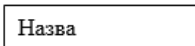
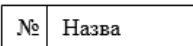
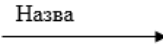
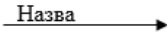
Зовнішня сутність - матеріальний об'єкт або фізична особа, що виступають в якості джерел або приймачів інформації, наприклад, замовники, персонал, постачальники, клієнти, банк і т.п.

Процес - перетворення вхідних потоків даних у вихідні відповідно до визначеного алгоритму. Кожен процес у системі має свій номер і зв'язаний з виконавцем, що здійснює дане перетворення. Як у випадку функціональних діаграм, фізично перетворення може здійснюватися комп'ютерами, вручну або спеціальними пристроями. На верхніх рівнях ієрархії, коли процеси ще не визначені, замість поняття «процес» використовують поняття «система» і «підсистема», що позначають відповідно систему в цілому чи її функціонально закінчену частину

Сховище даних - абстрактний пристрій для збереження інформації. Тип пристрою і способи приміщення, витягу і збереження для такого пристрою не деталізують. Фізично це може бути база даних, файл, таблиця в оперативній пам'яті, картотека на папері і т.п.

Потік даних — процес передачі деякої інформації від джерела до приймача. Фізично процес передачі інформації може відбуватися по кабелях під керуванням програми або програмної системи або вручну при участі пристроїв людей або поза проектованою системою.

Таким чином, діаграма ілюструє як потоки даних, породжені деякими зовнішніми сутностями, трансформуються відповідними процесами (або підсистемами), зберігаються накопичувачами даних і передаються іншим зовнішнім сутностям - приймачам інформації. У результаті ми одержуємо мережну модель збереження/обробки інформації. Для зображення діаграм потоків даних традиційно використовують два види нотацій: нотації Йордана і Гейна-Сарсона.

| Таблиця 4.1 | | |
|--------------------------------|---|---|
| Поняття | Нотація Йордана | Нотація Гейна-Сарсона |
| Зовнішня сутність |  |  |
| Система, підсистема або процес |  |  |
| Накопичувач даних |  |  |
| Потік |  |  |

Над лінією потоку, напрямом якого позначають стрілкою, указують, яка конкретно інформація в даному випадку передається.

Побудова ієрархії діаграм потоків даних починають з діаграми особливого виду - *контекстної діаграми*, що визначає найбільш загальний вид системи. На такій діаграмі показують, як система, що розробляється буде взаємодіяти з приймачами і джерелами інформації без указівки виконавців, тобто описують інтерфейс між системою і зовнішнім світом. Звичайно початкова контекстна діаграма має форму зірки. Якщо проєктована система містить велику кількість зовнішніх сутностей (більш 10-ти), має розподілену природу або включає вже існуючі підсистеми, то будують *ієрархії* контекстних діаграм.

Структурою даних називають сукупність правил і обмежень, що відбивають зв'язок, що існують між окремими частинами (елементами) даних.



Розрізняють *абстрактні структури* даних, використовувані для уточнення зв'язків між елементами, і *конкретні структури*, використовувані для представлення даних у програмах.

Всі абстрактні структури даних можна розділити на три групи: структури, елементи яких не зв'язані між собою, структури з неявними зв'язками елементів - таблиці і структури, зв'язок елементів яких указуються явно - графи. У першу групу входять *множини* та *кортежі*. Найбільш істотна характеристика елемента даних у цих структурах — його приналежність деякому набору, тобто *відношення входження*. Дані абстрактні структури використовують, якщо ніякі інші відносини елементів не є істотними для описуваних об'єктів. До другої групи відносять *вектори*, *матриці*, *масиви* (багатомірні), *записи*, *рядки*, а також *таблиці*, що включають перераховані структури в якості частин. Використання цих абстрактних типів може означати, що істотним є не тільки входження елемента даних у деяку структуру, але і їхній порядок, а також *відносини ієрархії* структур, тобто входження структури в структуру більш високого ступеня спільності. У тих випадках, коли істотні зв'язки елементів даних між собою, як модель структур даних використовують *графи*. Дуже істотно, що в реальності можливе вкладення структур даних, у тому числі і різних типів, а тому для їхнього опису можуть знадобитися спеціальні моделі. У залежності від описуваних типів відносин моделі структур даних прийнято поділяти на ієрархічні і мережні. *Ієрархічні моделі* дозволяють описувати упорядковані або не упорядковані відносини *входження* елементів даних у компонент більш високого рівня, тобто множини, таблиці і їхні комбінації. До ієрархічних моделей відносять модель Джексона-Орра, для графічного представлення якої можна використовувати:

- діаграми Джексона, запропоновані в складі методики проектування програмного забезпечення того ж автора в 1975 р.;
- діаграми побудовані на основі дужок Орра, запропоновані в складі методики проектування програмного забезпечення Варньє-Орра(1974).

Мережні моделі засновані на графах, а тому дозволяють описувати зв'язність елементів даних незалежно від виду відносини, у тому числі комбінації множин, таблиць і графів. До мережних моделей, наприклад, відносять модель «сутність-зв'язок» (ER - Entity-Relationship), звичайно використовувану при розробці баз даних.

Після постановки завдання слід найважливіший етап - вибір або розробка методу розв'язання задачі. Конкретний зміст етапу визначається розв'язуваною завданням і відбивається студентом у другому розділі пояснювальній записки, в якому пропонується висвітлити такі питання:

- Вибрати відповідно до постановкою завдання клас методів вирішення або можливих шляхів вирішення завдання;
- Розглянути суть методів, що входять в обраний клас, з аналізом їх переваг і недоліків;
- Вказати (розробити) радикальний спосіб, який буде використаний для вирішення завдання з обґрунтуванням його застосування;
- Скласти UML - моделі рішення задачі у вигляді діаграм варіантів використання, послідовності для варіантів використання, класів, розміщення і компонентів;

Дані питання в короткій формі повинні бути викладені в доповіді. Модель і алгоритм вирішення задачі повинні бути відбиті на слайдах презентації

29. Використання методу покрокової деталізації для проектування структури програмного забезпечення. Структурні карти Константайна.

Метод покрокової деталізації реалізує спадний підхід і базується на основних конструкціях структурного програмування. Він припускає покрокову розробку алгоритму. Кожен крок при цьому включає розкладання функції на підфункції. Так на першому етапі описують рішення поставленої задачі, виділяючи загальні підзадачі, на наступному аналогічно описують рішення підзадач, формулюючи при цьому підзадачі наступного рівня. Таким чином, на кожному кроці відбувається уточнення функцій проектованого програмного забезпечення. Процес продовжують, поки не доходять до підзадач, алгоритми рішення яким очевидні.

Декомпозируя програму методом покрокової деталізації, слід дотримуватися *основного правила структурної декомпозиції*, що виходить з принципу вертикального управління: в першу чергу деталізувати керуючі процеси декомпозируемого компонента, залишаючи уточнення операцій з даними накінець. Це пов'язано з тим, що пріоритетна деталізація управляючих процесів істотно спрощує структуру компонентів всіх рівнів ієрархії і дозволяє не відокремлювати процес прийняття рішення від його виконання: так, визначивши умови вибору деякої альтернативи, - відразу ж викликають модуль, що її реалізує.

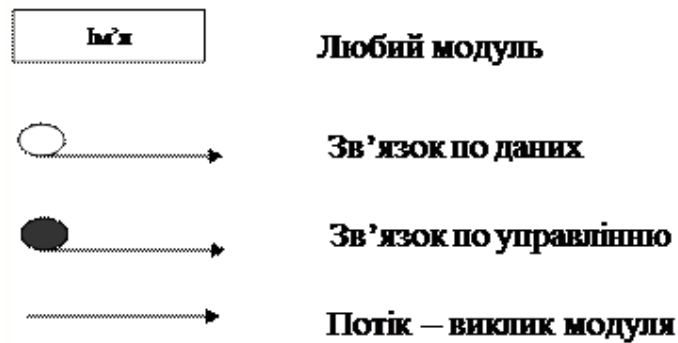
Деталізація операцій зі структурами в останню чергу дозволить відкласти уточнення їх специфікацій і забезпечить можливість відносно безболісної модифікації цих структур за рахунок скорочення кількості модулів, що залежать від цих даних.

Крім цього, *доцільно дотримуватися наступних рекомендацій*:

- не відокремлювати операції ініціалізації та завершення від відповідної обробки, так як модулі ініціалізації та завершення мають погану зв'язність (тимчасову) та сильне зчеплення (з управління);
- не проектувати занадто спеціалізованих або занадто універсальних модулів, так як проектування зайво спеціальних модулів збільшує їх кількість, а проектування зайво універсальних модулів підвищує їх складність;
- уникати дублювання дій у різних модулях, так як при їх зміні виправлення доведеться вносити у всі фрагменти програми, де вони виконуються - в цьому випадку доцільно просто реалізувати ці дії в окремому модулі;
- групувати повідомлення про помилки в один модуль за типом бібліотеки ресурсів, тоді буде легше узгодити формулювання, уникнути дублювання повідомлень, а також перенести повідомлення на іншу мову.

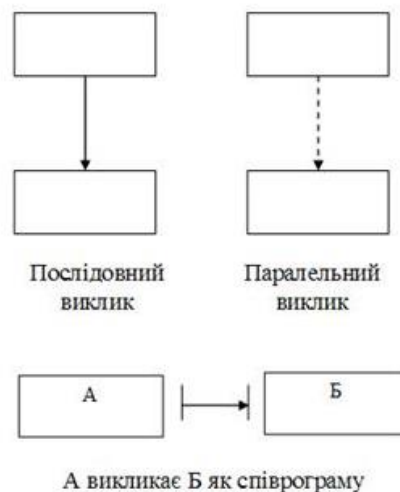
При цьому, описуючи рішення кожної задачі, бажано використовувати не більше 1-2-х структурних керуючих конструкцій, таких, як "цикл-поки" або розгалуження, що дозволяє чіткіше уявити собі структуру організованого обчислювального процесу.

Структурні карти Константайна є моделлю відносин ієрархії між програмними модулями.



Окремі частини програмної системи (програми, підпрограми) можуть **викликатися** послідовно, паралельно або як співпрограми. При послідовному виклику модулі викликаються в порядку їх слідування. При паралельному виклику модулі можуть викликатися у будь-якому порядку або одночасно (паралельно).

Найчастіше використовують *послідовний* виклик, при якому модулі, передавши керування чекають завершення виконання викликаного модуля або підпрограми, щоб продовжити перервану обробку.



Під *паралельним* викликом розуміють розпаралелювання обчислень на декількох обчислювачах, коли при активізації іншого процесу даний процес продовжує роботу. На однопроцесорних комп'ютерах в мультипрограмних середовищах в цьому випадку починається поперемінне виконання відповідних програм. Паралельні процеси бувають синхронні і асинхронні. Для синхронних процесів визначають точки синхронізації - моменти часу, коли проводиться обмін інформацією між процесами. Асинхронні процеси обмінюються інформацією тільки у момент активізації паралельного процесу.

Під *викликом співпрограми* розуміють можливість почергового виконання двох одночасно запущених програм, наприклад, якщо одна програма підготувала пакет даних для виводу, то друга може її вивести, а потім перейти в стан очікування наступного пакету. Причому в мультипрограмних системах основна програма, передавши дані, продовжує працювати, а не переходить в стан очікування.

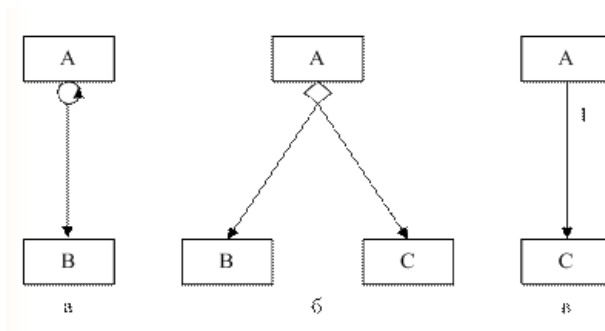
Якщо стрілка, що зображає виклик, стосується блоку, то звернення відбувається до модуля цілком, а якщо входить в блок, то - до елементу усередині модуля.

Для моделювання умовних і циклічних викликів застосовуються наступні вузли:

Умовний вузол використовується для умовного виклику модуля-нащадка. Він зображається за допомогою ромба, потоки - альтернативні виклики зображаються такими, що виходять з нього. Таким чином, якщо з ромба виходять два потоки, це відповідає конструкції *IF-THEN-ELSE*, якщо один потік - конструкції *IF-THEN*.

Ітераційний вузол використовується для того, щоб показати, що модуль-спадкоємець викликається в циклі. Він зображається півколом із стрілкою з потоками, що виходять з нього.

Якщо необхідно показати, що підлеглий модуль не викликається повторно при активації головного, це здійснюється вказівкою цифри "1" в головному модулі напроти потоку - виклику спадкоємця.



Структурні карти Константайна дозволяють наочно представити результат декомпозиції програми на модулі і оцінити її якість, тобто відповідність рекомендаціям структурного програмування (зчеплення і зв'язаність).

30. Проектування структур даних. Проектування програмного забезпечення, заснованого на декомпозиції даних.

Найважливішим елементом будь-якої бази даних, незалежно від її подання, є структура даних, яка відображає можливі подання відомостей для зберігання, вибірки і обробки. Структури даних необхідні для роботи з документарними даними, не пов'язаними з комп'ютерною обробкою, і з даними, що подаються в автоматизованих системах. У сенсі під термінів "Структура даних" розуміється одиниця відомостей, що дозволяє зберігати і обробляти безліч однотипних і (або) логічно пов'язаних даних. Одним з уявлень структури даних може бути форма документа, яка показує правила подання відомостей, відображаючи їх місце розташування, послідовність відображення, правила відображення, вид уявлення і т.д. Фактичне поява терміна "Структура даних" засноване на необхідності документарного подання відомостей і виникненні документа в якості джерела і засоби передачі інформації, що знайшло відображення у виникненні правил оформлення документів та подання в них відомостей. Структурованість документів, підпорядкована певним правилам, дозволяє людині краще розуміти особливості внесення в документ даних.

Декомпозиція — науковий метод, що використовує структуру завдання і дозволяє замінити вирішення одного великого завдання рішенням серії менших завдань, нехай і взаємопов'язаних, але більш простих. Декомпозиція, як процес розділення, дозволяє розглядати будь-яку досліджувану систему як складну, що складається з окремих взаємопов'язаних підсистем, які, в свою чергу, також можуть бути розділеними на частини. Як системи можуть виступати не тільки матеріальні об'єкти, а й процеси, явища і поняття. Існують різні типи декомпозиції, визначені в комп'ютерних науках: у **структурованому програмуванні** алгоритмічна декомпозиція розбиває процес на чітко визначені кроки. **Структурований аналіз** розбиває систему програмного забезпечення від системного контекстного рівня до системних функцій і об'єктів даних. **Об'єктно-орієнтована декомпозиція**, з іншого боку, розбиває велику систему на прогресивно менші класи або об'єкти, які відповідають за деяку частину проблемної області. **Алгоритмічна декомпозиція** є необхідною частиною об'єктно-орієнтованого аналізу та проектування, але об'єктно-орієнтовані системи починаються з розкладання на класи.

32. Розробка користуvalьницьких інтерфейсів. Типи

користуvalьницьких інтерфейсів й етапи їхньої розробки.

Психофізичні особливості людини, пов'язані зі сприйняттям, запам'ятовуванням й обробкою інформації. Користуvalьницька й програмна моделі інтерфейсу.

Інтерфейс користувача (ІК), (англ. *user interface, UI*) — засіб зручної взаємодії користувача з інформаційною системою. Сукупність засобів для обробки та відбиття інформації, якнайбільше пристосованих для зручності користувача; у графічних системах інтерфейс користувача, втілюється багатовіконним режимом, змінами кольору, розміру, видимості (прозорість, напівпрозорість, невидимість) вікон, їхнім розташуванням, сортуванням елементів вікон, гнучкими налаштуваннями як самих вікон, так і окремих їх елементів (файли, теки, ярлики, шрифти тощо), доступністю багатокористувацьких налаштувань. Види інтерфейсів

- Інтерфейс прямої обробки, це назва загального класу інтерфейсів користувача, який дозволяє користувачам маніпулювати наданими їм об'єктами, з використанням дій, котрі принаймні, відповідають фізичному світу.
- Графічні інтерфейси користувача (GUI) приймають вхідні дані за допомогою таких пристроїв, як комп'ютерна клавіатура та миша, й забезпечують графічний висновок на моніторі комп'ютера.
- Веб-інтерфейси користувача або веб-інтерфейси користувача (WUI), які приймають вхідні дані та забезпечують виведення, створенням веб-сторінок, які передаються Інтернетом і проглядаються користувачем за допомогою програми веб-браузера.

Можна виділити наступні етапи роботи над UX-проектом. Перед початком роботи над проектом формуються цілі і завдання з урахуванням всіх особливостей продукту. Наступним пунктом є аналіз бажаної аудиторії, яка матиме доступ до сайту. На основі перших двох параметрів опрацьовуються варіанти можливої взаємодії з тим розрахунком, щоб максимально задовольнити попит споживачів та завдання компанії. Виходячи з цього, проектується структура продукту і її зовнішній вигляд. UX / UI дизайн допомагає зробити додаток комфортнішим для користувача.

Перший тип проблем характеризується, серед іншого, **мінімальним впливом розробників комп'ютерних систем**. Тому, їх специфіка, з психологічної точки зору, не представляє суттєвого інтересу. Вочевидь, значно більше зацікавленості викликають друга і третя тенденції. Зокрема, йдеться про **властивості сприйняття просторової форми** предметів, які (як вказувалося вище) є на даний момент статичними й двовимірними; і, також, **ергономічні властивості** системи.

Розглянемо 4 моделі КІ – ментальну, користуvalьську, програміста, проектуvalьника. **Ментальна, або концептуальна модель** - лише внутрішнє відображення того, як користувач розуміє і взаємодіє з системою.

Модель користувача. Поганий дизайн інтерфейса викликає у користувачів сумніви в правильності своїх дій. Єдиний спосіб визначення вигляду користуvalьської моделі – поговорити з користувачем та подивитись, як він працює. Рекомендується 5 способів збирання інформації про користувача: аналіз їх задач; інтерв'ю з справжніми та потенційними користувачами; відвідування місць їх роботи; відгуки клієнтів; тести по придатності. Спілкуватись потрібно зі справжніми користувачами, а не з їх менеджерами та керівництвом.

Модель програміста – найлегша для відображення, оскільки вона може бути формально і недвозначно описана. Дана модель є функційною специфікацією програмного продукту.

Модель проектуvalьника. Проектуvalьник дізнається про ідеї, побажання користувача, поєднує їх зі своїми навичками та матеріалами, необхідними для програміста, та проектує ПЗ, яке повинно задовольняти потреби користувача. Модель проектуvalьника – це дещо середнє між моделлю користувача та моделлю програміста.

Кращою моделлю є та, що дозволяє вибрати необхідний рівень деталізації ПЗ. Іноді проста й нашвидкуруч створена модель програмного інтерфейсу є найприйнятнішим варіантом. В інших випадках доводиться працювати на рівні бітів (специфікація міжсистемних інтерфейсів тощо). У кожному випадку кращою моделлю буде та, яка дозволяє вибрати рівень деталізації залежно від того, хто із якою метою на неї дивиться. Для аналітика або користувача найбільший інтерес представляє питання «що робити», а для розробника — «як зробити». В обох випадках необхідна можливість розглядати систему на різних рівнях деталізації в різний час.

33. Класифікації діалогів і загальні принципи їхньої розробки. Основні компоненти графічних користувацьких інтерфейсів. Реалізація діалогів у графічному користувацькому інтерфейсі. Користувальницькі інтерфейси прямого маніпулювання і їхнє проектування. Інтелектуальні елементи користувацьких інтерфейсів.

Виділяють чотири основні структури типів діалогу: запитання і відповідь; меню; екранні форми; команди. Придатність структури до діалогу можна оцінювати за такими основними критеріями: природністю; послідовністю; стислістю (короткий); підтримкою користувача; гнучкістю.

Природним діалогом є такий, який не змушує користувача, котрий взаємодіє із системою, суттєво змінювати свої традиційні прийоми роботи.

Стислий діалог потребує від користувача введення лише мінімуму інформації, яка необхідна для роботи системи.

Інструкції для користувача виводяться у вигляді підказок чи довідкової інформації.

Повідомлення про помилки мають точно пояснювати, в чому помилка і які дії потрібно зробити, аби її виправити, а не обмежуватися загальними фразами «синтаксична помилка» чи таємничими кодами типу «ОС1».

У графічному інтерфейсі виділяють головне, системне, контекстне меню та меню додатка.

- **Головне меню** виникає на екрані після натискання кнопки Пуск (Start) робочого столу.
 - **Системне меню** призначається для керування вікнами і пропонує дії: згорнути, розгорнути, закрити, перемістити або змінити розміри вікна.
 - **Контекстне меню** виникає за натисканням правої кнопки миші в будь-якому місці робочого столу або вікна документу.
 - Програма, з якою працює користувач, у свою чергу, пропонує меню для взаємодії з нею.
- Це меню додатка.**

Пряме маніпулювання звільняє користувача від необхідності вивчати спеціальну комп'ютерну семантику і синтаксис для взаємодії з системою. Інтерфейси прямого маніпулювання характеризуються високим ступенем узгодженості. Велику різноманітність операцій можна виконати невеликим набором механізмів.

Інтелектуальні інтерфейси користувача, це людино-машинні інтерфейси, які спрямовано на підвищення якості, продуктивності і природності взаємодії між людиною та машиною, за допомогою уявлення, обґрунтування і впливу на моделі користувача, домену, завдання, дискурсу і середовища (наприклад, графіки, природна мова, жест).

34.Тестування програмних продуктів. Види контролю якості розроблюваного програмного забезпечення. Формування тестових наборів. Ручний контроль програмного забезпечення

Тестування - дуже важливий і трудомісткий етап процесу розробки програмного забезпечення, оскільки правильне тестування дозволяє виявити переважну більшість помилок, допущених при складанні програм.

Процес розробки програмного забезпечення передбачає три стадії тестування:

автономне, комплексне і системне, кожна з яких відповідає завершенню відповідної частині Системи.

Розрізняють два підходи до формування тестів: структурний і функціональний. Кожен з вказаних підходів має свої особливості і сфери застосування.

Недостатньо виконати проектування і кодування програмного забезпечення необхідно також забезпечити його відповідність вимогам і специфікаціям. Багато разів дослідження, що проводяться, показали, що чим раніше виявляються ті або інші невідповідності або помилки, тим більша ймовірність їх правильного виправлення і нижча вартість.

Сучасні технології розробки програмного забезпечення передбачають раннє виявлення помилок за рахунок виконання контролю результатів всіх етапів і стадій розробки.

На початкових етапах такий контроль здійснюють в основному уручну або з використанням CASE-засобів, на останніх - він набуває форми тестування.

Тестування - це процес виконання програми, метою якого є виявлення помилок.

забезпеченні. Для такого програмного забезпечення виконання повного тестування, тобто завдання всіх можливих комбінацій вихідних даних, стає неможливим, а, отже, завжди є вірогідність того, що в програмному забезпеченні залишилися не виявлені помилки. Проте дотримання основних правил тестування і науково обґрунтований підбір тестів може зменшити їх кількість.

Примітка. Зазвичай на питання про мету тестування початкуючі програмісти

Формування тестових наборів. Відповідно до визначення тестування на початку даного параграфа, вдалим слід рахувати тест, який виявляє хоч би одну помилку.

З цієї точки зору хотілося б використовувати такі набори тестів, кожен з яких з максимальною вірогідністю може виявити помилку.

Формування набору тестів має велике значення, оскільки тестування є одним з найбільш трудомістких етапів (від 30 до 60 % спільній трудомісткості) створення програмного продукту. Причому доля вартості тестування в спільній вартості розробки має тенденцію зростати при збільшенні складності програмного забезпечення і підвищенні вимог до їх якості.

Існують два принципово різних підходу до формування тестових наборів:

структурний і функціональний.

Структурний підхід базується на тому, що відома структура тестованого програмного забезпечення, у тому числі його алгоритми («скляний ящик»). В цьому випадку тести будують так, щоб перевірити правильність реалізації заданої логіки в коді програми.

Функціональний підхід ґрунтується на тому, що структура програмного забезпечення не відома («чорний ящик»). В цьому випадку тести будують, спираючись на функціональну специфікацію. Цей підхід називають також підходом, керованим даними, оскільки при його використанні тести будують на базі різних способів декомпозиції безлічі даних.

Набори тестів, отримані відповідно до методів цих підходів, зазвичай об'єднують забезпечуючи всестороннє тестування програмного забезпечення.

Детальніший розгляд перерахованих питань почнемо з обговорення методів

ручного контролю.

Ручний контроль, як вказано вище, зазвичай використовують на ранніх етапах розробки. Все проектні рішення, прийняті на тому або іншому етапі, повинні аналізуватися з точки зору їх правильності і доцільності якомога раніше, поки їх можна легко переглянути.

Оскільки можливість практичної перевірки подібних рішень на ранніх етапах розробки відсутня, велике значення має їх обговорення, яке проводять в різних формах.

Розрізняють статичний і динамічний підходи до ручного контролю. При статичному підході аналізують структуру, зв'язки, що управляють і інформаційні, програми, її вхідні і вихідні дані. При динамічному - виконують ручне тестування, тобто уручну моделюють процес виконання програми на заданих вихідних даних.

Вихідними даними для таких перевірок є: технічне завдання, специфікації, структурна і функціональна схеми програмного продукту, схеми окремих компонентів і т. д., а для пізніших етапів - алгоритми і тексти програм, а також тестові набори.

Доведено, що ручний контроль сприяє істотному збільшенню продуктивності і підвищенню надійності програм і з його допомогою можна знаходити від

30 до 70 % помилок логічного проектування і кодування. Отже, один або декілька з методів ручного контролю обов'язково повинні використовуватися в кожному програмному проекті.

Основними методами ручного контролю є:

- інспекції вихідного тексту
- скрізні перегляди
- перевірка за столом
- оцінки програм.

Інспекції вихідного тексту. Інспекції вихідного тексту є набором процедур і прийомів виявлення помилок при вивченні тексту групою фахівців. У цю групу входять: автор програми, проектувальник, фахівець з тестування і координатор - компетентний програміст, але не автор програми. Спільна процедура інспекції передбачає наступні операції:

- учасникам групи заздалегідь видається лістинг програми і специфікація на неї;

- програміст розповідає про логіку роботи програми і відповідає на питання

інспекторів;

- програма аналізується за списком питань для виявлення тих, що історично склалися спільних помилок програмування.

Список питань для інспекцій вихідного тексту залежить, як від використовуваної мови програмування, так і від специфіки програмного забезпечення, що розробляється. У якості прикладу нижче приведений список питань, який можна використовувати при аналізі правильності програм, написаних на мові Pascal.

· Контроль звернень до даних

- Чи всі змінні ініціалізували?
- Чи не перевищені максимальні (або реальні) розміри масивів і рядків?
- Чи не переплутані рядки із стовпцями при роботі з матрицями?
- Чи присутні змінні з схожими іменами?
- Чи використовуються файли? Якщо так, то при введенні з файлу чи перевіряється завершення файлу?
- Чи відповідають типи записуваних і читаних значень?
- Чи використані змінні, що не типізуються, відкриті масиви, динамічна пам'ять?

Якщо так, то чи відповідають типи змінних при «накладенні» формату? Чи не виходять індекси за межі масивів?

36.Складання програмної документації. Види програмних документів. Пояснювальна записка. Керівництво користувача. Основні правила оформлення програмної документації. Вимоги до оформлення розрахунково-пояснювальної записки при курсовому програмуванні.

Технічну Документацію зручно складати за допомогою спеціальних сервісів генераторів документації: (**Doxygen, NDoc, Javadoc, Visual Expert, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, Universal Report in.**) Генератори документації дуже зручно допомагають постійно підтримувати документацію в актуальному стані. Ці інструменти витягують коментарі із коду. За потреби зміст можна редагувати. Плюс їх можна ще використовувати при збірках програми.

Отже, Технічна Документація є складовою частиною вихідного коду. Вона записується у файлі **README**.

Також Технічна документація може бути сформована у текстові довідкові посібники до друку, кому як зручно.

Технічна Документація використовується усіма: розробниками, тестувальниками, адміністраторами, кінцевими користувачами, які використовують програмне забезпечення. Для усіх вона є важливою, бо у ній фіксуються усі зміни у програмі.

Види програмних документів

| Вид програмного документу | Зміст програмного документу |
|---------------------------------|---|
| Специфікація | Склад програми і документації на неї. |
| Утримувачі оригіналів | Перелік організацій, на яких зберігаються оригінали програмних документів. |
| Текст програми | Запис програми з необхідними коментарями. |
| Опис програми | Відомості про логічну структуру і функціонування програми. |
| Програма і методика випробувань | Вимоги, що підлягають перевірці при випробуванні програми, а також порядок і методи її контролю. |
| Технічне завдання | Призначення і галузь застосування програми технічні, техніко-економічні і спеціальні вимоги, задані програмі, необхідні стадії і строки розробки, види випробувань. |
| Пояснювальна записка | Схема алгоритму, загальний опис алгоритму та (або) функції програми, а також обґрунтування прийнятих технічних і техніко-економічних рішень. |

Експлуатаційні документи Відомості для забезпечення функціонування і експлуатації програми.

Пояснювальна записка - обґрунтування прийнятих і застосованих технічних і техніко-економічних рішень, схеми та опис алгоритмів, загальний опис роботи програмного виробу;

Керівництво системного програміста - містить відомості для перевірки, налаштування і функціонування програми при конкретному застосуванні;

Керівництво користувача - документ, призначення якого - НАДАТИ людям допомогу в використанні деякої системи. Документ входить до складу технічної документації на систему і, як правило, готується технічним письменником.

Основні правила оформлення програмної документації

Правила оформлення документа і його частин на кожному носії даних встановлюються стандартами ЕСПД на правила оформлення документів на відповідних носіях даних.

ГОСТ 19.106-78 ЕСПД. Вимоги до програмних документів, виконаним друкованим способом

Програмні документи оформляють:

- на аркушах формату А4 (ГОСТ 2.301-68) при виготовленні документа машинописним або рукописним способом;
- допускається оформлення на аркушах формату А3;
- при машинному способі виконання документа допускаються відхилення розмірів листів, відповідних форматів А4 та А3, обумовлені можливостями застосовуваних технічних засобів; на аркушах форматів А4 та А3, передбачаються вихідними характеристиками пристроїв виведення даних, при виготовленні документа машинним способом;
- на аркушах типографічних форматів при виготовленні документа друкарським способом.

Розташування матеріалів програмного документа здійснюється в наступній послідовності:

титульна частина:

- лист твердження (не входить в загальну кількість аркушів документа);
- титульний лист (перший аркуш документа);

інформаційна частина:

- анотація;
- лист змісту;

основна частина:

- текст документа (з малюнками, таблицями і т.п.)
- перелік термінів і їх визначень;
- перелік скорочень;
- додатка;
- предметний покажчик;
- перелік посилальних документів;

частина реєстрації змін:

- лист реєстрації змін.

Розрахунково-пояснювальна записка повинна містити наступні складові:

- титульний лист;
- завдання на роботу;
- реферат;
- перелік умовних позначень, символів та скорочень;
- зміст;
- вступ;
- основну частину;
- висновки;
- список використаної літератури;
- додатки.
- Розрахунково-пояснювальна записка є текстовим документом проекту і виконується відповідно з вимогами ДСТУ 3008-95. Пояснювальна записка виконується на 40-50 сторінках рукописного або машинописного тексту на одній сторінці білого паперу. На відміну від вимог до текстових документів [1] пояснювальну записку курсового проекту можна виконувати без рамок і основних надписів звичайним (не кресленим) рукописним шрифтом чорними, фіолетовими, синіми чорнилами (пастою) або з використанням комп'ютера.