Лекція 3

Прийоми забезпечення технологічності програмних продуктів.
Низхідне та висхідне програмування
Метод покрокової деталізації.

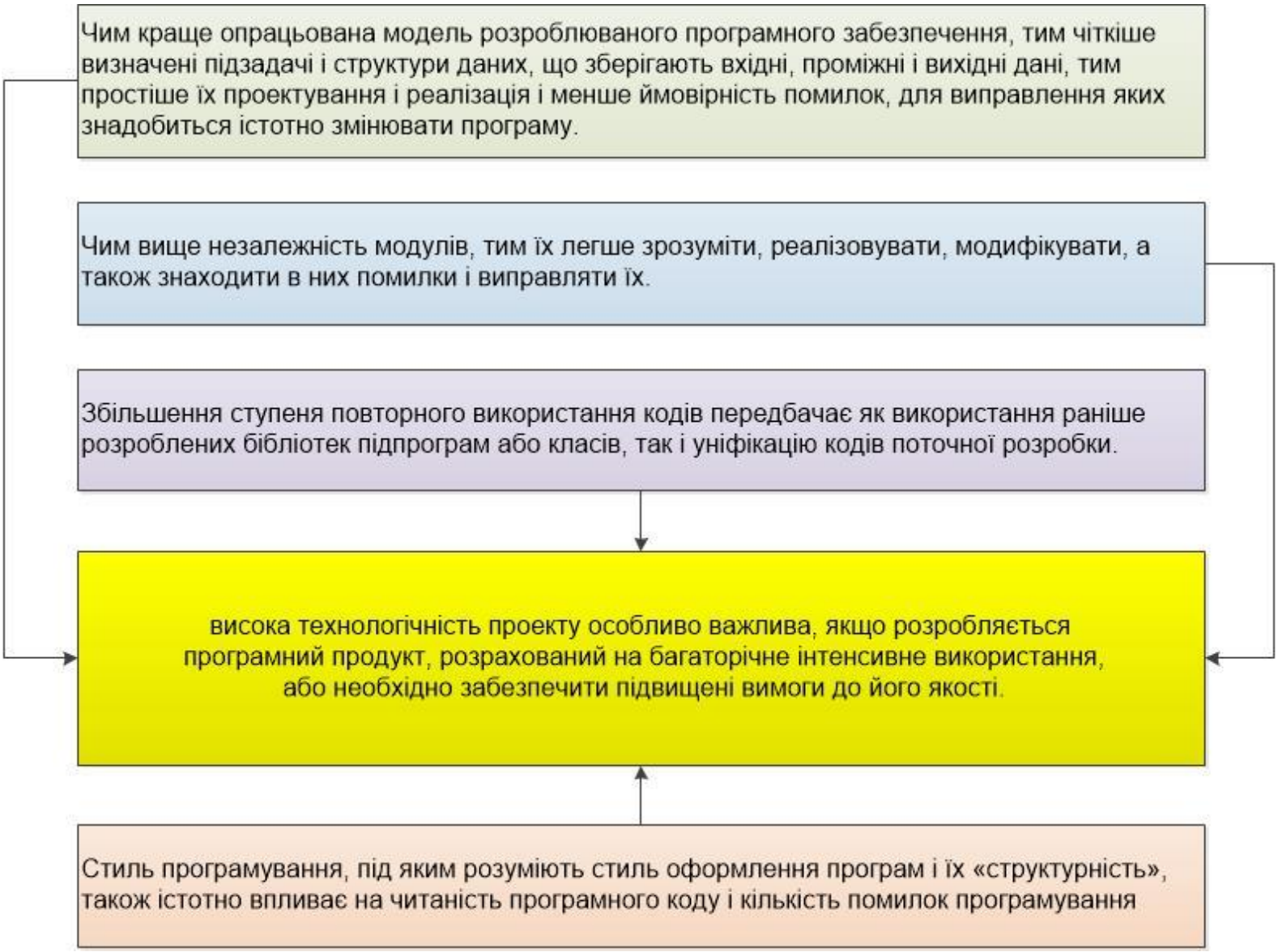
Прийоми забезпечення технологічності ПП

- Для забезпечення необхідних технологічних властивостей застосовують спеціальні технологічні прийоми і слідують певними методиками, сформульованим всім попереднім досвідом створення програмного забезпечення. До таких прийомів і методикам відносять правила декомпозиції, методи проектування, програмування і контролю якості. В його основу було покладено такі основні концепції:
  - низьхідна розробка;
  - модульне програмування;
  - структурне програмування;
  - наскрізний структурний контроль.

Поняття технологічності ПЗ

- Під ***технологічністю*** розуміють якість проекту програмного продукту, від якого залежать трудові та матеріальні витрати на його реалізацію і наступні модифікації.
- ***Хороший проект порівняно швидко і легко кодується, тестується, налагоджується і модифікується.***
- З досвіду кількох поколінь розробників програмного забезпечення відомо, що технологічність програмного забезпечення визначається розробленою його моделей, рівнем незалежності модулів, стилем програмування і ступенем повторного використання кодів.

Поняття технологічності ПЗ



Модулі та їх властивості

- При проектуванні досить складного програмного забезпечення після визначення його загальної структури виконують декомпозицію компонентів відповідно до обраного підходом до отримання елементів, які, на думку проектувальника, в подальшій декомпозиції не потребують.
- В даний час використовують два способи декомпозиції розроблюваного програмного забезпечення:
  - процедурний (або структурний);
  - об'єктний.

Модулі та їх властивості

Процедурна декомпозиція	Результатом є ієрархія підпрограм (процедур), в якій функції, пов'язані з прийняттям рішення, реалізуються підпрограмами верхніх рівнів, а безпосередньо обробка - підпрограмами нижніх рівнів.
Об'єктна декомпозиція	Результатом є сукупність об'єктів, які потім реалізують як змінні деяких спеціально розроблених типів (класів), що представляють собою сукупність полів даних і методів, які працюють е цими полями.

При будь-якому способі декомпозиції отримують набір пов'язаних е відповідними даними підпрограм, які в процесі реалізації організовують в модулі

Модулі

- Модулем ***називають автономно компільовані програмну одиницю.***
- Термін «модуль» традиційно використовується в двох сенсах.
  1. Спочатку, коли розмір програм був порівняно невеликий, і все підпрограми компілювалися окремо, під модулем розумілася підпрограма, тобто послідовність пов'язаних фрагментів програми, звернення до якої виконується по імені.
  2. 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.***

Модулі

Спочатку до модулів (ще розуміється як підпрограми) пред'являлися наступні вимоги:

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

Згодом, коли основні вимоги структурного підходу стали підтримуватися мовами програмування, і під модулем стали розуміти окремо компільовані бібліотеку ресурсів, вимога ***незалежності модулів стало основним.***

Модулі

Чим вище ступінь незалежності модулів, тим:

1. легше розібратися в окремому модулі і всій програмі і відповідно тестувати, налагоджувати і модифікувати її;
2. менша ймовірність появи нових помилок при виправленні ста-яких або внесення змін в програму, тобто ймовірність появи «хвильового» ефекту;
3. простіше організувати розробку програмного забезпечення групою програмістів і легше його супроводжувати.

Зчеплення модулів

- ***Зчеплення*** є мірою взаємозалежності модулів, яка визначає, наскільки добре модулі відокремлені один від одного.
- Модулі незалежні, якщо кожен з них нс містить про інше ніякої інформації. Чим більше інформації про інших модулях зберігає модуль, тим більше він з ними зчеплений.

Розрізняють п'ять типів зчеплення модулів:

- за даними;
- за зразком;
- управління;
- загальної області даних;
- вмісту.

Зчеплення модулів

Зчеплення за даними передбачає, що модулі обмінюються даними, представленими скалярними значеннями. При невеликій кількості переданих параметрів цей тип забезпечує найкращі технологічні характеристики програмного забезпечення.

Наприклад, функція Max передбачає зчеплення за даними через параметри скалярного типу:

```
int Max(int a, int b);
{
  if a>b
     return(a);
 else
   return(b);
};
```

Зчеплення модулів

Зчеплення за зразком передбачає, що модулі обмінюються даними, об'єднаними в структури. Цей тип також забезпечує непогані характеристики, але вони гірше, ніж у попереднього типу, так як конкретні дані, що передаються «заховані» в структури, і тому зменшується «прозорість» зв'язку між модулями. Крім того, при зміні структури даних, що передаються необхідно модифікувати всі використовують її модулі.

Так, функція MaxElement, описана нижче, передбачає зчеплення за зразком (параметр a - відкритий масив).

```
int MaxElement(int a[], int n);
{
int MaxEl :=a[0];
for(i=0; i=n; i++)
   if (a[i]>MaxEl)
      MaxEl=a[i];
return(MAxEl);
};
```

Зчеплення модулів

При зчепленні по управлінню, це коли один модуль посилає іншому деякий інформаційний об'єкт (прапор), призначений для управління внутрішньою логікою модуля. Таким способом часто виконують настройку режимів роботи програмного забезпечення. Подібні настройки також знижують наочність взаємодії модулів і тому забезпечують ще гірші характеристики технологічності розроблюваного програмного забезпечення в порівнянні з попередніми типами зв'язків.

Наприклад, функція MinMax передбачає зчеплення з управління, так як значення параметра flag впливає на логіку програми.

```
int MinMax(int a, int b, bool flag);
{
if ((a>b) && (flag))
 return(a);
else
 return(b);
}
```

Зчеплення модулів

Зчеплення за загальною областю даних передбачає, що модулі працюють із загальною областю даних. Цей тип зчеплення вважається неприпустимим, оскільки:

- програми, що використовують даний тип зчеплення, дуже складні для розуміння при супроводі програмного забезпечення;
- помилка одного модуля, що призводить до зміни загальних даних, можуть проявитися при виконанні іншого модуля, що істотно ускладнює локалізацію помилок;
- при посиланні до даних в загальній області модулі використовують конкретні імена, що зменшує гнучкість розроблюваного програмного забезпечення.

Наприклад, функція MaxA, яка використовує глобальний масив A, зчеплена з основною програмою по загальній області:

```
 int MaxA();
{
   int tmp=A[0];
  for (int  i=1; i<N; i++)
     if (a[i]>tmp)
       tmp=a[i];
return (tmp);
};
```

Зчеплення модулів

- У разі ***зчеплення по вмісту*** один модуль містить звернення до внутрішніх компонентів іншого (передає управління всередину, читає і / або змінює внутрішні дані або самі коди), що повністю суперечить блочно-ієрархічному підходу.
- Окремий модуль в цьому випадку вже не є блоком («чорним ящиком»): його вміст має враховується в процесі розробки іншого модуля.
- Сучасні універсальні мови процедурного програмування, наприклад Pascal, даного типу зчеплення в явному вигляді не підтримують, але для мов низького рівня, наприклад Ассемблера, такий вид зчеплення залишається можливим.

Зв’язність модулів.

- ***Зв’язність*** - міра міцності з'єднання функціональних та інформаційних об'єктів всередині одного модуля.
- Якщо зчеплення характеризує *якість відділення модулів*, то зв'язність *характеризує ступінь взаємозв'язку елементів*, що реалізуються одним модулем.
- Розміщення сильно пов'язаних елементів в одному модулі ***зменшує міжмодульні зв'язку і відповідно взаємовплив модулів***. У той же час розміщення сильно пов'язаних елементів в різні модулі не тільки *підсилює міжмодульні зв'язки, а й ускладнює розуміння їх взаємодії*. Об'єднання слабо пов'язаних елементів також зменшує технологічність модулів, так як такими елементами складніше маніпулювати.
- Розрізняють такі види зв'язності (в порядку убування рівня):
  - функціональну;
  - послідовну;
  - інформаційну (комунікативну);
  - процедурну;
  - тимчасову;
  - логічний;
  - випадкову.

Зв’язність модулів.

- При ***функціональній зв'язності*** *вага додаткових компонентів призначені для виконання однієї функції: операції, що об'єднуються для виконання однієї функції, або дані, пов'язані з однією функцією.*
- Модуль, елементи якого пов'язані функціонально, має чітко визначену мету, при його виклику виконується одне завдання. *Такий модуль має максимальну зв'язність, наслідком якої є його хороші технологічні якості: простота тестування, модифікації і супроводу.* ***Саме з цим пов'язане одне з вимог структурної декомпозиції «один модуль - одна функція».***
- З тих же міркувань слід уникати неструктурованого розподілу функції між модулями - бібліотеками ресурсів. Наприклад, якщо при проектуванні текстового редактора передбачається функція редагування, то краще організувати модуль - бібліотеку функцій редагування, ніж розмістити частину функцій в один модуль, а частину в інший.

Зв’язність модулів.

- При ***послідовній зв'язності*** функцій вихід однієї функції служить вихідними даними для іншої функції. Як правило, такий модуль має одну точку входу, тобто реалізує одну підпрограму, що виконує дві функції. Вважають, що дані, які використовуються послідовними функціями, також пов'язані послідовно. Модуль з послідовної зв'язністю функцій можна розбити на два або більше модулів, як з послідовною, так і з функціональної зв'язності.

- ***Інформаційно зв'язаними*** вважають функції, що обробляють одні й тіж дані. При використанні структурних мов програмування роздільне виконання функцій можна здійснити тільки, коли кожна функція реалізується своєї підпрограмою. Хоча раніше в подібних випадках зазвичай використовували різні точки входу в модуль, оформлений як одна підпрограма.

Зв’язність модулів.

- ***Процедурно зв'язані*** функції або дані, які є частинами одного процесу. Зазвичай модулі з процедурної зв’язністю функцій отримують, якщо в модулі об'єднані функції альтернативних частин програми.

- ***Тимчасова зв'язність*** функцій передбачає, що ці функції виконуються паралельно або протягом деякого періоду часу. Тимчасова зв'язність даних означає, що вони використовуються в деякому часовому інтервалі. Наприклад, тимчасову зв'язність мають функції, які виконуються під час ініціалізації деякого процесу.

Бібліотеки ресурсів.

- Розрізняють бібліотеки ресурсів двох типів:
  - бібліотеки підпрограм
  - бібліотеки класів.
- ***Бібліотеки підпрограм*** реалізують функції, близькі за призначенням, наприклад бібліотека графічного виведення інформації. Можливості підключення підпрограм між собою в такій бібліотеці - логічна, а зв'язність самих підпрограм - функціональна, так як кожна з них зазвичай реалізує одну функцію.
- ***Бібліотеки класів*** реалізують близькі за призначенням класи. Можливості підключення елементів класу - інформаційна, зв'язність класів між собою може бути функціональною - для родинних або асоційованих класів і логічної - для інших.
- Як засіб поліпшення технологічних характеристик бібліотек ресурсів в даний час широко використовують поділ тіла модуля на ***інтерфейсну частину і область реалізації.***

Низхідна та висхідна розробка ПЗ

- При проектуванні, реалізації та тестуванні компонентів структурної ієрархії, отриманої при декомпозиції, застосовують два підходи:
  - висхідний;
  - низхідний або спадний.

Висхідний підхід

- При використанні висхідного підходу спочатку проектують і реалізують компоненти нижнього рівня, потім попереднього і т. д. У міру завершення тестування і налагодження компонентів здійснюють їх складання, причому компоненти нижнього рівня при такому підході часто поміщають в бібліотеки компонентів.
- ***Для тестування і відлагодження компонентів проектують і реалізують спеціальні тестуючі програми.***
- ***Підхід має такі недоліки:***
  - збільшення ймовірності неузгодженості компонентів внаслідок неповноти специфікацій;
  - наявність витрат на проектування і реалізацію тестуючих програм, які не можна перетворити в компоненти;
  - пізне проектування інтерфейсу, а відповідно неможливість продемонструвати його замовнику для уточнення специфікацій і т. д.

Низхідний підхід

- Спадний підхід передбачає, що проектування і подальша реалізація компонентів виконується «***зверху-вниз»,*** тобто спочатку проектують компоненти верхніх рівнів ієрархії, потім наступних і так далі до самих нижніх рівнів. У тій же послідовності виконують і реалізацію компонентів. При цьому в процесі програмування компоненти нижніх, ще не реалізованих рівнів замінюють спеціально розробленими налагоджувальними модулями - «заглушками», що дозволяє тестувати і налагоджувати вже реалізовану частину.
- При використанні спадного підходу застосовують методи визначення послідовності проектування і реалізації компонентів:
  - ієрархічний,
  - операційний,
  - комбінований

Низхідний підхід

- ***Ієрархічний метод*** передбачає виконання розробки строго за рівнями. Винятки допускаються при наявності залежності за даними, тобто якщо виявляється, що деякий модуль використовує результати іншого, то його рекомендується програмувати після цього модуля. Основною проблемою даного методу є велика кількість досить складних заглушок. Крім того, при використанні даного методу основна маса модулів розробляється і реалізується в кінці роботи над проектом, що ускладнює розподіл людських ресурсів.
- ***Операційний метод*** пов'язує послідовність розробки модулів з порядком їх виконання під час запуску програми. Застосування методу ускладняються тим, що порядок виконання модулів може залежати від даних. Крім того, модулі виводу результатів, незважаючи на те, що вони викликаються останніми, повинні розроблятися одними з перших, щоб не проектувати складну заглушку, що забезпечує виведення результатів при тестуванні.

Низхідний підхід

Комбінований метод враховує такі фактори, що впливають на послідовність розробки:

- досяжність модуля - наявність всіх модулів в ланцюжку виклику даного модуля;
- залежність за даними - модулі, що формують деякі дані, повинні створюватися раніше обробних;
- забезпечення можливості видачі результатів - модулі виводу результатів повинні створюватися раніше обробних;
- готовність допоміжних модулів - допоміжні модулі, наприклад модулі закриття файлів, завершення програми, повинні створюватися раніше обробних;
- наявність необхідних ресурсів.

Низхідний підхід

Комбінований метод враховує такі фактори, що впливають на послідовність розробки:

- досяжність модуля - наявність всіх модулів в ланцюжку виклику даного модуля;
- залежність за даними - модулі, що формують деякі дані, повинні створюватися раніше обробних;
- забезпечення можливості видачі результатів - модулі виводу результатів повинні створюватися раніше обробних;
- готовність допоміжних модулів - допоміжні модулі, наприклад модулі закриття файлів, завершення програми, повинні створюватися раніше обробних;
- наявність необхідних ресурсів.

Низхідний підхід

Комбінований метод враховує такі фактори, що впливають на послідовність розробки:

- досяжність модуля - наявність всіх модулів в ланцюжку виклику даного модуля;
- залежність за даними - модулі, що формують деякі дані, повинні створюватися раніше обробних;
- забезпечення можливості видачі результатів - модулі виводу результатів повинні створюватися раніше обробних;
- готовність допоміжних модулів - допоміжні модулі, наприклад модулі закриття файлів, завершення програми, повинні створюватися раніше обробних;
- наявність необхідних ресурсів.

Вимоги до методології програмування

- ***Методологія програмування*** покликана:
- допомагати програмісту керувати складністю задачі, що розв'язується, і давати керівні вказівки для формулювання рішення задачі;
- вимагати від програміста дотримання правил запису процесу розробки програм; цю програмну розробку можуть потім читати інші програмісти, оцінити її і дати конструктивні поради або критикувати;
- допомагати створювати програми, які є зрозумілими;
- приводити до програм, правильність яких може бути доведена; якщо доведення правильності занадто складне, то методологія повинна надавати систематичний підхід до тестування програм;
- бути загально застосуванною, а не обмежуватися одним класом завдань;
- допомагати в створенні високоефективних програм;
- дозволяти створювати програми, які можна легко модифікувати.

Покрокова розробка програм

- ***Покрокова розробка*** - це спадний метод проектування програм (вперше був описаний Віртом), що задовольняє перерахованим вище критеріям.
- Вірт дав систематичне формулювання і опис цього методу, якого програмісти раніше дотримуватися інтуїтивно.
- Цей метод вважається багатьма фахівцями в галузі інформатики найважливішою формалізацією в програмуванні 70-х років.
- Даний підхід можна застосувати не тільки до проектування програм, але і до проектування складних систем.

Покрокова розробка програм

- При низхідному підході можна вирішити завдання деталізаціїї, або декомпозиції, на ряд підзадач, які потім необхідно вирішити.

- ***Декомпозиція***, чи ***деталізація***, повинна бути такою, щоб:
  1. підзадачі повинні бути вирішені;
  2. підзадачі вирішуються по можливості з найменшим впливом один на одного;
  3. рішення кожної підзадачі вимагає менше зусиль, ніж первинне завдання;
  4. якщо підзадачі вирішені, то рішення всієї задачі не повинно вимагати істотних додаткових зусиль.

Покрокова розробка програм

- Процес декомпозиції поширюється і на підзадачі. Звичайно, якщо рішення проблеми очевидно або тривіально, немає необхідності вдаватися до декомпозиції.

- Якщо P_0 - це початкове формулювання/рішення задачі, то кінцеве формулювання/рішення цієї ж завдання P_n (програма, що виконується) виходить після серії послідовних кроків деталізації.

- $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$

Рекомендації по деталізації

- Покрокова деталізація є ***ітеративним процесом***.

- Якщо отримане рішення не є остаточним, то процес деталізації повторюється з використанням додаткових відомостей, отриманих на попередніх етапах.

- При вирішенні великих або складних завдань можуть знадобитися додаткові ітерації, перш ніж програміст задовольнить всім вимогам правильності, елегантності і ефективності вирішення.

Рекомендації по деталізації

- Наведемо деякі ***рекомендації*** з розробки програм за допомогою покрокової деталізації.

1. Програма повинна розроблятися поступово. На кожному кроці деталізується один або більше операторів поточної деталізації. Процес деталізації завершується, коли оператори виражені потрібною мовою програмування або коли їх можна механічно відтранслювати в потрібну мову.
2. Деталізації повинні докладно відображати оператори, які вони описують.
3. Можна придумувати будь-які абстрактні оператори. Проте врешті-решт вони повинні переводитися у форму, що виконується.
4. При формулюванні абстрактних операцій необхідно використовувати інформацію про завдання і про область її визначення.
5. Необхідно використовувати позначення, природне для даної предметної області.
6. Кожна деталізація є деяке неявне проектне рішення. Програмісту слід пам'ятати про це і завжди розглядати можливі альтернативи. Необхідно вести протокол найбільш важливих прийнятих рішень.

Рекомендації по деталізації

7. Кожна деталізація є деяке неявне проектне рішення. Програмісту слід пам'ятати про це і завжди розглядати можливі альтернативи. Необхідно вести протокол найбільш важливих прийнятих рішень.
8. Рекурсію слід використовувати при необхідності. Якщо в мові програмування відсутня рекурсія, все ж рекомендується розглянути рекурсивне рішення і, якщо воно обране, перетворити його до нерекурсівного.
9. Типи даних необхідно деталізувати точно так, як і оператори. Програміст повинен виділяти абстрактні типи даних і відокремлювати їх від решти програми, тобто не деталізовувати операції над типами даних, а використовувати виклики процедур.

Рекомендації по деталізації

10. Представлення даних слід відкласти настільки, наскільки це можливо. Це зводить до мінімуму модифікації, якщо замість початкового використовується деяке альтернативне представлення даних.
11. Якщо оператор з'являється більше одного разу, програмісту слід розглянути можливість використання для нього виклику процедури. В цьому випадку деталізація проводиться тільки один раз. Виклики процедури слід використовувати і в тому випадку, якщо вони роблять структуру програми більш зрозумілою.
12. Програміст повинен використовувати інваріанти циклів, якщо програми, що розробляються, містять їх. Інваріанти циклів підказують цікаві рішення для операторів, що складають тіло циклу, і умов його завершення.

Стиль оформлення програми

- 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 чи модіфікувати. «***Пам'ятайте, програми читаються людьми***»,

  (Д. Ван Тассел).

- 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.

Стиль оформлення програми

- ***Правила іменування об'єктів програми.*** При виборі імен програмних об'єктів слід дотримуватися наступних правил:
- ім'я об'єкта має відповідати його змісту, наприклад:
  - MaxItem - максимальний елемент;
  - NextItem - наступний елемент;
- якщо дозволяє мову програмування, можна використовувати символ «_» для візуального поділу імен, що складаються з декількох слів, на-приклад:
  - Max_Item, Next_Item;
- необхідно уникати близьких за написанням імен, наприклад: Index і InDec.

Стиль оформлення програми

- ***Правила оформлення модулів.*** Кожен модуль повинен передувати заголовком, який, як мінімум, містить:
- назва модуля;
- короткий опис його призначення;
- короткий опис вхідних та вихідних параметрів із зазначенням одиниць виміру;
- список використовуваних (що викликаються) модулів;
- короткий опис алгоритму (методу) і (або) обмежень;
- ПІБ автора програми;
- ідентифікаційну інформацію (номер версії і (або) дату останнього коригування).

Стиль оформлення програми

- Наприклад:
- /*
- Функція: float Length_Path (int n, float L[]);
- Мета: визначення сумарної довжини відрізків
- Вхідні дані: N - кількість відрізків,
- L - масив довжин відрізків (в метрах)
- Результат: довжина (в метрах)
- Викликані модулі: немає
- Опис алгоритму: відрізки підсумовуються методом накопичення
- Дата: 25.01.2021 Версія 1.01.
- Автор: Іванов І.І.
- Виправлення: немає
- */

Стиль оформлення програми

- ***Стиль оформлення текстів модулів.*** Стиль оформлення текстів модулів визначає використання відступів, пропусків рядків і коментарів, що полегшують розуміння програми. Як правило, пропуски рядків і коментарів використовують для візуального поділу частин модуля, наприклад:

```
{Перевірка кількості відрізків і вихід, якщо відрізки не задані}
 if (n <0)
  {
    std :: cout << "Кількість відрізків негативно;
  }
{Цикл підсумовування довжин відрізків}
int S = 0;
for (int i = 0; n <n; i +)
  S: = S + Len [i];
```

Стиль оформлення програми

- Для таких мов, як Pascal, C ++ і Java, використання відступів дозволяє прояснити структуру програми: зазвичай додатковий відступ позначає вкладення операторів мови.
- Дещо складніше справа йде з коментарями. Досвід показує, що перекладати з англійської мови кожен оператор програми не потрібно: будь-який програміст, що знає мову програмування, на якому написана програма, без праці прочитає той чи інший оператор. Коментувати слід мети виконання тих чи інших дій, а також групи операторів, пов'язані спільним дією, тобто коментарі повинні містити деяку додаткову (неочевидні) інформацію, наприклад:
- {Перевірка кількості відрізків і вихід, якщо відрізки не задані}
- if (n <0)
- {
- std :: cout << "Кількість відрізків негативно;
- }

Дякую за увагу!